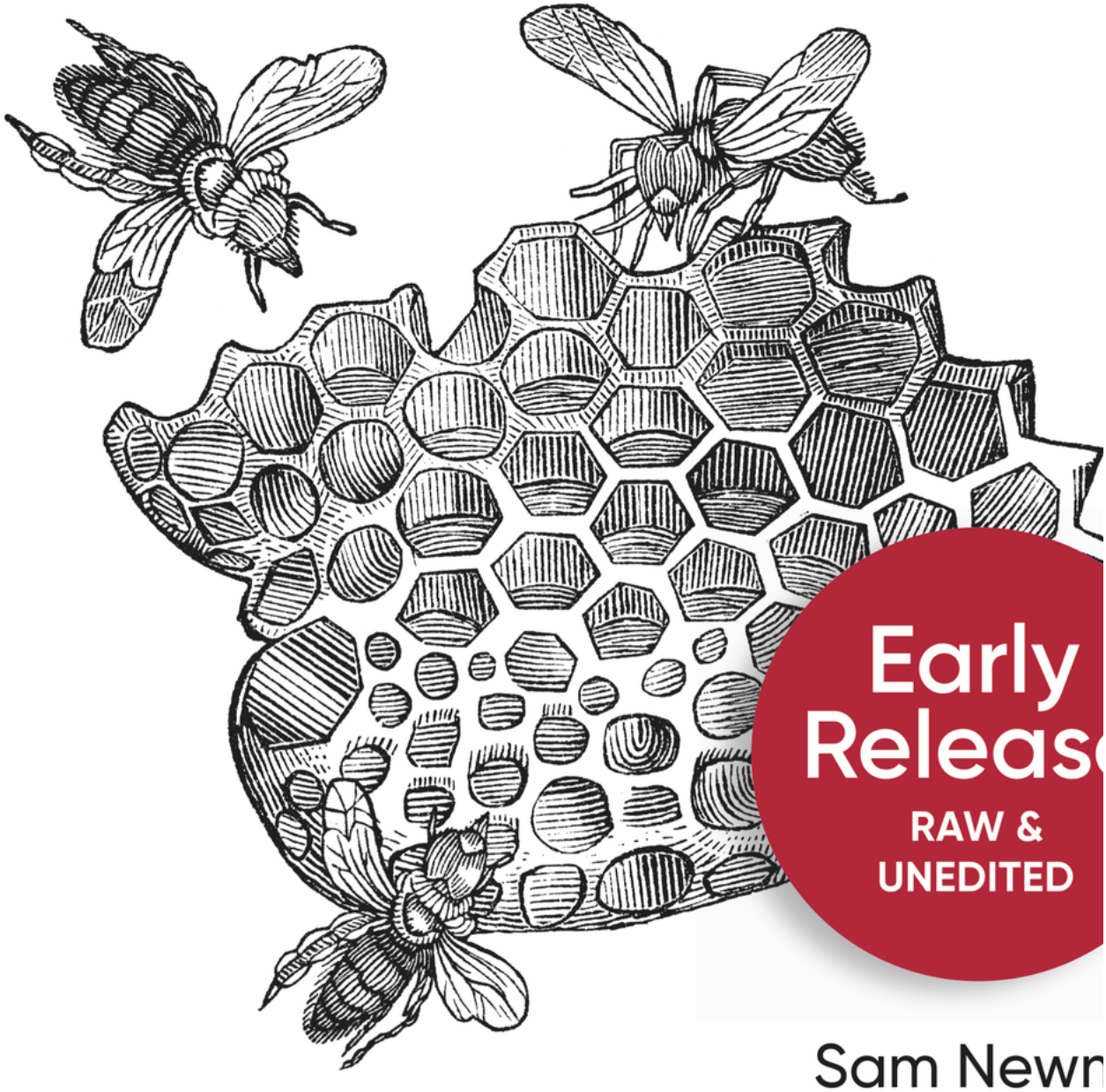
O'REILLY®

Building Microservices

Designing Fine-Grained Systems



Early Release

RAW & UNEDITED

Sam Newm

Building Microservices

Second Edition

Designing Fine-Grained Systems

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Sam Newman

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://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Melissa Duffiel
- Development Editor: Nicole Taché
- Production Editor: Deborah Baker
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- July 2021: Second Edition

Revision History for the Early Release

- 2020-05-22: First Release
- 2020-08-27: Second Release
- 2020-10-14: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492034025 for release details.

978-1-492-03395-0

[LSI]

Chapter 1. What Are Microservices?

Work In Progress

Please note that the text below is currently being reworked for the 2nd edition of the book, and is not in a complete state. This will be Chapter 1 of the final book.

If you have any feedback on the book, or suggestions for the 2nd edition, then please contact me on book-feedback@samnewman.io and/or complete a short survey here: *https://oreil.ly/Bldg_MicroServices_survey*.

Microservices have become an increasingly popular architecture choice in the five years since I wrote the first edition of this book. I can't claim credit for the subsequent explosion in popularity, but the rush of people to make use of microservice architectures means that while many of the ideas I captured previously are now tried and tested, new ideas have also come into the mix, at the same time as earlier practices have fallen out of favour. So it's once again time to distill down the essence of microservice architecture, highlighting the core concepts that make them work.

This book as a whole is designed to give a broad overview of the impact that microservices have on various aspects of software delivery. To start us off, this chapter will take a look at the core ideas behind microservices, the prior art that brought us here, and explore some of the reasons why these architectures are being used so widely.

Microservices At a Glance

Microservices are independently releasable services that are modelled around a business domain. A service encapsulates functionality and makes it accessible to other services via networks—you construct a more complex system from these building blocks. One service might represent inventory, another order management, and yet another shipping, but together they might constitute an entire ecommerce system. Microservices are an architecture choice that is focused on giving you many options for solving the problems you might face.

They are a *type* of service-oriented architecture, albeit one that is opinionated about how service boundaries should be drawn, and one in which independent deployability is key. They are technology agnostic, which is one of the advantages they offer.

From the outside, a single microservice is treated as a black box. It hosts business functionality on one or more network endpoints (for example, a queue or a REST API, as shown in Figure 1-1), over whatever protocols are most appropriate. Consumers, whether they're other microservices or other sorts of programs, access this functionality via these networked endpoints. Internal implementation details (for example, like the technology the service is written in or the way data is stored) are entirely hidden from the outside world. This means microservice architectures avoid the use of shared databases in most circumstances; instead, each microservice encapsulates its own database where required.

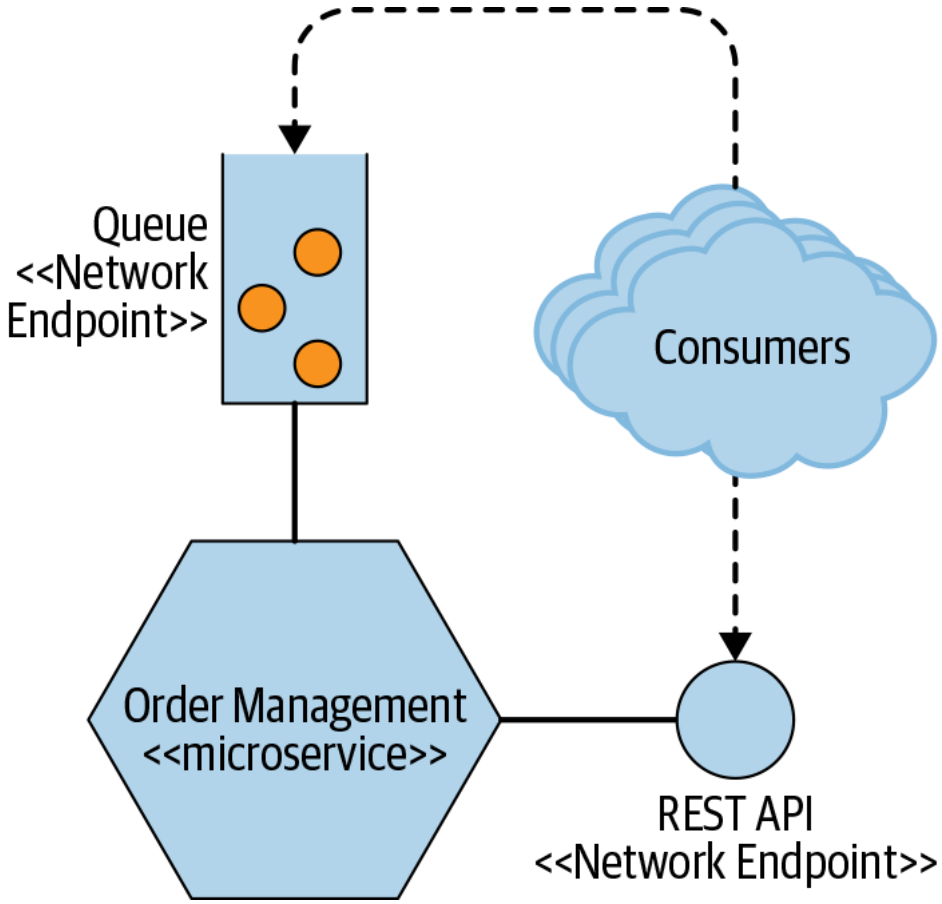


Figure 1-1. A microservice exposing its functionality over a REST API and a queue

Microservices embrace the concept of information hiding.[1] *Information hiding* describes hiding as much information as possible inside a component and exposing as little as possible via external interfaces. This allows for clear separation between what can change easily and what is more difficult to change. Implementation that is hidden from external parties can be changed freely as long as the networked interfaces the microservice exposes don't change in a backward-incompatible fashion. Changes inside a microservice boundary (as shown in Figure 1-1) shouldn't affect an upstream consumer, enabling independent releasability of functionality. This is essential in allowing our microservices to be worked on in isolation and released on demand. Having clear, stable service boundaries that don't change when the internal implementation changes results in systems that have looser coupling and stronger cohesion.

Are Service-Oriented Architecture and Microservices Different Things?

Service-oriented architecture (SOA) is a design approach in which multiple services collaborate to provide a certain 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 large monolithic applications. It is an approach that aims to promote the reusability of software; two or more end-user applications, for example, could 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 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. 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.

I've seen plenty of examples of SOA where teams were striving to make the services smaller, but still had everything coupled to a database and had to deploy everything together. Service Oriented? Yes. But it's not microservices.

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

Key Concepts of Microservices

A few core ideas are important to understand when exploring microservices. Given that some of these aspects are often overlooked, I think it's vital to explore these concepts further to help ensure that you better understand just what it is that makes microservices work.

Independently Deployability

Independent deployability is the idea that we can make a change to a microservice, deploy it, and release that change to our users, without having to deploy any other services. More important, it's not just the fact that we can do this, it's that this is *actually* how you manage deployments in your system. It's a discipline you adopt as your default release approach. This is a simple idea that is nonetheless complex in execution.

Tip

If you take only one thing from this book, and the concept of microservices in general, it should be this: ensure that you embrace the concept of independent deployability of your microservices. Get into the habit of deploying and releasing changes to a single microservice into production without having to deploy anything else. From this, many good things will follow.

To ensure independent deployability, we need to make sure our services are *loosely coupled*: we need to be able to change one service without having to change anything else. This means we need explicit, well-defined, and stable contracts between services. Some implementation choices make this difficult—the sharing of databases, for example, is especially problematic.

Independent deployability in and of itself is clearly incredibly valuable. But to achieve independent deployability, there are so many other things you have to get right that in turn have their own benefits. So you can also see the focus on independent deployability as a forcing function - by focusing on this as an outcome you'll also achieve a number of ancillary benefits.

The desire for loosely coupled services with stable interfaces guides our thinking about how we find service boundaries in the first place.

Modelled Around a Business Domain

Techniques like domain-driven design can allow you to structure your code to better represent the real-world domain that the software operates in.[2] With microservice architectures, we use this same idea to define our service boundaries. By modelling services around business domains, we can make it easier to roll out new functionality, and make it easier to recombine microservices in different ways to deliver new functionality to our users.

Rolling out a feature that requires changes to one or more microservices is expensive. You need to coordinate the work across each service (and potentially across separate teams) and carefully manage the order in which the new versions of these services are deployed. That takes a lot more work than making the same change inside a single service (or, for that matter, a monolith). It therefore follows that we want to find ways to make cross-service changes as infrequent as possible.

I often see layered architectures, as typified in Figure 1-2 by the three-tiered architecture. Here, each layer in this architecture represents a different service boundary, with each service boundary based on related technical functionality. If I need to make a change to just the presentation layer in this example, that would be fairly efficient. However, experience has shown that changes in functionality typically span multiple layers in these types of architectures—requiring changes in presentation, application, and data tiers. This problem is exacerbated if the architecture is even more layered than the simple example in Figure 1-2; often each tier may be split into further layers.

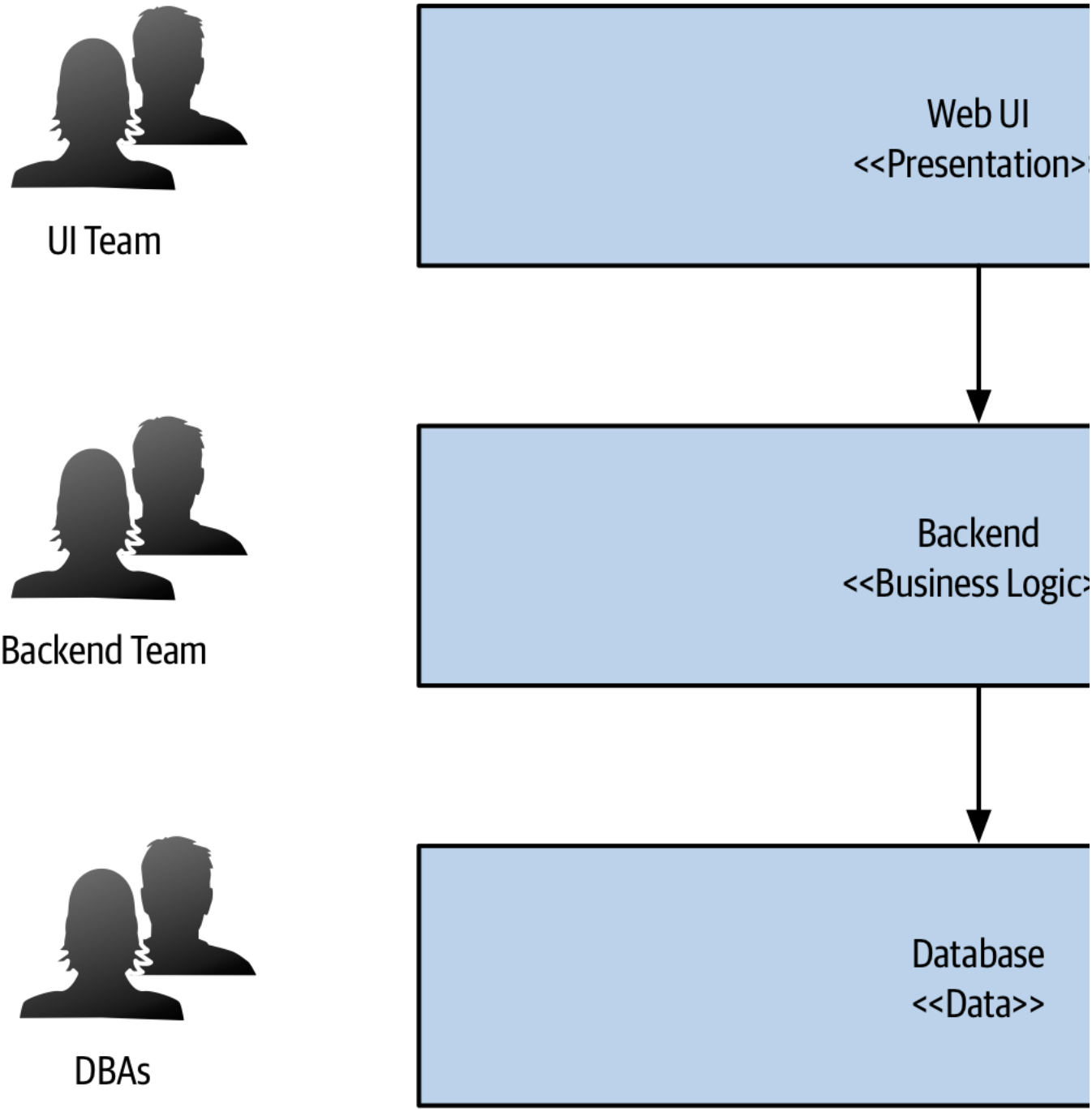


Figure 1-2. A traditional three-tiered architecture

By making our services end-to-end slices of business functionality, as shown in Figure 1-3, we ensure that our architecture is arranged to make changes to business functionality as efficient as possible. Each service, if needed, can encapsulate presentation, business logic, and data storage. Arguably, with microservices we have made a decision to prioritize high cohesion of business functionality over high cohesion of technical functionality.

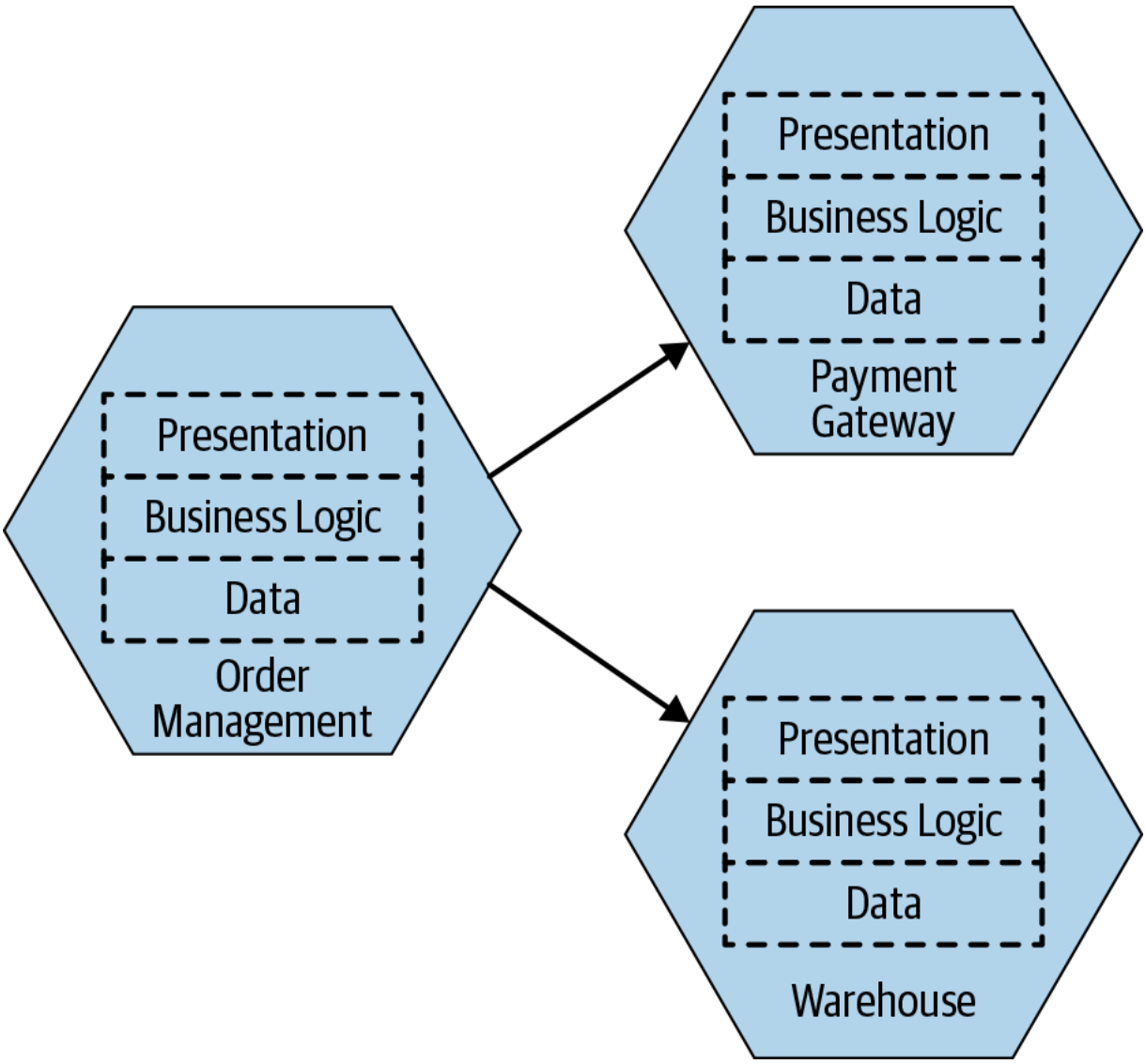


Figure 1-3. Each microservice, if required, can encapsulate presentation, business logic, and data storage functionality

We come back to the interplay of domain-driven design and how this interacts with organizational design later in this chapter.

Owning Their Own State

One of the things I see people having the hardest time with is the idea that microservices should not share databases. If one service wants to access data held by another service, it should go and ask that service for the data it needs. This gives the service the ability to decide what is shared and what is hidden. This allows us to clearly separate functionality that can change freely (our internal implementation) from the functionality that we want to change infrequently (the external contract that the consumers use).

If we want to make independent deployability a reality, we need to ensure that we limit making backward-incompatible changes to our microservices. If we break compatibility with upstream consumers, we will force them to change as well. Having a clean delineation between internal implementation detail and an external contract for a microservice can help reduce the need for backward-incompatible changes.

Hiding of internal state in a microservice is analogous with the practice of encapsulation in object-oriented (OO) programming. Encapsulation of data in OO systems is an example of information hiding in action.

Tip

Don’t share databases unless you really need to. And even then, do everything you can to avoid it. In my opinion, sharing databases is one of the worst things you can do if you’re trying to achieve independent deployability.

As discussed in the previous section, we want to think of our services as end-to-end slices of business functionality that, where appropriate, encapsulate user interface (UI), business logic, and data. This is because we want to reduce the effort needed to change business-related functionality. The encapsulation of data and behavior in this way gives us high cohesion of business functionality. By hiding the database that backs our service, we also ensure that we reduce coupling. We come back to coupling and cohesion in Chapter 2.

Size

“How big should a microservice be?” is one of the most common questions I hear. Considering the word “micro” is right there in the name, this comes as no surprise. However, when you get into what makes microservices work as a type of architecture, the concept of size is actually one of the least interesting things.

How do you measure size? Lines of code? That doesn’t make much sense to me. Something that might require 25 lines of code in Java could be written in 10 lines of Clojure. That’s not to say Clojure is better or worse than Java; it’s simply that some languages are more expressive than others.

James Lewis, technical director at ThoughtWorks, has been known to say “a microservice should be as big as my head”. On first glance, this is doesn’t seem terribly helpful. After all, how big is James’ head exactly? The rationale behind this statement is that a microservice should be kept to the size where it can be easily understood. The challenge here of course is that people’s ability to understand something isn’t the same, and as such you’ll need to make your own judgement regarding what size works for you. An experienced team may be able to better manage a larger codebase than another. So perhaps the read James’ quote here as “A microservice should be as big as *your* head”.

The closest I think I get to “size” having any meaning in terms of microservices is something *Microservice Patterns* author Chris Richardson once said—that the goal of microservices is to have “as small an interface as possible.” That aligns with the concept of information hiding again, but it does represent an attempt to find meaning in the term “microservices” that wasn’t there initially. When the term was first used to define these architectures, the focus, at least initially, was not specifically on size of the interfaces.

Ultimately, the concept of size is highly contextual. Speak to a person who has worked on a system for 15 years, and they’ll feel that their system with 100,000 lines of code is really easy to understand. Ask the opinion of someone brand-new to the project, and they’ll feel it’s much too big. Likewise, ask a company that has just embarked on its microservice transition, having perhaps 10 or fewer microservices, and you’ll get a different answer than you would from a similar-sized company where microservices have been the norm for many years, and it now has hundreds.

I urge people not to worry about size. When you are first starting out, it’s much more important that you focus on two key things. First, how many microservices can you handle? As you have more services, the complexity of your system will increase, and you’ll need to learn new skills (and perhaps adopt new technology) to cope with this. It’s for this reason that I am a strong advocate for incremental migration to a microservice architecture. Second, how do you define microservice boundaries to get the most out of them, without everything becoming a horribly coupled mess? These are the topics that are much more important to focus on when you start your journey.

Flexibility

James Lewis, has been known to say that “microservices buy you options.” Lewis was being deliberate with his words—they *buy* you *options*. They have a cost, and you must decide whether the cost is worth the options you want to take up. The resulting flexibility on a number of axes—organizational, technical, scale, robustness—can be incredibly appealing.

We don’t know what the future holds, so we’d like an architecture that can theoretically help us solve whatever problems we might face further down the road. Finding a balance between keeping your options open and bearing the cost of architectures like this can be a real art.

Think of adopting microservices as less like flicking a switch, and more like turning a dial. As you turn up the dial, and you have more microservices, you have increased flexibility. But you likely ramp up the pain points too. This is yet another reason I strongly advocate incremental adoption of microservices. By turning up the dial gradually, you are better able to assess the impact as you go, and stop if required.

Alignment of Architecture and Organization

Music Corp, an ecommerce company that sells CDs online, uses the simple three-tiered architecture shown earlier and depicted again in Figure 1-4. We’ve decided to move Music Corp kicking and screaming into the 21st century, and as part of that, we’re assessing the existing system architecture. We have a web-based UI, a business logic layer in the form of a monolithic backend, and data storage in a traditional database. These layers, as is common, are owned by different teams. We’ll be coming back to the trials and tribulations of Music Corp throughout the book.

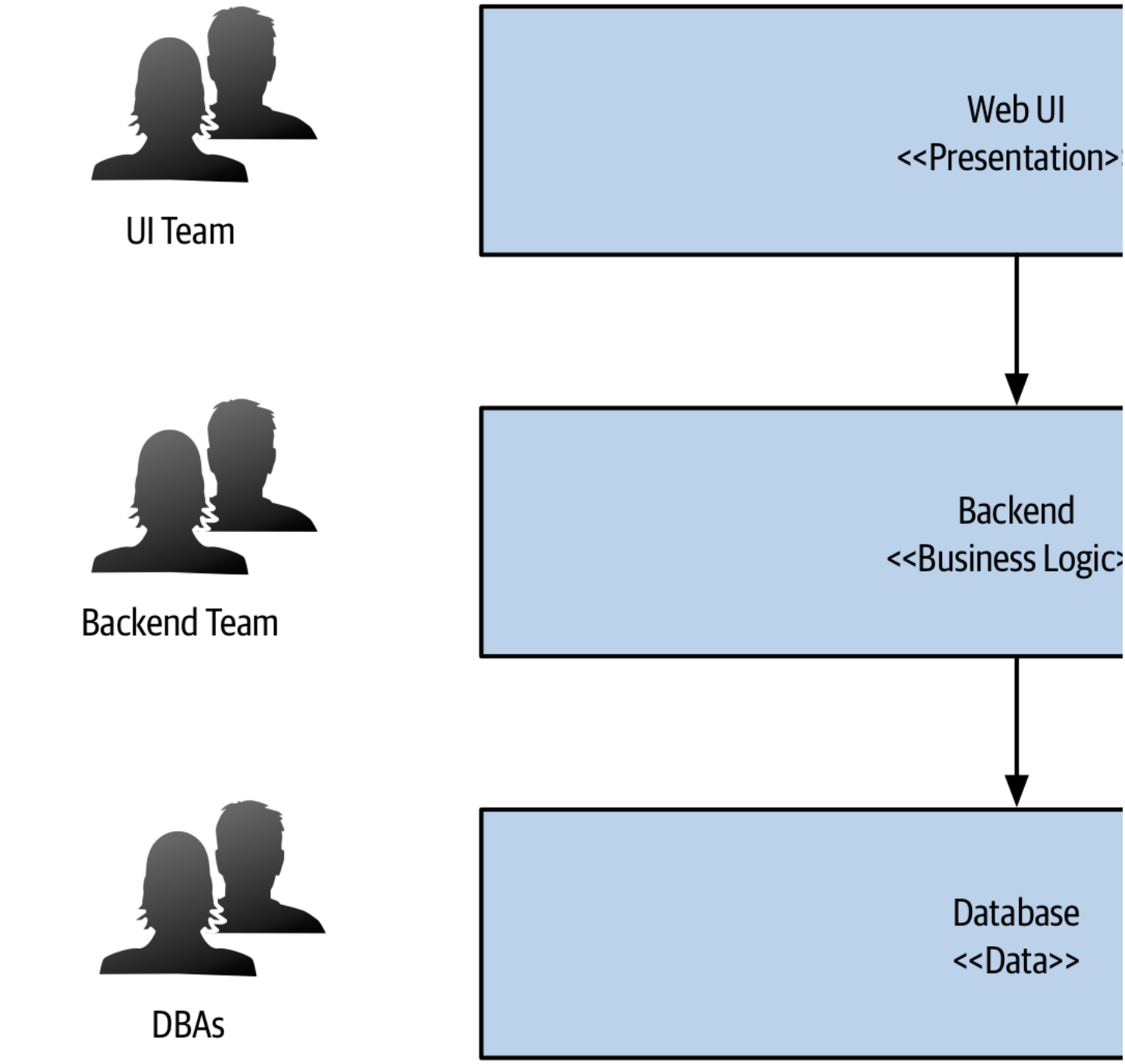


Figure 1-4. Music Corp's systems as a traditional three-tiered architecture

We want to make a simple change to our functionality: we want to allow our customers to specify their favorite genre of music. This change requires us to change the UI to show the genre choice UI, the backend service to allow for the genre to be surfaced to the UI and for the value to be changed, and the database to accept this change. These changes will need to be managed by each team and deployed in the correct order, as outlined in Figure 1-5.

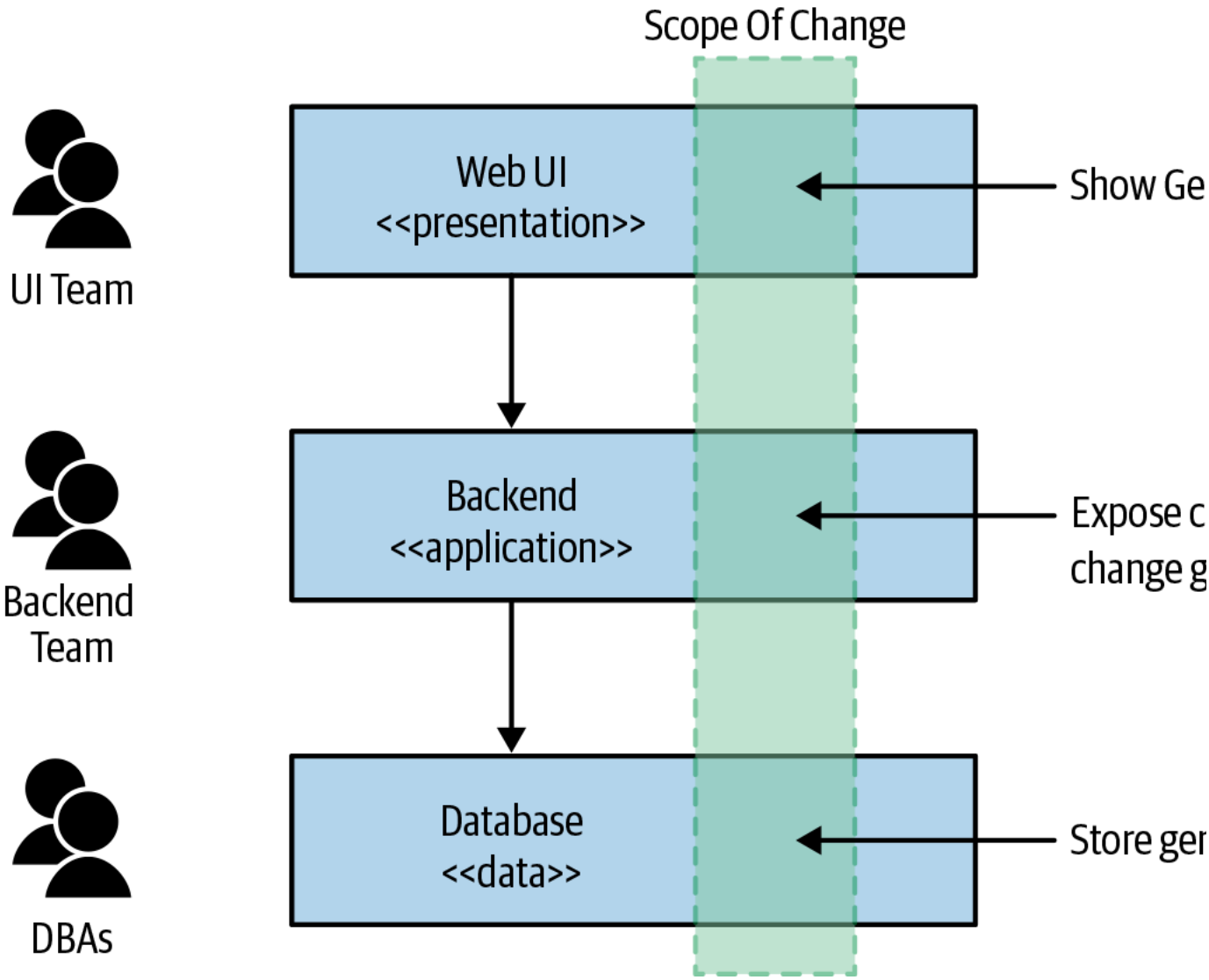


Figure 1-5. Making a change across all three tiers is more involved

Now this architecture isn't bad. All architecture ends up getting optimized around a set of goals. The three-tiered architecture is so common partly because it is universal—everyone has heard about it. So picking a common architecture that you might have seen elsewhere is often one reason we keep seeing this pattern. But I think the biggest reason we see this architecture again and again is because it is based on how we organize our teams.

The now famous Conway's law states the following:

> Any organization that designs a system…will inevitably produce a design whose structure is a copy of the organization's communication structure
>
> Melvin Conway, *How Do Committees Invent?*

The three-tiered architecture is a good example of this in action. In the past, the primary way IT organizations grouped people was in terms of their core competency: database admins were in a team with other database admins; Java developers were in a team with other Java developers; and frontend developers (who nowadays know exotic things like JavaScript and native mobile application development) were in yet another team. We group people based on their core competency, so we create IT assets that can be aligned to those teams.

So that explains why this architecture is so common. It's not bad; it's just optimized around one set of forces—how we traditionally grouped people, around familiarity. But the forces have changed. Our aspirations around our software have changed. We now group people in poly-skilled teams, to reduce hand-offs and silos. We want to ship software much more quickly than ever before. That is driving us to make different choices about the way we organize our teams, and therefore to organize them in terms of the way we break our systems apart.

Most changes that we are asked to make to our system relate to changes in business functionality. But in Figure 1-5, our business functionality is, in effect, spread across all three tiers, increasing the chance that a change in functionality will cross layers. This is an architecture that has high cohesion of related technology but low cohesion of business functionality. If we want to make it easier to make changes, instead we need to change how we group code—we choose cohesion of business functionality rather than technology. Each service may or may not then end up containing a mix of these three layers, but that is a local service implementation concern.

Let's compare this with a potential alternative architecture, illustrated in Figure 1-6. We have a dedicated Customer service, which exposes a UI to allow customers to update their information, and the state of the customer is also stored within this service. The choice of a favorite genre is associated with a given customer, so this change is much more localized. In Figure 1-6, we also show the list of available genres being fetched from a Catalog service, likely something that would already be in place. We also see a new Recommendations service accessing our favorite genre information, something that could easily follow in a subsequent release.

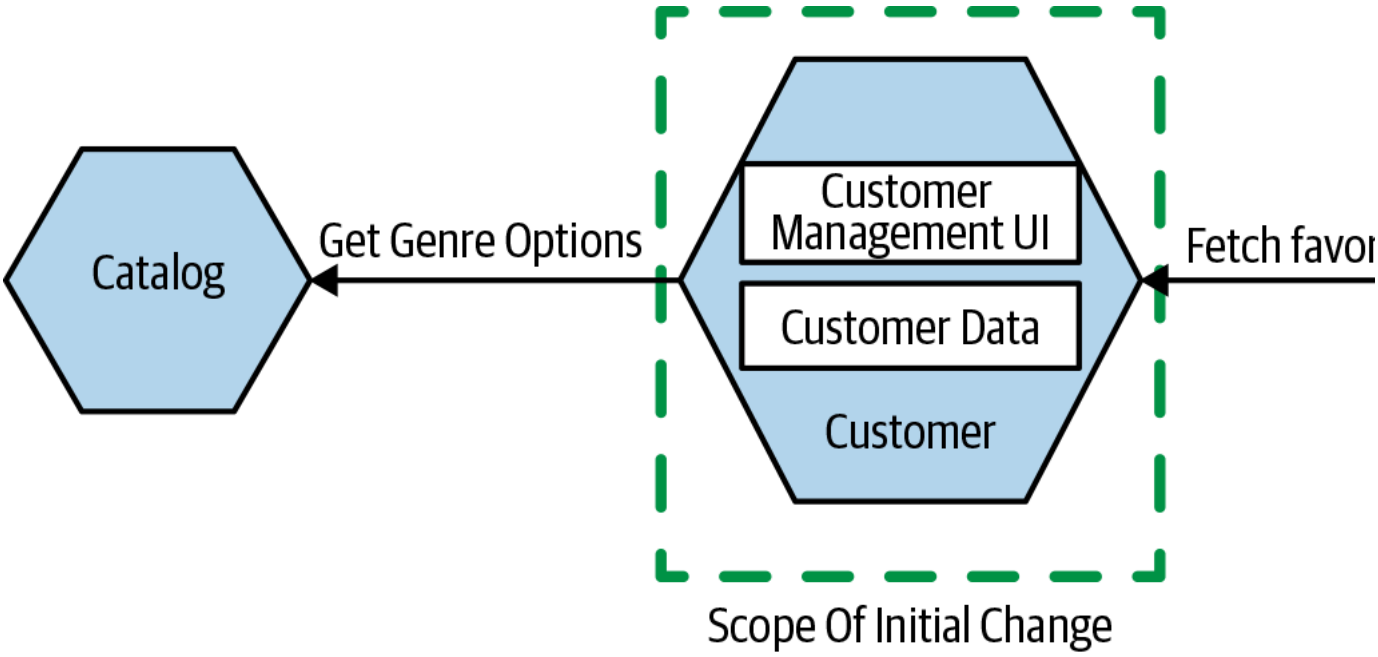


Figure 1-6. A dedicated Customer service can make it much easier to record the favorite musical genre for a customer

In such a situation, our Customer service encapsulates a thin slice of each of the three tiers—it has a bit of UI, a bit of application logic, and a bit of data storage—but these layers are all encapsulated in the single service. Our business domain becomes the primary force driving our system architecture, hopefully making it easier to make changes, as well as making it easier for us to align our teams to lines of business within the organization.

The Monolith

We've spoken about microservices, but microservices are most often discussed as an architectural approach that is an alternative to monolithic architecture. To better help distinguish the microservice architecture, and to help you better understand whether microservices are worth considering, I should also discuss what exactly I mean by *monoliths*.

When I talk about monoliths throughout this book, I am primarily referring to a unit of deployment. When all functionality in a system must be deployed together, we consider it a monolith. Arguably, multiple architectures fit this definition, but I'm going to discuss those I see most often: the single-process monolith, the modular monolith, and the distributed monolith.

The Single-Process Monolith

The most common example that comes to mind when discussing monoliths is a system in which all of the code is deployed as a *single process*, as in Figure 1-7. You may have multiple instances of this process for robustness or scaling reasons, but fundamentally all the code is packed into a single process. In reality, these single-process systems can be simple distributed systems in their own right because they nearly always end up reading data from or storing data into a database, or presenting information to web or mobile applications.

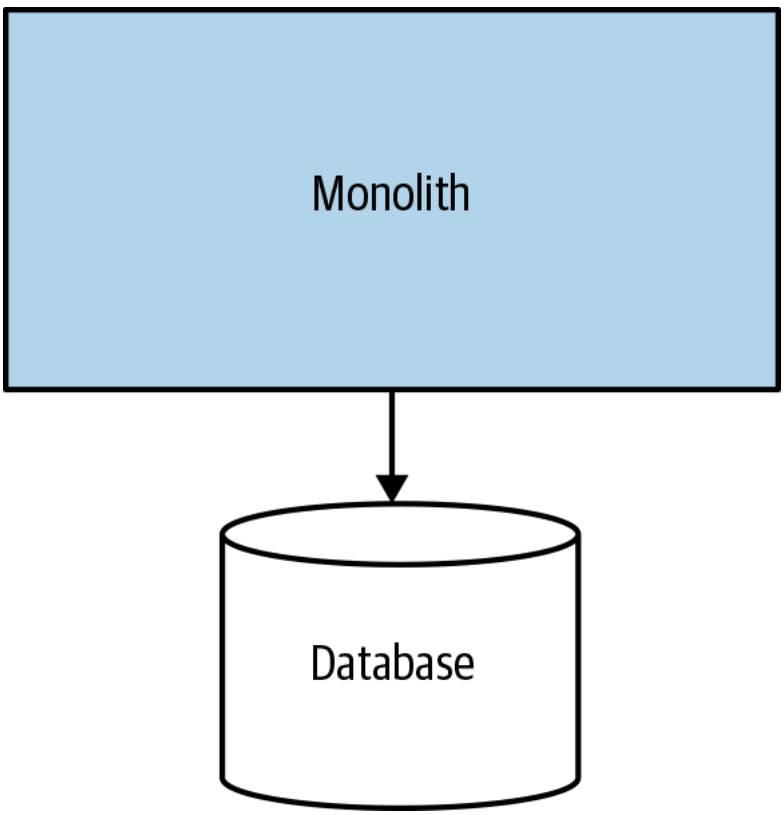


Figure 1-7. In a single-process monolith, all code is packaged into a single process

Although this fits most people's understanding of a classic monolith, most systems I encounter are somewhat more complex than this. You may have two or more monoliths that are tightly coupled to one another, potentially with some vendor software in the mix.

The Modular Monolith

As a subset of the single-process monolith, the *modular monolith* is a variation in which the single process consists of separate modules. Each can be worked on independently, but all still need to be combined together for deployment, as shown in Figure 1-8. The concept of breaking software into modules is nothing new; modular software has its roots in work done around structured programming from the 1970s, and even further back than that. Nonetheless, this is still not an approach that I see enough organizations properly engage with.

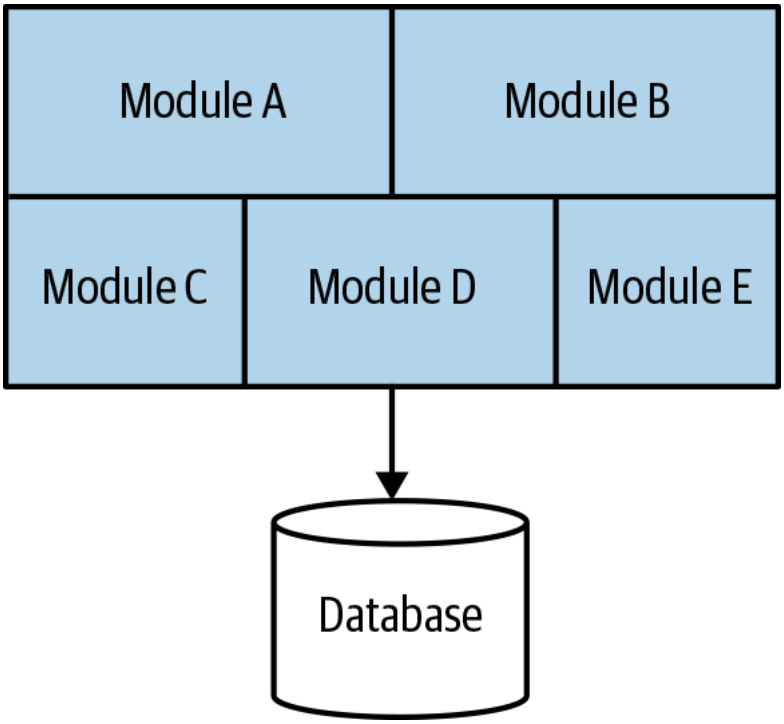


Figure 1-8. In a modular monolith, the code inside the process is divided into modules

For many organizations, the modular monolith can be an excellent choice. If the module boundaries are well defined, it can allow for a high degree of parallel work, while avoiding the challenges of the more distributed microservice architecture by having a much simpler deployment topology. Shopify is a great example of an organization that has used this technique as an alternative to microservice decomposition, and it seems to work really well for that company.[3]

One of the challenges of a modular monolith is that the database tends to lack the decomposition we find in the code level, leading to significant challenges if you want to pull apart the monolith in the future. I have seen some teams attempt to push the idea of the modular monolith further, having the database decomposed along the same lines as the modules, as shown in Figure 1-9.

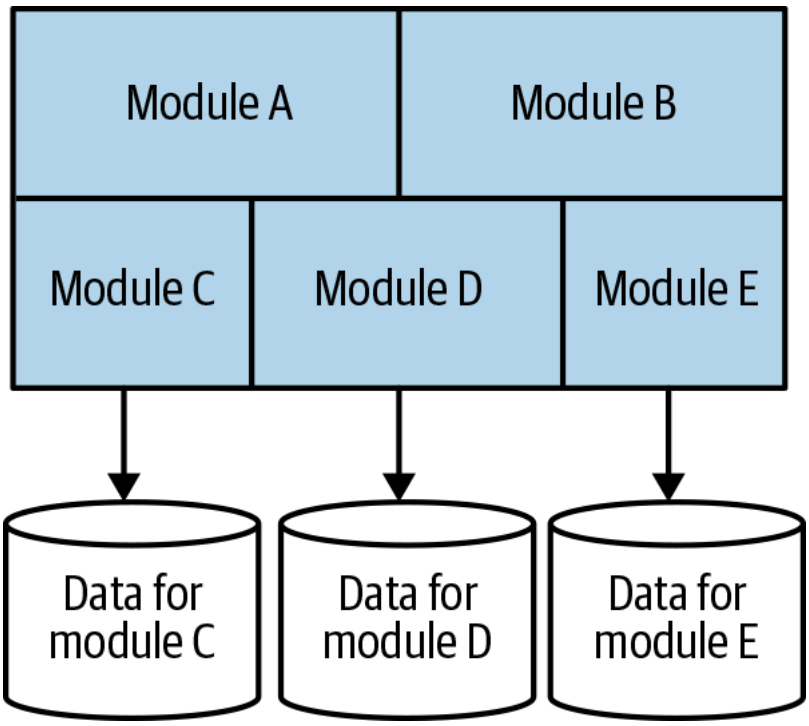


Figure 1-9. A modular monolith with a decomposed database

The Distributed Monolith

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.[4]
>
> Leslie Lamport

A *distributed monolith* is a system that consists of multiple services, but for whatever reason, the entire system must be deployed together. A distributed monolith might well meet the definition of an SOA, but all too often, it fails to deliver on the promises of SOA. In my experience, distributed monoliths have all the disadvantages of a distributed system, *and* the disadvantages of a single-process monolith, without having enough upsides of either. Encountering a number of distributed monoliths in my work has in large part influenced my own interest in microservice architecture.

Distributed monoliths typically emerge in an environment in which not enough focus was placed on concepts like information hiding and cohesion of business functionality. Instead, highly coupled architectures cause changes to ripple across service boundaries, and seemingly innocent changes that appear to be local in scope break other parts of the system.

Monoliths and Delivery Contention

As more and more people work in the same place, they get in one another's way. For example, different developers wanting to change the same piece of code; different teams wanting to push functionality live at different times (or delay deployments); confusion around who owns what, and who makes decisions. A multitude of studies have been done that show the challenges of confused lines of ownership.[5] I refer to this problem as *delivery contention*.

Having a monolith doesn't mean you will definitely face the challenges of delivery contention any more than having a microservice architecture means that you won't ever face the problem. But a microservice architecture does give you more concrete boundaries around which ownership lines can be drawn in a system, giving you much more flexibility regarding how to reduce this problem.

Advantages of Monoliths

Some monoliths, such as the single-process or modular monoliths, have a whole host of advantages too. Its much simpler deployment topology can avoid many of the pitfalls associated with distributed systems. This can result in much simpler developer workflows, and monitoring, troubleshooting, and activities like end-to-end testing can be greatly simplified as well.

Monoliths can also simplify code reuse within the monolith itself. If we want to reuse code within a distributed system, we need to decide whether we want to copy code, break out libraries, or push the shared functionality into a service. With a monolith, our choices are much simpler, and many people like that simplicity—all the code is there; just use it!

Unfortunately, people have come to view the monolith as something to be avoided—as something inherently problematic. I've met multiple people for whom the term *monolith* is synonymous with *legacy*. This is a problem. A monolithic architecture is a choice, and a valid one at that. I'd go further and say that in my option it is the sensible default choice as an architectural style. In other words, I am looking for a reason to be convinced to use microservices, rather than looking for a reason not to use them.

If we fall into the trap of systematically denigrating the monolith as a viable option for delivering our software, we're at risk of not doing right by ourselves or by the users of our software.

Enabling Technology

As I touched on earlier, I don't think you need to adopt lots of new technology when you first start using microservices. In fact, that can be counterproductive. Instead, as you ramp up your microservice architecture, you should be constantly on the lookout for issues caused by your increasingly distributed system, and then look for technology that might help.

That said, technology has played a large part in the adoption of microservices as a concept. Understating the tools that are available to you to help get the most out of this architecture is going to be a key part to making any implementation of microservices a success. In fact, I would go as far to say that microservices require an understanding of the supporting technology to such a degree that previous distinctions between logical and physical architecture can be problematic - if you are involved in helping shape a microservice architecture, you'll need a breadth of understanding of these two worlds.

We'll be exploring a lot of this technology in detail in subsequent chapters, but before that, let's briefly introduce some of the enabling technology that might help you should you decide to make use of microservices.

Log Aggregation and Distributed Tracing

With the number of processes you are managing increasing, it can be difficult to understand how your system is behaving in a production setting. This, in turn, can make troubleshooting much more difficult. We'll be exploring these ideas in more depth in [Link to Come], but at a bare minimum, I strongly advocate for the implementation of a log aggregation system as a prerequisite for adopting a microservice architecture.

Tip

Be cautious in taking on too much new technology when you start off with microservices. That said, a log aggregation tool is so essential that you should consider it a pre-requisite for adopting microservices.

These systems allow you to collect and aggregate logs from across all your services, providing you a central place from which logs can be analyzed, and even made part of an active alerting mechanism. Many options in this space can cater to numerous situations. I'm a big fan of Humio for several reasons, but the simple logging services provided by the main public cloud vendors might be good enough to get you started.

These log aggregation tools can be made even more useful by implementing correlation IDs, in which a single ID is used for a related set of service calls—for example, the chain of calls that might be triggered due to user interaction. By logging this ID as part of each log entry, isolating the logs associated with a given flow of calls becomes much easier, making troubleshooting much easier.

As your system grows in complexity, it becomes essential to consider tools that allow you to better explore what your system is doing, providing the ability to analyze traces across multiple services, detect bottlenecks, and ask questions of your system that you didn't know you would want to ask in the first place. Open source tools can provide some of these features. One example is Jaeger, which focuses on the distributed tracing side of the equation.

But products like LightStep and Honeycomb (shown in Figure 1-10), take these ideas further. They represent a new generation of tools moving beyond traditional monitoring approaches, making it much easier to explore the state of your running system. You might already have more conventional tools in place, but you really should look at the capabilities these products provide. They've been built from the ground up to solve the sorts of problems that operators of microservice architectures have to deal with.

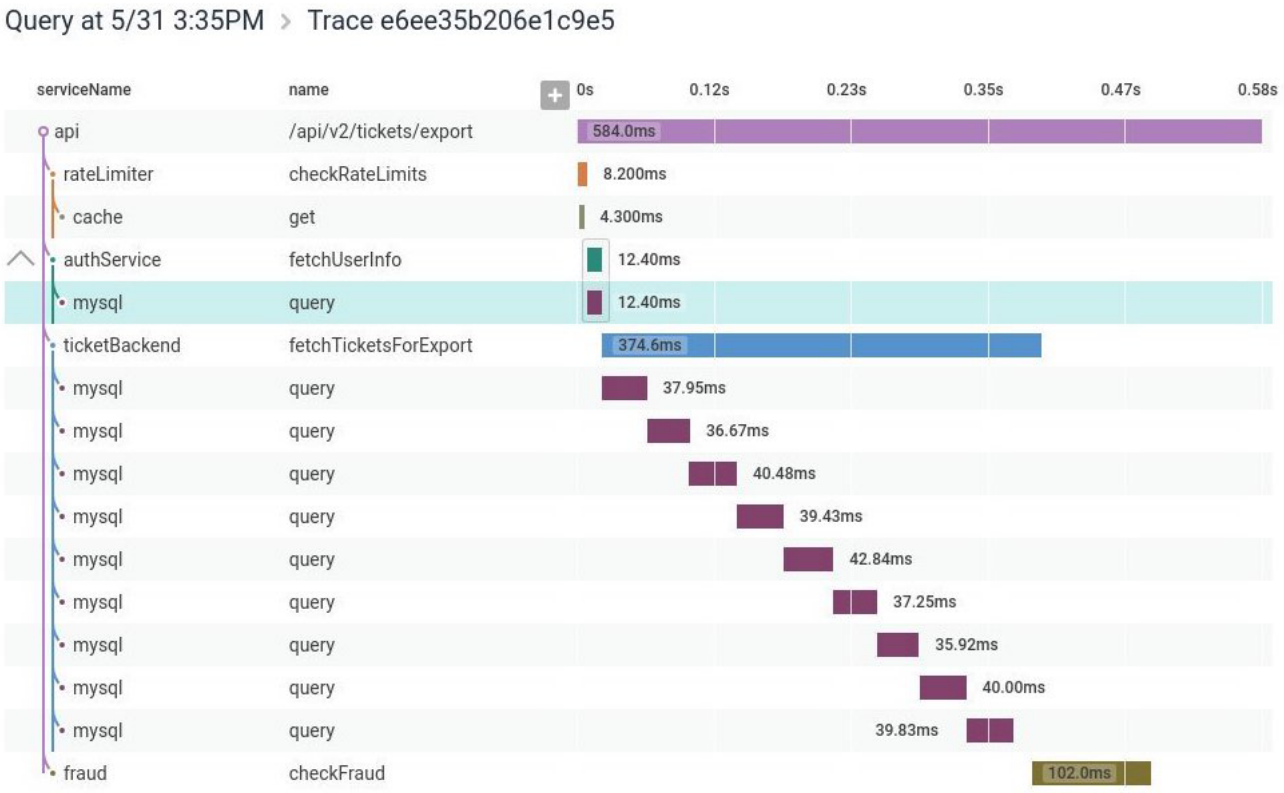


Figure 1-10. A distributed trace shown in Honeycomb, allowing you to identify where time is being spent for operations that can span multiple microservices

Containers and Kubernetes

Ideally, you want to run each microservice instance in isolation. This ensures that issues in one microservice can't affect another—for example, by gobbling up all the CPU. Virtualization is one way to create isolated execution environments on existing hardware, but normal virtualization techniques can be quite heavy when we consider the size of our microservices. *Containers*, on the other hand, provide a much more lightweight way to provision isolated execution for service instances, resulting in faster spin-up times for new container instances, along with being much more cost effective for many architectures.

After you begin playing around with containers, you'll also realize that you need something to allow you to manage these containers across lots of underlying machines. Container orchestration platforms like *Kubernetes* do exactly that, allowing you to distribute container instances in such a way as to provide the robustness and throughput your service needs, all while allowing you to make efficient use of the underlying machines. In Chapter 7 we'll come back and explore the concepts of operational isolation, containers, and kubernetes.

Don't feel the need to rush to adopt Kubernetes, or even containers, for that matter. They absolutely offer significant advantages over more traditional deployment techniques, but it's difficult to justify if you have only a few services. After the overhead of managing deployment begins to become a significant headache, start considering containerization of your service and the use of Kubernetes. But if you do end up doing that, do your best to ensure that someone else is running the Kubernetes cluster for you, perhaps by making use of a managed service on a public cloud provider. Running your own Kubernetes cluster can be a significant amount of work!

Streaming

Although with microservices we are moving away from monolithic databases, we still need to find ways to share data between services. This is happening at the same time as organizations are wanting to move away from batch reporting operations, toward more real-time feedback, allowing them to react more quickly. Products that allow for the easy streaming and processing of what can often be large volumes of data have therefore become popular with people using microservice architectures.

Apache Kafka has become the de facto choice for many for streaming data in a microservice environment, and for good reason. Capabilities like message permanence, compaction, and the ability to scale to handle large volumes of messages can be incredibly useful. Kafka has also started adding stream-processing capabilities in the form of KSQL, but you can also use it with dedicated stream-processing solutions like Apache Flink. Debezium is an open source tool developed to help stream data from existing datasources over Kafka, helping ensure that traditional datasources can become part of a stream-based architecture. In Chapter 3 we'll look at how streaming technology can play a part in microservice integration.

Public Cloud and Serverless

Public cloud providers, or more specifically the main three—Google Cloud, Microsoft Azure, and Amazon Web Services (AWS)—provide a huge array of managed services and deployment options for managing your application. As your microservice architecture grows, more and more work will be pushed into the operational space. Public cloud providers offer a host of managed services, from managed database instances or Kubernetes clusters, to message brokers or distributed filesystems. By making use of these managed services, you are offloading a large amount of this work to a third party that is arguably better able to deal with these tasks.

Of particular interest among the public cloud offerings are the products that sit under the banner of *serverless*. These products hide away the underlying machines, allowing you to work at a higher level of abstraction. Examples of serverless products include message brokers, storage solutions, and databases. Function as a Service (FaaS) platforms are of special interest because they provide a nice abstraction around the deployment of code. Rather than worrying about how many servers you need to run your service, you just deploy your code and let the underlying platform handle spinning up instances of your code on demand. We'll be looking at servleress in more detail in Chapter 7.

Advantages of Microservices

The advantages 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 because they take a more opinionated stance in the way service boundaries are defined. By combining the concepts of information hiding and domain-driven design along with the power of distributed systems, they can help deliver significant gains over other forms of distributed architectures.

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 might also decide that the way 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 1-11.

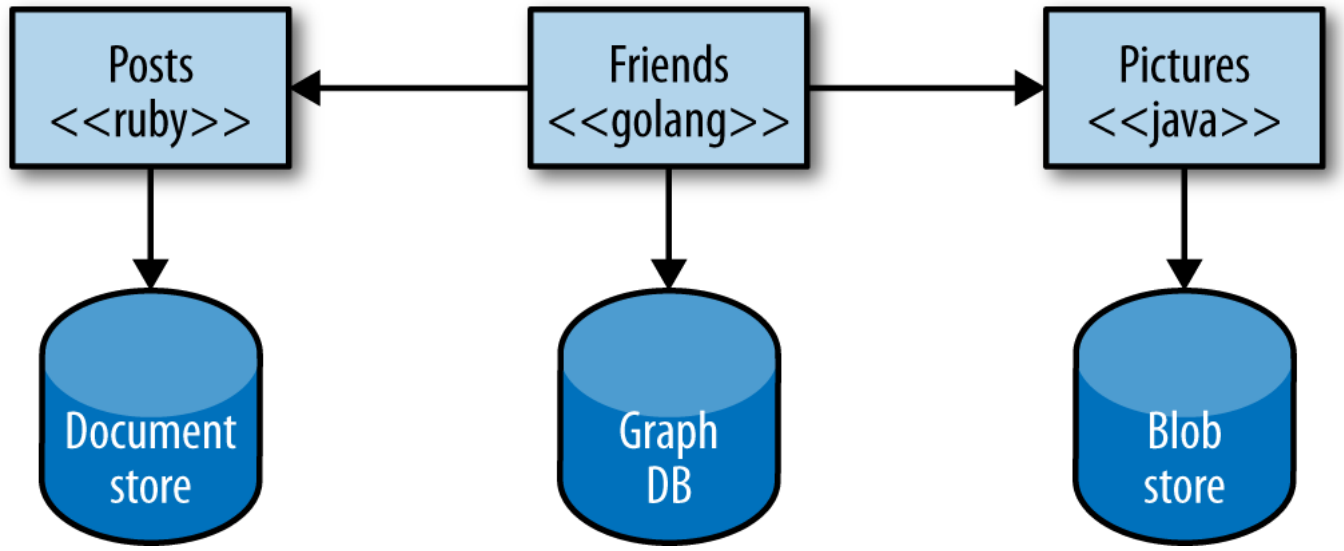


Figure 1-11. Microservices can allow you to more easily embrace different technologies

With microservices, we are also able to more quickly adopt technology and to understand how new advancements might 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 affect much of my system. With a system consisting of multiple services, I have multiple new places 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.

Embracing multiple technologies doesn't come without 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 because those companies 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.

Robustness

A key concept in improving robustness of your application 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 some of the constituent services and degrade functionality accordingly.

We do need to be careful, however. To ensure that our microservice systems can properly embrace this improved robustness, 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) those failures should have on the end users of our software.

Scaling

With a large, monolithic service, we need to scale everything together. Perhaps one small part of our overall system is constrained in performance, but if that behavior is locked up in a giant monolithic application, we need to handle scaling everything as a piece. With smaller services, we can scale just those services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware, as illustrated in Figure 1-12.

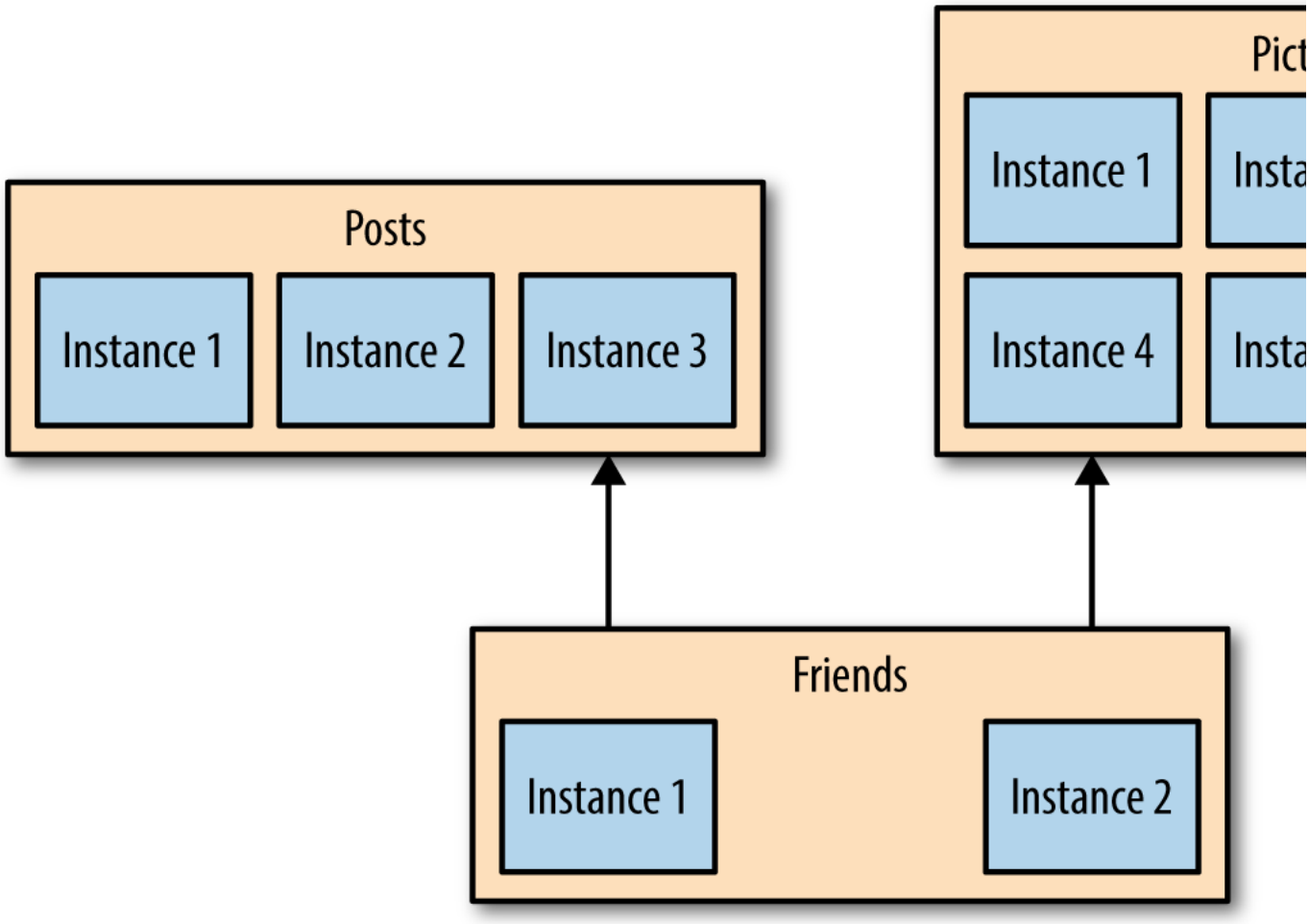


Figure 1-12. You can target scaling at just the microservices that need it

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 it has more than 450 microservices, each one running on multiple separate machines.

When embracing on-demand provisioning systems like those provided by AWS, 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 monolithic application requires the entire application to be deployed in order to release the change. That could be a large-impact, high-risk deployment. In practice, deployments such as these end up happening infrequently because of understandable fear. Unfortunately, this means that our changes continue to build up between releases, until the new version of our application entering 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 quickly isolated to an individual service, making fast rollback easy to achieve. It also means that we can get our new functionality out to customers faster. This is one of the main reasons organizations like Amazon and Netflix use these architectures—to ensure that they remove as many impediments as possible to getting software out the door.

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. Microservices also allow us to change ownership of services as the organization changes—enabling us to maintain the alignment between architecture and organization in the future.

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 applications 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!

Microservice Pain Points

Microservice architectures bring a host of benefits, as we've already seen. But they also bring a host of complexity. If you are considering adopting a microservice architecture, it's important that you do so being able to compare the good with the bad. In reality, most of these pain points can be laid at the door of distributed systems, and so would just as likely to be evident in a distributed monolith as a microservice architecture.

We'll be covering many of these issues in depth throughout the rest of the book - in fact I'd argue that the bulk of this book is about dealing with the pain, suffering, and horror of owning a microservice architecture.

Developer Experience

As you have more and more services, the developer experience can begin to suffer. More resource-intensive runtimes like the JVM can limit the number of microservices that can be run on a single developer machine. I could probably run four or five JVM-based microservices as separate processes on my laptop, but could I run 10 or 20? Probably not. Even with less-taxing runtimes, there is a limit to the number of things you can run locally, which inevitably will start conversations about what to do when you can't run the entire system on one machine. This can become even more complicated if you are using cloud services that you cannot run locally.

Extreme solutions can involve "developing in the cloud," where developers move away from being able to develop locally anymore. I'm not a fan of this, because feedback cycles can suffer greatly. Instead, I think limiting the scope of which parts of a system a developer needs to work on is likely to be a much more straightforward approach. However, this might be problematic if you want to embrace more of a "collective ownership" model in which any developer is expected to work on any part of the system.

Technology Overload

The sheer weight of new technology that has sprung up to enable the adoption of microservice architectures can be overwhelming. I'll be honest and say that a lot of this technology has just been re-branded as "microservice friendly," but some advances have legitimately helped in dealing with the complexity of these sorts of architectures. There is a danger, though, that this wealth of new toys can lead to a form of technology fetishism. I've seen so many companies adopting microservice architecture also deciding that now is the best time to introduce vast arrays of new, and often alien, technology.

Microservices may well give you the *option* for each microservice to be written in a different programming language, have it run on a different runtime, or use a different database - but these are options, not requirements. You have to carefully balance the breadth and complexity of the technology you use against the costs that a diverse array of technology can bring.

When you start adopting microservices, some fundamental challenges are inescapable: you'll need to spend a lot of time understanding issues around data consistency, latency, service modelling, and the like. If you're trying to understand how these ideas change the way you think about software development at the same time that you're embracing a huge amount of new technology, you'll have a hard time of it. It's also worth pointing out that the bandwidth taken up by trying to understand all of this new technology will also reduce the time you have for actually shipping features to your users.

As you (gradually) increase the complexity of your microservice architecture, look to introduce new technology as you need it. You don't need a Kubernetes cluster when you have three services! In addition to ensuring that you're not overloaded with the complexity of these new tools, this gradual increase has the added benefit of allowing you to gain new, better ways of doing things that will no doubt emerge over time.

Reporting

With a monolithic system, you typically have a monolithic database. This means that stakeholders who want to analyze all of the data together, often involving large join operations across data, have a ready-made schema against which to run their reports. They can just run them directly against the monolithic database, perhaps against a read replica, as shown in Figure 1-13.

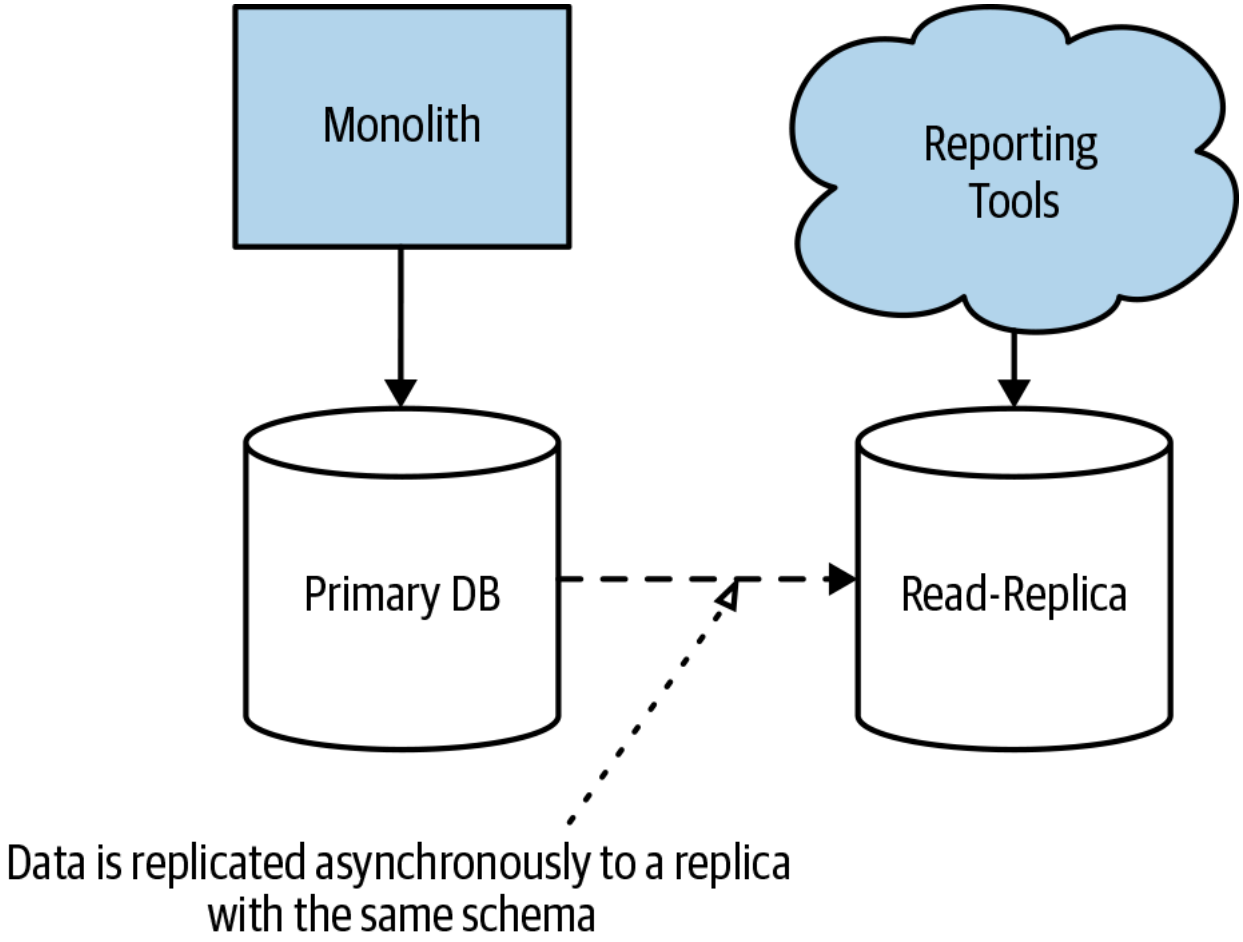


Figure 1-13. Reporting carried out directly on the database of a monolith

With a microservice architecture, we have broken up this monolithic schema. That doesn't mean that the need for reporting across all of our data has gone away; we've just made it much more difficult because now our data is scattered across multiple logically isolated schemas.

More modern approaches to reporting, such as using streaming to allow for real-time reporting on large volumes of data, can work well with a microservice architecture but typically require the adoption of new ideas and associated technology. Alternatively, you might simply need to publish data from your microservices into central reporting databases (or perhaps less structured data lakes) to allow for reporting use cases.

Monitoring and Troubleshooting

With a standard monolithic application, we can have a fairly simplistic approach to monitoring. We have a small number of machines to worry about, and the failure mode of the application is somewhat binary—the application is often either all up or all down. With a microservice architecture, do we understand the impact if just a single instance of a service goes down?

With a monolithic system, if our CPU is stuck at 100% for a long time, we know that's a big problem. With a microservice architecture with tens or hundreds of processes, can we say the same thing? Do we need to wake someone up at 3 a.m. when just one process is stuck at 100% CPU?

Luckily, there are a whole host of ideas in this space that can help. If you'd like to explore this concept in more detail, I recommend *Distributed Systems Observability* by Cindy Sridharan (O'Reilly) as an excellent starting point, although we'll also be taking our own look in [Link to Come].

Security

With a single-process monolithic system, much of our information flowed within that process. Now, more information flows over networks between our services. This can make our data more vulnerable to being observed in transit, but also potentially manipulated as part of man-in-the-middle attacks. This means that you might need to direct more care to protecting data in transit and to ensuring that your microservice endpoints are protected such that only authorized parties are able to make use of them. [Link to Come] is dedicated entirely to looking at the challenges in this space.

Testing

With any type of automated functional test, you have a delicate balancing act. The more functionality a test executes—the broader the scope of the test—the more confidence you have in your application. On the other hand, the larger the scope of the test, the harder it is to set up test data and supporting fixtures, the longer it can take to run, and the harder it can be to work out what is broken when it fails. In [Link to Come] I'll share a number of techniques for making testing work in this more challenging environment.

End-to-end tests for any type of system are at the extreme end of the scale in terms of functionality they cover, and we are used to them being more problematic to write and maintain than smaller-scoped unit tests. Often this is worth it, though, because we want the confidence that comes from having an end-to-end test use our systems in the same way a user might.

But with a microservice architecture, the scope of our end-to-end tests becomes *very* large. We would now need to run tests across multiple services, all of which need to be deployed and appropriately configured for the test scenarios. We also need to be prepared for the false negatives that occur when environmental issues, such as service instances dying or network time-outs of failed deployments, cause our tests to fail.

These forces mean that as your microservice architecture grows, you will get a diminishing return on investment when it comes to end-to-end testing. The testing will cost more but won't manage to give you the same level of confidence that it did in the past. This will drive you toward new forms of testing, such as contract-

driven testing, as well as exploring release remediation techniques and ideas like testing in production.

Latency

With a microservice architecture, processing that might previously have been done locally on one processor can now end up being split across multiple separate microservices. Information that previously flowed within only a single process now needs to be serialized, transmitted, and deserialized over networks that you might be exercising more than ever before. All of this can result in worsening latency of your system.

Although it can be difficult to measure the exact impact on latency of operations at the design or coding phase, this is another reason it's important to undertake any microservice migration in an incremental fashion. Make a small change and then measure the impact. This assumes that you have some way of measuring the end-to-end latency for the operations you care about—distributed tracing tools like Jaeger can help here. But you also need to have an understanding of what acceptable latency is for these operations too. Sometimes making an operation slower is perfectly acceptable, as long as it is still fast enough!

Data Consistency

Shifting from a monolithic system, in which data is stored and managed in a single database, to a much more distributed system, in which multiple processes manage state in different databases, causes potential challenges with respect to consistency of data. Whereas in the past you might have relied on database transactions to manage state changes, you'll need to understand that similar safety cannot easily be provided in a distributed system. The use of distributed transactions in most cases proves to be highly problematic in coordinating state changes.

Instead, you might need to start using concepts like *sagas* (something I'll detail at length in Chapter 3) and eventual consistency to manage and reason about state in your system. These ideas can require fundamental changes in the way you think about data in your systems, something that can be quite daunting when migrating existing systems. Yet again, this is another good reason to be cautious in how quickly you decompose your application. Adopting an incremental approach to decomposition so that you are able to assess the impact of changes to your architecture in production is really important.

Should I Use Microservices?

Despite the drive in some quarters to make microservice architectures the default approach for software, I feel that because of the numerous challenges I've outlined, adopting them still requires careful thought. You need to assess your own problem space, skills, and technology landscape and understand what you are trying to achieve before deciding whether microservices are right for you. They are *an* architectural approach, not *the* architectural approach. Your own context should play a huge part in deciding whether you want to go down that path.

That said, I do want to outline a few situations that would typically tip me away from—or toward—picking microservices.

Who They Might Not Work For

Given the importance of defining stable service boundaries, I feel that microservice architectures are often a bad choice for brand-new products or startups. In either case, the domain that you are working with is typically undergoing significant change as you iterate on the fundamentals of what you are trying to build. This shift in domain models will, in turn, result in more changes being made to service boundaries, and coordinating changes across service boundaries is an expensive undertaking. In general, I feel it more appropriate to wait until enough of the domain model has stabilized before looking to define service boundaries.

I do see a temptation for startups to go microservice first. The reasoning goes, "If we're really successful, we'll need to scale!" The problem is that you don't necessarily know if anyone is even going to want to use your new product. And even if you do become successful enough to require a highly scalable architecture, the thing you end up delivering to your users might be very different from what you started building in the first place. Uber initially focused on limos, and Flickr spun out of attempts to create a multiplayer online game. The process of finding product market fit means that you might end up with a very different product at the end than the one you thought you'd build when you started.

Startups also typically have fewer people available to build the system, which creates more challenges with respect to microservices. Microservices bring with them sources of new work and complexity, and this can tie up valuable bandwidth. The smaller the team, the more pronounced this cost will be. When working with smaller teams with just a handful of developers I'm always very hesitant in suggesting microservices for this reason.

The challenge of microservices for startups is compounded by the fact that normally your biggest constraint is people. For a small team, a microservice architecture can be difficult to justify because there is work required just to handle the deployment and management of the microservices themselves. Some people have described this as the "microservice tax." When that investment benefits lots of people, it's easier to justify. But if one person out of your five-person team is spending their time on these issues, that's a lot of valuable time not being spent building your product. It's much easier to move to microservices later, after you understand where the constraints are in your architecture and what your pain points are—then you can focus your energy on using microservices in the most sensible places.

Finally, I encounter a surprising number of organizations creating software that will be deployed and managed by their customers. As we've already covered, microservice architectures can push a lot of complexity into the deployment and operational domain. If you are running the software yourselves, you are able to offset this new complexity by adopting new technology, developing new skills, and changing working practices. This isn't something you can expect your customers to do. If they are used to receiving your software as a Windows installer, it's going to come as an awful shock to them when you send out the next version of your software and say, "Just put these 20 pods on your Kubernetes cluster!" In all likelihood, they will have no idea what a pod, Kubernetes, or a cluster even is.

Where They Work Well

Probably the single biggest reason that I see organizations adopt microservices is to allow for more developers to work on the same system without getting in each other's way. Get your architecture and organizational boundaries right, and you allow more people to work independently from one anothers, reducing delivery contention. A five-person startup is likely to find a microservice architecture a drag. A hundred-person scale-up that is growing rapidly is likely to find that its growth is much easier to accommodate with a microservice architecture properly aligned around its product development efforts.

Software as a Service (SaaS) applications are, in general, also a good fit for a microservice architecture. These products are typically expected to operate 24-7, which creates challenges when it comes to rolling out changes. The independent releasability of microservice architectures is a huge boon in this area. Further, the microservices can be scaled up or down, as required. This means that as you establish a sensible baseline for your system's load characteristics, you get more control over ensuring that you can scale your system in the most cost-effective way possible.

The technology-agnostic nature of microservices ensures that you can get the most out of cloud platforms. Public cloud vendors provide a wide array of services and deployment mechanisms for your code. You can much more easily match the requirements of specific services to the cloud services that will best help you implement them. For example, you might decide to deploy one service as a set of functions, another as a managed virtual machine (VM), and another on a managed Platform as a Service (PaaS) platform.

Although it's worth noting that adopting a wide range of technology can often be a problem, being able to try out new technology easily is a good way to rapidly identify new approaches that might yield benefits. The growing popularity of FaaS platforms is one such example. For the appropriate workloads, it can drastically reduce the amount of operational overhead, but at present, it's not a deployment mechanism that would be suitable in all cases.

Microservices also present clear benefits for organizations looking to provide services to their customers over a variety of new channels. A lot of digital transformation efforts seem to involve trying to unlock functionality hidden away in existing systems. The desire is to create new customer experiences that can support the needs of users via whatever interaction mechanism makes the most sense.

Above all, a microservice architecture is one that can give you a lot of flexibility as you continue to evolve your system. That flexibility has a cost, of course, but if you want to keep your options open regarding changes you might want to make in the future, it could be a price worth paying.

Summary

Microservice architectures can give you a huge degree of flexibility in choosing technology, handling robustness and scaling, organizing teams, and more. This flexibility is in part why many people are embracing microservice architectures. But microservices bring with them a significant degree of complexity, and you need to ensure that this complexity is warranted. For many, they have become a default system architecture, to be used in virtually all situations. On the contrary, I still think that they are an architectural choice whose use must be justified by the problems you are trying to solve; often, simpler approaches can deliver much more easily.

Nonetheless, many organizations, especially larger ones, have shown how effective microservices can be. When the core concepts of microservices are properly understood and implemented, they can help create empowering, productive architectures that can help systems become more than the sum of their parts.

I hope this chapter has served as a good introduction into these topics. Next, we're going to look at how we define microservice boundaries, exploring the topics of structured programming and domain-driven design along the way.

[1] This concept was first outlined by David Parnas in 1971, in "Information Distributions Aspects of Design Methodology," *Proceedings of IFIP Congress '71*.

[2] For an in-depth introduction to domain-driven design, see *Domain-Driven Design* by Eric Evans (Addison-Wesley Professional), or for a more condensed overview, *Domain-Driven Design Distilled* by Vaughn Vernon (Addison-Wesley Professional).

[3] For an overview of Shopify's thinking behind the use of a modular monolith rather than microservices, "Deconstructing the Monolith" by Kirsten Westeinde has some useful insights.

[4] Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on May 28, 1987.

[5] Microsoft Research has carried out studies in this space, and I recommend all of them, but as a starting point, I suggest "Don't Touch My Code! Examining the Effects of Ownership on Software Quality" by Christian Bird, et al.

Chapter 2. How to Model Microservices

Work In Progress

Please note that the text below is currently being reworked for the 2nd edition of the book, and is not in a complete state. This will be Chapter 2 of the final book.

If you have any feedback on the book, or suggestions for the 2nd edition, then please contact me on book-feedback@samnewman.io and/or complete a short survey here: *https://oreil.ly/Bldg_MicroServices_survey*.

> *My opponent's reasoning reminds me of the heathen, who, being asked on what the world stood, replied, "On a tortoise." But on what does the tortoise stand? "On another tortoise."*
>
> Joseph Barker (1854)

So you know what microservices are, and hopefully have a sense of their key benefits. You're probably eager now to go and start making them, right? But where to start? In this chapter, we'll be looking at some foundational concepts such as information hiding, coupling, and cohesion and understand how they'll shift our thinking about drawing boundaries around our microservices. We'll also be looking at different forms of decomposition you might use, as well as focusing more deeply on domain-driven design as a being a hugely useful technique in this space.

We'll look at how to think about the boundaries of your microservices so as to maximize the upsides and avoid some of the potential downsides. But first, we need something to work with.

Introducing MusicCorp

Books about ideas work better with examples. Where possible, I'll be sharing stories from real-world situations, but I've found it's also useful to have a fictional scenario with which to work. Throughout the book, we'll be returning to this scenario, seeing how the concept of microservices works within this world.

So let's turn our attention to the cutting-edge online retailer MusicCorp. MusicCorp was recently a brick-and-mortar retailer, but after the bottom dropped out of the gramophone record business it focused more and more of its efforts online. The company has a website, but feels that now is the time to double-down on the online world. After all, those smart phones for music are just a passing fad (Zunes are way better, obviously) and music fans are quite happy to wait for CDs to arrive at their doorsteps. Quality over convenience, right? And while they may have just learned that Spotify is in fact a digital music service rather than some sort of skin treatment for teenagers, MusicCorp are pretty happy with their own focus, and are sure all of this streaming business will blow over soon.

Despite being a little behind the curve, MusicCorp has grand ambitions. Luckily, it has decided that its best chance of taking over the world is by making sure it can make changes as easily as possible. Microservices for the win!

What Makes a Good Microservice Boundary?

Before the team from MusicCorp tears off into the distance, creating service after service in an attempt to deliver eight-track tapes to all and sundry, let's put the brakes on and talk a bit about the most important underlying idea we need to keep in mind. We want our microservices to be able to be changed, deployed, and their functionality released to our users in an independent fashion. The ability to change one microservice in isolation from another is vital. So what things do we need to bear in mind when we think about how we draw the boundaries around them?

In essence, microservices are just another form of modular decomposition, albeit one that has network-based interaction between the models and all the associated challenges that brings. Luckily, this means we can rely on a lot of prior art in the space of modular software and structured programming to help guide us in terms of working out how to define our boundaries. With that in mind, let's look more deeply at three key concepts which we touched on briefly in Chapter 1 and which are vital to grasp when it comes to working out what makes for a good microservice boundary - information hiding, cohesion, and coupling.

Information Hiding

We introduced information hiding in Chapter 1 - a concept developed by David Parnas to look at the most effective way to define module boundaries. Information hiding describes a desire to hide as many details as possible behind a module (or in our case microservice) boundary. Parnas looked at the benefits that modules should theoretically give us[1], namely:

Improved Development Time

> By allowing modules to be developed independently, we can allow for more work to be done in parallel, and reduce the impact of adding more developers to a project.

Comprehensability

> Each module can be looked at in isolation, and understood in isolation. This in turn makes it easier to understand what the system as a whole does.

Flexibility

> Modules can be changed independently from one another, allowing for changes to be made to the functionality of the system without requiring other modules to change. In addition, modules can be combined in different ways to deliver new functionality.

This list of desirable characteristics nicely complements what we are trying to achieve with microservice architectures - and indeed I now see microservices as just another form of modular architecture. Adrian Colyer has actually looked back at a number of David Parnas' papers from this period and examined them with respect to microservices, and his summaries are well worth reading[2].

The reality as Parnas explored through much of his work, is that having modules doesn't result in you actually achieving these outcomes. A lot depends on *how* the module boundaries are formed. From his own research information hiding was a key technique to help get the most out of our modular architectures, and with a modern eye, the same applies to microservices too.

From another of Parnas' papers[3], we have this gem:

> The connections between modules are the assumptions which the modules make about each other.
>
> David Parnas

By reducing the number of assumptions that one module (or microservice) makes about another, we directly impact the connections between them. By keeping the number of assumptions small, it is easier to ensure that we can change one module without impacting others. If a developer changing a module has a clear understanding as to how the module is used by others, it will be easier for them to make changes safely in such a way that upstream callers won't also have to change.

With microservices, this applies as well, except that we also have the opportunity to deploy that changed microservice without having to deploy anything else, arguably amplifying the three desireable characteristics that Parnas describes of improved development time, comprehensability and flexibility.

The implications of information hiding play out in so many ways, and I'll pick up this theme throughout the book.

Cohesion

One of the most succinct definitions I've heard for describing cohesion is this: "the code that changes together, stays together." For our purposes, this is a pretty good definition. As we've already discussed, we're optimizing our microservice architecture around ease of making changes in business functionality—so we want the functionality grouped in such a way that we can make changes in as few places as possible.

We want related behavior to sit together, and unrelated behavior to sit elsewhere. Why? Well, if we want to change behavior, we want to be able to change it in one place, and release that change as soon as possible. If we have to change that behavior in lots of different places, we'll have to release lots of different services (perhaps at the same time) to deliver that change. Making changes in lots of different places is slower, and deploying lots of services at once is risky—both of which we want to avoid.

So we want to find boundaries within our problem domain that help ensure that related behavior is in one place, and that communicate with other boundaries as loosely as possible. If the related functionality is spread across the system, we say that cohesion is weak - whereas for our microservice architectures we're aiming for strong cohesion.

Coupling

When services are loosely coupled, a change to one service should not require a change to another. The whole point of a microservice is being able to make a change to one service and deploy it, without needing to change any other part of the system. This is really quite important.

What sort of things cause tight coupling? A classic mistake is to pick an integration style that tightly binds one service to another, causing changes inside the service to require a change to consumers.

A loosely coupled service knows as little as it needs to about the services with which it collaborates. This also means we probably want to limit the number of different types of calls from one service to another, because beyond the potential performance problem, chatty communication can lead to tight coupling.

Coupling though comes in many forms, and I've seen a number of misunderstandings about the nature of coupling as it pertains to a service-based architecture. With that in mind, I think it's important that we explore this topic in more detail, something we'll do shortly.

The Interplay of Coupling And Cohesion

As we've already touched on, the concepts of coupling and cohesion are obviously related. Logically, if related functionality is spread across our system, changes to this functionality will ripple across those boundaries, implying tighter coupling. Constantine's Law[4], named for structured design pioneer Larry Constantine, sums this up neatly:

> A structure is stable if cohesion is strong and coupling is low.
>
> Constantine's Law, *Albert Endres and Dieter Rombach*

The concept here of stability is important to us. For our microservice boundaries to deliver on the promise of independent deployability, allowing us to work on microservices in parallel and reduce the amount of co-ordination between teams working on these services, we need some degree of stability in the boundaries themselves. If the contract that a microservice exposes is constantly changing in a backwards incompatible fashion, then this will cause upstream consumers to constantly have to change too.

Based on this thinking, if we can keep cohesion strong and coupling loose, then stability should follow. The one wrinkle here is that sometimes parts of your system may be going through so much change that stability might be impossible. We'll look at one such example later in this chapter when I share the experiences of the product development team behind SnapCI.

Types Of Coupling

It's possible that you could infer from the overview above that all coupling is bad. This isn't strictly true. Ultimately, some coupling in our system will be unavoidable. What we want to do is reduce how much coupling we have.

A lot of work has been done to look at the different forms of coupling in the context of structured programming, which was largely considering modular (non-distributed, monolithic) software. Many of these different models for assessing coupling overlap or clash, and in any case they speak primarily about things at the code level, rather than considering service-based interactions. As microservices are a style of modular architecture (albeit with the added complexity of distributed systems), we can use a lot of these original concepts and apply them in the context of our microservice-based systems.

Prior Art In Structured Programming

Much of our work in computing involves building on the work that came before. It is sometimes impossible to recognize everything that came before, but I have aimed with this second edition to highlight prior art where I can, partly to give credit where credit is due, partly as a way of ensuring that I lay down some breadcrumbs for those readers who want to explore certain topics in more detail, but also to show that many of these ideas are tried and tested.

When it comes to building on the work that came before, there are few areas in this book that have quite as much prior art as structured programming. We've already' mentioned Larry Constantine, and his book "Structured Design[5]" with Edward Yourdon is considered one of the most important texts in this area. Meilir Page-Jones's "Practical Guide to Structured Systems Design"[6] was also useful. Unfortunately, one thing all of these books have in common is how hard they can be to get hold of now, as they are out of print and aren't made available in ebook form. Yet another reason to support your local library!

Not all the ideas map cleanly, so I have done my best to synthesize a working model for the different types of coupling for microservices. Where these ideas map cleanly to previous definitions, I've stuck with those terms. In other places I have had to come up with new terms or blend in ideas from elsewhere. So please consider what follows to be built on top of a lot of prior art in this space, which I am attempting to give more meaning in the context of microservices.

In Figure 2-1 we see a brief overview of the different types of coupling, with them organized from low (desirable) coupling to high (undesirable).

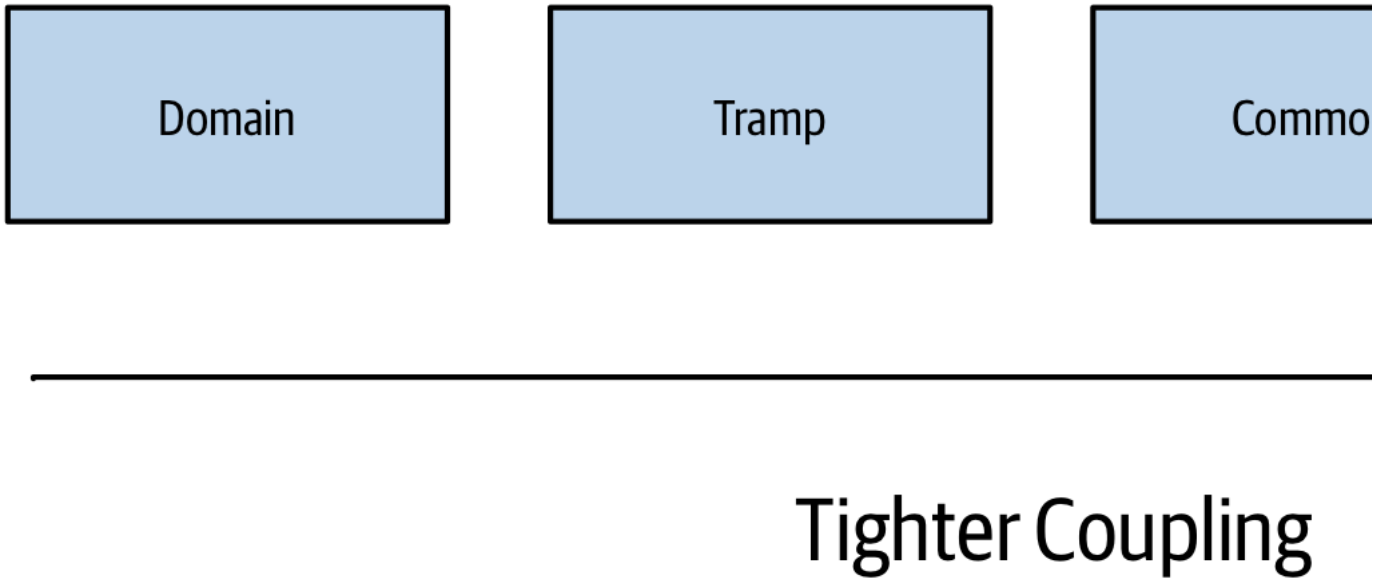


Figure 2-1. The different types of coupling, loose to tight coupling

Next, we'll look at each form of coupling in turn, and look at examples that show how they may manifest themselves in our microservice architecture.

Domain Coupling

Domain Coupling describes the situation where one microservice needs to interact with another microservice, because it needs to make use of the functionality that the other microservice provides[7].

In Figure 2-2, we see part of how orders for CDs are managed inside MusicCorp. In this example, `Order Processor` calls the `Warehouse` microservice to reserve stock, and the `Payment` microservice to take payment. The `Order Processor` is therefore dependent, and coupled, on the `Warehouse` and `Payment` microservices for this operation. We see no such coupling between `Warehouse` and `Payment` though, as they don't interact.

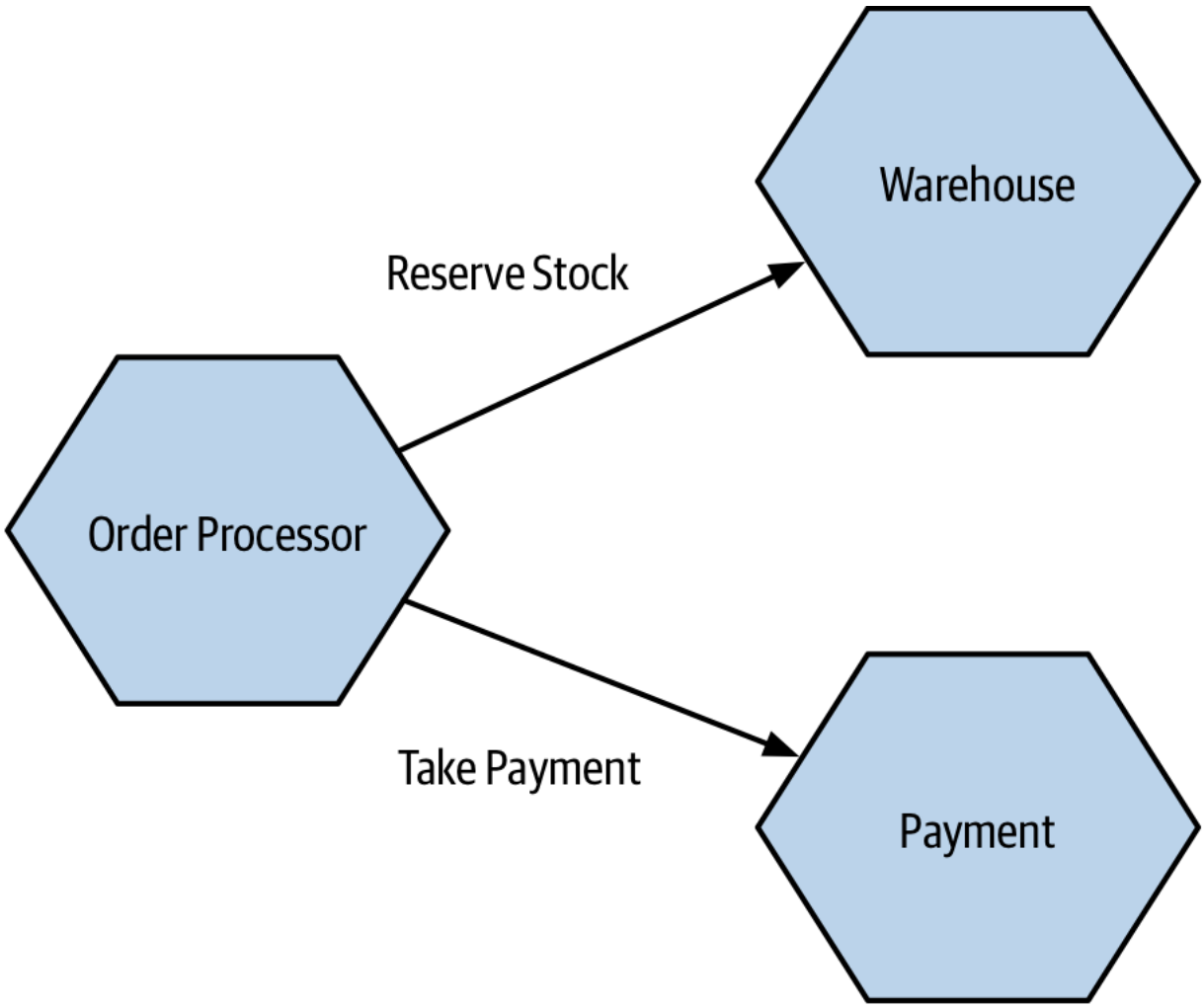


Figure 2-2. An example of Domain Coupling, where Order Processor needs to make use of the functionality provided by other microservices

In a microservice architecture, this type of interaction is largely unavoidable. A microservice-based system relies on multiple microservices collaborating in order for it to do its work. We still want to keep this to a minimum though - whenever you see a single microservice depending on multiple downstream services in this way it can be a cause for concern - it might imply a microservice that is doing too much.

As a general rule, domain coupling is considered to be a loose form of coupling, although even here we can hit problems. A microservice which needs to talk to lots of downstream microservices might point to a situation where too much logic has been centralized. Domain Coupling can also become problematic as more complex sets of data are sent between services - this can often point to the more problematic forms of coupling we'll explore shortly.

Just remember the importance of information hiding. Only share what you absolutely have to, and only send the absolute minimum amount of data that you need.

A Brief Note On Temporal Coupling

Another form of coupling you may have heard of is *temporal coupling*. Technically speaking, this type of coupling doesn't fit into the model we are exploring here, as it primarily speaks to runtime concerns.

In a situation where one microservice needs to call another microservice in a synchronous way, we say that these microservices are temporally coupled. They both need to be up and available and communicate with each other at the same time in order for the operation to complete. So in Figure 2-3, where MusicCorp's `Order Processor` is making a synchronous HTTP call to the `Warehouse` service, for the operation to complete `Warehouse` needs to be up and available at the same time the call is made.

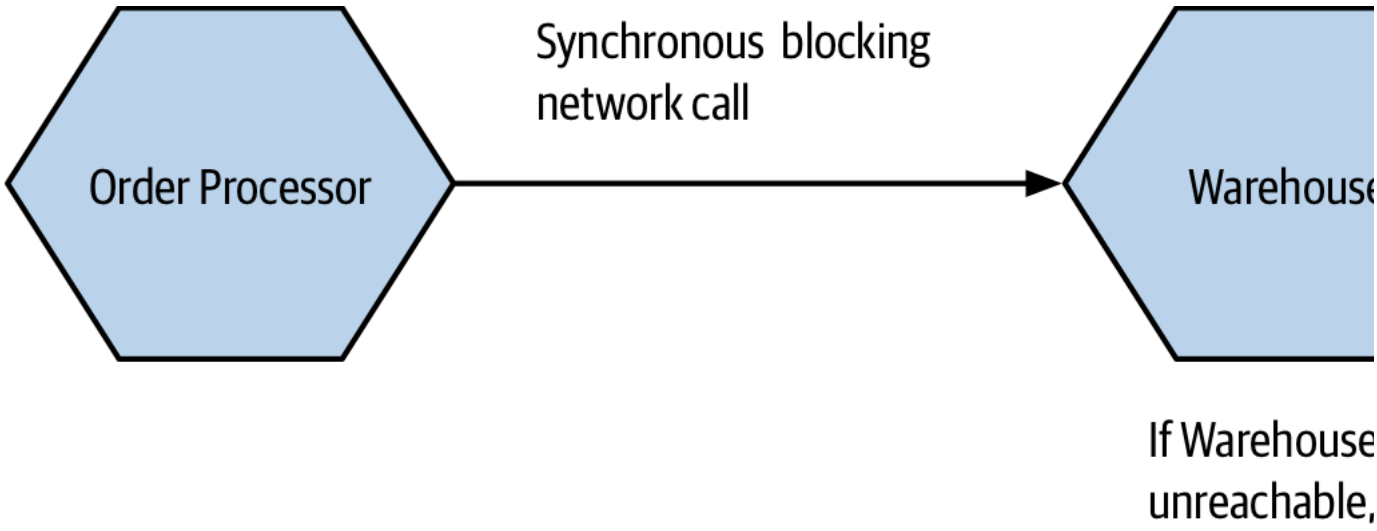


Figure 2-3. An example of Temporal Coupling, where Order Processor makes a synchronous HTTP call to the Warehouse microservice

If for some reason `Warehouse` isn't currently reachable by the `Order Processor`, then the operation fails, as we can't reserve the CDs to be sent out. `Order Processor` will also have to block and wait for a response from `Warehouse` as well, potentially causing issues in terms of resource contention.

Temporal coupling isn't always bad, it's just something to be aware of. As you have more microservices, and more complex interactions between them, the challenges of temporal coupling can increase to such a point that it becomes more difficult to scale your system and keep it working. One of the ways to avoid temporal coupling is to use some form of asynchronous communication, such as a message broker. We'll be coming back to the concept of temporal coupling and the associated issues in much more detail in Chapter 3

Pass Through Coupling

Pass through coupling[8] describes a situation where one microservice passes data to another microservice purely because it is needed by some other further downstream microservice. In many ways it's one of the most problematic forms of implementation coupling, as it implies that the caller knows not just that the microservice it is invoking calls yet another microservice, but also potentially that it needs to know how that one-step-removed microservice works.

As an example of pass through coupling, let's look deeper at part of how MusicCorp's order processing works, in Figure 2-4. Here, we have an `Order Processor`, which is sending a request to `Warehouse` to prepare an order for dispatch. As part of the request payload, we send along a `Shipping Manifest`. This `Shipping Manifest` consists not just of the address of the customer, but also the shipping type. The `Warehouse` just passes this manifest on to the downstream `Shipping` microservice.

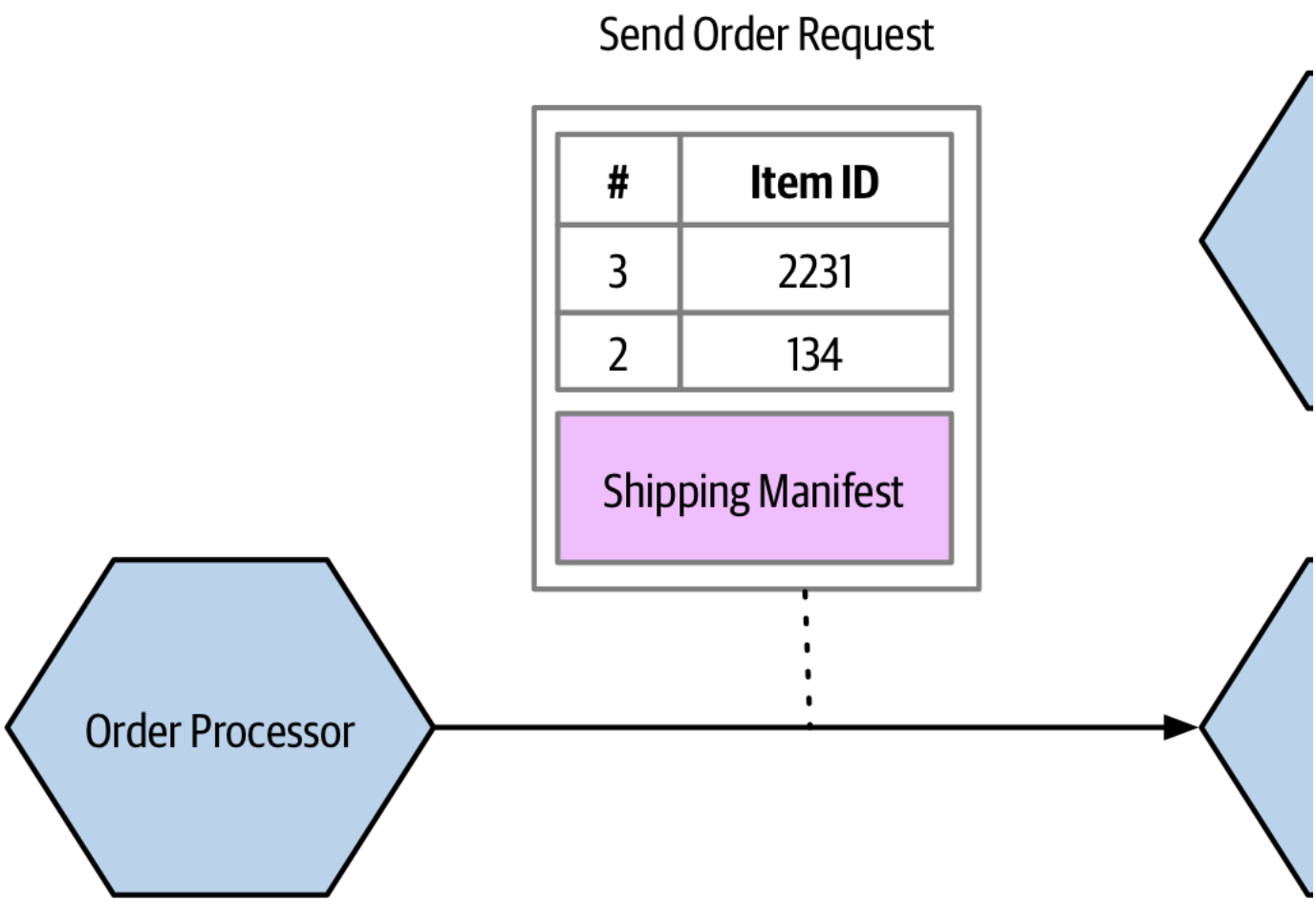


Figure 2-4. Pass through coupling, where data is passed to a microservice purely because another downstream service needs it

The major issue with pass through coupling is that a change to the required data downstream can cause a more significant upstream change. In our example, if the `Shipping` now needs the format or content of the data to be changed, then both `Warehouse` and `Order Processor` would likely need to change.

There are a few ways this can be fixed. The first is to consider if it makes sense for the calling microservice to just bypass the intermediary. In our example, this might mean `Order Processor` speaks directly to `Shipping`. Now, in this specific situation, this causes some other headaches. Our `Order Processor` is increasing its domain coupling, as `Shipping` is yet another microservice it needs to know about - if that was the only issue, this might still be fine, as domain coupling is a looser form of coupling of course. This solution gets more complex here though as stock has to be reserved with `Warehouse` before we dispatch the package using `Shipping`, and after the shipping has been done we need to update the stock accordingly. This pushes more complexity and logic into the `Order Processor` which was previously hidden inside `Warehouse`.

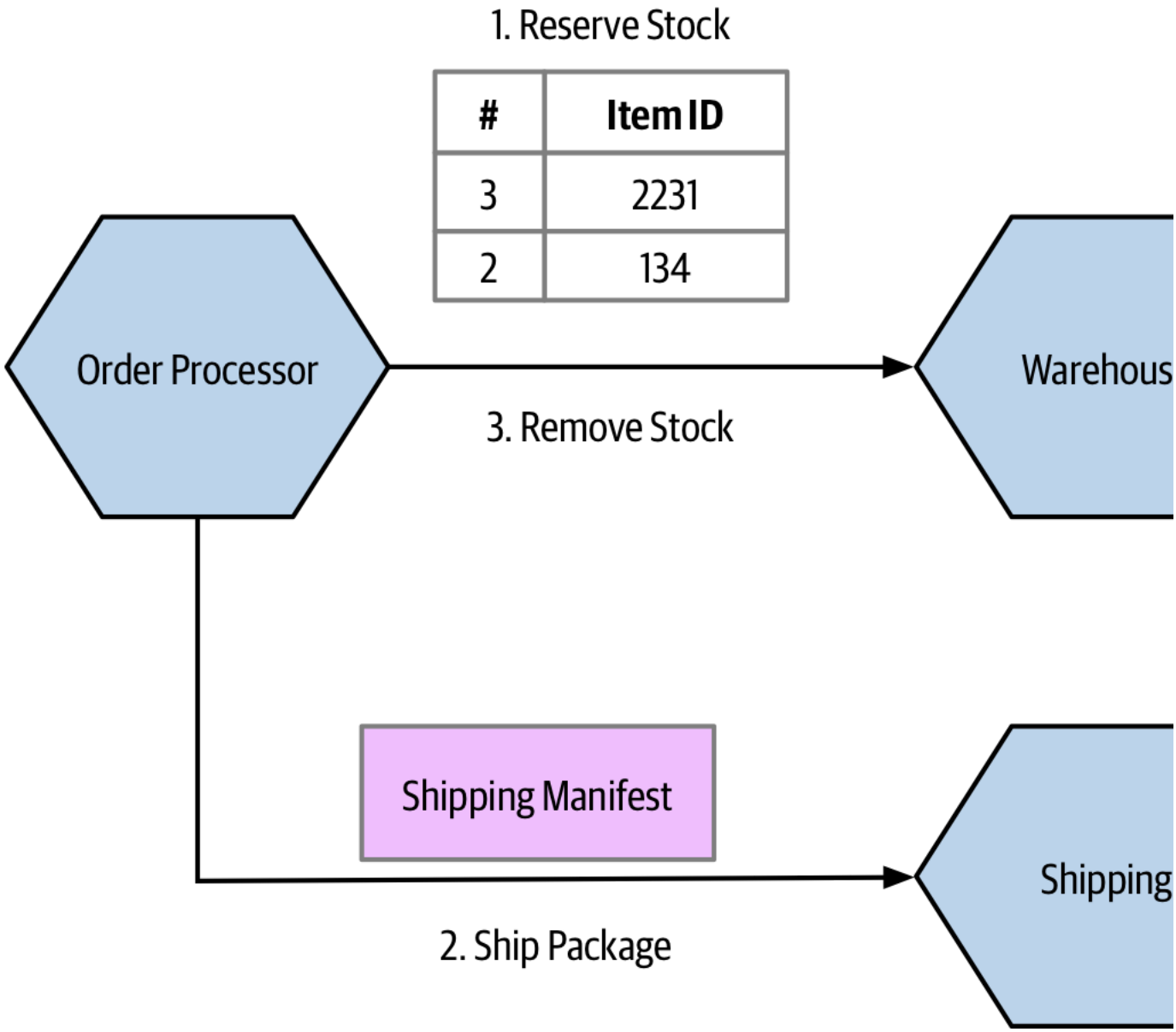


Figure 2-5. One way to work around pass through coupling involves communicating directly with the downstream service

For this specific example, I might consider a simpler (albeit more nuanced) change, namely to totally hide the requirement for a `Shipping Manifest` from `Order Processor`. The idea of delegating the work of both managing stock and arranging for dispatch of the package to our `Warehouse` service makes sense, but we don't like the fact that we have leaked some lower-level implementation, namely the fact that the `Shipping` microservice wants a `Shipping Manifest`. One way to hide this detail would be to have `Warehouse` take in the required information as part of its contract, and then have it construct the Shipping Manifest locally. Now, this means that if the `Shipping` service changes its service contract, as long as the required data is collected by the `Warehouse`, then this change will be invisible from the viewpoint of the `Order Processor`.

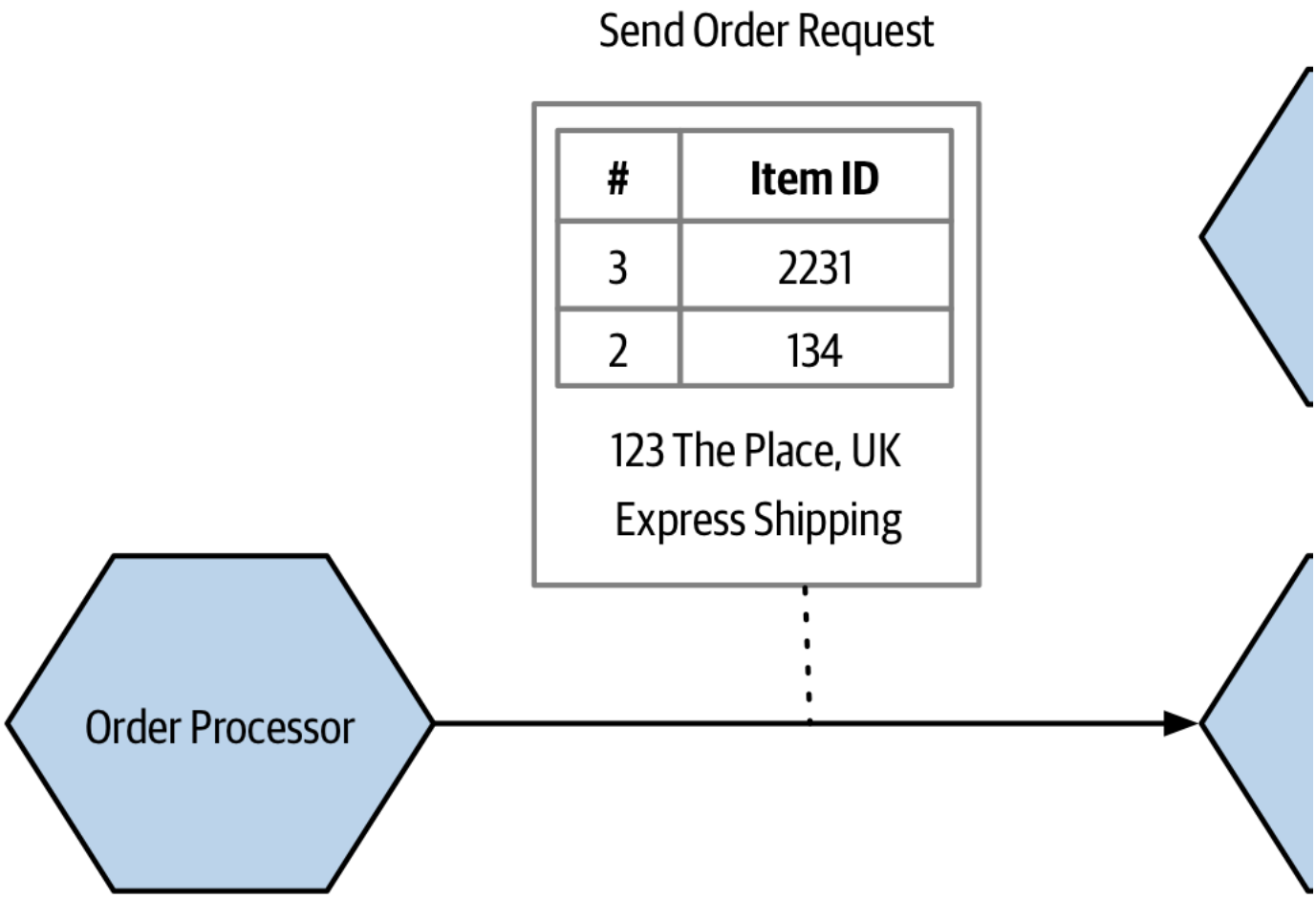


Figure 2-6. Hiding the need for a Shipping Manifest from the Order Processor

Whilst this will help protect the `Warehouse` microservice from some changes to `Shipping`, there are some things that would still require all parties to change. Let's consider the idea that we want to start shipping internationally. As part of this, the `Shipping` service needs a `Customs Declaration` as part of the `Shipping Manifest`. If this is an optional parameter, then we could deploy a new version of the `Shipping` microservice without issue. If this was a required parameter though, then the `Warehouse` would need to create one. It might be able to do this with existing information that it has (or is given), but if it required additional information this might require additional information to be passed to it by the `Order Processor`.

Although in this case we haven't eliminated the need for a change to be made across all three microservices, we have been given much more power about when and how these changes could be made. If we had the tight (pass through) coupling of the initial example, adding this new required `Customs Declaration` may require a lock-step rollout of all three microservices. At least by hiding this detail we could much more easily phase deployment.

One final approach which could help reduce the pass through coupling would be for the order processor to still send the shipping manifest to the `Shipping` microservice via the `Warehouse`, but to have the `Warehouse` be totally unaware of the structure of the `Shipping Manifest` itself. The `Order Processor` sends the manifest as part of the order request, but the `Warehouse` makes no attempt to look at or process the field - it just treats it like a blob of data and doesn't care about the contents. Instead it just sends it along. A change in the format of the the `Shipping Manifest` would still require a change to both the `Order Processor` and `Shipping` microservice, but as the `Warehouse` doesn't care about what is actually in the manifest itself it doesn't need to change.

Common Coupling

Common coupling occurs when two or more microservices make use of a common set of data. A simple and common example of this form of coupling would be multiple microservices making use of the same shared database, but this could also manifest itself in the use of shared memory or a shared filesystem.

The main issue with common coupling is that changes to the structure of the data can impact multiple microservices at once. Consider the example of some of MusicCorp's services in Figure 2-7. As we discussed earlier, MusicCorp operate around the world, so need various bits of information about the countries in which they operate. Here, multiple services are all reading static reference data from a shared database. If the schema of this database changed in a backwards-incompatible way, it would require changes to each of the consumers of the database. In practice, shared data like this tends to be very difficult to change as a result.

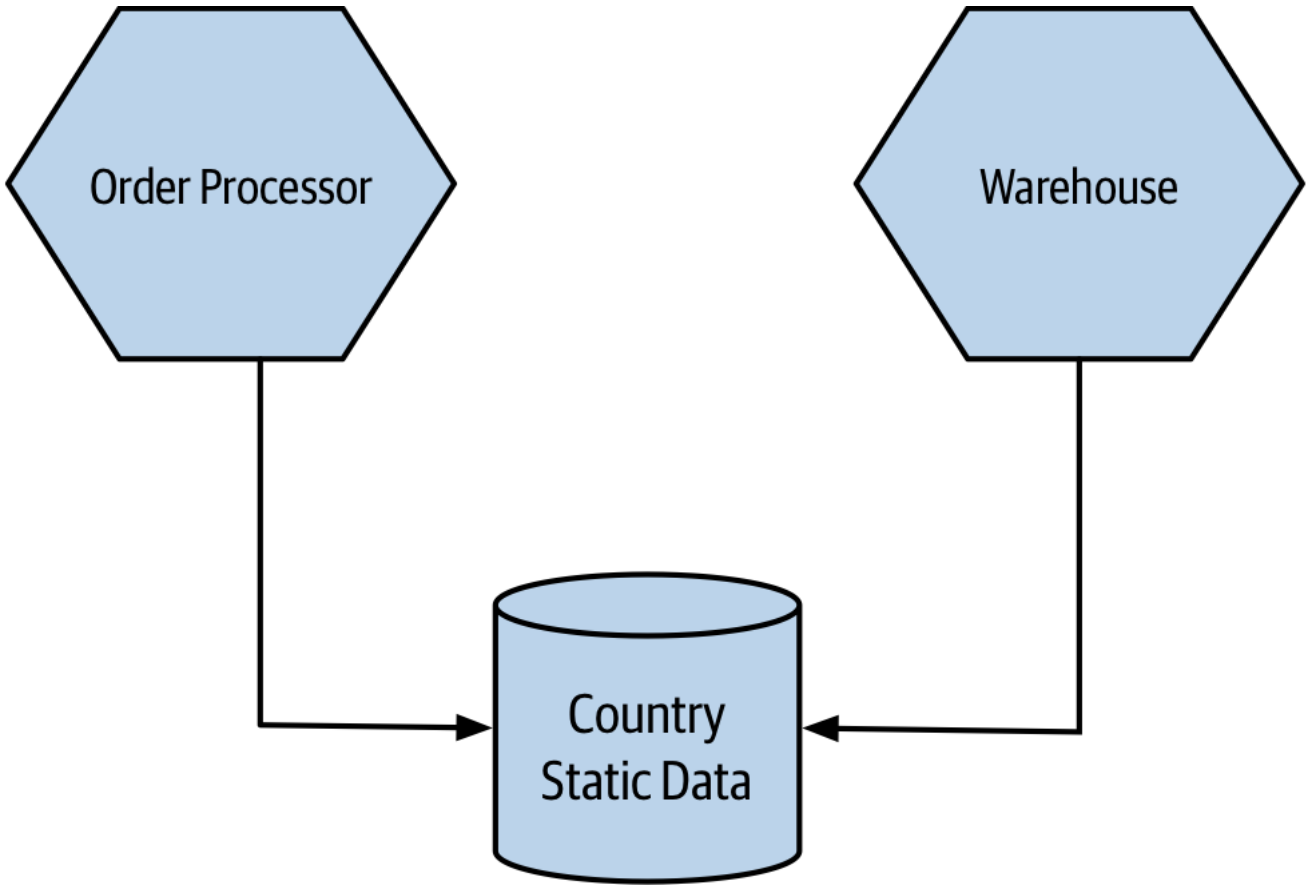


Figure 2-7. Multiple services accessing shared static reference data related to countries from the same database

The example in Figure 2-7 is, relatively speaking, fairly benign. This is because by its very nature static reference data doesn't tend to change often, and also because this data is read-only - as a result I tend to be relaxed about sharing static reference data in this way. Common coupling though becomes more problematic if the structure of the common data changes more frequently, or if multiple microservices are reading and writing to the same data.

Figure 2-8 shows us a situation where the `Order Processor` and `Warehouse` service are both reading and writing from a shared `Order` table, to help manage the process of dispatching CDs to MusicCorp's customers. Both microservices are updating the `STATUS` column. The `Order Processor` can set the `PLACED`, `PAID`, and `COMPLETED` statuses, whereas the `Warehouse` will apply `PICKING` or `SHIPPED` statuses.

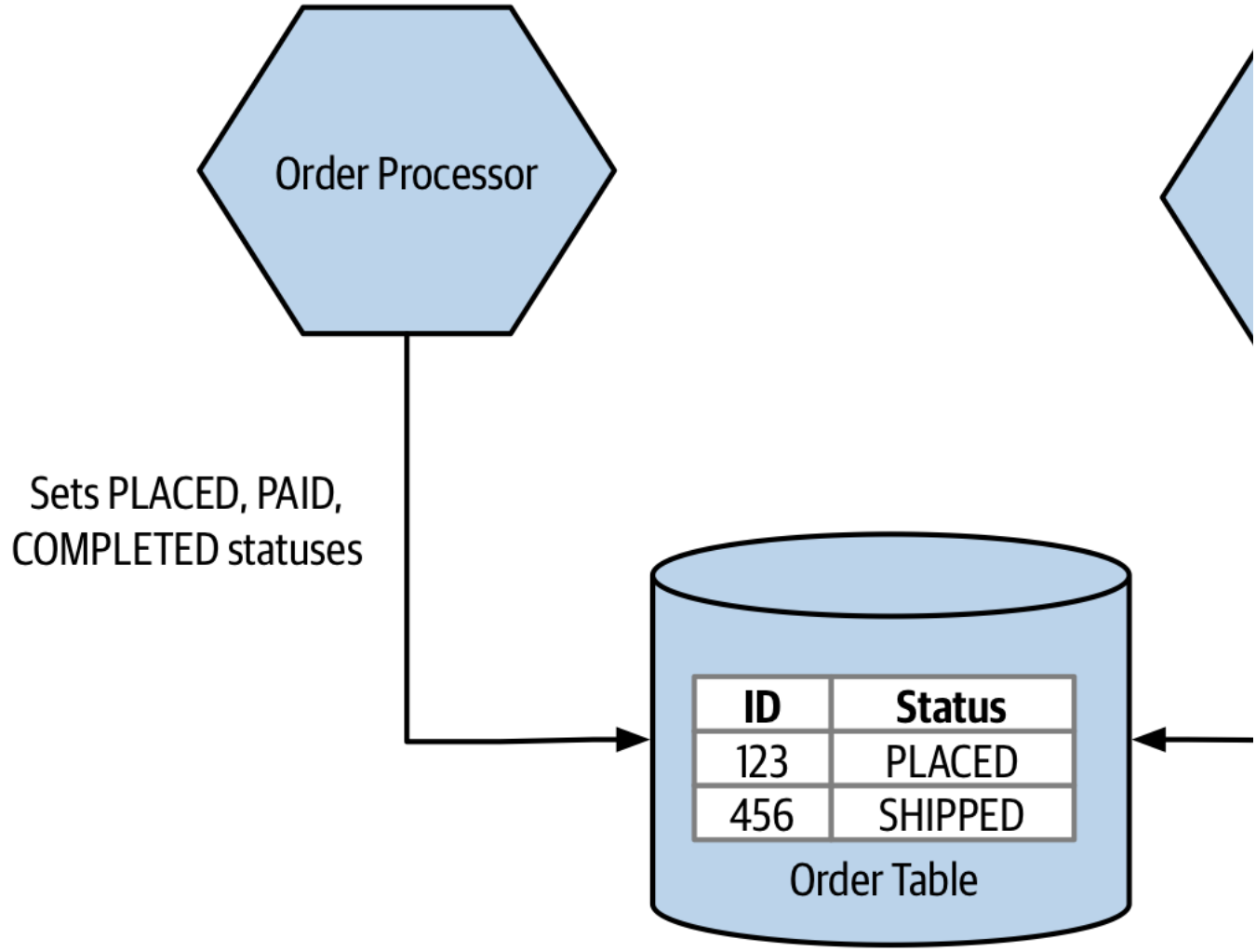


Figure 2-8. An example of common coupling where both Order Processor and Warehouse are updating the same order record

Although you might consider Figure 2-8 to be a somewhat contrived example, this nonetheless straightforward example of common coupling helps illustrate a core problem. Conceptually, we have both the `Order Processor` and the `Warehouse` microservices managing different aspects of the lifecycle of an order. When making changes in `Order Processor`, can I be sure that I am not changing the order data in such a way that it breaks `Warehouse`'s view of the world, or vice-versa?

One way to ensure that the state of something is changed in a correct fashion, would be to create a finite state machine. A state machine can be used to manage the transition of some entity from one state to another, ensuring invalid state transitions are prohibited. In Figure 2-9, see the allowed transitions of state for an order in MusicCorp. An order can go from `PLACED` to `PAID`, but not straight from `PLACED` to `PICKING` (this state machine likely wouldn't be sufficient for the real-world business processes involved in full end-to-end buying and shipping of goods, but I wanted to give a simple example to illustrate the idea).

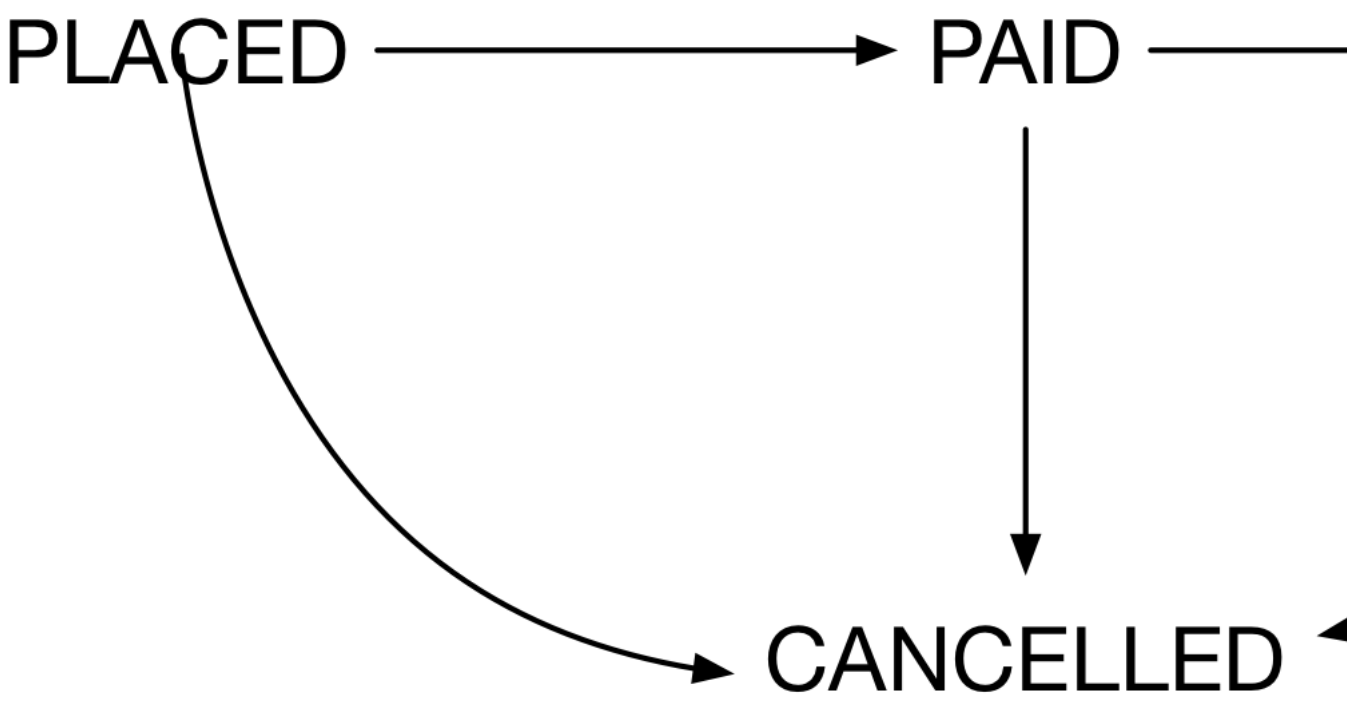


Figure 2-9. An overview of the allowable state transitions for an order in MusicCorp

The problem in this specific example is that both `Warehouse and +Order Processor` share responsibilities for managing this state machine. How do we ensure that they are both in agreement as to what transitions are allowed? There are ways to manage processes like this across microservice boundaries, and we will return to this topic when we discuss Sagas in [Link to Come].

A potential solution here would be to ensure that one single microservice manages the order state. In Figure 2-10, either `Warehouse` or `Order Processor` can send status update requests to the `Order` service. Here, the `Order` microservice is the source of truth for any given order. In this situation, it is really important that we see the requests from `Warehouse` and `Order Processor` as just that - *requests*. In this scenario, it is the job of the `Order` service to manage the acceptable state transitions associated with an order aggregate. As such, if it received a request from `Order Processor` to move a status from `PLACED` straight to `COMPLETED` it is free to reject that request if that is an invalid change.

Tip

Make sure you see a request that is sent to a microservice as something that the downstream microservice can reject if it is invalid.

An alternative approach I see in such cases is to implement the `Order` service as little more than a wrapper around database CRUD operations, where requests just map directly to database updates. This is akin to an object having private fields but public getters and setters - the behavior has leaked from the microservice to upstream consumers (reducing cohesion), and we're back in the world of managing acceptable state transitions across multiple different services.

Warning

If you see a microservice that just looks like a thin wrapper around database CRUD operations, that is a sign that you may have weak cohesion and tighter coupling, as logic that should be in that service to manage the data is instead spread elsewhere in your system.

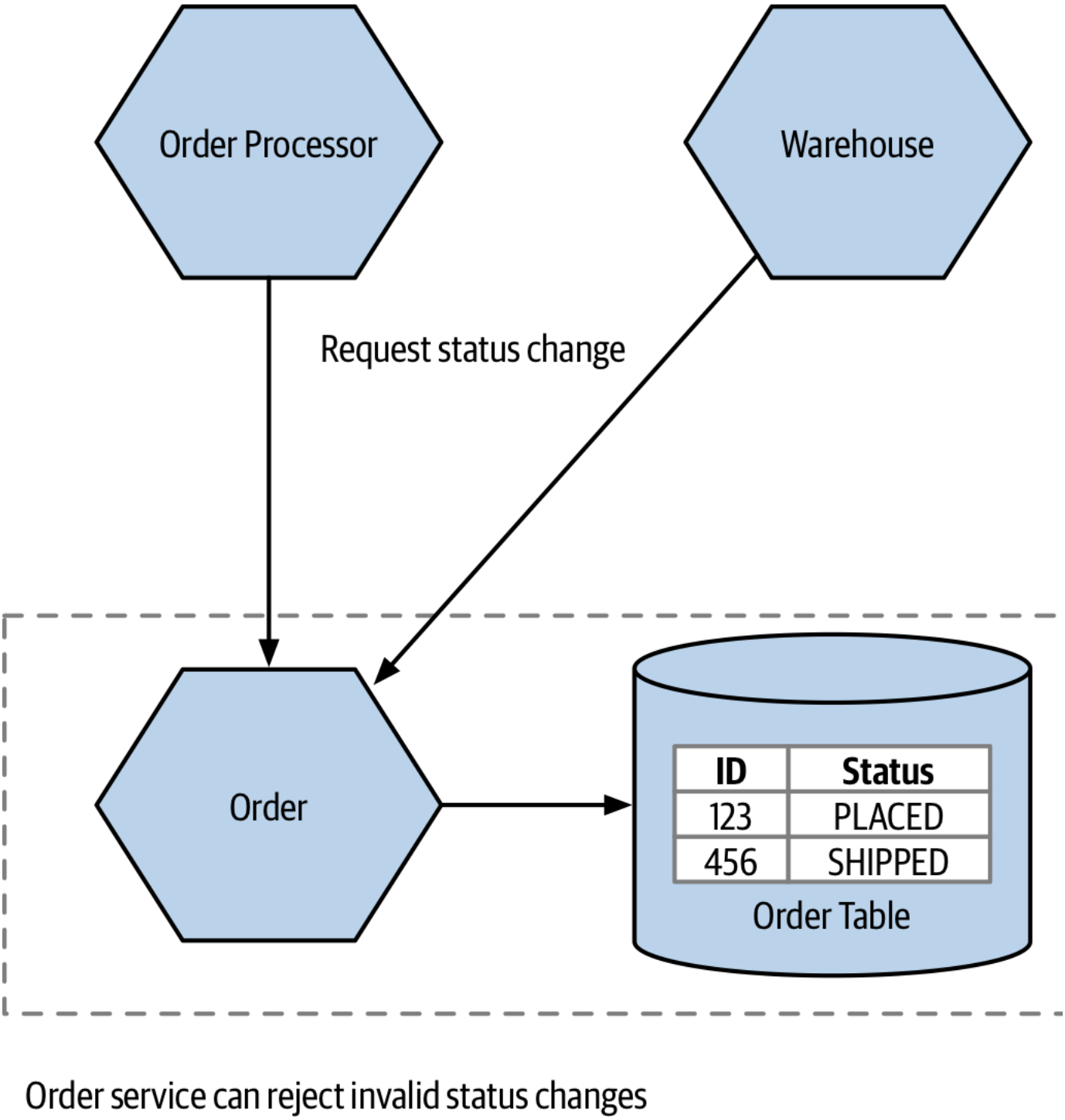


Figure 2-10. Both Order Processor and Warehouse can request changes are made to an order, but the Order microservice decides what requests are acceptable

Sources of common coupling are also potential sources of resource contention. Multiple microservices making use of the same file system or database could overload that shared resource, potentially causing significant problems if the shared resource becomes slow or even entirely unavailable. Shared databases are especially prone to this problem, as multiple consumers can run arbitrary queries against the database itself, which in turn can have wildly different performance characteristics. I've seen more than one database brought to its knees by an expensive SQL query - I may have even been the culprit once or twice[9].

So common coupling is **sometimes** ok, but often not. Even when it's benign, it means that we are limited in what changes can be made to the shared data, but it often speaks to a lack of cohesion in our code. It can also cause us problems in terms of operational contention too. It's for those reasons that we consider common coupling to be one of the least desirable forms of coupling, but it can get worse.

Content Coupling

Content coupling describes a situation where an upstream service reaches into the internals of a downstream service and changes its internal state. The most common manifestation of this is an external service directly accessing another microservice's database and changing it directly. The difference between content coupling and common coupling are subtle. On the face of it, in both cases two or more microservices are reading and writing to the same set of data. With common coupling, you understand that you are making use of a shared, external dependency. You know it's not under your control. With content coupling, the lines of ownership become less clear, and it becomes more difficult for developers to change a system.

Let's revisit our earlier example from MusicCorp. In Figure 2-11, we have an `Order` service which is supposed to manage the allowable state changes to orders in our system. The `Order Processor` is sending requests to the `Order` service, delegating not just the exact change in state that will be made, but also delegating to the `Order` service responsibility for deciding what state transitions are allowable. On the other hand, the `Warehouse` service is directly updating the table where order data is stored, bypassing any functionality in the `Order` service which might check for allowable changes. We have to hope that the `Warehouse` service has a consistent set of logic to ensure that only valid changes are made. Best case, this represents a duplication of logic. Worst case, the checking around allowable changes in `Warehouse` is different to that in the `Order` service, and as a result we could end up with orders in very odd, confusing states.

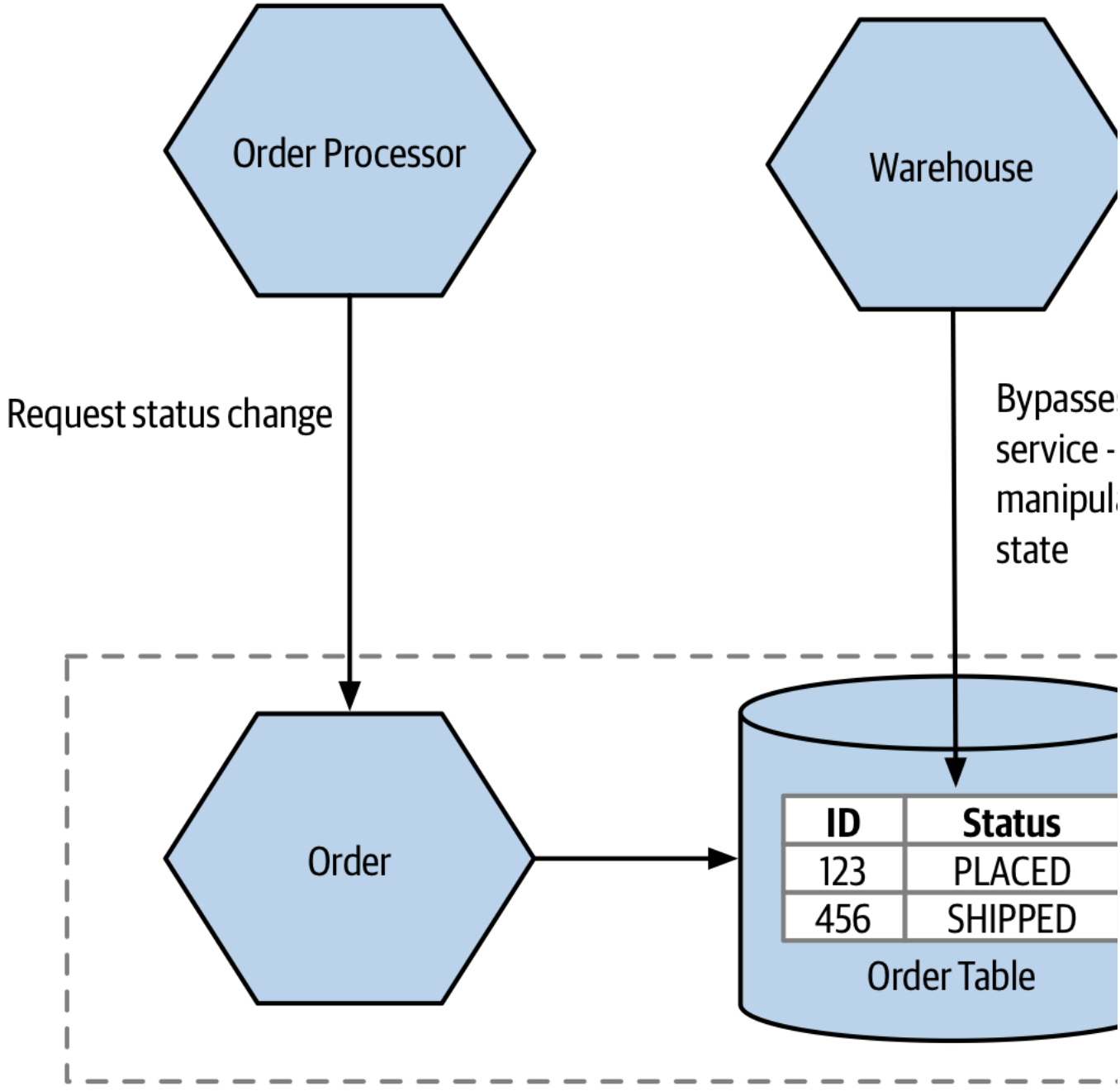


Figure 2-11. An example of content coupling, where the Warehouse is directly accessing the internal data of the Order service

In this situation, we also have the issue that the internal data structure of our order table is exposed to an outside party. When changing the `Order` service, we now have to be extremely careful about making changes to that particular table - that's even assuming it's obvious to us that this table is being directly accessed by an

outside party. The easy fix here is to have the `Warehouse` send requests to the `Order` service itself, where we can vet the request, but also hide the internal detail making subsequent changes to the `Order` service much easier.

If you are working on a microservice, it's vital that you have a clear separation between what can be changed freely, and what cannot. To be explicit, as a developer you need to know when you are changing functionality that is part of the contract your service exposes to the outside world. You need to ensure that if you make changes here that you will not break upstream consumers. Functionality that doesn't impact the contract your microservice exposes can be changed without concern.

It's certainly the case that all the problems that occur with common coupling also apply with content coupling, but content coupling has some additional headaches which make it so problematic - problematic enough that some people refer to this form of coupling as *pathological coupling*.

When you allow an outside party to directly access *your* database, your database in effect becomes part of that external contract, albeit one you cannot easily reason about what can, or cannot, be changed. You've lost the ability to define what is shared (and therefore cannot be changed easily), and what is hidden. Information hiding has gone out of the window.

In short, avoid content coupling.

Alternatives to Domain-Oriented Decomposition

So far, we've looked at the interplay of cohesion and coupling as they apply to microservices that arrived pre-formed. And as I introduced in Chapter 1, the primary mechanism we use for finding microservice boundaries is around the domain itself. Domain-oriented boundaries give us benefits in terms of making it easier to align to organizational structures, make it easier to combine functionality in different ways to deliver different experiences to users, and experience has shown that the boundaries also tend to be more stable than other forms of decomposition.

While I think that using the domain as the primary mechanism for identifying boundaries for microservices makes the most sense in general, there are other approaches that can be useful on occasion, either as an alternative to domain-oriented decompositon, or else as an additional tool.

Volatility

I've increasingly heard of a push back against domain-oriented decomposition, often by advocates promoting instead that volatility should be the primary driver for decomposition. Volatility based decomposition has you identify the parts of your system going through more frequent change, and extract that functionality into their own services where they can be more effectively worked on. Conceptually, I don't have a problem with this, but promoting it as the only way to do things isn't helpful, especially when we consider the different drivers we might have that are pushing us towards microservices. If my biggest issue is related to the need to scale my application, a volatility-based decomposition is unlikely to deliver much of a benefit for example.

The mindset behind volatility-based decomposition is also evident in approaches like Bimodal IT. A concept put forward by Gartner, Bimodal IT neatly breaks the world down into the snappily named "Mode 1" (aka Systems Of Record) and "Mode 2" (aka Systems Of Innovation) categories based on how fast (or slow) different systems need to go. Mode 1 systems, otherwise known as we are told, don't change much, don't need much business involvement. Mode 2 is where the action is, with systems needing to change fast and needing a high-touch from the business. Putting aside for one moment the drastic oversimplification inherent in such a categorization scheme, it also implies a very fixed view of the world, and belies the sorts of transformations that are evident across industry as companies look to "go digital". Parts of their system that didn't need to change much in the past suddenly do, in order to open up new market opportunities and provide services to their customers in ways that they previously didn't imagine.

Let's come back to MusicCorp. Their first foray into what we now call digital was just having a webpage. All it offered back in the mid-90s was a listing of what was for sale, but you just had to phone up to place the order. It was little more than an advert in a newspaper. Then, online ordering was a thing - now the entire warehouse which had up until that point been just handled with paper had to be digitized. Who knows, perhaps MusicCorp will at some stage have to consider making music available digitally? Although you might consider that MusicCorp are behind the times, you can still appreciate the amount of upheaval that companies have been going through as they understand how changing technology and customer behavior can require significant changes in parts of a business that couldn't be easily forseen.

I also dislike bimodal IT as a concept, as it becomes a way for people to dump stuff that is hard to change into a nice neat box and say "we don't need to deal with the issues in there - it's Mode 1". It's yet another model that a company can adopt to ensure that nothing actually has to change. It also avoids the fact that quite often changes in functionality also require changes in "Systems of record" (Mode 1) to allow for changes in "Systems of Innovation" (Mode 2). In my experience, organizations adopting bimodal IT do end up having two speeds - slow and slower.

To be fair to proponents of volatility-based decomposition, many of them aren't necessarily recommending such simplistic models as bimodal IT. In fact I find this technique to be highly useful to help determine boundaries if the main driver is about fast time to market - extracting functionality that is changing (or needs to change) frequently makes perfect sense in such a situation. But again, the goal determines the most appropriate mechanism.

Data

The nature of the data you hold and manage can drive you towards different forms of decomposition. For example you might want to limit what services handle personally identifiable information (PII), to reduce your risk of data breaches, but to also simplify oversight and implementation of things like GDPR.

For one of my recent clients, a payment company we'll call PaymentCo, the use of certain types of data directly influenced the decisions we made about system decomposition. PaymentCo handle credit card data, and as a result it means that their system needed to comply to various requirements set down by Payment Card Industry (PCI) about how this data needs to be managed. As part of this, their system and processes needed to be audited. They had a need to handle the full credit card data, and at a volume that meant their system had to comply with PCI Level 1, which is the most stringent level, and requires quarterly external assessment of the systems and practices related to how this data is managed.

Many of the PCI requirements are common sense, but the requirement to ensure that the whole system complied with these requirements, not least the need for the system to be audited by an external party, was proving to be quite onerous. As a result, they wanted to split out the part of the system which handled the full credit card data - meaning that only a subset of the system required this additional level of oversight. In Figure 2-12 we see a simplified form of the design we came up with. Services operating in the green zone (shown here enclosed by a dotted lone) never see full credit card information - this is limited to processes (and networks) in red zone (surrounded by dashes). The gateway diverted calls to the appropriate services (and the appropriate zone) - as the credit card information passed through this gateway, it was in effect also in the Red zone.

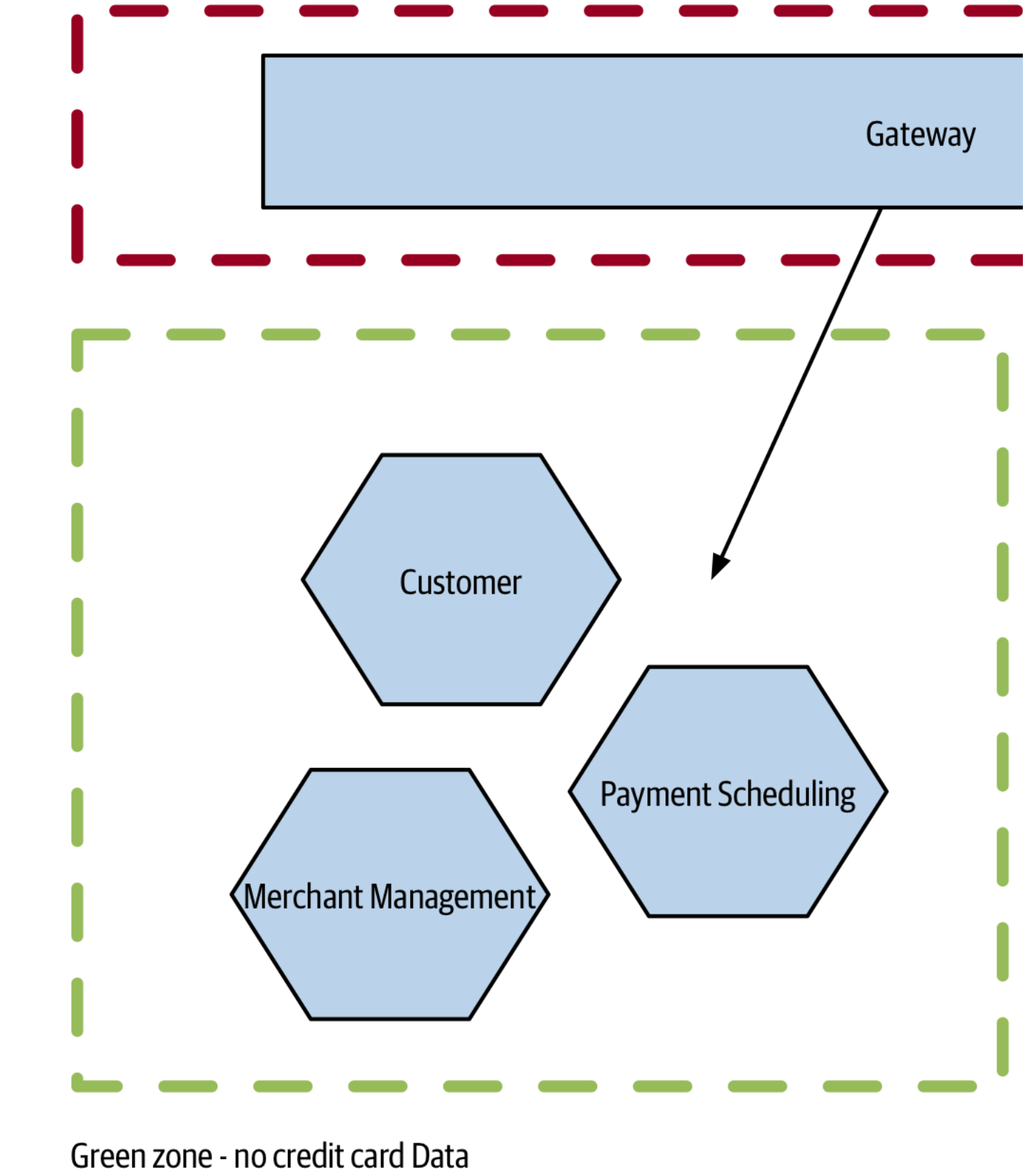


Figure 2-12. PaymentCo, who segregate processes based on their use of credit card information in order to limit the scope ot PCI

As credit card information never flowed into the green zone, all services in this area could be exempted from a full PCI audit. Services in the red zone were in scope for such oversight. When working through the design we did everything we could to limit what had to be in this red zone. It's key to note that we had to make sure that the credit card information never flowed to the green zone at all - if a microservice in the green zone could request this information, or that information was sent by a microservice in the red zone back to the green zone, then the clear lines of separation would break down.

Segregation of data is often driven by a variety of privacy and security concerns - we'll come back to this topic and the example of PaymentCo later on in [Link to Come].

Technology

The need to make use of different technology can also be a factor in terms of finding a boundary. You can accommodate different databases in a single running microservice, but if you wanted to mix different runtime models you may face a challenge. If you identify that part of your functionality needs to be implemented in a runtime like rust which enables you to eke out additional performance improvements, this ends up being a major forcing factor.

Of course we have to be aware of where this can drive us if adopted as a general means of decomposition. The classic three tiered architecture that we discussed in the opening chapter, and show again in Figure 2-13, is an example where related technology is grouped together. As we've already explored, this is often a less than ideal architecture.

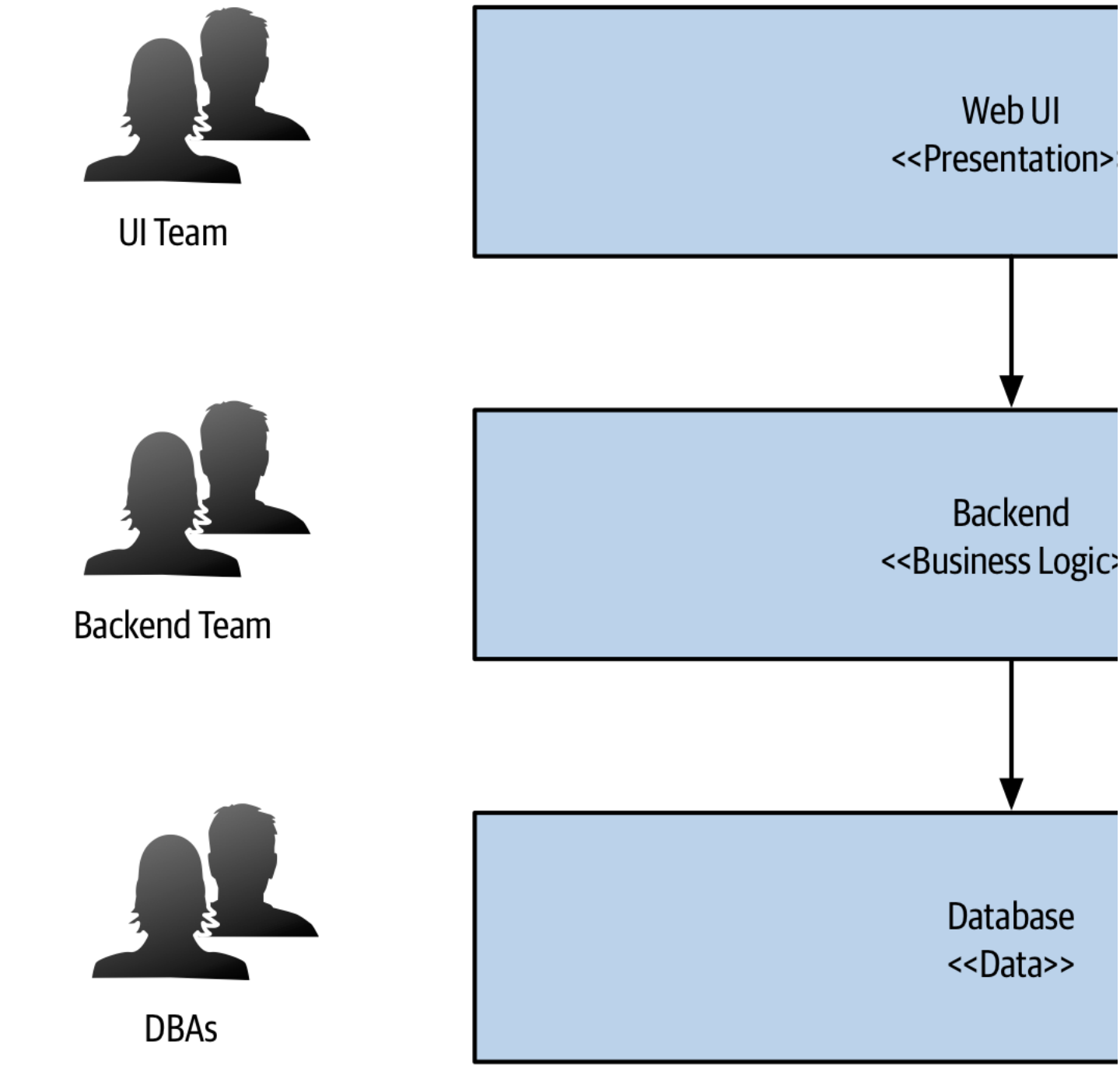


Figure 2-13. A traditional three-tiered architecture is often driven by technological boundaries

Organizational

As we established when I introduced Conway's law back in Chapter 1, there is an inherent interplay between organizational structure and the system architecture you end up with. Quite aside from the studies that have shown this link, in my own anecdotal experience I have seen this play out time and time again. How you organize yourselves ends up driving your systems architecture, for good or ill. When it comes to helping us define our service boundaries, we have to consider this as a key part of our decision making.

Defining a service boundary whose ownership would cut across multiple different teams is unlikely to yield the outcomes we would desire - as we'll explore further in [Link to Come], shared ownership of microservices is a fraught affair. It therefore follows that we must take account of the existing organizational structure when considering where and when to define boundaries, and in some situations perhaps even consider changing the organizational structure to support the architecture we want.

Even when we do work within an existing organizational structure, there is a danger that we will not get our boundaries in the right place. Many years ago, a few colleagues and I were working with a client in California, helping the company adopt some cleaner code practices and move more toward automated testing. We'd started with some of the low-hanging fruit, such as service decomposition, when we noticed something much more worrying. I can't go into too much detail as to what the application did, but it was a public-facing application with a large, global customer base.

The team, and system, had grown. Originally one person's vision, the system had taken on more and more features, and more and more users. Eventually, the organization decided to increase the capacity of the team by having a new group of developers based in Brazil take on some of the work. The system got split up, with the front half of the application being essentially stateless, implementing the public-facing website, as shown in Figure 2-14. The back half of the system was simply a remote procedure call (RPC) interface over a data store. Essentially, imagine you'd taken a repository layer in your codebase and made this a separate service.

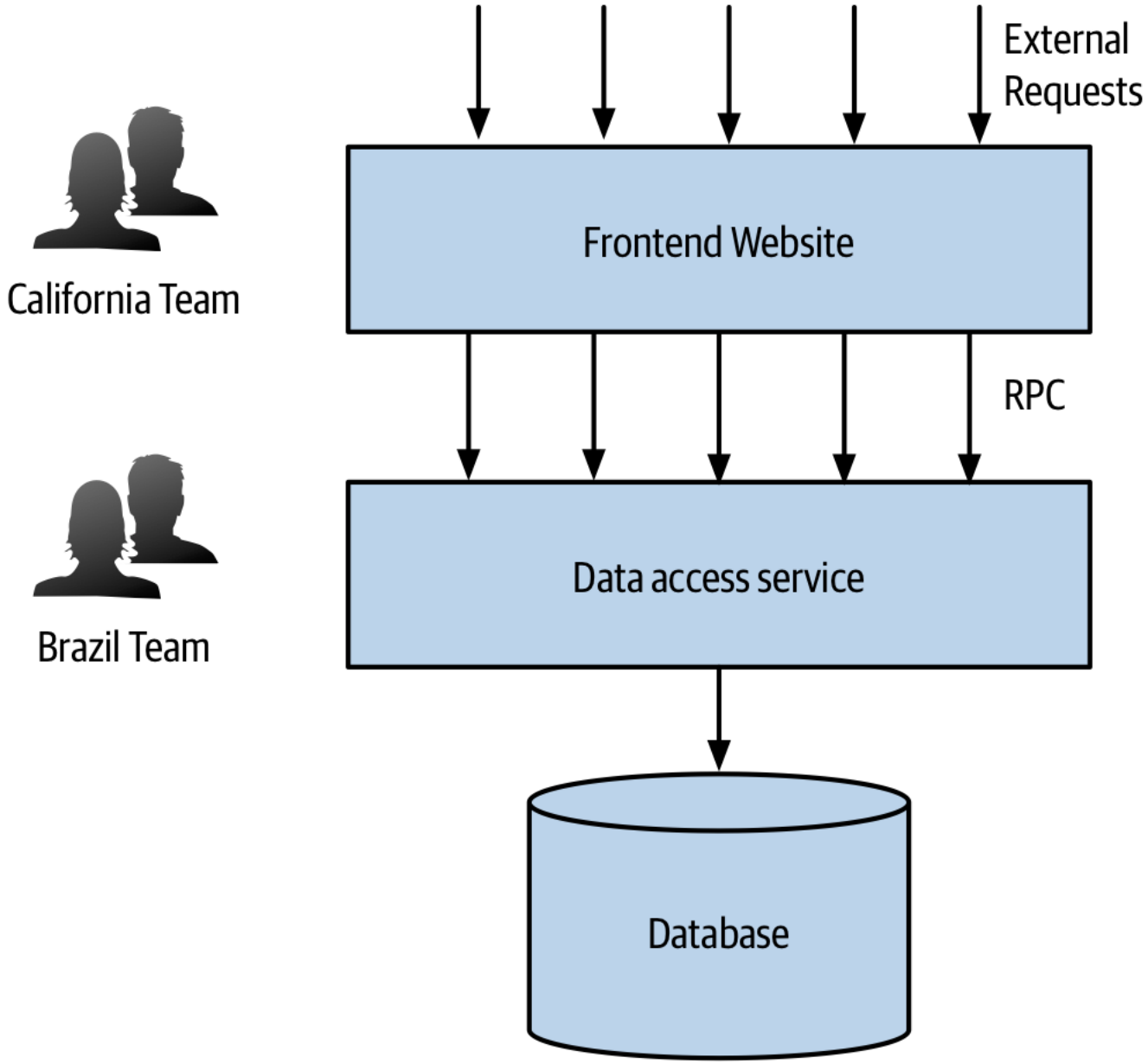


Figure 2-14. A service boundary split across a technical seam

Changes frequently had to be made to both services. Both services spoke in terms of low-level, RPC-style method calls, which were overly brittle (we'll discuss this further in Chapter 3). The service interface was also very chatty too, resulting in performance issues. This resulted in the need for elaborate RPC-batching mechanisms. I called this *onion architecture*, as it had lots of layers and made me cry when we had to cut through it.

Now on the face of it, the idea of splitting the previously monolithic system along geographical/organizational lines makes perfect sense, as we'll expand on in [Link to Come]. Here, however, rather than taking a vertical, business-focused slice through the stack, the team picked what was previously an in-process API and made a horizontal slice. A better model would have been for the team in California to have one end-to-end vertical slice, consisting of the related parts of the front end and data access functionality, with the team in Brazil taking another slice.

Layering Inside Vs Layering Outside

As you can hopefully see by now, I'm not a fan of horizontally layered architecture. Layering though can have it's place. Within a microservice boundary, it can be totally sensible to delineate between different layers in order to make the code easier to manage. The problem occurs when this layering becomes the mechanism by which your microservice and ownership boundaries are drawn.

Different Goals, Different Drivers

Microservices are not the goal. You don't "win" by having microservices. Adopting a microservice architecture should be a conscious decision, one based on rational decision-making. You should be thinking of adopting a microservice architecture in order to achieve something that you can't currently achieve with your existing system architecture.

Without having a handle on what you are trying to achieve, how are you going to inform your decision-making process about what options you should take? What you are trying to achieve by adopting microservices will greatly change where you focus your time, and how aggressive you are in creating microservices. It will also help guide you in which approach makes the most sense for finding microservice boundaries. PaymentCo for example are guided by their overriding concern about data.

In other words, while I can share my own views on how we should define a microservice boundary, and the fact that I think modelling them around a business domain is a sensible starting point, other factors may well come into play in determining what sort of decomposition you want to pick.

Mixing Models And Exceptions

As I hope is clear so far, I am not dogmatic in terms of how you find these boundaries. If you follow the guidelines of information hiding and appreciate the interplay of coupling and cohesion, then chances are that you'll avoid some of the worst pitfalls of whatever mechanism you pick. I happen to think that by focusing on these ideas that you are *more* likely to end up with a domain-oriented architecture, but that is by the by. The fact is though that there can often be reasons to mix models, even if "domain-oriented" is what you decide to pick as your main mechanism for defining microservice boundaries.

The different mechanisms we've outlined so far also have a lot of potential interplay between them. Being too narrow in your choices here will cause you to follow the dogma, rather than do the right thing. Volatility-based decomposition can make a lot of sense if your focus is on improving the speed of delivery, but if this causes you to extract a service which crosses organizational boundaries, then expect your pace of change to suffer due to delivery contention.

I might define a nice `Warehouse` service based on my understanding of the business domain, but if part of that system needs to be implemented in C++, and another part in Kotlin, then you'll need to decompose further along those technical lines.

For me, organizational and domain-driven service boundaries are my starting point. It's the usual tool I pick up, because as a general rule of thumb it's the one that tends to work best. But it's just that, a general model. It's also extremely rare that the domain is the only factor driving this decision making. Typically, a number of the factors we outlined above come into play, and which ones influence your own decisions will be based on what problems you are trying to solve. You need to look at your own specific circumstances to determine what works best for you - and hopefully I've given you a few different options here to consider. Just remember, if someone says "The only way to do this is X!" they are likely just selling you more dogma. You can do better than that.

With all that said, let's dive deeper into the topic of domain modelling, by exploring domain-driven design in a little more detail.

Just Enough Domain-Driven Design

So as we see, modeling our services around a business domain has significant advantages for our microservice architecture. The question is how to come up with that model—and this is where domain-driven design (DDD) comes in.

The desire to have our programs better represent the real world in which the programs themselves will operate is not a new idea. Object-oriented programming languages like Simula were developed to allow us to model real domains. But it takes more than program language capabilities for this idea to really take shape.

Eric Evans' Domain-Driven Design[10] presented a series of important ideas that helped us better represent the problem domain in our programs. A full exploration of these ideas is outside the scope of this book, but I'll provide a brief overview of the most important ideas in the context of microservice architectures.

Ubiquitous Language

Ubiquitous language refers to the idea that we should strive to use the same terms in our code as the users use . The idea is that by having a common language between the delivery team and the actual people who use the system it will be easier to model the real-world domain, and should also improve communication.

As a counter-example, I recall a situation when working at a large, global bank. We were working in the area of corporate liquidity, a fancy term that basically refers to the ability to move cash between different accounts held by the same corporate entity. The product owner was really great to work with, and she had a fantastic deep understanding of the various products that she wanted to bring to market. When working with her, we'd have discussions about things like haircuts and end-of-day sweeps, all things which made a lot of sense in her world and which had meaning to her customers.

The code on the other hand had none of this language in there. At some point previously, a decision had been made to use a standard data model for the database. It was widely refereed to as "The IBM banking model", but I never got to grips as to whether or not this was a standard IBM product, or just the creation of a consultant from IBM. By defining the loose concept of an "arrangement", the theory went that any banking operation could be modeled. Taking out a loan? That was an arrangement. Buying a share? That's an arrangement! Applying for a credit card? Guess what - that's an arrangement too!

The data model had polluted the code to such an extent that the codebase was shorn of all real understanding of the system we were building. We weren't building a generic banking application. We were building a system specifically to manage corporate liquidity. The problem was that we had to map the rich domain language of the product owner to the generic code concepts - meaning a lot of work in helping translate. Our business analysts were often just spending their time explaining the same concepts over and over again as a result.

By working the real world language into the code, things became much easier. A developer picking up a story written using the terms that had come straight from the product owner was much more likely to understand their meaning and work out what needed to be done.

Aggregate

In DDD, an *aggregate* is a somewhat confusing concept, with many different definitions out there. Is it just an arbitrary collection of objects? The smallest unit I should take out of a database? The model that has always worked for me is to first consider an aggregate as a representation of a real domain concept—think of something like an Order, Invoice, Stock Item, etc. Aggregates typically have a life cycle around them, which opens them up to being implemented as a state machine.

As an example in the MusicCorp domain, an Order aggregate might contain multiple line items that represent the items in the order. Those line items only have meaning as part of the overall Order aggregate.

We want to treat aggregates as self-contained units; we want to ensure that the code that handles the state transitions of an aggregate are grouped together, along with the state itself. So one aggregate should be managed by one microservice, although a single microservice might own management of multiple aggregates.

In general though, think of an aggregate as something which has state, has identity, and has a lifecycle that will be managed as part of the system. They typically refer to real-world concepts.

When thinking about aggregates and microservices, a single microservice will handle the life cycle and data storage of one or more different types of aggregates. If functionality in another service wants to change one of these aggregates, it needs to either directly request a change in that aggregate, or else have the aggregate itself react to other things in the system to initiate its own state transitions, perhaps by subscribing to events issued by other microservices.

The key thing to understand here is that if an outside party requests a state transition in an aggregate, the aggregate can say no. You ideally want to implement your aggregates in such a way that illegal state transitions are impossible.

Aggregates can have relationships with other aggregates. In Figure 2-15, we have a Customer aggregate, which is associated with one or more Orders, and one or more Wishlists. Each of these aggregates could be managed by the same microservice, or different microservice.

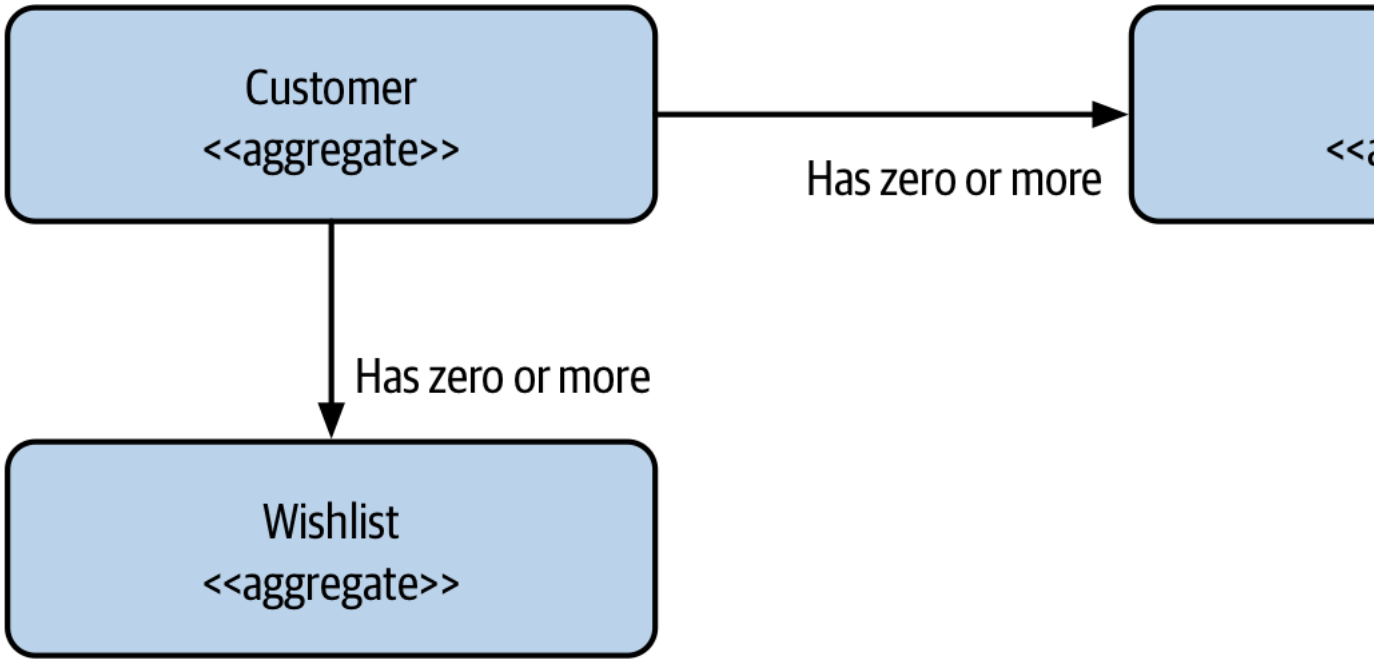


Figure 2-15. One Customer aggregate may be associated with one or more Order or Wishlist aggregates

If these relationships between aggregates exist inside the scope of a single microservice, the relationships could easily be stored using something like a foreign key relationship if using a relational database. If the relationships between these aggregates span microservice boundaries though, we need some way to model this relationship. In Figure 2-16, we have an entry in a financial ledger being made against a customer - this could represent a Payment aggregate. In the ledger table, we store a reference for the customer, here in the form of a URI which we might use if building a REST-based system[11]. We'll revisit the topic of cross-service relationships of this nature in Chapter 3 to explore the nature and use of these references in more detail.

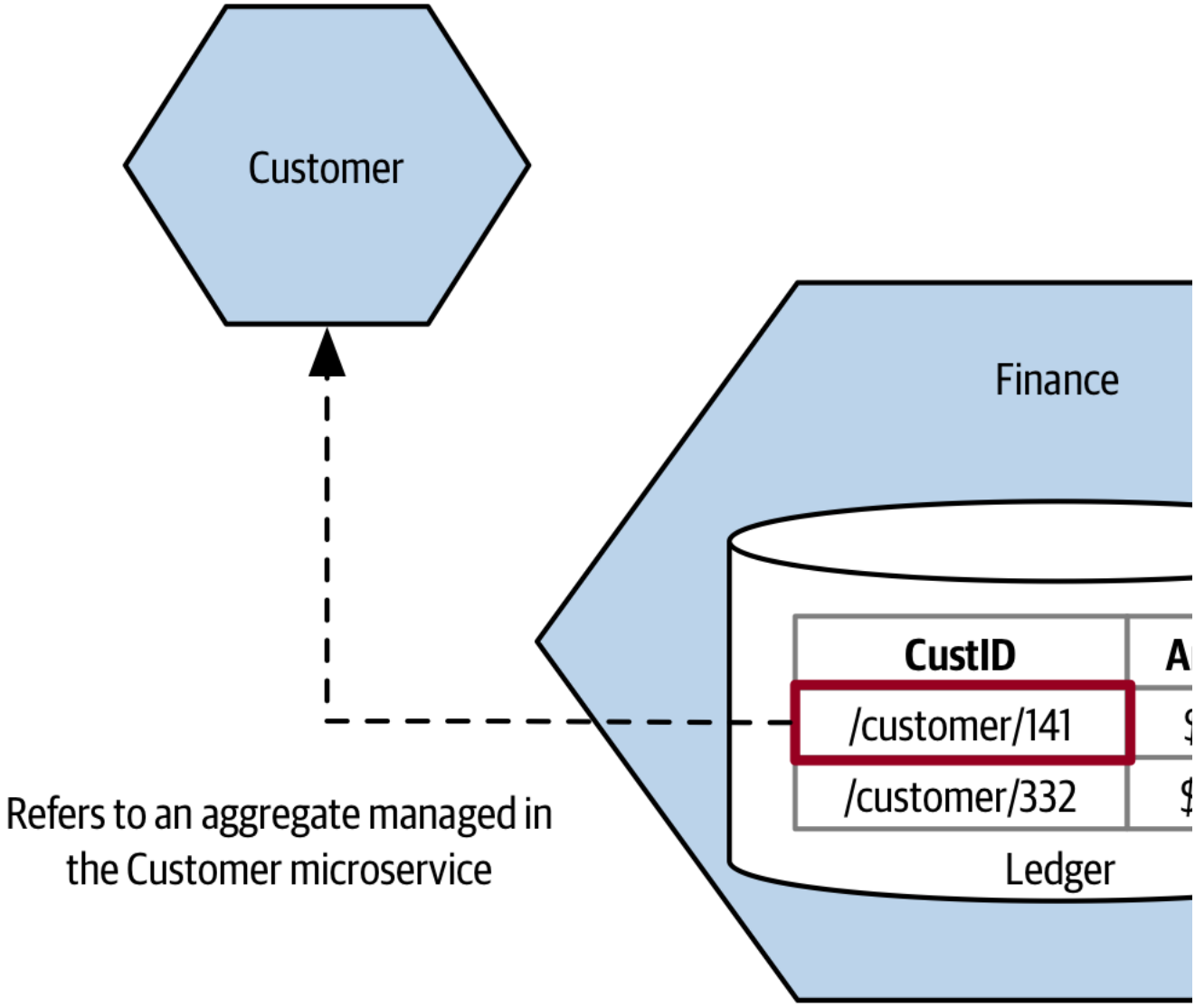


Figure 2-16. An example of how a relationship between two aggregates in different microservices can be implemented

There are lots of ways to break a system into aggregates, with some choices being highly subjective. You may, for performance reasons or ease of implementation, decide to reshape aggregates over time. To start with, though, I consider implementation concerns to be secondary, initially letting the mental model of the system users be my guiding light on initial design until other factors come into play.

Bounded Context

A *bounded context* typically represents a larger organizational boundary inside an organization. Within the scope of that boundary, explicit responsibilities need to be carried out. That's all a bit wooly, so let's look at another specific example.

At Music Corp, our warehouse is a hive of activity—managing orders being shipped out (and the odd return), taking delivery of new stock, having forklift truck races, and so on. Elsewhere, the finance department is perhaps less fun-loving, but still has an important function inside our organization, handling payroll, paying for shipments, and the like.

Bounded contexts hide implementation detail. There are internal concerns—for example, the types of forklift trucks used is of little interest to anyone other than the folks in the warehouse. These internal concerns should be hidden from the outside world—they don't need to know, nor should they care.

From an implementation point of view, bounded contexts contain one or more aggregates. Some aggregates may be exposed outside the bounded context; others may be hidden internally. As with aggregates, bounded contexts may have relationships with other bounded contexts—when mapped to services, these dependencies become inter-service dependencies.

Let's return for a moment to the MusicCorp business. Our domain is the whole business in which we are operating. It covers everything from the warehouse to the reception desk, from finance to ordering. We may or may not model all of that in our software, but that is nonetheless the domain in which we are operating. Let's think about parts of that domain that look like the bounded contexts that Evans refers to.

Hidden Models

For MusicCorp, we can then consider the finance department and the warehouse to be two separate bounded contexts. They both have an explicit interface to the outside world (in terms of inventory reports, pay slips, etc.), and they have details that only they need to know about (forklift trucks, calculators).

Now the finance department doesn't need to know about the detailed inner workings of the warehouse. It does need to know some things, though—for example it needs to know about stock levels to keep the accounts up to date. Figure 2-17 shows an example context diagram. We see concepts that are internal to the warehouse, like Picker (people who pick orders), shelves that represent stock locations, and so on. Likewise, entries in the general ledger is integral to finance but is not shared externally here.

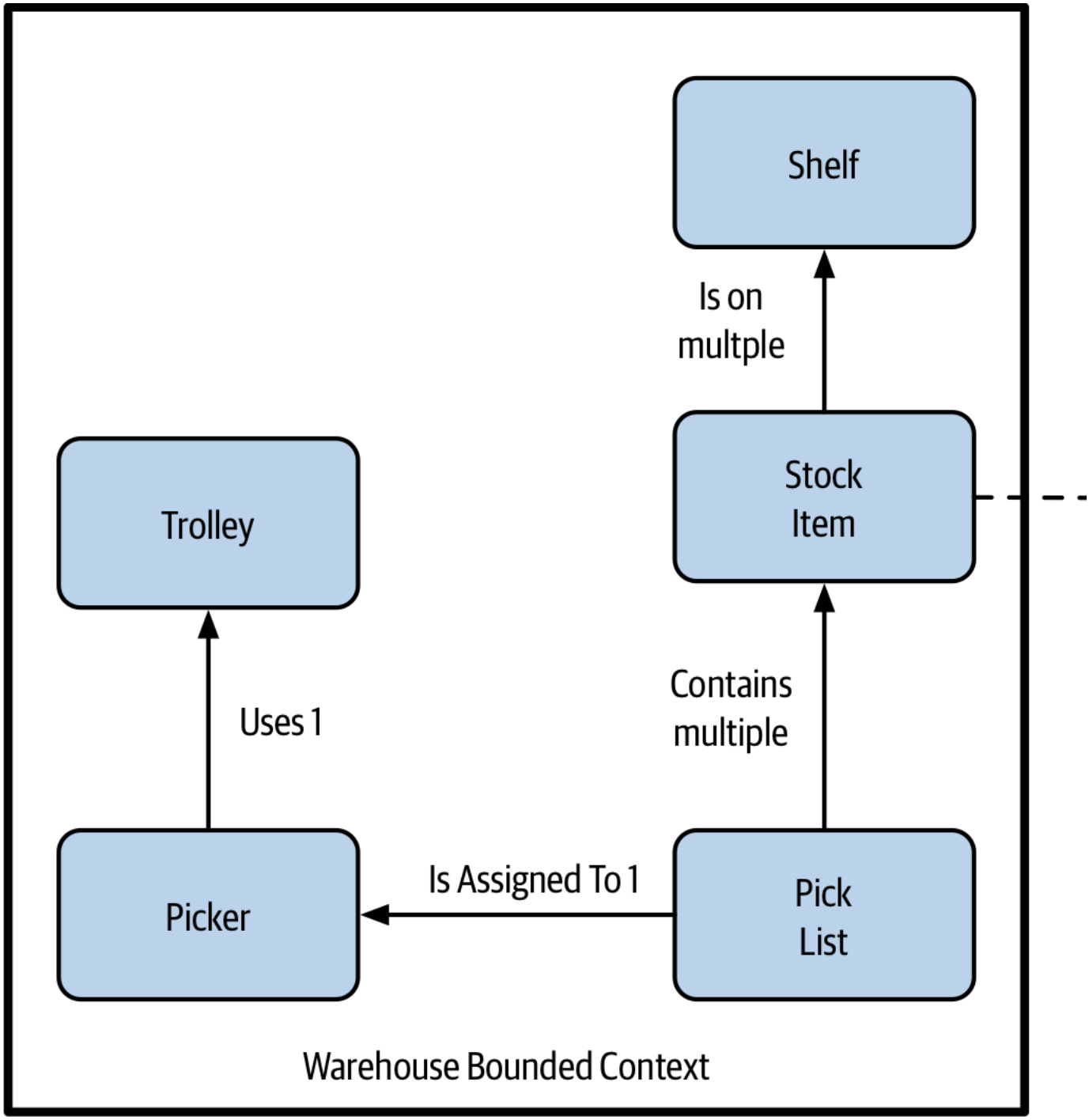


Figure 2-17. A shared model between the finance department and the warehouse

To be able to work out the valuation of the company, though, the finance employees need information about the stock we hold. The stock item then becomes a shared model between the two contexts. However, note that we don't need to blindly expose everything about the stock item from the warehouse context. In Figure 2-18, we see how `Stock Item` inside the warehouse bounded context contains references to the shelf locations, but the shared representation just contains a a count. So there is the internal-only representation, and the external representation we expose. Often, when you have different internal and external representations, it may be beneficial to name them differently to avoid confusion - in this situation, one approach could be to call the shared `Stock Item` a `Stock Count` instead.

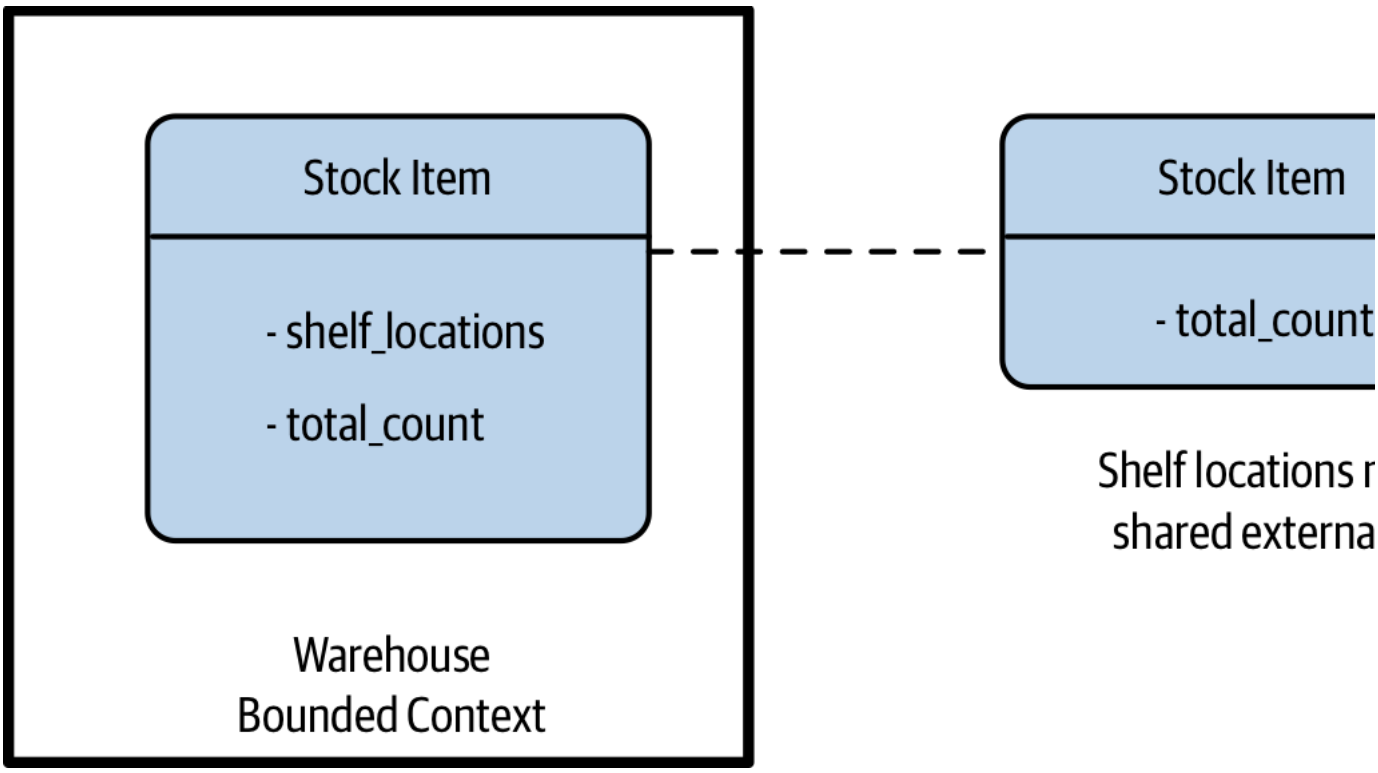


Figure 2-18. A model which is shared can decide to hide information that should not be shared externally

Shared Models

We can also have concepts which appear in more than one bounded context. In Figure 2-17 we saw that a customer exists in both locations. What does this mean? Is the customer copied? The way to think about this is that conceptually, both finance and warehouse needs to know something about our customer. Finance need to know about the financial payments made to a customer, whereas the Warehouse needs to know about the customer to the extent that it knows what packages have been sent to allow for deliveries to be traced.

When you have a situation like this, a shared model like customer can have different meanings in the different bounded contexts, and therefore might be called different things. We might be happy to keep the name "customer" in Finance, but in Warehouse we might call them a "recipient", as that is the role they play in that context. We store information about the customer in both locations, but the information is different. Finance stores information about the customer's financial payments (or refunds), the warehouse stores information related to the goods shipped. We still may need to link both local concepts to a global customer, and we may want to look up common, shared information about that customer like their name or email address - we could use a technique like that shown in Figure 2-16 to achieve this.

Mapping Aggregates and Bounded Contexts to Microservices

Both the aggregate and the bounded context give us units of cohesion with well-defined interfaces with the wider system. The aggregate is a self-contained state machine that focuses on a single domain concept in our system, with the bounded context representing a collection of associated aggregates, again with an explicit interface to the wider world.

Both can therefore work well as service boundaries. When starting out, as I've already mentioned, you want to reduce the number of services you work with. As a result, you should probably target services that encompass entire bounded contexts. As you find your feet, and decide to break these services into smaller services, look to split them around aggregate boundaries.

Turtles All the Way Down

At the start, you will probably identify a number of coarse-grained bounded contexts. But these bounded contexts can in turn contain further bounded contexts. For example, you could decompose the warehouse into capabilities associated with order fulfillment, inventory management, or goods receiving. When considering the boundaries of your microservices, first think in terms of the larger, coarser-grained contexts, and then subdivide along these nested contexts when you're looking for the benefits of splitting out these seams.

A trick here is that even if you decide to split a service that models an entire bounded context into smaller services later on, you can still hide this decision from the outside world—perhaps by presenting a coarser-grained API to consumers. The decision to decompose a service into smaller parts is arguably an implementation decision, so we might as well hide it if we can. in Figure 2-19 we see an example of this. We've split `Warehouse` down into `Inventory` and `Shipping`. As far as the outside world is concerned, there is still just the `Warehouse` microservice. Internally though, we've further decomposed things to allow `Inventory` to manage `Stock Items` and have `Shipping` manage `Shipments`. Remember, we want to keep the ownership of a single aggregate inside a single microservice.

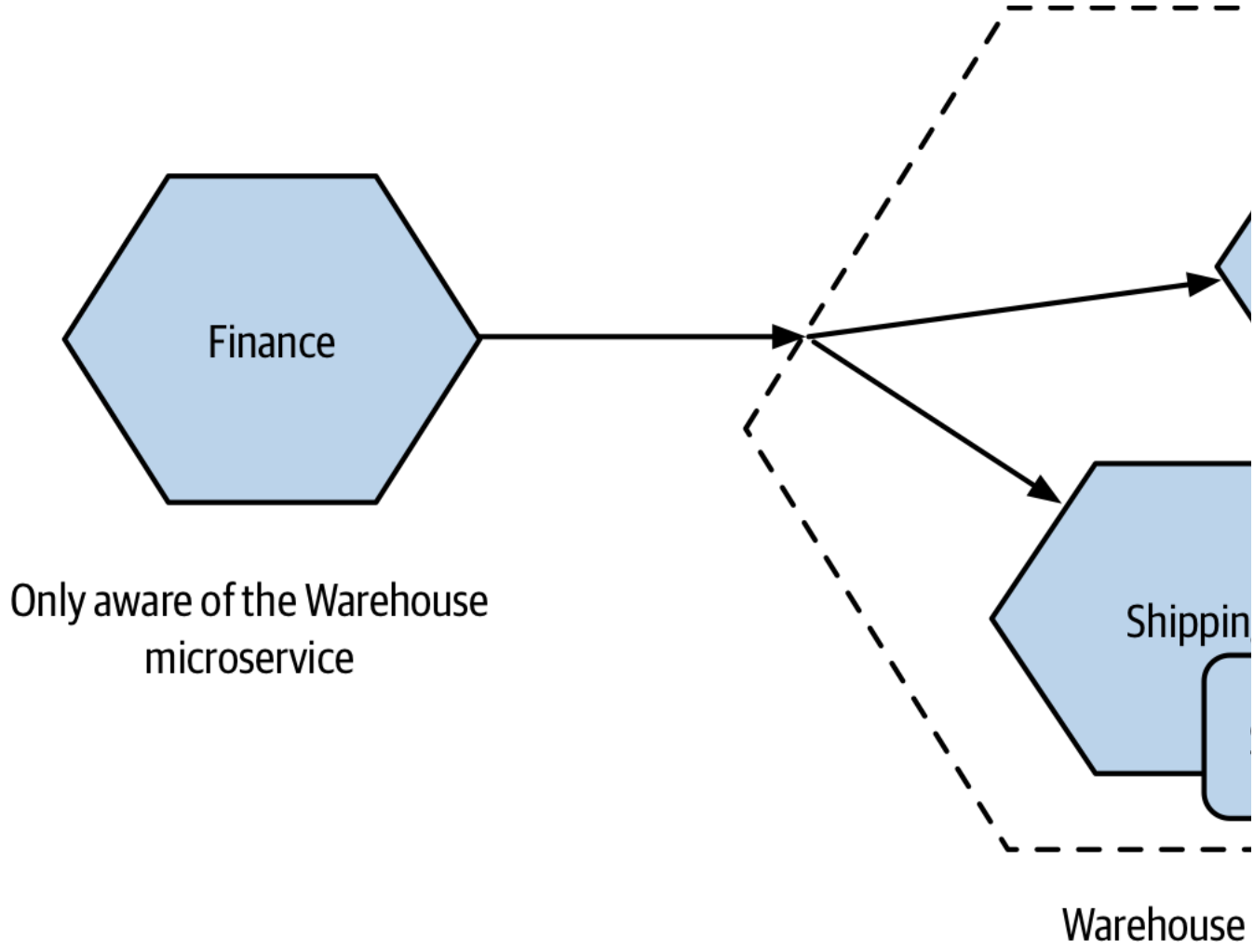


Figure 2-19. The Warehouse service internally has been split into a Finance and Warehouse microservice

This is another form of information hiding - we've hidden a decision about internal implementation in such a way that if this implementation detail changes again in the future then our consumers will be unaware.

Another reason to prefer the nested approach could be to chunk up your architecture to simplify testing. For example, when testing services that consume the warehouse, I don't have to stub each service inside the warehouse context, just the more coarse-grained API. This can also give you a unit of isolation when considering larger-scoped tests. I may, for example, decide to have end-to-end tests where I launch all services inside the warehouse context, but for all other collaborators I might stub them out. We'll explore more about testing and isolation in [Link to Come].

The Dangers Of Premature Decomposition

There is a danger in creating decomposing microservices based on an unclear understanding of the domain. One such example comes from my previous company, ThoughtWorks. One of their products was called SnapCI, a hosted continuous integration and continuous delivery tool (we'll discuss those concepts later in [Link to Come]). The team had previously worked on another similar tool, Go-CD, a now open source continuous delivery tool that can be deployed locally rather than being hosted in the cloud.

Although there was some code reuse very early on between the SnapCI and Go-CD projects, in the end SnapCI turned out to be a completely new codebase. Nonetheless, the previous experience of the team in the domain of CD tooling emboldened them to move more quickly in identifying boundaries, and building their system as a set of microservices.

After a few months, though, it became clear that the use cases of SnapCI were subtly different enough that the initial take on the service boundaries wasn't quite right. This led to lots of changes being made across services, and an associated high cost of change. Eventually the team merged the services back into one monolithic system, giving them time to better understand where the boundaries should exist. A year later, the team was then able to split the monolithic system apart into microservices, whose boundaries proved to be much more stable. This is far from the only example of this situation I have seen. Prematurely decomposing a system into microservices can be costly, especially if you are new to the domain. In many ways, having an existing codebase you want to decompose into microservices is much easier than trying to go to microservices from the beginning for this very reason.

Communication in Terms of Business Concepts

The changes we implement to our system are often about changes the business wants to make to how the system behaves. We are changing functionality—capabilities—that are exposed to our customers. If our systems are decomposed along the bounded contexts that represent our domain, the changes we want to make are more likely to be isolated to one, single microservice boundary. This reduces the number of places we need to make a change, and allows us to deploy that change quickly.

It's also important to think of the communication between these microservices in terms of the same business concepts. The modeling of your software after your business domain shouldn't stop at the idea of bounded contexts. The same terms and ideas that are shared between parts of your organization should be reflected in your interfaces. It can be useful to think of forms being sent between these microservices, much as forms are sent around an organization.

Event-storming

Event Storming, a technique developed by Alberto Brandolini, is a collaborative brainstorming exercise designed to help surface a domain-model. Rather than having an architect sit in a corner and come up with their own representation of what the domain model is[12], event storming brings together technical and non-technical stakeholders in a joint exercise. The idea is that by making the development of the domain model a joint activity, that you end up with a shared, joined-up view of the world.

It's worth mentioning at this point that while the domain models defined using event storming can be used to implement event-driven systems, and indeed the mapping is very straightforward, you can also use such a domain model to build a more request/response oriented system too.

Logistics

Alberto has some very specific views as to how event storming should be run, and on some of these points I am very much in agreement. Firstly, get everyone in a room together. This is often the most difficult step - getting people's calendars to line up can be a problem, as can finding a room big enough. Those issues were all true in a pre-covid world, but as I write this during the virus-related lockdown in the UK, I'm aware that this might be even more problematic in the future. The key here though is to have all stakeholders present at the same time. You want representatives for all parts of the domain that you plan to model - users, subject matter experts, product owners, whoever is best placed to help represent that part of the domain.

Once in a room together, Alberto suggests the removal of all chairs, in order to make sure that everyone gets up and gets involved. As someone with a bad back, while this is something I understand, it may not work for everyone. One thing I do agree with Alberto about is the need to have a large space where the modelling can be done. A common solution here is to pin large rolls of brown paper to the walls of the room, allowing for all the walls to be used for capturing information.

The main modelling tool is post-it notes to capture different concepts, with different coloured post-it notes representing different concepts.

The Process

The exercise starts with the participants identifying the *domain events*. These represent things that happen in the system - they are the facts that you care about. "Order Placed" would be a good event that we would care about in the context of MusicCorp, as would "Payment Received". These are captured on orange post-it notes. It is at this point that I have another disagreement with Alberto's structure here, as the events are far and away the most numerous things you'll be capturing, and orange post-it notes are surprisingly hard to get hold off[13].

Next, participants identify the commands that cause these events to happen. Commands are decisions made by a human to do something (a user of the software). Here you are trying to understand the boundary of the system, and identify the key human actors in the system. Commands are captured on blue post-it notes.

For the techies in the event storming session, at this stage they should be listening to what their non-technical colleagues come up with here. A key part of this exercise is not to let any current implementation warp the perception of what the domain is (that comes later). At this stage you want to create a space where you can get the concepts out of the heads of the key stakeholders, and get these ideas out into the open.

With events and commands captured, aggregates come next. The events you have at this stage are useful sharing not just what happens in the system, but also it starts to highlight what the potential aggregates might be. Think of the aforementioned domain event "Order Placed". The noun here - Order - could well be a potential aggregate. And "Placed" is something that can happen to an order, so this may well be part of the life-cycle of the aggregate. Aggregates are represented by yellow post-it notes, and the commands and events associated with that aggregate are moved and clustered around the aggregate. This also helps you understand how aggregates are related to each other - events from one aggregate might trigger behavior in another.

With the aggregates identified, they are then grouped into bounded contexts. Bounded contexts most commonly follow a company's organizational structure, and the participants of the exercise are well placed to understand what aggregates are used by which parts of the organization.

There is more to event storming than this - it was just meant as a brief overview. For a more detailed overview of how Event Storming works I'd suggest you read the (currently in progress) book "Event Storming[14]" by Alberto

Summary

In this chapter, you've learned a bit about what makes a good microservice boundary, and how to find seams in our problem space that give us the dual benefits of both low coupling and strong cohesion. Having a detailed understanding of your domain can be a vital tool in helping us find these seams, and by aligning our microservices to these boundaries we ensure that the resulting system has every chance of keeping those virtues intact. We've also got a hint about how we can subdivide our microservices further, something we'll explore in more depth later. And we also introduced MusicCorp, the example domain that we will use throughout this book.

The ideas presented in Eric Evans's *Domain-Driven Design* are very useful to us in finding sensible boundaries for our services, and I've just scratched the surface here. I recommend Vaughn Vernon's book *Implementing Domain-Driven Design* (Addison-Wesley) to help you understand the practicalities of this approach.

Although this chapter has been mostly high-level, we need to get much more technical in the next. There are many pitfalls associated with implementing interfaces between services that can lead to all sorts of trouble, and we will have to take a deep dive into this topic if we are to keep our systems from becoming a giant, tangled mess.

[1] Parnas, David, "On the criteria to be used in decomposing systems into modules", 1971 *https://kilthub.cmu.edu/articles/On_the_criteria_to_be_used_in_decomposing_systems_into_modules/6607958*

[2] The obvious starting point is Adrian's summary of "On the criteria…" *https://blog.acolyer.org/2016/09/05/on-the-criteria-to-be-used-in-decomposing-systems-into-modules/*, but the coverage of Parnas' earlier work "Information Distribution Aspects of Design Methodology" contains some great insights along with commentary from Parnas himself: *https://blog.acolyer.org/2016/10/17/information-distribution-aspects-of-design-methodology/*

[3] Parnas, David, "Information distribution aspects of design methodology", 1971

[4] In my book, Monolith To Microservices, I attributed this to Larry Constantine himself. While the statement neatly sums up much of Constantine's work in this space, the quote should really be attributed to Albert Endres and Dieter Rombach from their book "A Handbook of Software and Systems Engineering".

[5] Constantine, Larry and Edward Yourdon, Structured Design, Yourdon Press

[6] Page-Jones, Meilir, Practical Guide to Structured Systems Design (Yourdon Press Computing)

[7] This concept is similar to the Domain Application Protocol which defines the rules by which components interact in a REST-based system.

[8] Pass through coupling is my name for what was originally described as Tramp coupling by M. Page-Jones: The Practical Guide to Structured Systems Design. I chose to use a different term here due to the fact that I found the original term somewhat problematic, and not terribly meaningful to a wider audience

[9] OK, more than once or twice. A **lot** more than once or twice…

[10] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley, 2004).

[11] I know some people object to the use of templated URIs in REST systems, and I understand why - I just want to keep things simple for this example

[12] I mean no disrespect if this is you - I've done this myself more than once

[13] I mean, why not yellow? It's the most common colour!

[14] Event Storming, Alberto Brandolini, Leanpub (work in progress) *https://leanpub.com/introducing_eventstorming*

Chapter 3. Microservice Communication Styles

Work In Progress

Please note that the text below is currently being reworked for the 2nd edition of the book, and is not in a complete state. This will be Chapter 3 of the final book.

If you have any feedback on the book, or suggestions for the 2nd edition, then please contact me on book-feedback@samnewman.io and/or complete a short survey here: *https://oreil.ly/Bldg_MicroServices_survey*.

Getting communication between microservices right is problematic for many, in great part due to the fact that I feel that people gravitate towards a chosen technological approach without first considering the different types of communication you might want. In this chapter, I'll try and tease apart the different styles of communication, to help you understand the pros and cons of each, and also help you understand which approach will best fit your problem space.

We'll be looking at synchronous blocking and asynchronous non-blocking communication mechanisms, as well as comparing request-response collaboration with event-driven collaboration.

By the end of this chapter you should be much better prepared to understand the different options available to you, and will have a foundational knowledge that will help when we start looking at more detailed implementation concerns in the following chapters.

From In-Process To Inter-Process

OK, let's get the easy stuff out of the way first - or at least what I *hope* is the easy stuff. Namely, calls *between* different processes across a network (inter-process) are **very** different to calls *within* a single process (in-process). At one level, we can ignore this distinction. It's easy, for example, to think of one object making a method call on another object, then just map this interaction to two microservices communicating via a network. Putting aside the fact that microservices aren't just objects, this thinking can get us into a lot of trouble.

Let's look at some of these differences now, and how they might change how you think about the interactions between your microservices.

Performance

The performance of an in-process call and an inter-process call is fundamentally different. When I make an in-process call, the underlying compiler and runtime can carry out a whole host of optimizations to reduce the impact of the call, including inlining the invocation so it's as though there was never a call in the first place. No such optimizations are possible with inter-process calls. Packets have to be sent. Expect the overhead of an inter-process call to be significant compared to the overhead of an in-process call. The former is very measurable - just round-tripping a single packet in a data centre is measured in milliseconds - whereas the overhead of making a method call is something you don't need to worry about.

This can often lead you to want to rethink APIs. An API that makes sense in-process may not make sense in inter-process situations. I can make 1000 calls across an API boundary in-process without concern. Do I want to make 1000 network calls between two microservices? Perhaps not.

When I pass a parameter into a method, the data structure I pass in typically doesn't move - what's more likely is that I pass around a pointer to a memory location. Passing in an object or data structure to another method doesn't necessitate more memory to be allocated in order to copy the data.

When making calls between microservices over a network on the other hand, the data actually has to be serialized into some form that can be transmitted over a network. The data then needs to be sent, and deserialized at the other end. We therefore may need to be more mindful about the size of payloads being sent between processes. When was the last time you were aware of how big a data structure was that you were passing around inside a process? The reality is that you likely didn't need to know - now, you do. This might lead you to reduce the amount of data being sent or received (perhaps not a bad thing if we think about information hiding), pick more efficient serialization mechanisms, or even offload data to a file system and pass around pointers to that data instead.

These differences may not cause you issues straight away, but you certainly need to be aware of them. I've seen a lot of attempts to hide from the developer the fact that a network call is even taking place. Our desire to create abstractions to hide detail is a big part of what allows us to do more things more efficiently, but

sometimes we create abstractions that hide too much. A developer needs to be aware if they are doing something that will result in a network call, otherwise do not be surprised if you end up with some nasty performance bottlenecks further down the line.

Changing Interfaces

When we consider changes to an interface inside a process, the act of rolling out the change is straightforward. Both the code implementing the interface, and the code calling the interface, are all packaged together in the same process. In fact if I change a method signature using an IDE with refactoring capability, often the IDE itself will automatically refactor calls to this changing method. Rolling out such a change can be done in an atomic fashion - both sides of the interface are packaged together in a single process.

With communication between microservices, however, the microservice exposing an interface, and the consuming microservices using that interface, are separately deployable microservices. When making a backwards incompatible change to a microservice interface, we either need to do a lock-step deployment with consumers, making sure they are updated to use the new interface, or else find some way to phase the rollout of the new microservice contract. We'll explore this concept in more detail later in this chapter.

Error handling

Within a process, if I call a method, the nature of the errors tends to be pretty straightforward. Simplistically, the errors are either expected and easy to handle, or else they are catastrophic to the point where we just propagate the error up the call stack. Errors, on the whole, are deterministic.

With a distributed system, the nature of errors can be different. You are vulnerable to a host of errors that are outside of your control. Networks time out. Downstream microservices might be temporarily unavailable. Networks get disconnected, containers get killed due to consuming too much memory, and in extreme situations, bits of your data centre can catch fire[1].

Many of these errors are often transient in nature - they are short-lived problems that might go away, and therefore are things you might want to retry - think of a simple network timeout. Other problems can't be dealt with easily. As a result, it can become important to have a richer set of semantics for returning errors in a way that can allow for clients to take appropriate action.

HTTP is an example of a protocol that understands the importance of this. Every HTTP response has a code, with the 400 and 500 series codes being reserved for errors. 400 series error codes are request errors - essentially, a downstream service is telling the client that there is something wrong with the original request. As such, it's probably something you should give up with - is there any point retrying a `404 Not Found` for example? The 500 series response codes relate to downstream issues, a subset of which indicate to the client that the issue might be temporary. A `503 Service Unavailable` for example indicates that the downstream serer is unable to handle the request, but that this could be a temporary state. In which case, an upstream client might decide to retry this request. On the other hand, if a client received a `501 Not Implemented` response, a retry is unlikely to help much.

Whether or not you pick a HTTP-based protocol for communication between microservices, if you have a rich set of semantics around the nature of the error, you'll make it easier for clients to carry out compensating actions, which in turn should help you build more robust systems.

Technology for Inter-process Communication: So Many Choices

> "And in a world where we have too many choices and too little time, the obvious thing to do is just ignore stuff."
>
> Seth Godin

The range of technology available to us for inter-process communication is vast. As a result, we can often be overburdened with choice. Often, I find people just gravitate to technology which is familiar to them, or perhaps just the latest hot technology they learned about from a conference. The problem with this is that when you buy into a specific technology choice, you are often buying into a set of ideas (and constraints) that come along for the ride. These constraints might not be the right ones for you - and the mindset behind the technology may not actually line up with the problem you are trying to solve.

If you're trying to build a website, single page app technology like Angular or React is a bad fit. Likewise, trying to use Kafka for request-response really isn't a good idea, as it was designed for more event-based interactions (topics we'll get to in just a moment). And yet I see technology used in the wrong place time and time again. People pick the new shiny tech (like microservices!) without considering whether or not it fits their problem.

When it comes to the bewildering array of technology available to us for communication between microservices, I therefore think it is important to talk first about the style of communication you want, and only then look for the right technology to implement these styles. With that in mind, let's take a look at a model I've been using for several years to help distinguish between the different approaches for microservice-to-microservice communication, which in turn can help you filter the technology options you'll want to look at.

Styles of Microservice Communication

In Figure 3-1 we see an outline for the model I use for thinking about different styles of communication. This model is not meant to be entirely exhaustive (I'm not trying to present a grand unified theory of inter-process communication here), more that it provides a good high-level overview for considering the different styles of communication which are most widely used for microservice architectures.

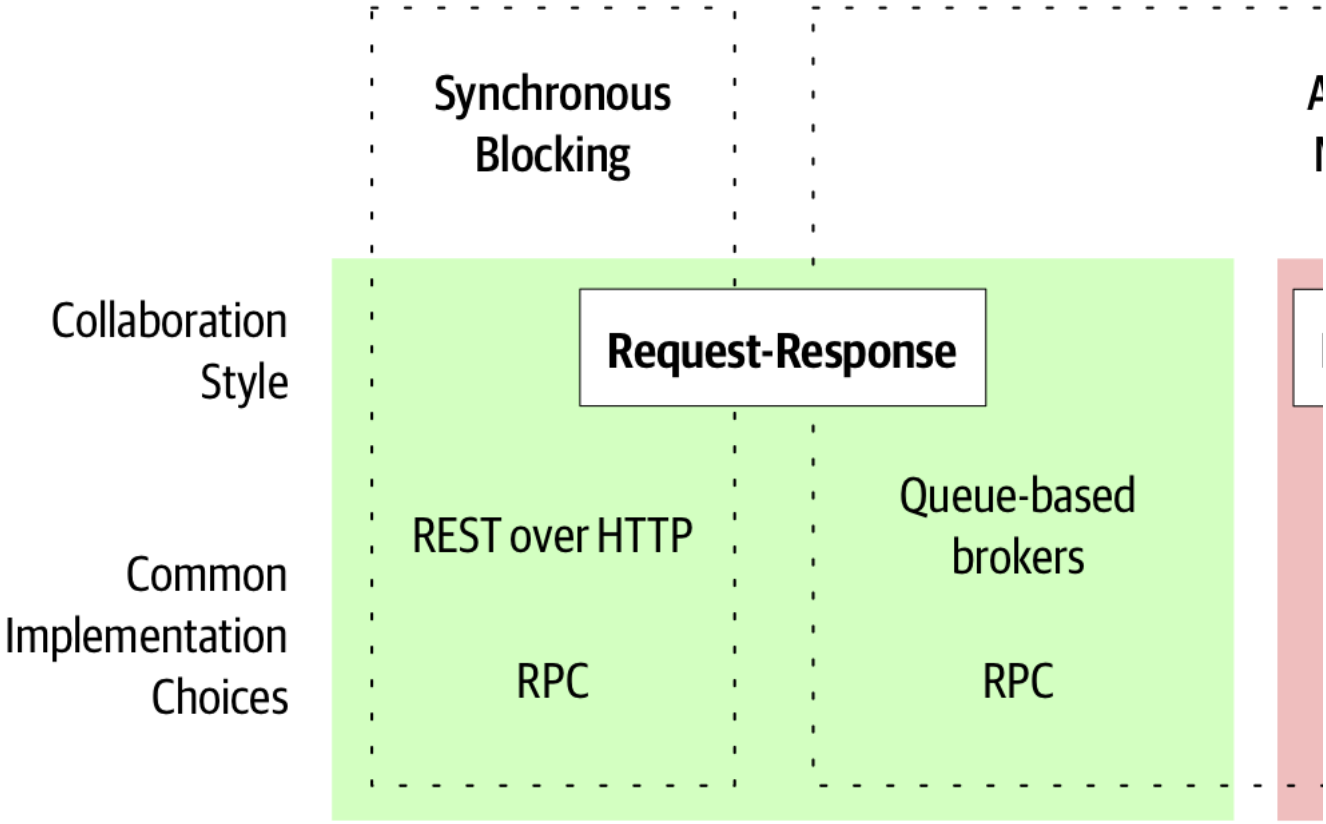


Figure 3-1. Different styles of inter-microservice communication along with example implementing technologies

We'll look at each element in more detail shortly, but first I'd like to briefly outline the different elements of this model.

Synchronous Blocking

A microservice makes a call to another microservice and blocks operation waiting for the response.

Asynchronous Non-Blocking

The microservice emitting a call is able to carry on processing whether or not the call is received.

Request-response

A Microservice sends a request to another microservice asking for something to be done. It expects to receive a response to the request informing it of the result.

Event-Driven

Microservices emit events, which other microservices consume and react to accordingly. The microservice emitting the event is unaware of which microservices, if any, consume the events it emits.

Common Data

Not often seen as a communication style, microservices collaborate via some shared data source.

When using this model to help teams decide on the right approach, I spend a lot of time understanding the context in which they are operating. Their needs in terms of reliable communication, acceptable latency and volume of communication are all going to play a part in making a technology choice. But in general, I tend to start with deciding if synchronous or asynchronous communication is more appropriate for the given situation. If synchronous communication is an option, then I am firmly in the world of request-response communication. If asynchronous communication makes more sense, then I have a second choice to make, which is whether or not event-driven, request-response-based or common data-based communication is more appropriate. As we'll explore, event-driven communication is fundamentally asynchronous, but request-response calls can be implemented synchronously or asynchronously.

Pattern: Synchronous Blocking

With a synchronous blocking call, a microservice sends a call of some kind to a downstream process (likely another microservice), and blocks until the call has completed, and potentially until a response has been received. In Figure 3-2, the `Order Processor` sends a call to the `Loyalty` microservice to inform it that some points should be added to a customer's account.

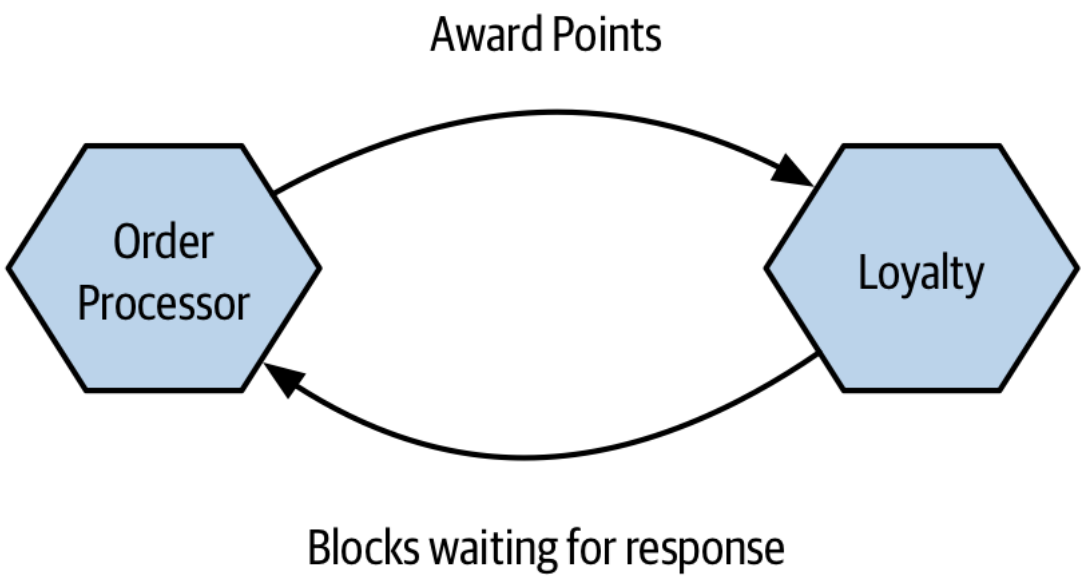


Figure 3-2. Order Processor sends a synchronous call to the Loyalty microservice, blocks and waits for a response

Typically, a synchronous blocking call is one that is waiting for a response from the downstream process. This may be because the result of the call is needed for some further operation, or just because it wants to make sure the call worked and if not carry out some sort of retry. As a result, virtually all synchronous blocking calls I see would also constitute being a request-response call, something we'll look at shortly.

Advantages

There is something simple and familiar about a blocking, synchronous call. Many of us learned to program in a fundamentally synchronous style, reading a piece of code like a script, with each line executing in turn, with the next line of code waiting its turn to do something. Most of the situations where you would have used inter-process calls were probably done so in a synchronous, blocking style. Running a SQL query on a database for example, or making a HTTP request of a downstream API.

When moving from a less distributed architecture, like that of a single process monolith, it can make sense to stick with those ideas that are familiar when there is so much else going on that is brand new.

Disadvantages

The main challenge with synchronous calls is the inherent temporal coupling that occurs, a topic we explored briefly in Chapter 2. When the `Order Processor` makes a call to `Loyalty` in the example above, the `Loyalty` microservice needs to be reachable in order for the call to work. If the `Loyalty` microservice is unavailable, then the call will fail and `Order Processor` needs to work out what kind of compensating action to carry out - this might involve an immediate retry, buffering the call to retry later, or perhaps giving up altogether.

As the sender of the call is blocking and waiting for the downstream microservice to respond, it also follows that if the downstream microservice responds slowly, or if there is an issue with the latency of the network, then the sender of the call will be blocked for a prolonged period of time waiting for a response. If the `Loyalty` microservice is under significant load, and is responding slowly to requests, this in turn will cause the `Order Processor` to respond slowly.

The use of synchronous calls can therefore make a system more vulnerable to cascading issues caused by downstream outages more readily than asynchronous calls.

Where To Use It

For simple microservice architectures, I don't have a massive problem with the use of synchronous, blocking calls. Their familiarity for many people is an advantage when getting to grips with distributed systems.

For me, where these types of calls start to be problematic is when you start having more chains of calls - in Figure 3-3 for example, we have an example flow from MusicCorp, where we are checking a payment for potentially fraudulent activity. The `Order Processor` calls the `Payment` service to take payment. The `Payment` service in turn wants to check with the `Fraud Detection` microservice as to whether or not this should be allowed. The `Fraud Detection` microservice in turn needs to get information from the `Customer` microservice.

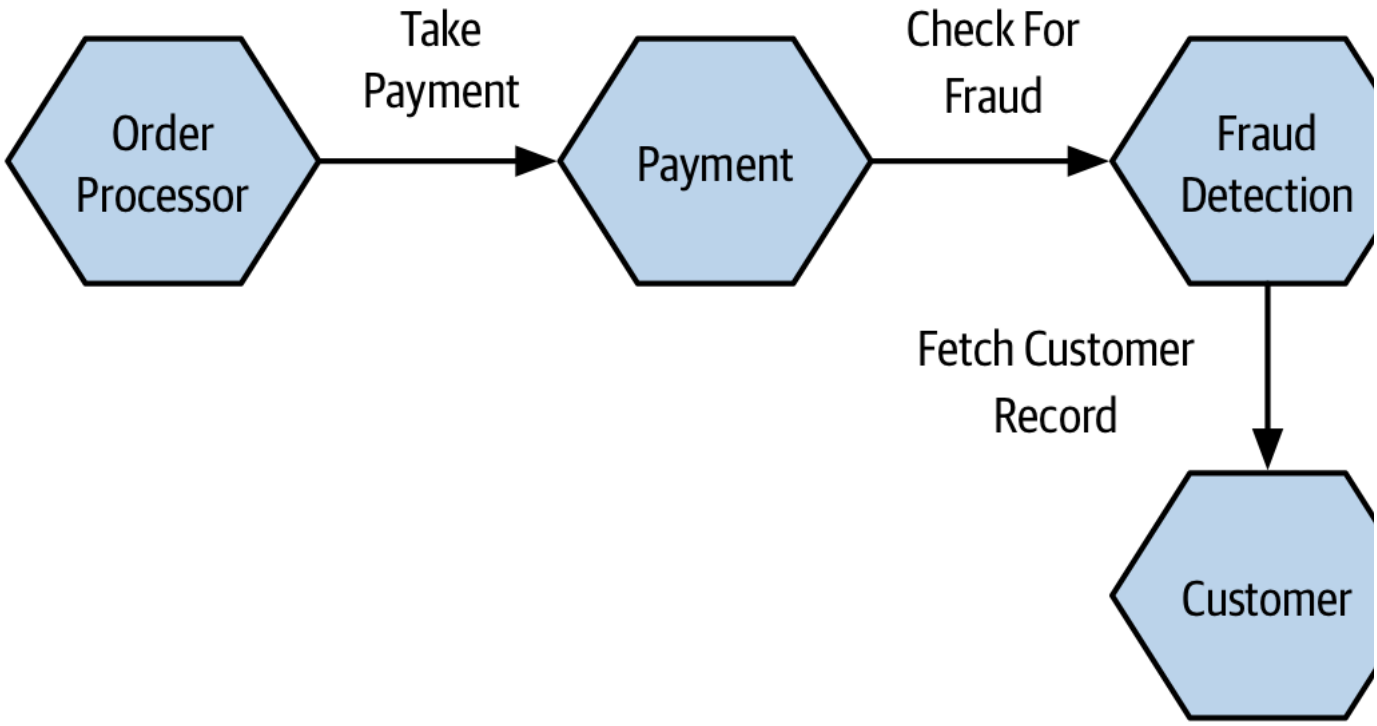


Figure 3-3. Checking for potentially fraudulent behavior as part of order processing flow

If all of these calls are synchronous and blocking, there are a number of issues we might face. A issue in any of the four involved microservices, or in the network calls between them, could cause the whole operation to fail - we arguably have a greater surface area for failure. This is quite aside from the fact that these kinds of long chains can cause significant *resource contention*. Behind the scenes, the `Order Processor` likely has a network connection open waiting to hear back from `Payment`. `Payment` in turn has a network connection open waiting for a response from `Fraud Detection` and so on. Having a lot of connections that need to be kept open can have an impact on the running system - you are much more likely to experience issues where you run out of available connections, or suffer from increased network congestion as a result.

To improve this situation, we could re-examine the interactions between the microservices in the first place. For example, maybe we take the use of the `Fraud Detection` out of the main purchase flow, as shown in Figure 3-4, and instead have it run in the background. If it finds a problem with a specific customer their records are updated accordingly, and this is something that could be checked earlier in the payment process. Effectively, this means we're doing some of this work in parallel. By reducing the length of the call chain we'll see the overall latency of the operation improve, and take one of our microservices (Fraud Detection) out of the critical path for the purchase flow, giving us one fewer dependencies to worry about for what is a critical operation.

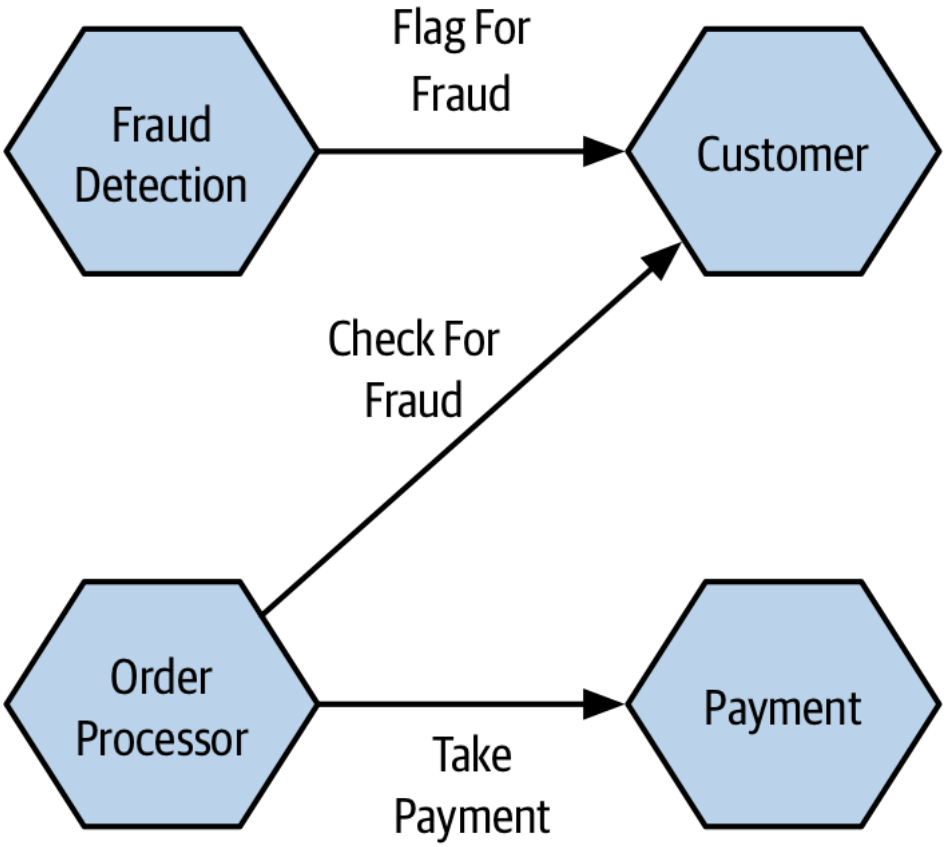


Figure 3-4. Moving fraud detection to a background process can reduce the concerns around the length of the call chain

We could also of course replace the use of blocking calls with some style of non-blocking interaction without changing the workflow here, something we'll explore next.

Pattern: Asynchronous Non-blocking

With asynchronous communication, the act of sending a call out over the network doesn't block the microservice issuing the call. It is able to carry on with any other processing without having to wait for a response. If a response is needed, it is able to handle that response when it returns. Non-blocking asynchronous communication comes in many forms, but we'll be looking in more detail at the three most common styles I see in microservice architecture. They are:

Communication Though Common Data

: The upstream microservice changes some common data, which one or more microservices later make use of.

Request-Response

: A microservice sends a request to another microservice asking it to do something. When the requested operation completes, successfully or not, the upstream microservice receives the response.

Event-Driven Interaction

: A microservice broadcasts an event, which can be thought of as a factual statement as to something that has happened. Other microservices can listen for the events they are interested in and react accordingly.

Advantages

With non-blocking asynchronous communication the microservice making the initial call, and the microservice (or microservices) receiving the call, are decoupled temporarily. The microservices that receive the call do not need to be reachable at the same time the call is made. This means we avoid the concerns of temporal decoupling that we discussed in Chapter 2 (see "A Brief Note On Temporal Coupling").

This style of communication is also beneficial if the functionality being triggered by a call will take a long time to process. Let's come back to our example of MusicCorp, and specifically the process of sending out a package. In Figure 3-5, the `Order Processor` has taken payment, and decided that it is time to dispatch the package, so it sends a call to the `Warehouse` microservice. The process of finding the CDs, taking them off the shelf, packaging them up, and having them picked up, could take many hours, potentially days, depending on how the actual dispatch process works. It makes sense therefore for the `Order Processor` to issue a non-blocking asynchronous call to the `Warehouse`, and have the `Warehouse` call back to the `Order Processor` later on to inform it of progress. This is a form of asynchronous request-response communication.

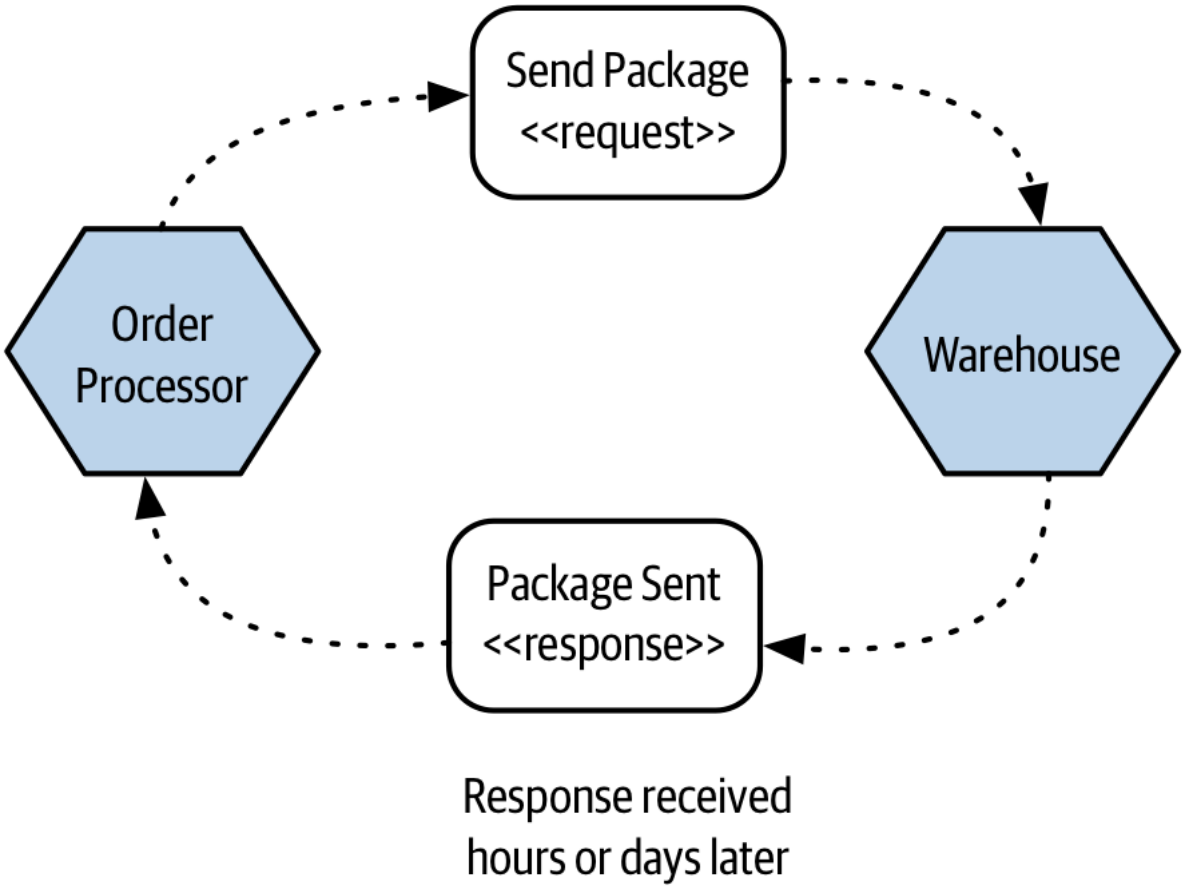


Figure 3-5. The Order Processor kicks off the process to package and ship an order, which is done in an asynchronous fashion

If we tried doing something similar with synchronous blocking calls, then we'd either have to restructure the interactions between `Order Processor` and `Warehouse` - it wouldn't be feasible for `Order Processor` to open a connection, send a request, block any further operations in the calling the thread, and wait for what might be hours or days waiting for a response.

Disadvantages

The main downsides of non-blocking asynchronous communication, relative to blocking synchronous communication, is the level of complexity and range of choice. As we've already outlined, there are different styles of asynchronous communication to choose from - which is right for you? When we start digging into how these different styles of communication are implemented, there is a potentially bewildering list of technology we could look at.

If asynchronous communication doesn't map to your mental models of computing, adopting an asynchronous style of communication will be challenging at first. And as we'll explore further when we look at detail at the various styles of asynchronous communication, there are a lot of different, interesting ways in which you can get yourself into a **lot** of trouble.

Async/await, and When Asynchronous Is Still Blocking

As with many areas of computing, we can use the same term in different contexts to have very different meaning. A style of programming, which appears to be especially popular in JavaScript, is the use of constructs like async/await to work with a potentially asynchronous source of data, but work with it in a blocking, synchronous style.

In Example 3-1 we see a very simple example of this in action. The currency exchange rates fluctuate frequently through the day, and we recieve these via a message broker. We define a `Promise`. Generically, a promise is something that will resolve to a state at some point in the future. In our case, our `eurToGbp` will eventually resolve to being the next Euro to GBP exchange rate.

Example 3-1. An example of working with a potentially asyncrhnous call in a blocking, sychronous fashion.

```
async function f() {

  let eurToGbp = new Promise((resolve, reject) => {
    //code to fetch latest exchange rate between USD and GBP
    ...
  });

  var latestRate = await usdToGbp; 1
  process(latestRate);2
}
```

❶ Wait until the latest USD to GBP exchange rate is fetched

❷ Won't run until the promise is fulfilled

When we reference `eurToGbp` using `await`, we block until `latestOrder`'s state is fulfilled - `process` isn't reached until we resolve the state of `eurToGbp` [2].

Even though our exchange rates are being received in an asynchronous fashion, from the use of `await` in this context means we are *blocking* until the state of `latestOrder` is resolved. So even if the underlying technology we are using to get the order status could be considered to be asynchronous in nature (for example waiting for the order stat), from the point of our code, this is inherently a synchronous, blocking interaction.

Where To Use It

Ultimately, when considering if asynchronous communication is right for you, you also have to consider which *type* of asynchronous communication you want to pick, as each as it's own tradeoffs. In general though, there are some specific use cases that would have me reaching for some form of asynchronous communication. Long running processes are an obvious candidate, as we explored in Figure 3-5 above. Also, situations where you have long call chains you can't easily restructure could be a good candidate. We'll dive deeper into this though when we look at three of the most common forms of asynchronous communication - request-response calls, event-driven communication, and communication through common data.

Pattern: Communication Through Common Data

A style of communication which spans a multitude of implementations is communication through common data. This pattern is used in a situation where one microservice puts data into a defined location, and another microservice (or potentially multiple) then make use of this data. It can be as simple as one microservice dropping a file in a location, and at some point later on another microservice picking that file up and doing something with it. This integration style is fundamentally asynchronous in nature.

An example of this is shown in Figure 3-6, where the `New Product Importer` creates a file that is then read by the downstream `Inventory` and `Catalog` microservices.

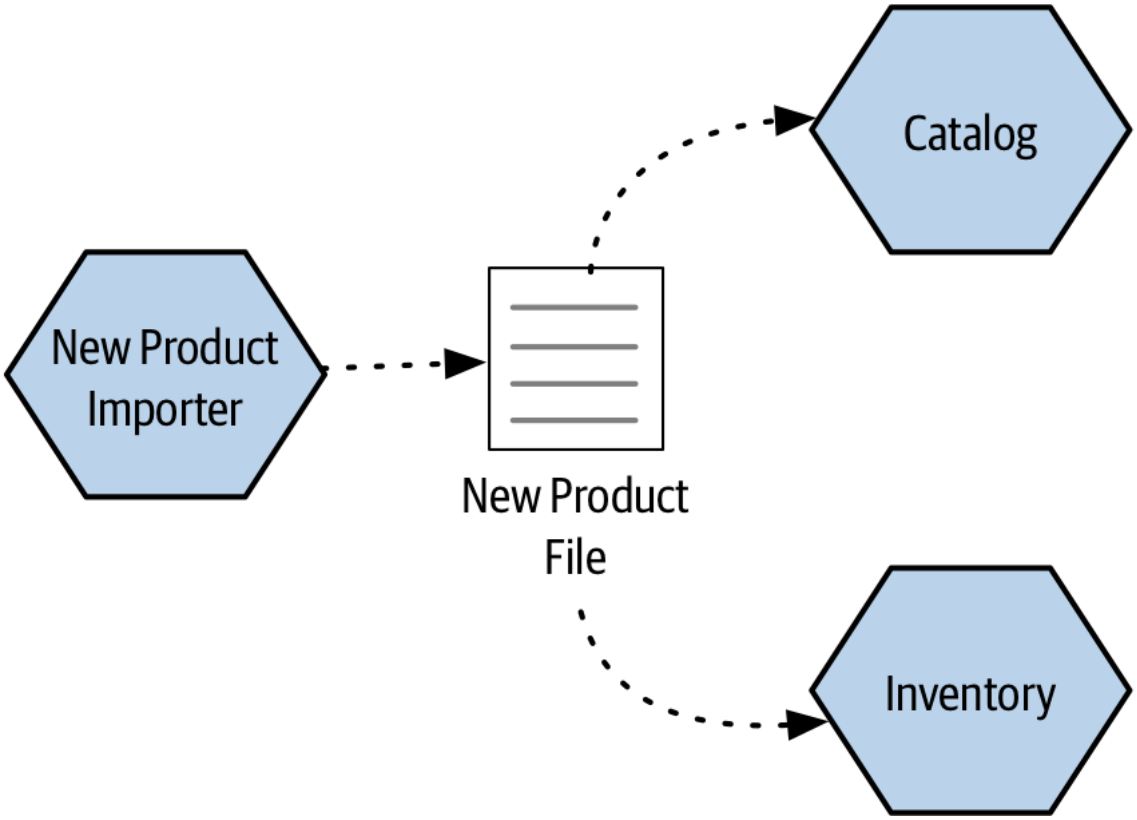


Figure 3-6. One microservice writes out a file which other microservices make use of

This pattern is in some ways the most common general inter-process communication pattern that you'll see, and yet we sometimes fail to see it as a communication pattern at all - largely I think because the communication between processes is often so indirect as to be hard to spot.

Implementation

To implement this pattern, you need some sort of persistent store for the data. A file system in many cases can be enough. I've built many systems which just periodically scan a file system, note the presence of a new file, and react on it accordingly. You could also use some sort of robust distributed memory store as well of course. It's worth noting that any downstream microservice which is going to act on this data will need it's own mechanism to identify that new data is available - polling is a frequent solution to this problem.

Two common examples of this pattern are the data lake and the data warehouse. In both cases, these solutions are typically designed to help processing large volumes of data, but arguably they exist at opposite ends of the spectrum regarding coupling. With data lake, sources upload raw data in whatever format they see fit, and downstream consumers of this raw data are expected to know how to process that information. With a data warehouse, the warehouse itself is a structured data store. Microservices pushing data to the data warehouse need to know the structure of the data warehouse - if the structure changes in a backwards compatible way, then these producers will need to be updated.

With both the data warehouse or data lake, the assumption is that the flow of information is in a single direction. One microservice publishes data to the common data store, and downstream consumers read that data and carry out appropriate actions. This unidirectional flow can make it easier to reason about the flow of information. A more problematic implementation can be the use of a shared database where multiple microservices both read and write to the same data store, an example of which we discussed in Chapter 2 when we explored common coupling - Figure 3-7 shows both the `Order Processor` and `Warehouse` updating the same record.

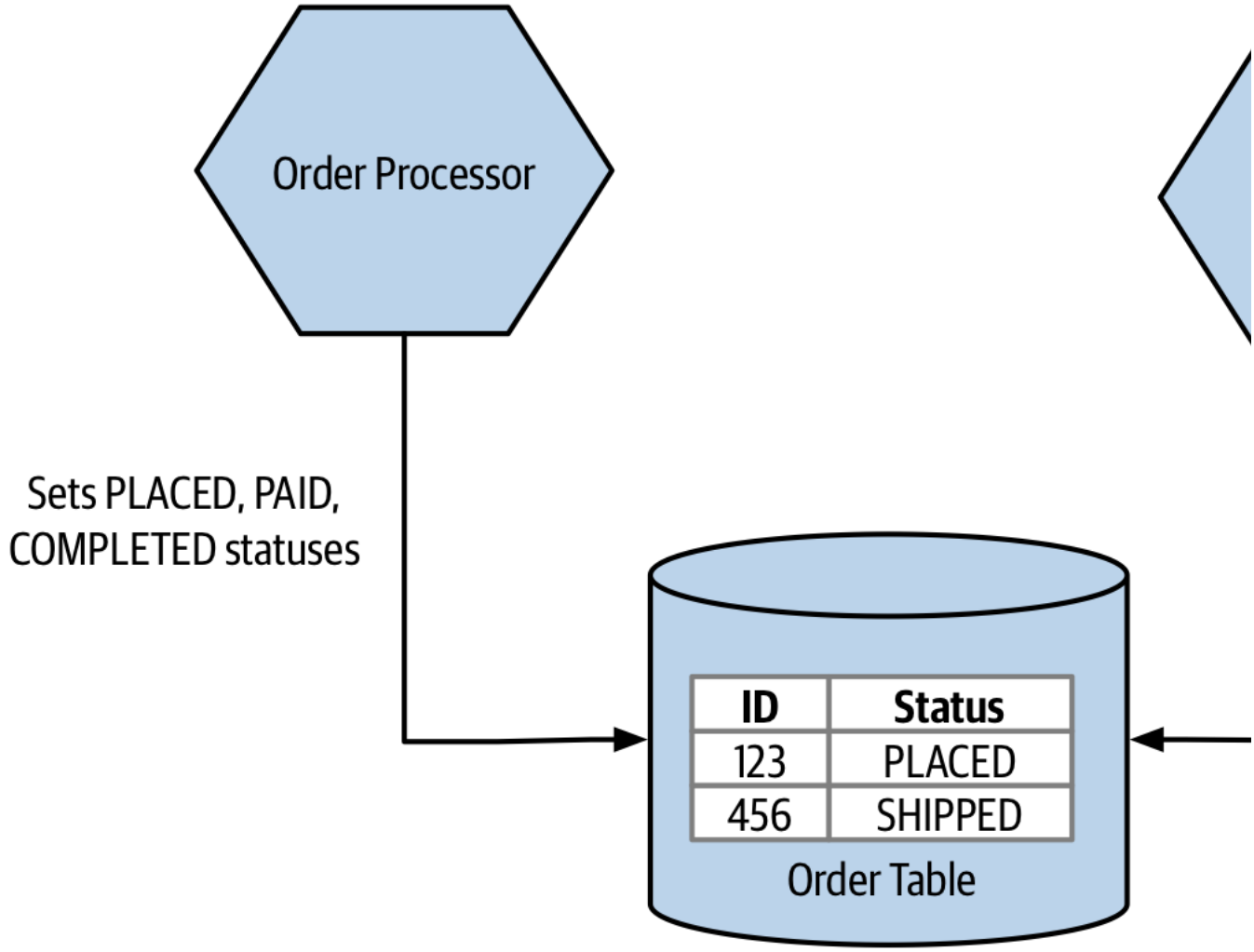


Figure 3-7. An example of common coupling where both Order Processor and Warehouse are updating the same order record

Advantages

This pattern can be implemented very simply, using commonly understood technology. If you can read or write to a file, or read and write to a database, you can use this pattern. The use of prevalent and well understood technology also enables interoperability between different types of systems, including older mainframe applications or customizable of the shelf software (COTS) products. Data volumes are also less of a concern here - if you're sending lots of data in one big go, this pattern can work well.

Disadvantages

Downstream consuming microservices will typically be aware that there is new data to process via some sort of polling mechanism, or else perhaps through a periodically triggered timed job. That means that this mechanism is unlikely to be useful in low-latency situations. You can of course combine this pattern with some other sort of call, informing a downstream microservice that new data is available. For example I could write a file to a shared filesystem, then send a call to the interested microservice informing it that there is new data that it may want. This can close the gap between data being published and data being processed. In general though, if you're using this pattern for very large volumes of data, it's less likely that low latency is high on your list of requirements. If you are interested in sending larger volumes of data and have them processed more in "real time", then using some sort of streaming technology like Kafka would be a better fit.

Another big disadvantage, and something that should be fairly obvious if you remember back to our exploration of common coupling in Figure 3-7, is that the common data store becomes a potential source of coupling. If that data store changes structure in some way, it can break communication between microservices.

The robustness of the communication will also come down to the robustness of the underlying data store. This isn't a disadvantage strictly speaking, but something to be aware of. If you're dropping a file on a file system, you might want to make sure that the filesystem itself isn't going to fail in interesting ways.

Where To Use It

Where this pattern really shines is in enabling interoperability between processes which might have restrictions in what technology they can use. Having an existing system talk to your microservice's GRPC interface or subscribe to its Kafka topic might well be more convenient from the point of view of the microservice, but not from the point of view of a consumer. Older systems may have limitations on what technology they can support, and may have high costs of change. Even old mainframe systems should be able to read data out of a file on the other hand. This does of course all depend on using data store technology which is widely supported - I could also implement this pattern using something like a redis cache. But can your old mainframe system talk to redis?

Another major sweet spot for this pattern is when sharing large volumes of data. If you need to send a multi gigabyte file onto a file system, or load in a few million rows into a database, then this pattern is the way to go.

Pattern: Request-Response Communication

With request-response, a microservice sends a request to a downstream service asking it to do something, and expects to receive a response with the result of the request. This interaction can be undertaken via a synchronous blocking call, or could be implemented in an asynchronous non-blocking fashion. A simple example of this interaction is shown in Figure 3-8, where the `Chart` microservice, which collates the best selling CDs for different genres, sends a request to the `Inventory` service asking for the current stock levels for some CDs.

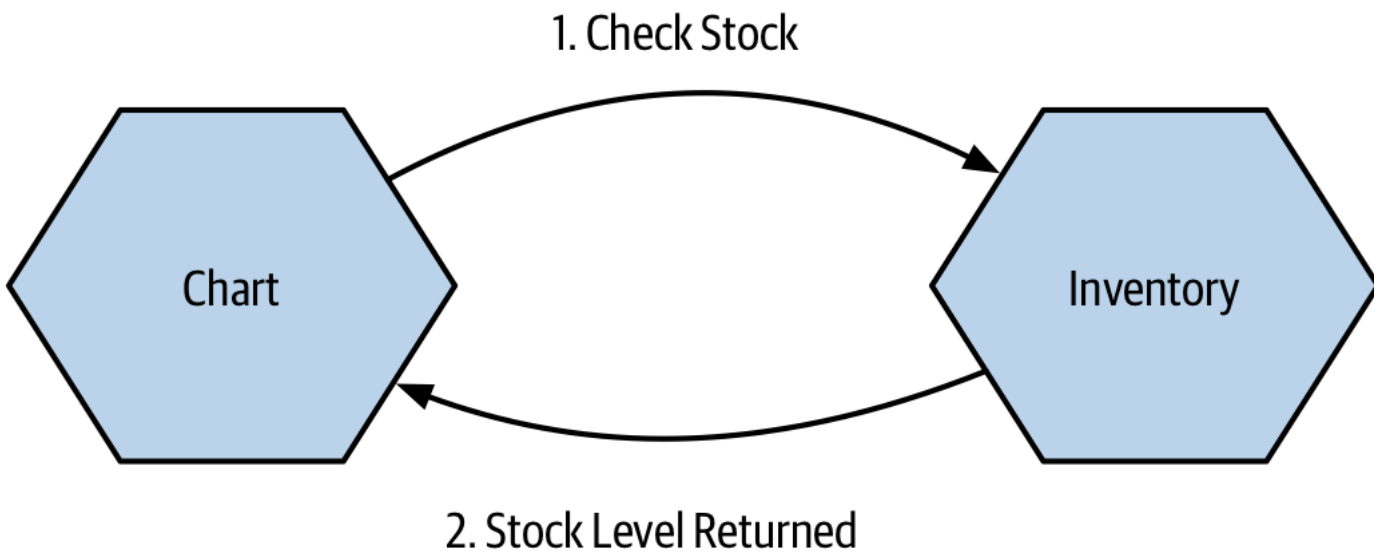


Figure 3-8. The Chart microservice sends a request to Inventory asking for stock levels

Retrieving data from other microservices like this is a common use case for a request-response call. Sometimes though, you just need to make sure something gets done. In Figure 3-9, the `Warehouse` microservice is sent a request from `Order Processor`, asking it to reserve stock. The `Order Processor` just needs to know that stock has been successfully reserved if it wants to carry on with taking payment. If the stock can't be reserved - perhaps because an item is no longer available - then the payment can be cancelled. Using request-response calls in situations where calls need to be completed in a certain order like this is common place.

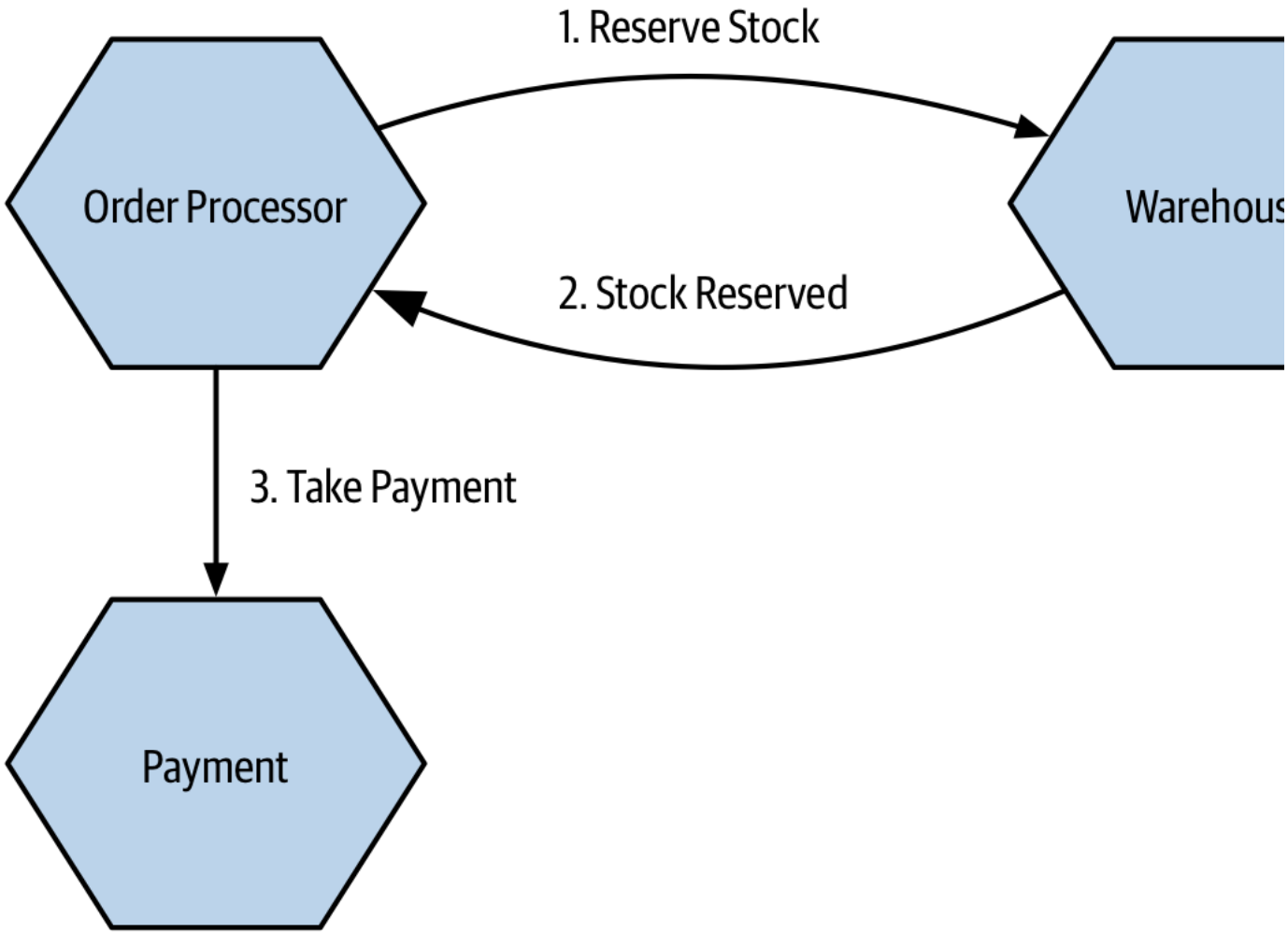


Figure 3-9. Order Processor needs to ensure stock can be reserved before payment can be taken

Commands vs Requests

I've heard some people talk about sending commands, rather than requests, specifically in the context of asynchronous request-response communication. The intent behind the term command is arguably the same as that of request - namely an upstream microservice is asking a downstream microservice to do something.

Personally speaking though, I much prefer the term request. Command implies a directive that must be obeyed, and it can lead to the situation where people feel that a command has to be acted on. A request implies something that can be rejected. It is right that a microservice examines each request on its merits, and based on it's own internal logic decides if the request should be auctioned. If the request it has been sent violates internal logic, it should be rejected. Although it's a subtle nuance, I don't feel that the term command conveys the same meaning.

Although I'll stick to using request over command, whatever term you decide to use, just remember that a microservice gets to reject the request/command if appropriate.

Implementation: Synchronous vs Asynchronous

Request-response calls like this can be implemented in either a blocking synchronous, or non-blocking asynchronous style. With a synchronous call, what you'd typically see is a network connection being opened with the downstream microservice, with the request being sent along this connection. The connection is kept open, waiting for the downstream microservice to respond. In this case, the microservice sending the response doesn't really need to know anything about the microservice that sent the request - it's just sending stuff back over an inbound connection.

With a asynchronous request response, things are less straight forward. Let's revisit the process associated with reserving stock. In Figure 3-10 the request to reserve stock is sent as a message over some sort of message broker (we'll explore message brokers later in this chapter). Rather than the message going directly to the `Inventory` microservice from `Order Processor`, it instead sits in a queue. The `Inventory` consumes messages from this queue when it is able. It reads the request, carries out the associated work of reserving the stock, and now it needs to send the response back to a queue that the `Order Processor` is reading from. The `Inventory` microservice needs to know where to route the response. In our example, it sends this response back over another queue which is in turn consumed by `Order Processor`.

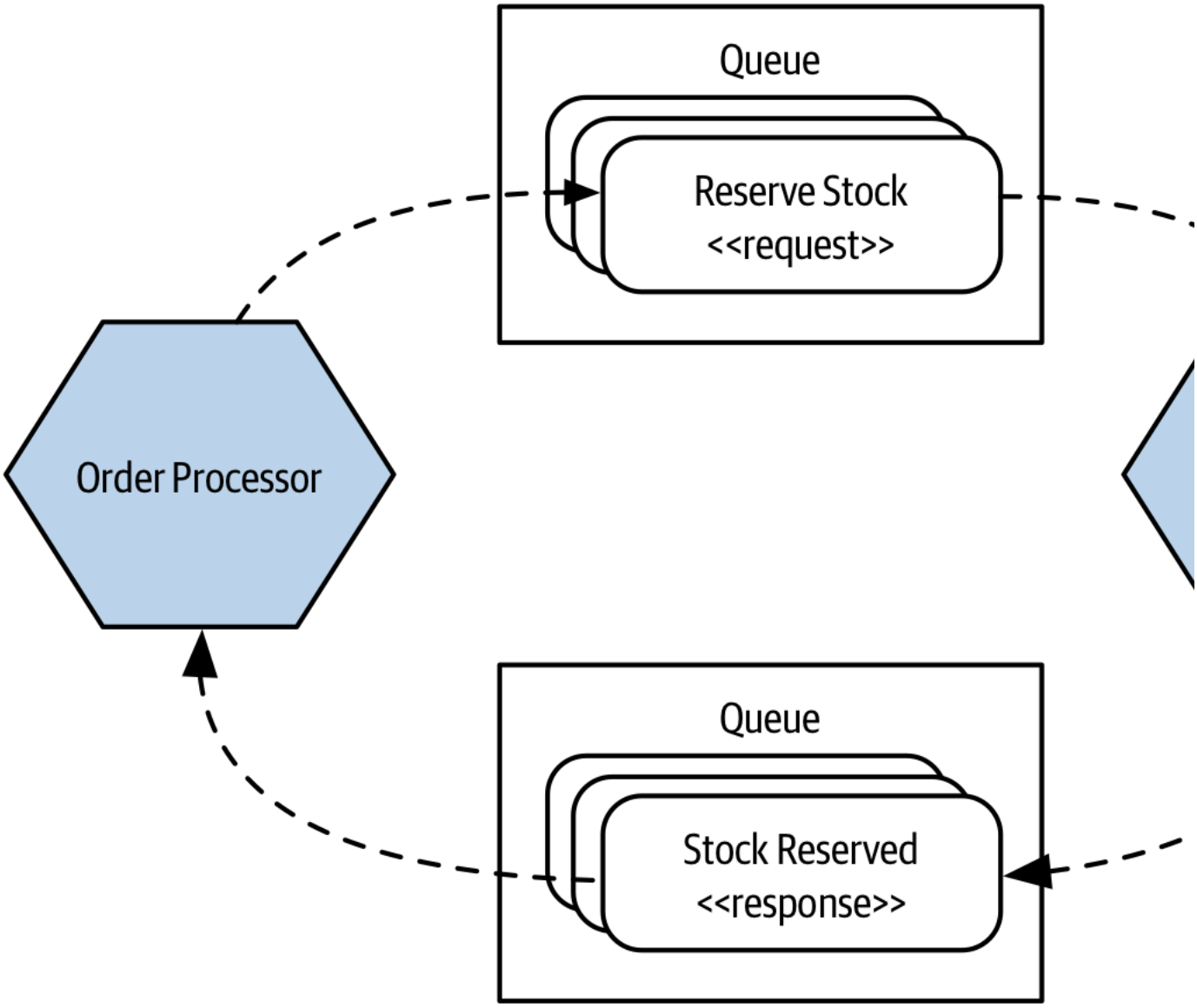


Figure 3-10. Using a queue to send stock reservation requests

So with a non-blocking asynchronous interaction, the microservice that receives the request either needs to implicitly know where to route the response, or else be told where the response should go. When using a queue, we have the added benefit that multiple requests could be buffered up in the queue waiting to be handled. This can help in situations where the requests can't be handled quickly enough. The microservice can consume the next request when it is ready, rather than being overwhelmed by too many calls. A lot of course then depends on the queue absorbing these requests.

When a microservice receives a response in this way, it might need to relate the response to the original request. This can be challenging as a lot of time may have passed, and depending on the nature of the protocol being used, the response may not come back to the same instance of the microservice that sent the request. In our example of reserving stock as part of placing an order, we'd need to know how to associate the stock reserved response with a given order, so we can carry on processing that particular order. An easy way to handle this would be to store any state associated with the original request into a database, such that when the response comes in, the receiving instance can reload any associated state and act accordingly.

Parallel vs Sequential Calls

When working with request-response interactions, you'll often encounter a situation where you need to make multiple calls before you can continue with some processing.

Consider a situation where MusicCorp needs to check on the price for a given item from three different stockists, which we do by issuing API calls. We want to get the prices back from all three stockists before deciding which one we want to order new stock from. We could decide to make the three calls in sequence - waiting

for each one to finish, before proceeding with the next. In such a situation, we'd be waiting for the sum of latencies of each of the calls. If the API call to each provider took 1 second to return, we'd be waiting 3 seconds before we can decide who we should order from.

A better option would be to run these three requests in parallel - then the overall latency of the operation would be based on the slowest API call, rather than the sum of latencies of each API call.

Reactive extensions, and mechanisms like async/await can be very useful to help run calls in parallel, and this can result in significant improvements in the latency of some operations.

Where To Use It

Request-response calls make perfect sense for any situation where the result of a request is needed before further processing can take place. It also fits really well in situations where a microservice wants to know if a call didn't work, so that it can carry out some sort of compensating action, like a retry. If that fits your situation, request-response is a sensible approach - the only remaining question then is to decide on a synchronous vs asynchronous implementation, with the same tradeoffs we discussed earlier.

Pattern: Event-Driven Communication

Event-driven communication looks quite odd compared to request-response calls. Rather than a microservice asking some other microservice to do something, instead a microservice emits events which may or may not be received by other microservices. It is an inherently asynchronous interaction, as the event listeners will be running on their own thread of execution.

An event is a statement about something that has occurred, nearly always something that has happened inside the world of the microservice that is emitting the event. The microservice emitting the event has no knowledge of the intent of other microservices to use the event, and indeed may not even be aware that any other microservice exists. It emits the event when required, and that is the end of it's responsibilities.

In Figure 3-11, we see the `Warehouse` emitting events related to the process of packaging up of an order. These events are received by two microservices, `Notifications` and `Inventory`, and they react accordingly. The `Notifications` microservice sends an email to update our customer about changes in order status, where the `Inventory` microservice can update stock levels as items are packaged into customer orders.

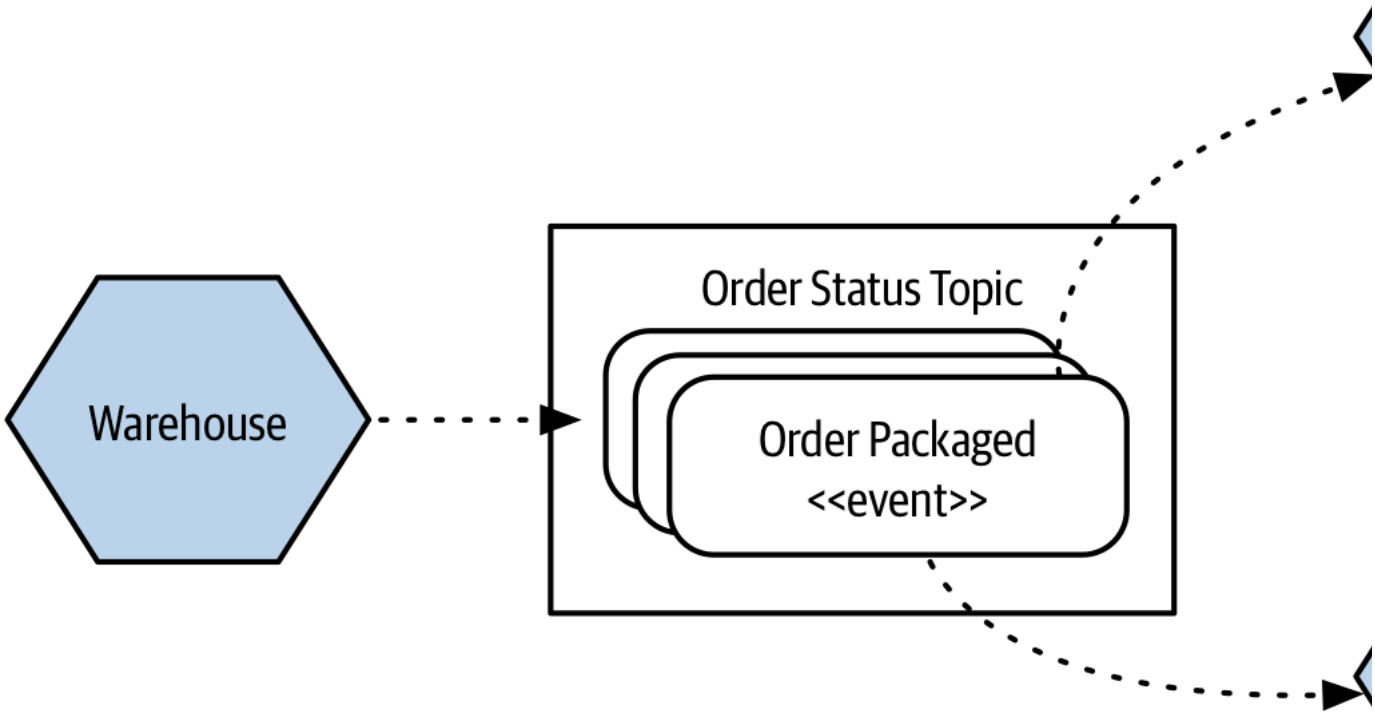


Figure 3-11. The Warehouse emits events which some downstream microservices care about

This is an inversion of responsibilities, when compared to a request-response model. With events, the `Warehouse` is just broadcasting events, assuming that interested parties will react accordingly. It is unaware of who the recipients of the events are, making event-driven interactions much more loosely coupled in general. When compared to a request-response call though, this is an inversion of responsibility that it can take a while to get your head around. With request-response, I might instead expect `Warehouse` to tell the `Notifications` microservice to send emails when appropriate. In such a model, the `Warehouse` would need to know what events require notifying a customer about. With an event-driven interaction, we are instead pushing that responsibility into the `Notifications` microservice.

This distribution of responsibility we see with our event-driven interactions can mirror the same distribution of responsibility we see with organizations trying to create more autonomous teams. Rather than holding all the responsibility centrally, instead we want to push it into the teams themselves to allow them to operate in a more autonomous fashion - a concept we will revisit in [Link to Come]. Here, we are pushing responsibility from `Warehouse` into `Notifications` and `Payment` - this can help us reduce the complexity of microservices like `Warehouse`, and lead to a more even distribution of "smarts" in our system. We'll explore that idea in more detail when we compare choreography and orchestration later.

Events & Messages

On occasion I've seen the term messages and events get confused. An event is a fact - a statement that something happened, along with some information about exactly what happened. A message is a thing we send over an asynchronous communication mechanism, like a message broker.

With event-driven collaboration, we want to broadcast that event, and a typical way to implement that broadcast mechanism would be to put that event into a message. The message is the medium, the event is the payload.

Likewise, we might want to send a request as the payload of a message - in which case we would be implementing a form of asynchronous request-response.

Implementation

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 ages 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 (we'll come back to that later in [Link to Come]). 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.

What's In An Event?

In Figure 3-12, we see an event being broadcast from the `Customer` microservice, informing interested parties that a new customer has registered with the system. Two of the downstream microservices, `Loyalty` and `Notifications` care about this event. The `Loyalty` microservice reacts to receiving the event by setting up an account for the new customer so that they can start earning points, whereas the `Notifications` microservice sends an email to the newly registered customer welcoming them to the wondrous delights of MusicCorp.

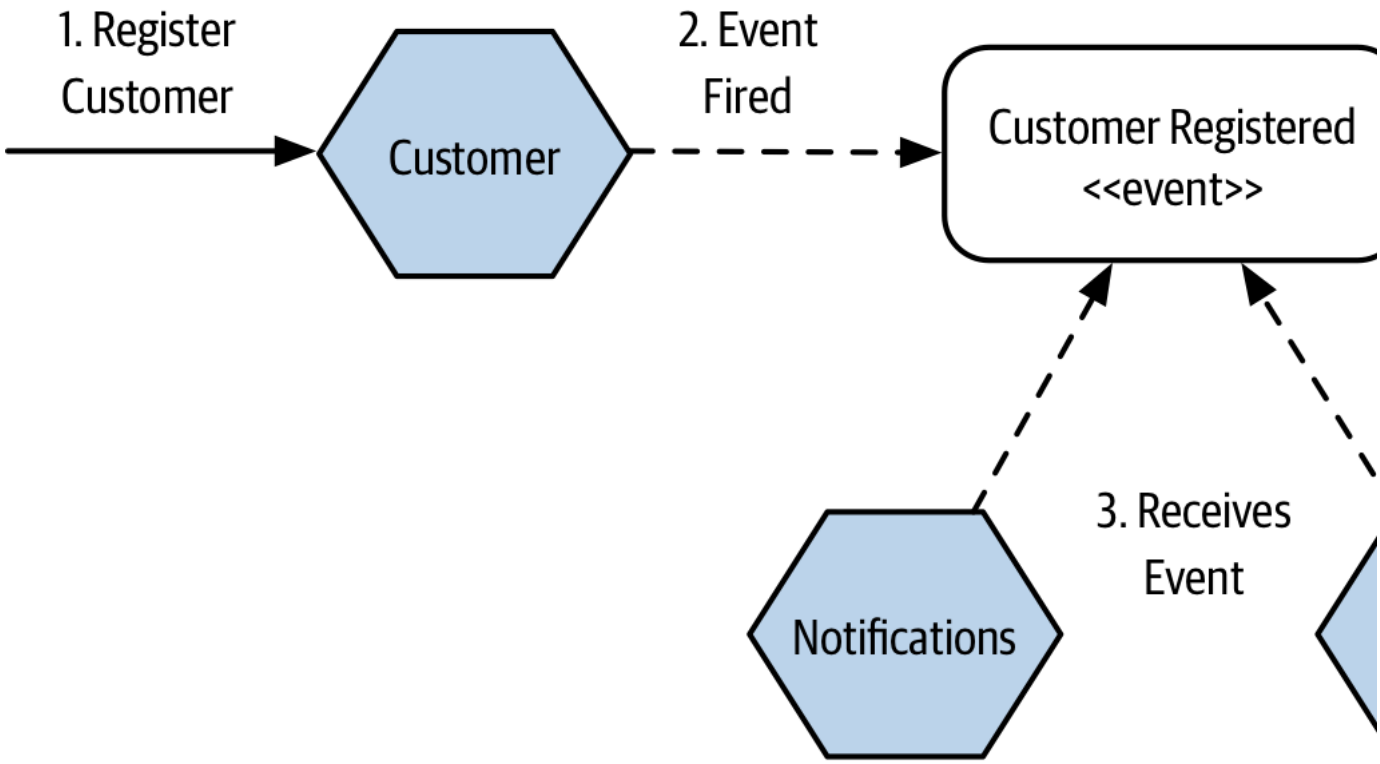


Figure 3-12. Notifications and Loyalty microservices receive an event when a new customer is registered.

With a request, we are asking a microservice to do something, and providing the required information for the requested operation to be carried out. With an event we are broadcasting a fact that other parties **might** be interested in, but as the microservice emitting an event can't and shouldn't know who receives the event, how do we know what information other parties might need from the event? So what, exactly, should be inside the event?

Just An ID

One option, is for the event to just contain an identifier for the newly registered customer, as shown in Figure 3-13. The `Loyalty` microservice only needs this identifier to create the matching loyalty account, so it has all the information it needs. However, while the `Notifications` microservice knows that it needs to send a welcome email when this type of event is received, it will need additional information to do its job - at least an email address, and probably the name of the customer as well to give the email that personal touch. As this information isn't in the event that the `Notifications` microservice receives then it has no choice but to fetch this information from the `Customer` microservice, something we see in Figure 3-13.

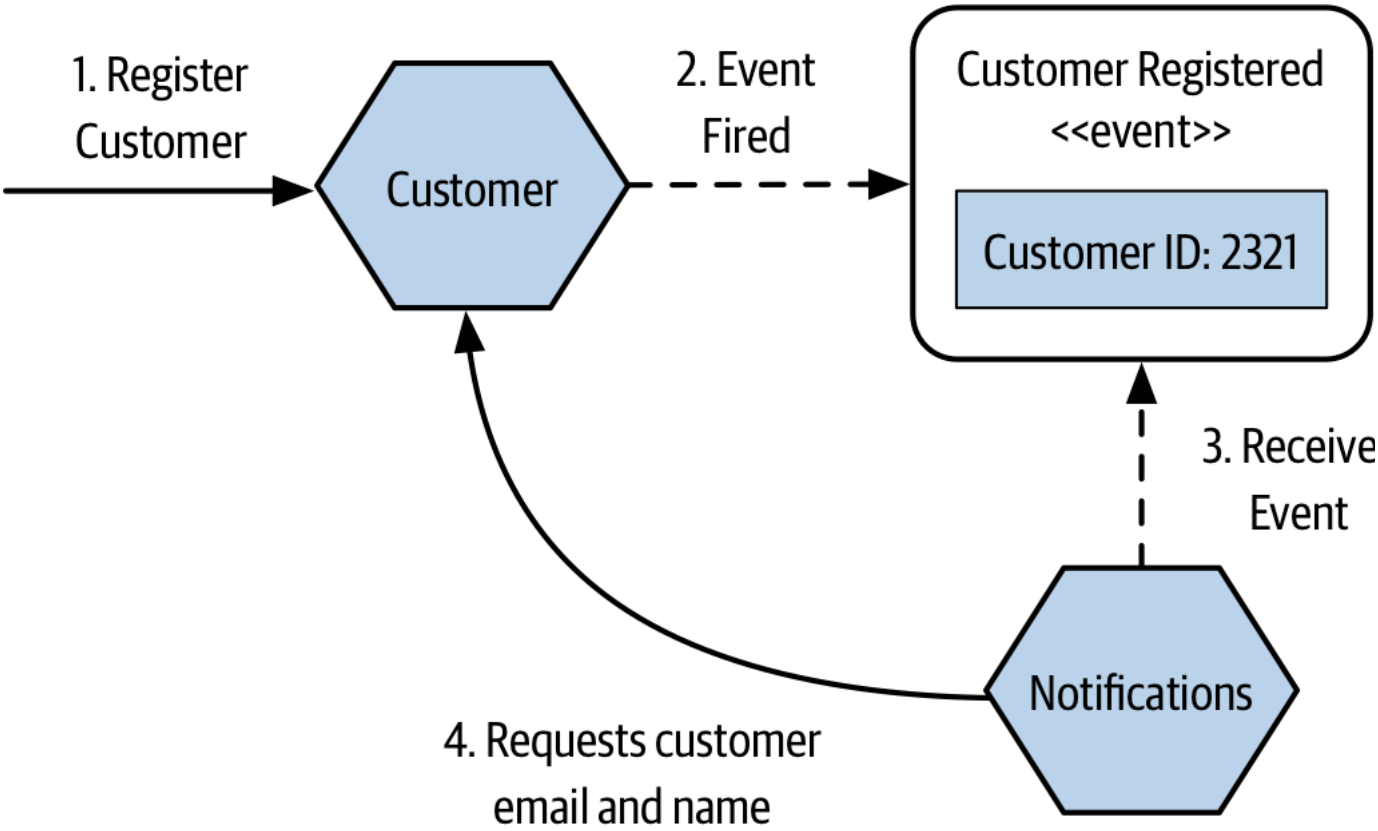


Figure 3-13. The Notification microservice needs to request further details from the Customer microservice as they aren't in the event

There are some downsides with this approach. Firstly, the `Notification` microservice now has to know about the `Customer` microservice, adding additional domain coupling. While domain coupling, as we discussed in Chapter 2, is on the looser end of the coupling spectrum, we'd still like to avoid it where possible. If the event that `Notification` received contained all the information it needed, then this call back wouldn't be required. This call back from the receiving microservice can also lead to the other major downside - namely that in a situation with a large number of receiving microservices, the microservice emitting the event might get a barrage of requests as a result. Imagine if five different microservices all received the same customer creation event, and all needed to request additional information - they'd all need to immediately send a request to the `Customer` microservice to get what they needed. As the number of microservices interested in a particular event increases, the impact of these calls could become significant.

Fully Detailed Events

The alternative, which I prefer, is to put everything into an event that you would be happy otherwise sharing via an API. If you'd let the `Notifications` microservice ask for the email address and name of a given customer, why not just put that in the event in the first place? In Figure 3-14, we see this approach - `Notification` is now more self-sufficient, and able to do its job without needing to communicate with the `Customer` microservice. In fact, it might never need to know the `Customer` microservice even exists.

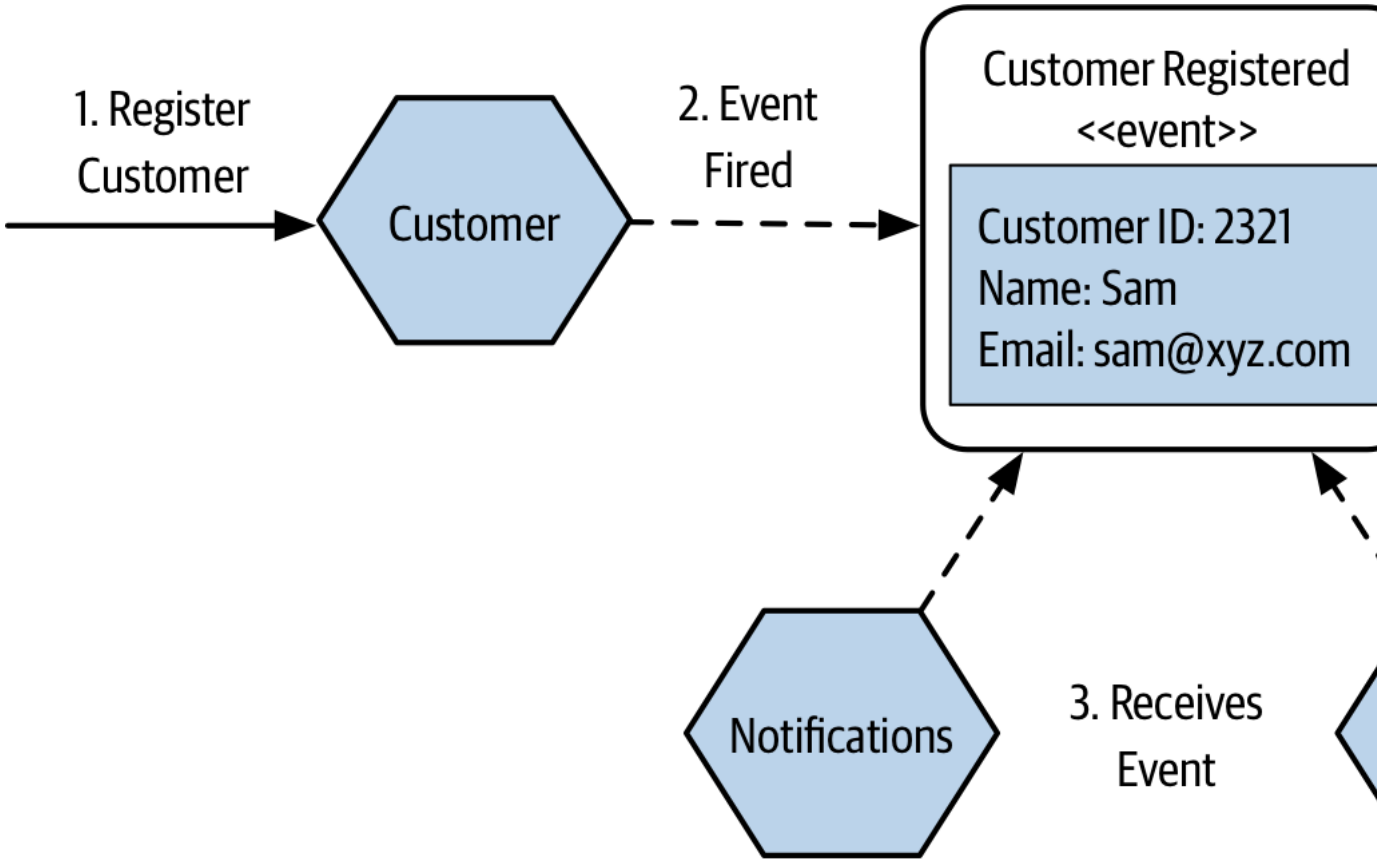


Figure 3-14. An event with more information in it can allow receiving microservices to act without requiring further calls to the source of the events.

In addition to the fact that events with more information can allow for looser coupling, events with more information can double up as an historical record as to what happened to a given entity. This could help you as part of implementing an auditing system, or perhaps even provide the ability to reconstitute an entity at given points of time - meaning that these events could be used as part of an event sourcing, a concept we'll explore briefly in a moment.

Whilst this approach is definitely my preference, it's not without some downsides. Firstly, if the data associated with an event is large, we might have concerns about the size of the event. Now, modern message brokers (assuming you're using one to implement your event broadcast mechanism) have fairly generous limits for message size. The default maximum size for a message in Kafka is 1MB, and the latest release of RabbitMQ has a theoretical upper limit of 512MB for a single message (down from the previous limit of 2GB!), even though one could expect there to be some interesting performance issues with large messages like this. But even the 1MB afforded to us as the maximum size of a message on Kafka gives us a lot of scope to send quite a bit of data. Ultimately, if you're venturing into a space where you are starting to worry about the size of your events, then a hybrid approach where some information is in the event but other (larger) data can be looked up if required.

In Figure 3-14, `Loyalty` doesn't need to know the email address or name of the customer, and yet because it is being sent this information via the event it nonetheless receives it. This could lead to concerns if we are trying to limit the scope of which microservices can see what kind of data - for example I might want to limit what microservices can see personally identifiable information (or PII), payment card details, or similar sensitive data. A way to solve this could be to implement something like Split Horizon Communication, which we'll explore later in [Link to Come].

Another consideration is that once we put data into an event, it becomes part of our contract with the outside world. We have to be aware that if we remove a field from an event that we may break external parties. Information hiding is still an important concept in event-driven collaboration - the more data we put into an event, the more assumptions external parties will have about an event. My general rule is that I am OK putting information into an event if I'd be happy sharing the same data over a request-response API.

Did It Work?

TODO: Move to workflow discussion?

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 [Link to Come].

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

Summary

In this chapter, we broke down some of the key styles of microservice communication, and discussed the various tradeoffs. There isn't always a single **right** option, but hopefully I've detailed enough information regarding synchronous and asynchronous calls, event-driven and request-response styles of communication, to help you make the right call for your given context.

Where this chapter focused primarily on how one microservice talks to another, in our next chapter we look beyond that to how we can get multiple microservices collaborating to implement workflows.

[1] True story

[2] Please note, this is very simplified - I've completely omitted error handling code for example. If you want to know more about async/await, specifically in JavaScript, the The Modern JavaScript Tutorial is a great place to start: *https://javascript.info/*

Chapter 4. Implementing Microservice Communication

Work In Progress

Please note that the text below is currently being reworked for the 2nd edition of the book, and is not in a complete state. This will be Chapter 4 of the final book.

If you have any feedback on the book, or suggestions for the 2nd edition, then please contact me on book-feedback@samnewman.io and/or complete a short survey here: *https://oreil.ly/Bldg_MicroServices_survey*.

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? GRPC?

Well, as we discussed in the previous chapter, your choice of technology should be driven in large part based on the style of communication you want. Deciding between blocking synchronous or non-blocking asynchronous calls, request-response or event-driven collaboration, will help you whittle down what might otherwise be a very long list of technology.

In this chapter, we're going to now look at some of the common technology used for microservice communication. But new options are always coming up, so before we discuss specific technology, let's think about what we want out of whatever technology we pick.

Make Backwards Compatibility Easy

When making changes to our microservices, we need to make sure we don't break compatibility with any consuming microservices. As such, we want to ensure that whatever technology we pick makes it easy to make backwards compatible changes. Simple operations like adding new fields shouldn't break clients. We also ideally want the ability to validate that the changes we have made are backwards compatible - and have a way to get that feedback before we deploy our microservice into production.

Make Your Interface Explicit

It is important that the interface that a microservice exposes to the outside world is explicit. This means that it is clear to a consumer of a microservice as to what functionality that microservice exposes. But it also means that it is clear to a developer working on a microservice as to what functionality needs to remain intact for external parties - we want to avoid a situation where a change to a microservice causes an accidental breakage in compatibility.

Schemas can go a long way to helping ensure that the interface a microservice exposes is explicit. Some of the technology we can look at requires the use of a schema, for others the use of a schema is optional. Either way, I strongly encourage the use of a schema, as well as enough supporting documentation to be clear about what functionality a consumer can expect a microservice to provide.

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 microservice. 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—exactly what 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.

There is a whole host of technology we could look at, but rather than looking broadly at a long list of options in this space, I will highlight some of the most popular and interesting choices. Here are the options we'll be looking at:

Remote Procedure Calls (RPC)

: Frameworks that allow for local method calls to be invoked on a remote process. Common options include SOAP and GRPC.

REST

: An architectural style where you expose resources (Customer, Order etc) that can be accessed using a common set of verbs (GET, POST). There is a bit more to REST than that, but we'll get to that shortly.

GraphQL

: A relatively new protocol that allows for consumers to define custom queries that can fetch information from multiple downstream microservices, filtering the results to return only what is needed.

Message Brokers

: Middleware that allows for asynchronous communication either via queues or topics.

TODO: Show this tech against styles we outlined previously?

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. Most of the technology in this space requires an explicit schema, such as SOAP or GRPC. The use of a separate schema makes 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 remote call look like a local call.

Typically, using an RPC technology means you are buying into a serialization protocol. The RPC framework defines how data is serialized and deserialized. GRPC for example uses the protocol buffer serialization format for this purpose. 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.

RPC frameworks that have an explicit schema make it very easy to generate client code. This can avoid the need for client libraries, as any client can just generate their own code against this service specification. For client side code generation to work though, the client needs some way to get the schema out of band - in other words the consumer needs to have access to the schema before it plans to make calls. AVRO RPC is an interesting outlier here, as it has the option to send the full schema along with the payload, allowing for clients to dynamically interpret the schema.

The ease of generation of client-side code is 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.

Challenges

As we've seen, RPC offers some great advantages, but it's not without it's downsides - and some RPC implementations can be more problematic than others. Many of these issues can be dealt with, but they deserve further exploration.

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.

To be fair, there are a number of RPC implementations that don't have this restriction - GRPC, SOAP and Thrift are all examples that allow for interoperability between different technology stacks.

Local Calls Are Not Like Remote Calls

The core idea of RPC is to hide the complexity of a remote call. This can though lead to hiding 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 un-marshalling 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 [Link to Come].

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 4-1 declares the methods we are going to expose remotely. Java RMI then generates the client and server stubs for our method.

Example 4-1. 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, `createCustomer` 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.

Where To Use It

Despite its shortcomings, I actually quite like RPC, and the more modern implementations, such as GRPC, are excellent, whereas other implementations have significant issues which would cause me to give them a wide berth. Java RMI for example has a number of issues regarding brittleness and limited technology choices, and SOAP is pretty heavyweight from a developer perspective, especially when compared with more modern choices.

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.

If I was looking at options in this space, GRPC would be top of my list. Built to take advantage of HTTP/2, it has some impressive performance characteristics and good general ease of use. I also appreciate the ecosystem around GRPC, including tools like Protolock[1], something we'll discuss later in this chapter when we discuss schemas.

GRPC fits a synchronous request-response model well, but can also work in conjunction with reactive extensions. It's high on my list whenever I'm in situations where I have a good deal of control over both the client and server ends of the spectrum. If you're having to support a wide variety of other applications that might need to talk to your microservices, the need to compile client-side code against a server-side schema can be problematic. In which case, some form of REST over HTTP API would likely be a better fit.

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 when thinking about REST 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 completely 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 [Link 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. GRPC on the other hand has been designed to take advantage of the capabilities of HTTP/2 such as the ability to send multiple request-response streams over a single connection.

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 how 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 4-2. Along with information about the album, we see a number of hypermedia controls.

Example 4-2. Hypermedia controls used on an album listing

```
<album>
  <name>Give Blood</name>
  <link rel="/artist" href="/artist/theBrakes" /> 1
  <description>
    Awesome, short, brutish, funny and loud. Must buy!
  </description>
  <link rel="/instantpurchase" href="/instantPurchase/1234" /> 2
</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.

The theory is that by using these controls to decouple the client and server we gain significant benefits over time that hopefully offset the increase in the time it takes to get these protocols up and running. Unfortunately, although these ideas all seem sensible in theory, I've found that this form of REST is rarely practiced, for reasons I've not entirely got to grips with. This makes HATEOS specifically a much harder concept for me to promote for those already committed to the use of REST. Fundamentally, many of the ideas in REST are predicated on creating distributed hypermedia systems, and this isn't what most people end up building.

Challenges

In terms of ease of consumption, historically you wouldn't be able to generate client-side code for your REST over HTTP application protocol like you can with RPC implementations. This has often lead to people creating REST APIs providing client libraries for consumers to make use of. These client libraries give you a binding to the API to make client integration easier. The problem is that client libraries can cause some challenges with regards to coupling between the client and server, something we'll discuss in "DRY and the Perils of Code Reuse in a Microservice World".

In recent years this problem has been somewhat alleviated. The OpenAPI specification[2], that grew out of the Swagger documentation format, now provides you with the ability to define enough information on a REST endpoint to allow for the generation of client-side code in a variety of languages. In my experience, I haven't seen many teams actually making use of this functionality even if they were already using Swagger for documentation. I have a suspicion that this may be due to the difficulties of retrofitting its use into current APIs. I do also have concerns about a specification previously just being used for documentation now being used to define a more explicit contract. This can lead to a much more complex specification - comparing an OpenAPI schema with a protocol buffer schema for example is quite a stark contrast. Despite my reservations though, it's good that this option now exists.

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. All mainstream HTTP protocols in current use require the use of Transmission Control Protocol (TCP) under the hood, which has inefficiencies

compared with alternative networking protocols, and some RPC implementations can allow you to use alternative networking protocols to TCP such as User Datagram Protocol (UDP).

The limitations placed on HTTP due to the requirement to use TCP are being addressed. HTTP/3, which is currently in the process of being finalized, is looking to shift over to using the newer QUIC protocol. QUIC provides the same sorts of capabilities as TCP (such as improved guarantees over UDP) but has some significant improvements over TCP, which have been shown to deliver improvements in latency and reductions in bandwidth. It's likely that HTTP/3 will take several years before it has a widespread impact on the public internet, but it seems reasonable to assume that organizations can benefit earlier than this within their own networks.

With respect to HATEOS specifically, you can encounter additional performance issues. As clients need to navgiate multiple controls to find the right endpoints for a given operation, this can lead to very chatty protocols - multiple round trips may be required for each operation. Ultimately, this is a trade-off. If you decide to adopt a HATEOS-style of REST, 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.

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)[3], which covers the topic of REST over HTTP in depth.

Where To Use It

Due to its widespread use in the industry, a REST over HTTP based API is an obvious choice for a synchronous request-response interface if you are looking to allow access from as wide a variety of clients as possible. It would be a mistake to think of a REST over HTTP as just being a "good enough for most things" choice, but there is something to that. It's a widely understood style of interface, that most people are familiar with, and guarantees interoperability from a huge variety of technologies.

Due in large part to the capabilities of HTTP, and the extent to which REST builds upon these capabilities (rather than hiding them), these APIs excel in situations where you want large scale and effective caching of requests. It's for this reason that they are the obvious choice for exposing APIs to external parties or client interfaces. They may well suffer when compared to more efficient communication protocols, and although you can construct asynchronous interaction protocols over the top of REST-based APIs, that's not really a great fit compared to the alternatives for general microservice-to-microservice communication.

Despite intellectually appreciating the goals behind HATEOS, I haven't in my experience seen the additional work to implement this style of REST deliver worthwhile benefits in the long run, nor can I recall in the last few years talking to any teams implementing a microservice architecture that can speak to the value of using HATEOS. My own experiences are obviously only one set of data points, and I don't doubt that for some people it may have worked well. But this concept does not seem to have caught on as much as I thought it would. It could be that the concepts behind HATEOS are too alien for us to grasp, or it could be the lack of tools or standards in this space, or perhaps the model just doesn't work for the sorts of systems we have ended up building.

So for use at the perimeter, it works fantastically well, and for synchronous request-response based communication between microservices, it's great.

GraphQL

In recent years, GraphQL[4] has gained more popularity, due in large part to the fact that it excels in one specific area. Namely, it makes it possible for a client-side device to define queries that can avoid the need to make multiple requests to retrieve the same information. This can offer significant improvements in terms of the performance of constrained client-side devices, and also avoid the need to implement bespoke server-side aggregation.

TODO: Do I need a picture?

To take a simple example, imagine a mobile device that wants to display a page showing an overview of a customer's latest orders. The page needs to contain some information about the customer. along with information about the 5 most recent orders the client placed. The screen only needs a few fields from the customer record, and only needs the date, value and shipped status of each order. The mobile device could issue calls to two downstream microservices to retrieve the required information, but this would involve making multiple calls, including pulling back information that isn't actually required. Especially with mobile devices, this can be wasteful - it uses up more of a mobile device's data plan than is needed, and can take longer.

GraphQL allows for the mobile device to issue a single query that can pull back all the required information. For this to work, you need a microservice which exposes a GraphQL endpoint to the client device. This GraphQL endpoint is the entry for all client queries, and exposes a schema for the client devices to use. This schema exposes the types available to the client, and a nice graphical query builder is also available to make creating these queries easier. By reducing the amount of calls and amount of data retrieved by the client device, you can deal neatly with some of the challenges that occur when building user interfaces with microservice architectures.

Challenges

Early on, one challenge was lack of language support for the GraphQL specification, with JavaScript being your only choice initially. This has improved greatly, with all major technologies now having support for the specification. In fact across the board there have been significant improvements in GraphQL and the various implementations, making it a much less risky prospect than it might have been a few years ago. That said, a few challenges do remain with the technology which you might want to be aware of.

As the client device can issue dynamically changing queries, this can potentially cause an issue with server-side load. I've heard of teams who have had issues with GraphQL queries causing significant load on the server-side as a result of this. To compare GraphQL with something like SQL, we have the same issue there. An expensive SQL statement can cause significant problems for a database, potentially having a large impact on the wider system. The same problem applies with GraphQL. The difference is that at least with SQL we have tools like query planners for our databases, which can help us diagnose problematic queries, whereas a similar problem with GraphQL can be harder to track down. Server-side throttling of requests is one potential issue, but as the execution of the call may be spread across multiple microservices, this is far from straightforward.

Compared with normal REST-based HTTP APIs, caching is also more complex. With REST-based API, I can set one of many response headers to help client side devices, or intermediate caches like content delivery networks, cache responses so they don't need to be requested again. This isn't possible in the same way with GraphQL. The advice I've seen around this issue seems to revolve around just associating an ID with every returned resource (and remember, a GraphQL query could contain multiple resources), and then having the client device cache the request against that ID. As far as I can tell, this makes the use of Content Delivery Networks (CDNs) or caching reverse proxies incredibly difficult without additional work, or additional tooling.

Although I've seen some implementation-specific solutions to this problem (such as those found in the JavaScript Apollo implementation), caching feels like it was either consciously or unconsciously ignored as part of the initial development of GraphQL. If the queries you are issuing are highly specific in nature to a particular user, then this lack of request-level caching may not be a deal breaker of course, as your cache-hit ratio is likely to be low. I do wonder though if this limitation means that you'll still end up with a hybrid solution for client devices, with some (more generic) requests going over normal REST-based HTTP APIs, with other requests going over GraphQL.

Another issue, is that while GraphQL theoretically can handle writes, it doesn't seem to fit as well as reads. This does lead to situations where teams are using GraphQL for read, but REST for writes.

The last issue is something which may be entirely subjective, but I still think it's worth raising. GraphQL makes it feel like you are just working with data, which can reinforce the idea that the microservices you are talking to are in fact just wrappers over databases. I've seen multiple people in fact compare GraphQL with OData, a technology which is designed as a generic API for accessing data from databases. As we've already discussed at length, the idea of just treating microservices as wrappers over databases can be very problematic. Microservices expose functionality over networked interfaces. Some of that functionality might require or result in data being exposed, but they should still have their own internal logic and behavior. Just because you are using GraphQL, don't slip into thinking of your microservices as little more than an API on a database - it's essential that your GraphQL API isn't coupled to the underlying datastores of your microservices.

Where To Use It

GraphQL's sweet spot is for use at the perimeter of the system, exposing functionality to external clients. These clients are typically GUIs, and it's an obvious fit for mobile devices given their constraints in terms of their limited ability to surface data to the end user and nature of mobile networks. GraphQL has also seen use though for external APIs, GitHub being an early adopter of GraphQL. If you have an external API which often requires external clients to make multiple calls to get the information they need, then GraphQL can help make these APIs much more efficient and friendly.

TODO: Reference picture in intro for GraphQL above

Fundamentally, GraphQL is a call aggregation mechanism, so in the context of a microservice architecture it would be used to aggregate calls over multiple downstream microservices, as we saw in <<>>. As such, it's not something that would replace general microservice-to-microservice communication.

An alternative to the use of GraphQL would be to consider an alternative pattern like the Backend For Frontend (BFF) pattern - we'll look at that and compare with GraphQL and other aggregation techniques further in [Link to Come].

Message Brokers

Message brokers are intermediaries, often called middleware, that sit between processes to manage communication between them. They are a popular choice to help implement asynchronous communication between microservices as they offer a variety of powerful capabilities.

As we discussed earlier, a message is a generic concept which defines the thing that a message broker sends. A message could contain a request, a response, or an event. Rather than one microservice directly communicating with another microservice, instead, it gives a message to a message broker, with information about how the message should be sent.

Topics and Queues

Brokers tend to provide either queues, topics, or both. Queues are typically point to point. A sender puts a message on a queue, and a consumer reads from that queue. With a topic-based system, multiple consumers are able to subscribe to a topic, and each subscribed consumer will receive a copy of that message.

A consumer could represent one or more microservices - typically modelled as a consumer group. This would be useful when you have multiple instances of a microservice, and you want any one of them to be able to receive a message. In Figure 4-1, we see an example where the `Order Processor` has three deployed instances, all as part of the same consumer group. When a message is put into the queue, only one member of the consumer group will receive that message - this means the queue works as a load distribution mechanism - this is an example of the Competing Consumers pattern we touched on briefly in Chapter 3.

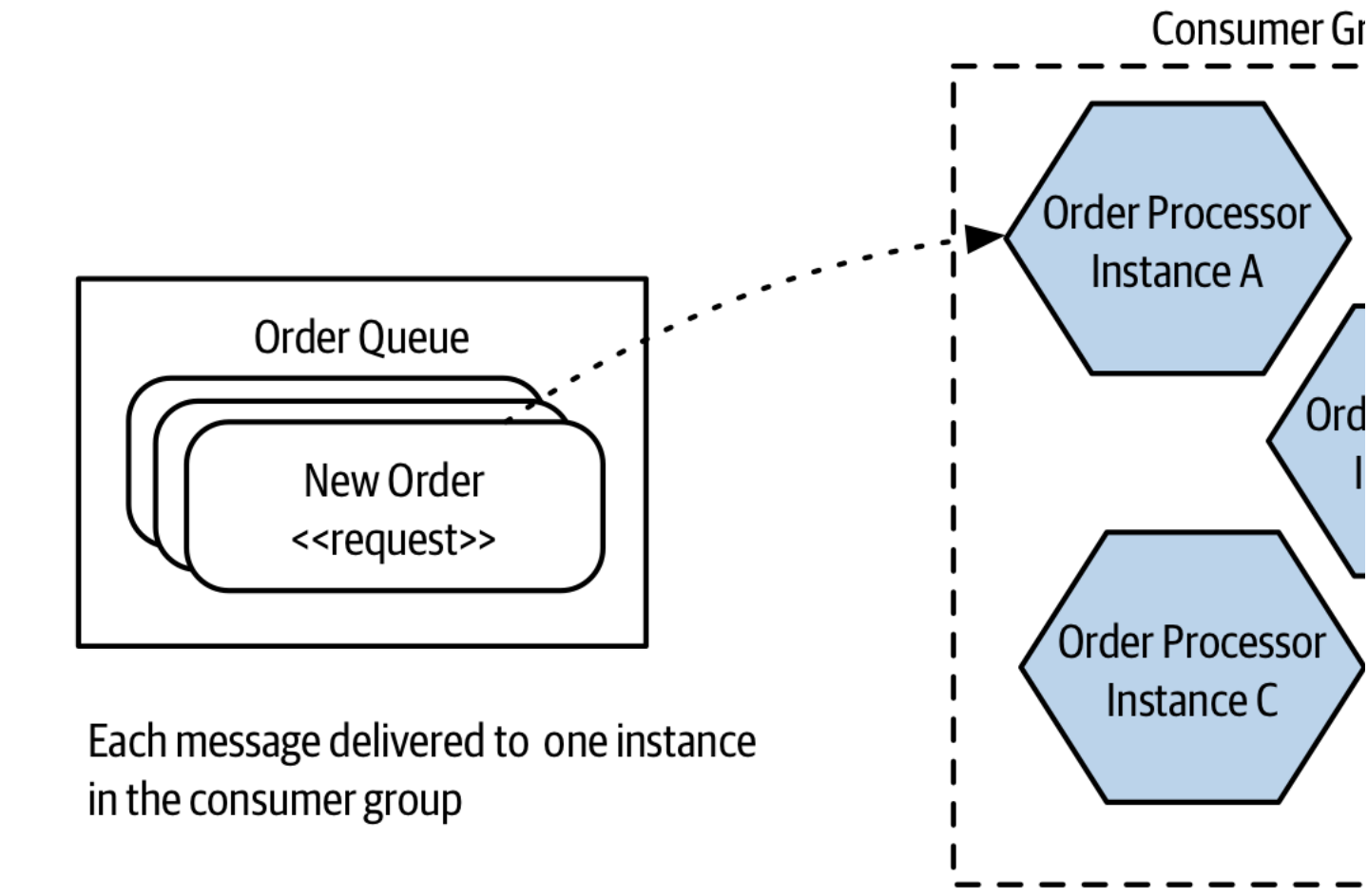


Figure 4-1. A queue allows for one consumer group

With topics, you can have multiple consumer groups. In Figure 4-2, an event representing an order being paid for is put onto the `Order Status` topic. A copy of that event is received by both the `Warehouse` microservice, and the `Notifications` microservice, both of which are in separate consumer groups. Only one instance of each consumer group will see that event.

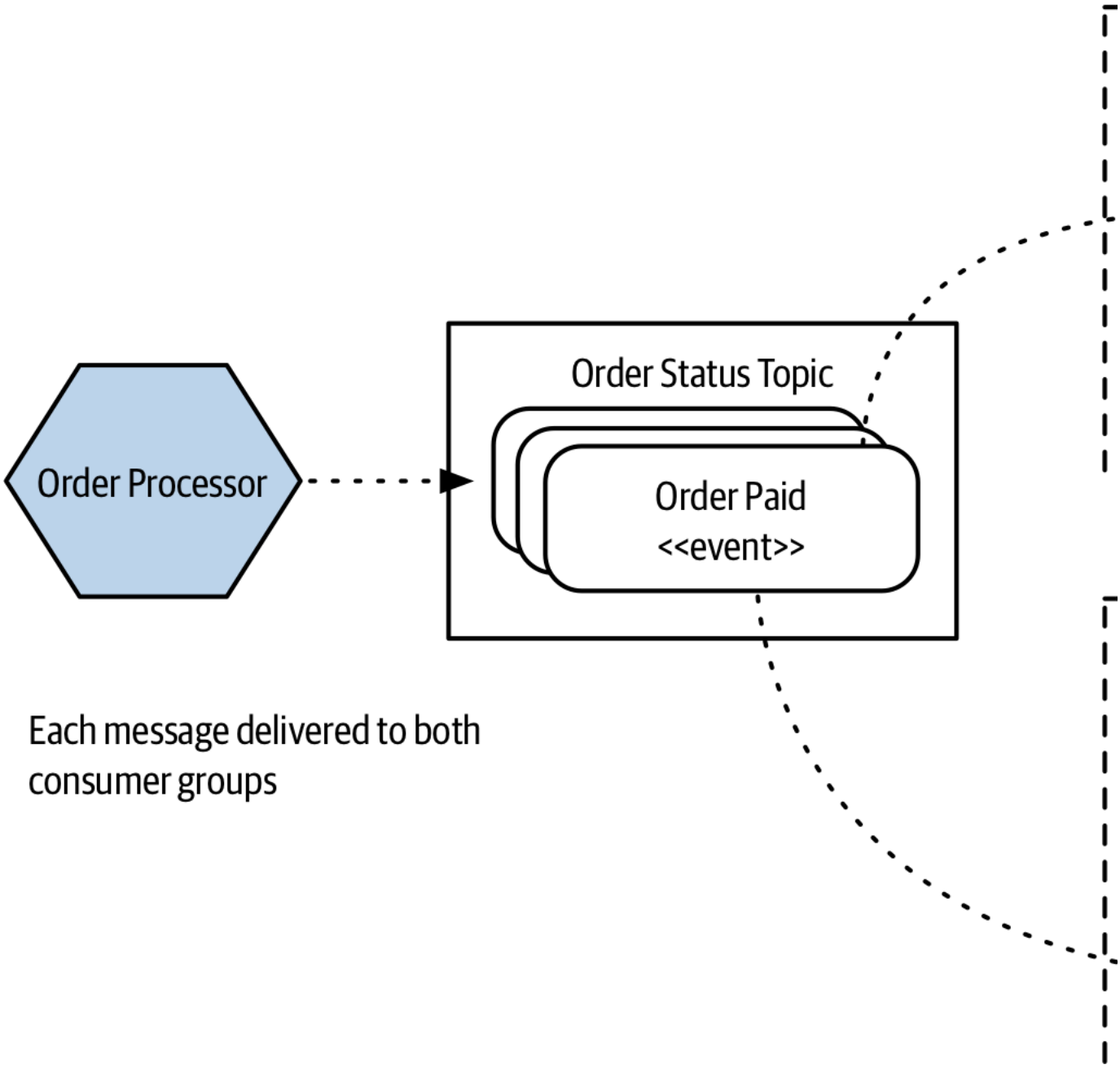


Figure 4-2. Topics allow for multiple subscribers to receive the same messages, useful for event broadcast

At first glance, a queue just looks like a topic with a single consumer group. A large part of the distinction between the two is that when sending a message over a queue, there is knowledge of what the message is being sent to. With a topic, this information is hidden from the sender of the message - they are unaware of who (if anyone) will end up receiving the message.

Topics are a good fit for event-based collaboration, where queues would be more appropriate for request/response communication. This should be considered as general guidance though rather than a strict rule.

Guaranteed Delivery

So why use a broker? Fundamentally, they provide some capabilities that can be very useful for asynchronous communication. The properties they provide vary, but the most interesting feature is that of guaranteed delivery, something which all widely used brokers support in some way. Guaranteed delivery describes a commitment by the broker to ensure that the message is delivered.

From the point of view of the microservice sending the message, this can be very useful. If the downstream destination is unavailable, then this isn't a problem - the broker will hold on to the message until it can be delivered. This can reduce the number of things an upstream microservice needs to worry about. When compared to a synchronous direct call, for example an HTTP request, if the downstream destination isn't reachable, the upstream microservice will need to work out what to do with the request - should it retry the call, or give up?

For guaranteed delivery to work, a broker will need to ensure that any messages not yet delivered are going to be held in a durable fashion until they are able to be delivered. To deliver on this promise, a broker will normally run as some sort of cluster-based system, ensuring that the loss of a single machine doesn't cause the message to be lost. There is typically a lot involved in running a broker correctly, partly due to the challenges in managing cluster-based software. Often, the promise

of guaranteed delivery can be undermined if the broker isn't setup correctly. As an example, RabbitMQ requires instances in a cluster to communicate over relatively low-latency networks, otherwise the instances can start to get confused about the current state of messages being handled, resulting in data loss. I'm not highlighting this particular limitation as a way of saying that RabbitMQ is in anyway bad, all brokers have restrictions as to how they need to be run to deliver the promise of guaranteed delivery. If you plan to run your own broker, make sure you read the documentation carefully.

It's also worth noting that what any given broker means by guaranteed delivery can vary. Again, reading the documentation is a great start.

Trust

One of the big draws of a broker is the property of guaranteed delivery. But for this to work, you need to trust not only the people who created the broker, but also the way that broker has operated. If you've built a system that is based on the assumption that delivery is guaranteed, and that turns out not to be the case due to an issue with the underlying broker, it can cause significant issues. The hope of course is that you are offloading that work to software created by people who can do that job better than you can. Ultimately, you have to decide how much you want to trust the broker you are making use of.

Other Characteristics

Aside from guaranteed delivery, there are other characteristics that brokers can provide that you may find to be useful.

Most brokers can guarantee the order in which messages will be delivered, but this isn't universal, and even then the scope of this guarantee can be limited. With Kafka for example, ordering is only guaranteed within a single partition. If you are unable to be certain that messages will be received in order, your consumer may need to compensate for this, perhaps by deferring processing of messages that are received out of order, until the missing messages are received.

Some brokers provide transactions on write - Kafka as an example allows you to write to multiple topics in a single transaction. Some brokers can also provide read transactionality, and this is something I've made use of when using a number of brokers via the Java Messaging Service (JMS) APIs. This can be useful if you want to ensure the message can be processed by the consumer before removing it from the broker.

Another, somewhat controversial feature promised by some brokers is that of exactly once delivery. One of the easier ways to provide guaranteed delivery is allowing the message to be resent. This can result in a consumer seeing the same message more than once (even if this is a rare situation). Most brokers will do what they can to reduce the chance of this, or hide this fact from the consumer, but some brokers have gone further and guarantee exactly once delivery. This is a complex topic, as I've spoken to some experts who state that guaranteeing this in all cases is impossible, while other experts say you basically can do this with a few simple workarounds. Either way, if your broker of choice claims to implement this, then pay **really** careful attention to how this is implemented. Even better, build your consumers in such a way that they are prepared for the fact that they might receive a message more than once, and can handle this situation. A very simple example would be for each message to have an ID which a consumer can check when the message is received. If a message with that ID has already been processed, the message can be ignored.

Choices

A variety of message brokers exist. Popular examples include RabbitMQ, ActiveMQ, and Kafka (which we'll explore further shortly). The main public cloud vendors also provide a variety of products that play this role, from managed versions of those brokers you could install on your own infrastructure, to bespoke implementations that are specific to a given platform. AWS for example has the Simple Queue Service (SQS), Simple Notification Service (SNS), and Kinesis, all of which provide different flavours of fully managed brokers. SQS was in fact the first ever product released by AWS, launched back in 2006.

Kafka

Kafka is worth highlighting as a specific broker, due in large part to its popularity in recent years. Part of this popularity is due to its use in helping moving large volumes of data around as part of implementing stream processing pipelines. This can help move from batch-oriented processing to more real-time processing.

There are a few characteristics of kafka which are worth highlighting. Firstly, it is designed for very large scale - it was built at LinkedIn to replace multiple existing message clusters with a single platform. Kafka is built to allow for multiple consumers and producers - I've spoken to one expert at a large technology company who had over 50K producers and consumers working on the same cluster. To be fair, very few organizations have problems at that level of scale, but for some organizations, the ability to scale kafka easily (relatively speaking) can be very useful.

Another fairly unique feature of kafka is message permanence. With a normal message broker, once the last consumer has received a message, the broker no longer needs to hold on to that message. With Kafka, messages can be stored for a configurable period. This means that messages can be stored forever. This can allow consumers to re-ingest messages that they had already processed, or allow newly deployed consumers to process messages that were sent previously.

Finally, Kafka has been rolling out built-in support for stream processing. Rather than using Kafka to send messages to a dedicated stream processing tool like Apache Flink, instead some tasks can be done inside Kafka itself. Using KSQL, you can define SQL-like statements that can process one or more topics on the fly. This can give you something akin to a dynamically updating materialized database view, with the source of data being Kafka topics rather than a database. These capabilities open up some very interesting possibilities about how data is managed in distributed systems. If you'd like to explore these ideas in more detail, I can recommend "Designing Event-Driven Systems" by Ben Stopford[5] (I have to recommend Ben's book, as I wrote the foreword for it!). For a deeper dive on Kafka in general, I'd suggest "Kafka: The Definitive Guide"[6].

TODO: Challenges

TODO: When To Use It

Serialization Formats

Some of the technology choices we've looked at - specifically some of the RPC implementations - make choices for you regarding how data in serialized and deserialized. When picking GRPC for example, any data sent will be converted into protocol buffer format. Many of the technology options though give us a lot of freedom in terms of how we covert data for network calls. Pick Kafka as your broker of choice, and you can send messages in a variety of formats. So which format should you chose?

Textual Formats

The use of standard textual formats gives clients a lot of flexibility as to how they consume resources. REST APIs mostly typically use a textual format for the request and response bodies, even if theoretically you can quite happily send binary data over HTTP. In fact, this is how GRPC works - using HTTP underneath, but sending binary protocol buffers.

JSON has usurped XML as the text serialization format of choice. You can point to a number of reasons why this occurred, but the main reason is that one of the main consumers of APIs is often a browser, where JSON is a great fit. JSON became popular partly as a result of the backlash against XML, and proponents cite its relative compactness and simplicity when compared to XML as another winning factor. The reality is that the size of a JSON vs XML payload is rarely a massive

differential, especially as these payloads are typically compressed. It's also worth pointing out that some of the simplicity of JSON comes at a cost - in our rush to adopt simpler protocols, schemas went out of the window (more on that later).

AVRO is an interesting serialization format. It takes JSON as an underlying structure and uses it to define a schema-based format. AVRO has found a lot of popularity as a format for message payloads, partly due to the ability to send the schema as part of the payload, which can make supporting multiple different messaging formats much easier.

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 "Handling Change Between Microservices") 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 as 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!

Binary Formats

Where textual formats have benefits like making it easy for humans to read them, and provide a lot of interoperability with different tools and technologies, the world of binary serialization protocols is where you want to be if you start getting worried about payload size, or the efficiencies of writing and reading the payloads. Protocol buffers have been around for a while, and are often used outside the scope of GRPC - they probably represent the most popular binary serialization format for microservice-based communication.

This space though is large, and there are a number of other formats out there that have been developed with a variety of requirements in mind. Simple Binary Encoding[7], Cap'n Proto[8], and FlatBuffers[9] all come to mind. Although benchmarks abound for each of these formats, highlighting their relevant benefits compared to protocol buffers, JSON, or other formats, benchmarks suffer from a fundamental problem that they may not necessarily represent how you are going to use them. If you're looking to eek the last few bytes out of your serialization format, or shave microseconds off the time taken to read or write these payloads, I strongly suggest you carry out your own comparison of these various formats. In my experience, the vast majority of systems rarely have to worry about such optimizations though, as they can often achieve the improvements they are looking for by sending less data, or not making the call at all. If you are building an ultra-low latency distributed system though, make sure you're prepared to dive head first into the world of binary serialization formats.

Schemas

One discussion that comes up time and again is should we use schemas to define what our endpoints expose, and what they accept? Schemas can come in lots of different types, and typically picking a serialization format will define which schema technology you can use. If you're working with raw XML, you'd use XML Schema Definition (XSD), raw json, you'd use JSON-Schema. Some of the technology choices we've touched on (specifically a sizable subset of the RPC options) require the use of explicit schemas, so if you picked those technologies you'd have to make use of schemas. SOAP works through use of a schema specification called the Web Service Definition Language (WSDL), while GRPC requires the use of a protocol buffer specification. Other technology choices we've explored make the use of schemas optional, and this is where things get more interesting.

Personally speaking, I am in favour of having explicit schemas for microservice endpoints. This is for two key reasons. Firstly, it goes a long way to being an explicit representation of what a microservice endpoint exposes, and what it can accept. This makes life easier for both developers working on the microservice, but also their consumers. Schemas may not replace the need for good documentation, but they certainly can help reduce the amount of documentation required.

The other reason I like explicit schemas though, is how they help in terms of catching accidental breakages of microservice endpoints. We'll explore how to handle changes between microservices in a moment, but it's first worth exploring the different types of breakages and the role schemas can play.

Structural vs Semantic Contract Breakages

Broadly speaking, we can break contract breakages down into two categories - *structural* breakages, and *semantic* breakages. Structural breakages refer to situations where the structure of the endpoint changes in such a way that a consumer is now incompatible - this could represent fields or methods being removed, or new required fields being added. Semantic breakages refer to situations where the structure of the microservices endpoint remains the same, but the behavior changes in such a way as to break consumers expectations.

Let's take a simple example. You have a highly complex `Hard Calculations` microservice that exposes a `calculate` method on its endpoint. This `calculate` method takes two integers, both of which are required fields. If you changed `Hard Calculations` such that the `calculate` method now takes only one integer, then consumers would break - they'd be sending requests with two integers which the `Hard Calculations` microservice would reject. This is an example of a structural change, and in general these changes can be easier to spot.

Semantic changes are more problematic. This is where the structure of the endpoint doesn't change, but the behavior of the endpoint does. Coming back to our `calculate` method, imagine that in the first version, the two provided integers are addded together and the results returned. So far so good. Now, we change `Hard Calculations` so that the `calculate` method now multiplies the integers together and returns the result. The semantics of the `calculate` method have changed in a way that could break expectations of the consumers.

Should You Use Schemas?

By using schemas, and comparing different versions of schemas, we can catch structural breakages. Catching semantic breakages requires the use of testing. If you don't have schemas, or have schemas but decide to not compare schema changes for compatibility, then the burden of catching structural breakages before you get to production also falls on testing. Arguably, the situation is somewhat analogous with static vs dynamic typing in programming languages. With a statically typed language, the types are fixed at compile time - if your code does something with an instance of a type that isn't allowed (like calling a method that doesn't exist), then the compiler can catch that mistake. This can leave you to focus testing efforts on other sorts of problems. With a dynamically typed language though, some of your testing will need to catch mistakes that a compiler picks up for statically typed languages.

Now, I'm pretty relaxed about static vs dynamically typed languages, and I've found myself to be very productive (relatively speaking) in both. Certainly, dynamically typed languages give you some significant benefits which for many people justify giving up on compile time safety. Personally speaking though, if we bring the discussion back to microservice interactions, I haven't found that a similar balanced tradeoff exists when it comes to schema vs schemaless communication. Put simply, I think that having an explicit schema more than offsets any perceived benefit of having schema-less based communication.

The main argument for schemaless endpoints seems to be that schemas need more work and don't give enough value. This IMHO is partly a failure of imagination, and partly a failure of good tooling to help schemas have more value when it comes to using them to catch structural breakages.

Really, the question isn't actually if you have a schema or not - it's whether or not that schema is *explicit*. If you are consuming data from a schemaless API, you still have expectations as to what data should be in there, and how that data should be structured. Your code that will handle the data will be written with a set of assumptions in mind as to how that data is structured. In such a case the schema is arguably totally implicit, rather than explicit[10]. A lot of my desire for an explicit schema is driven by the fact that I think it's important to be as explicit as possible as to what a microservice does (or doesn't) expose.

Ultimately, a lot of what schemas provide is an explicit representation of part of the structure contract between a client and server. They help make things explicit, and can greatly aid communication between teams as well as work as a safety net. In situations where the cost of change is reduced, for example where both client

and server are owned by the same team, then I am more relaxed about you not having schemas.

Handling Change Between Microservices

Probably the most common question I get about microservices, after “how big should they be?” is “how do you handle versioning?”. When this question gets asked, it’s rarely a query regarding what sort of numbering scheme you should use, it’s more about how you handle changes in the contracts between microservices.

How you handle change really breaks down into two topics. In a moment, we’ll look at what happens if you need to make a breaking change. But before that, we’ll look at what you can do to avoid making a breaking change in the first place.

Avoiding Breaking Changes

If you want to avoid making breaking changes, there are a few key ideas which are worth exploring - many of which we’ve already touched on at the start of the chapter. If you can put these ideas into practice, you’ll find it much easier to allow for microservices to be changed independently from one another.

Expansion Changes

: Add new things to a microservice interface, don’t remove old things

Tolerant Reader

: When consuming a microservice interface, be flexible in what you expect.

Right Technology

: Pick technology that makes it easier to make backwards compatible changes to the interface.

Explicit Interface

: Be explicit about what a microservice exposes. This makes things easier for the client, and easier for the maintainers of the microservice to understand what can be changed freely.

Catch Accidental Breaking Changes Early

: Have mechanisms in place to catch interface changes that will break consumers in production, before those changes are deployed.

These ideas do reinforce each other, and many build upon that key concept of information hiding that we’ve discussed frequently so far. Let’s look at each in turn.

Expansion Changes

Probably the easiest place to start is by only adding new things to a microservice contract, and don’t remove anything else. Consider the example of adding a new field to a payload - assuming the client is in some way tolerant of such changes, this shouldn’t have a material impact. Adding a new `dob` field to a customer record should be fine for example.

Tolerant Reader

How the consumer of a microservice is implemented can have a lot to say regarding making backwards compatible changes easy. Specifically, we want to avoid client code binding too tightly to the interface of a microservice. Let’s consider an email microservice, 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 a 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 4-3.

Example 4-3. 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, the email microservice only needs 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 4-4? 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 4-4. 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 microservice-based interactions, it leads us to try and structure our client code to be tolerant of changes to payloads.

Right Technology

As we've already explored, some technology can be more brittle when it comes to allowing us to change interfaces - I've already highlighted my own personal frustrations with Java RMI. On the other hand, some integration implementations go out of their way to make it as easy as possible for changes to be made without breaking clients. At the simple end of the spectrum, protocol buffers, the serialization format used as part of GRPC, has the concept of field number. Each entry in a protocol buffer has to define a field number, which client code expects to find. If new fields are added, the client doesn't care. AVRO allows for the schema to be sent along with the payload, allowing clients to potentially interpret a payload much like a dynamic type.

At the more extreme end of the spectrum, the REST concept of HATEOS is largely all about enabling clients to make use of REST endpoints even when they change by making use of the previously discussed hypermedia links. This does call for you to buy into the entire HATEOS mindset of course.

Explicit Interface

I am a **big** fan of a microservice exposing an explicit schema denoting what it's endpoints do. Having an explicit schema makes it clear to consumers as to what they can expect, but it also makes it much more clear to a developer working on a microservice as to what things should remain untouched to ensure you don't break consumers. Put another way, an explicit schema goes a long way to making the boundaries of information hiding more explicit - what's exposed in the schema is by definition not hidden.

Having an explicit schema for RPC is long established, and is in fact a requirement for many RPC implementations. REST on the other hand has typically viewed the concept of a schema as optional, to the point where I find explicit schemas for REST endpoints to be vanishingly rare. This is changing, with things like the aforementioned OpenAPI specification gaining traction, and the JSON Schema specification also gaining in maturity.

Asynchronous messaging protocols have struggled more in this space. You can have a schema for the payload of a message easily enough, and in fact this is an area where AVRO is frequently used. However having an explicit interface needs to go further than this. If we consider a microservice that fires events, which events does it expose? There are a few attempts at making explicit schemas for event-based endpoints underway. One is AsyncAPI[11] which has picked up a number of big name users, but the one gaining most traction seems to be CloudEvents specification[12] which is backed by the Cloud Native Computing Foundation. Azure's event grid product supports the CloudEvents format, a sign of different vendors supporting this format which should help with interoperability. This is still a fairly new space, so it will be interesting to see how things shake out over the next few years.

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 seen this approach used enough in distributed systems to understand its effectiveness in that context.

Catch Accidental 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, it's possible that an innocent change of a microservice could cause consumers to break. As we've already touched on, using schemas can help us pick up structural changes, assuming we use some sort of tooling to help compare schema versions. There is a wide range of tooling out there to do this for different schema types. We have ProtoLock[13] for protocol buffers, json-schema-diff-validator for JSON-Schema[14], or openapi-diff for the openAPI specification[15]. More tools seem to be cropping up all the time in this space - what you're looking for though is something that doesn't just report on the differences between two schemas, but something that will pass or fail based on compatibility - this would allow you to fail a CI build if incompatible schemas are found, ensuring that your microservice won't get deployed.

The open source Confluent schema registry[16] supports JSON-schema, AVRO and protocol buffers, and is capable of comparing newly uploaded versions for backwards compatibility. Although it was built to help as part of an ecosystem where Kafka is being used, the registry isn't tied to kafka in anyway, and you could make use of this in other situations to ensure backwards compatibility based on schema comparison.

Schema comparison tools can help us catch structural breakages, but what about semantic breakages? Or what if you aren't making use of schemas in the first place? Then we're looking at testing. This is a topic we'll explore in more detail in [Link to Come], but I wanted to highlight consumer-driven contract testing which explicitly helps in this area. Just remember, if you don't have schemas, expect your testing to have to do more work to catch breaking changes.

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.

Managing Breaking Changes

So you've gone as far as you can to ensure that the changes you're making to a microservice's interface are backwards compatible, but you've realized that you just have to make a change that will constitute a breaking change. What can you do in such a situation? You've got three main options:

Lock-step Deployment

: Require that the microservice exposing the interface and all consumers of that interface are changed at the same time

Coexist Incompatible Microservice Versions

: Run old and new versions of the microservice side by side

Emulate The Old Interface

: Have your microservice expose the new interface, and also emulate the old interface

Lock-Step Deployment

Of course, lock-step deployment flies in the face of independent deployability. If we want to be able to deploy a new version of our microservice with a breaking change to it's interface, but still do this in an independent fashion, we need to give our consumers time to upgrade to the new interface. That leads us on to the next two options I'd consider.

Coexist Incompatible Microservice 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 4-3. 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.

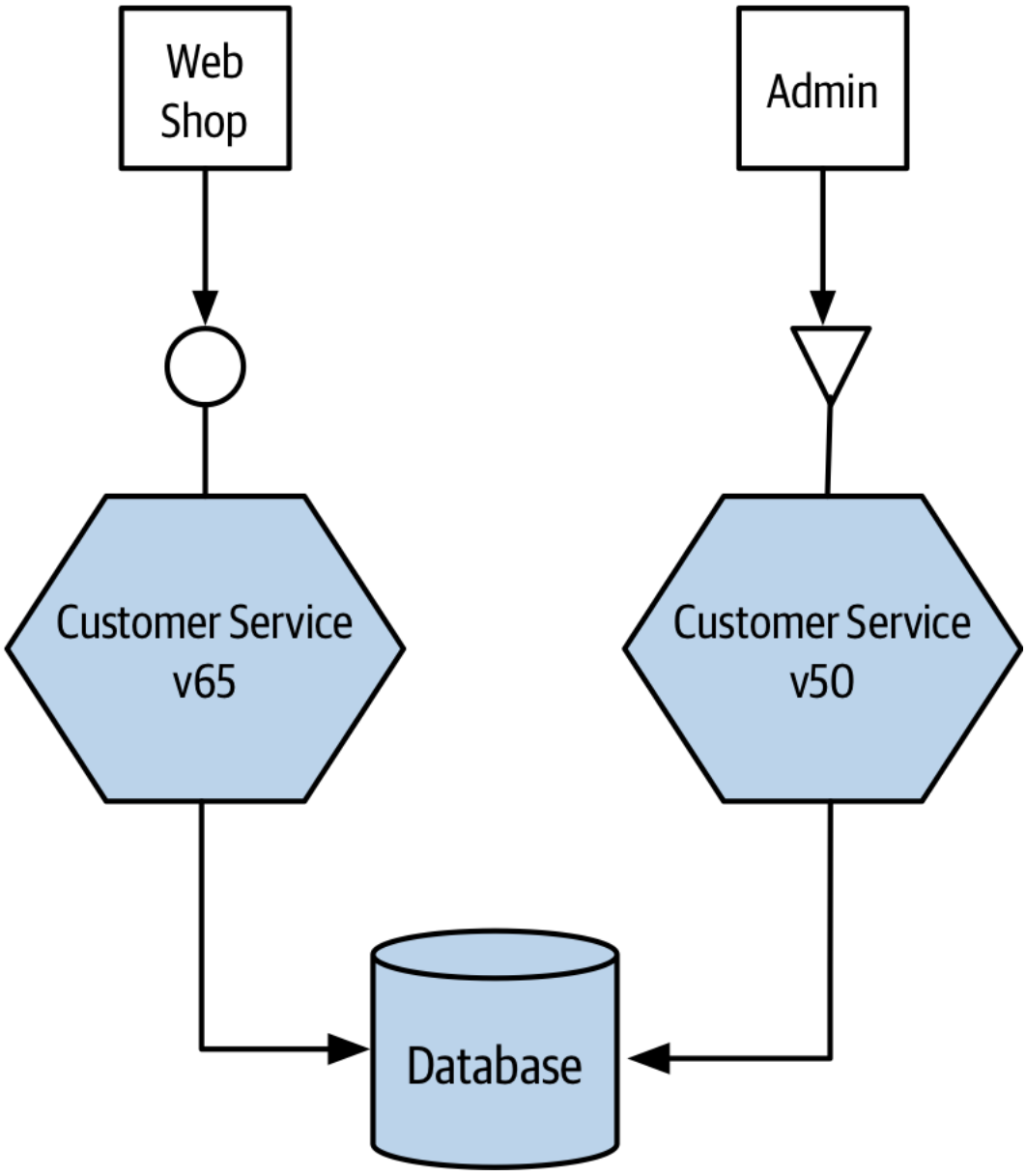


Figure 4-3. Running multiple versions of the same service to support old endpoints

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 [Link 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 this work is worthwhile for the average project.

Emulate The Old Interface

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 4-4.

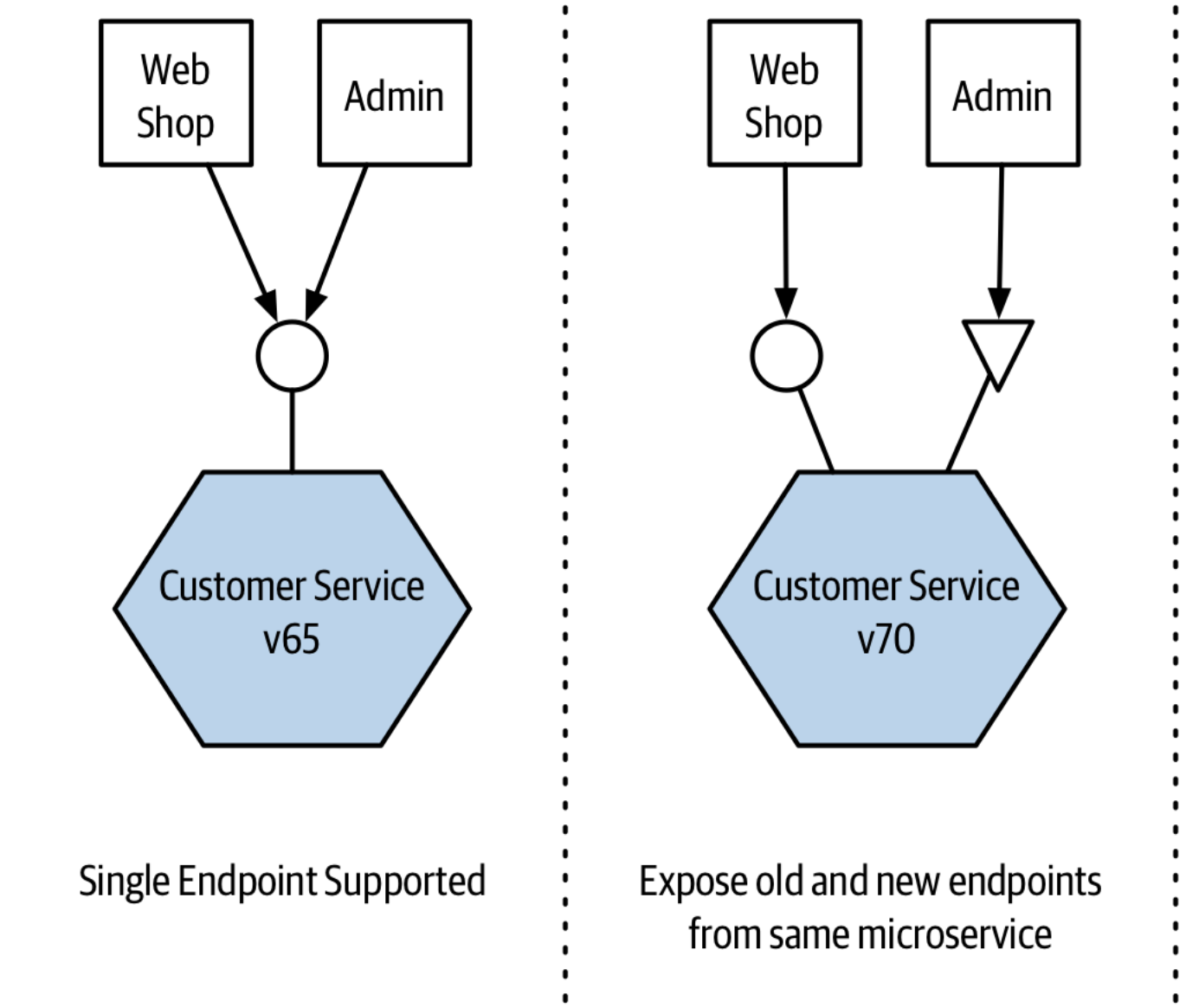


Figure 4-4. One microservice emulating the old endpoint and exposing the new backwards incompatible endpoint

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 hard-coding 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.

Which Approach Do I Prefer?

For situations where the same team manages both the microservice and all consumers, I am somewhat relaxed about a lock-step release in limited situations. Assuming it really is a one-off situation, then doing this in a situation where the impact is limited to a single team can be justifiable. I am very cautious about this though, as there is the danger that a one-off activity becomes business as usual, and there goes independent deployability. Use lock-step deployments too often, and you'll end up with a distributed monolith before long.

Co-existing different versions of the same microservice can be problematic, as we discussed. I'd only consider doing this in situations where we only planned to run the microservice versions side by side for a short period of time. The reality is that when you need to give consumers time to upgrade, you could be looking at weeks or more. In other situations where you might co-exist microservice versions, perhaps as part of a blue/green deployment or canary release, the durations involved are much shorter, offsetting the downsides of this approach.

My general preference is where possible to use emulation of old endpoints. The challenges of implementing emulation are in my opinion much easier to deal with than co-existing of microservice versions.

The Social Contract

Which approach you pick will be due in large part to the expectations consumers have of how these changes will be made. Keeping the old interface lying around can have a cost, and ideally you'd like to turn it off and remove associated code and infrastructure as soon as possible. On the other hand, you want to give consumers as much time as possible to make a change. And remember, in many cases the backwards-incompatible changes you are making are often things that have been asked for by the consumers, or will actually end up benefiting them. There is a balancing act of course, between the needs of the microservice maintainers, and the consumers - and this needs to be discussed.

I've found that in many situations, how these changes will be handled has never been discussed, leading to all sorts of challenges. As with schemas, having some degree of explicitness in how backwards-incompatible changes will be made can greatly simplify things.

You don't need reams of paper and huge meetings necessarily to agree these things. But both the owner and consumer of a microservice need to be clear on a few things. Assuming you aren't going down the route of lock-step releases, I'd suggest being clear on a few things:

- How will you raise the issue that the interface needs to change?
- How will the consumers and microservice teams collaborate to agree on what the change will look like?
- Who is expected to do the work to update the consumers?
- When the change is agreed, how long will consumers have to shift over to the new interface before it is removed?

Remember, one of the secrets to an effective microservice architecture is to embrace a consumer-first approach. Your microservices exist to be called by other consumers. Their needs are paramount, and if you are making changes to a microservice that are going to cause upstream consumers problems, this needs to be taken into account.

In some situations of course it might not be possible to change the consumers. I've heard from Netflix that they had issues (at least historically), with old set-top boxes using older versions of the Netflix APIs. These set-top boxes cannot be upgraded easily, so the old endpoints have to remain available unless and until the number of older set-top boxes drops to a level where they can have their support disabled. Decisions to stop old consumers being able to access your endpoints can sometimes end up being financial - how much money does it cost you to support the old interface, balanced against how much money you make from those consumers.

Tracking Usage

Even if you do agree on a time by which consumers should stop using the old interface, would you know if they had actually stopped using it? Making sure you have logging in place for each endpoint your microservice exposes can help, as can ensuring that you have some sort of client identifier so you can chat to the team in question if you need to work with them to get them to migrate away from your old interface. This could be something as simple as asking consumers to put their identifer in the user agent header when making HTTP requests, or you could require that all calls go via some sort of API Gateway where clients need keys to identify themselves.

Extreme Measures

So, assuming you know a consumer is still using an old interface that you want to remove, and they are dragging their heels about moving to the new version, what can you do about it? Well, the first thing to do is talk to them. Perhaps you can lend them a hand to make the changes happen. If all else fails, and they still don't upgrade even after agreeing to, then there are some extreme techniques I've seen used.

In one large tech company, I discussed with them how they handled this issue. Internally, they had a very generous period of one year before old interfaces would be retired. I asked how they knew if consumers were still using the old interfaces, and they replied that they didn't bother tracking that information really. After one year they just turned the old interface off. It was recognized internally that if this caused a consumer to break, then in that company it was accepted that it was the fault of the consuming microservice's team - they'd had a year to make the change, and hadn't done it. Of course, this approach won't work for many (I said it was extreme!). It also leads to a large degree of inefficiency. By not knowing if the old interface was used, they denied themselves the opportunity to remove it before the year had passed. Personally, even if I was to suggest just turning the endpoint off after a certain period of time, I'd still definitely want tracking of who was going to be impacted.

Another extreme measure I saw was actually in the context of deprecating libraries, but it could also theoretically be used for microservice endpoints. The example given was of an old library that people were trying to retire from use inside the organization, in favour of a newer, better one. Despite lots of work, other teams were still dragging their heels. The solution was to insert a sleep in the old library, so that it responded more slowly to calls (with logging to show what was happening). Over time, the team just kept increasing the duration of the sleep, until eventually the other teams got the message. You obviously have to be extremely sure that you've exhausted other reasonable efforts to get consumers to upgrade before considering something like this!

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! It turns out though that sharing code in a microservice environment is a bit more involved than that. As always, we have more than one option to consider.

Sharing Code Via Libraries

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.

The really important point about sharing code via libraries is that you cannot update all uses of the library at once. Although multiple microservices might all use the same library, they do so typically by packaging that library into the microservice deployment. To upgrade the version of the library being used, you'd therefore need to redeploy the microservice. If you want to update the same library everywhere at exactly the same time, this could lead to a widespread deployment of multiple different microservices all at the same time, with all the associated headaches.

So, if using libraries for code reuse across microservice boundaries, you have to accept that multiple different versions of the same library might be out there at the same time. You can of course look to update all of these to the last version over time, but as long as you are OK with this fact, then by all means reuse code via libraries. If you really do need to update that code for all users of it at exactly the same time, then you'll actually want to look at reusing code via a dedicated microservice instead.

There is one specific use case associated with reuse through libraries which is worth exploring further, though.

Client Libraries

I've spoken to more than one team that 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 increase 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!

Service Meshes and API Gateways

Service Meshes and API Gateways do offer a potential way to share code between microservices without requring the creation of new client libraries, or new microservices. Put (very) simply, service meshes and API gateways can work as proxies between microservices. This can mean that they can be used to to implement some microservice-agnostic behaviour which might otherwise have to be done in code, such as service discovery or logging.

If you are using either an API gateway or a service mesh to implement shared, common behaviour for your microservices, it's essential that this behaviour is totally generic - in other words, the behaviour in the proxy bares no relation to any specific behaviour of an individual microservice.

API Gateways and Service Meshes are topics which need to be explored more fully - we'll come back to them both in [Link to Come].

Summary

So, we've covered a lot of ground here - let's break down some of what we've covered.

- Firstly, ensure that the problem you are trying to solve guides your technology choice. Based on your context, and your preferred communication style, use that to select the technology that is most appropriate to you - don't fall into the trap of picking the technology first. The model shared again in Figure 4-5 can help guide your decision making, but just following this model isn't a replacement for sitting down and thinking about your own situation.

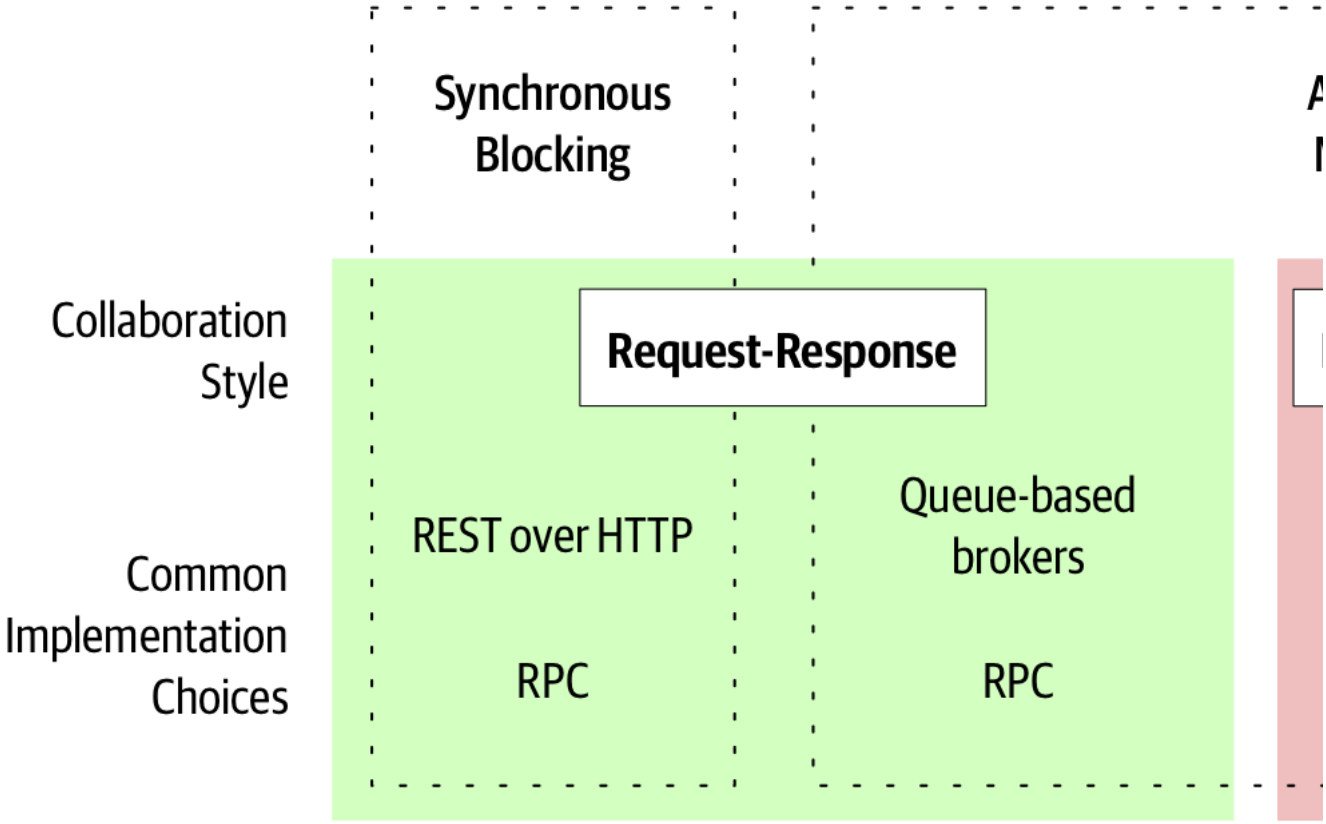


Figure 4-5. Different styles of inter-microservice communication along with example implementing technologies

- Whatever choice you make, consider the use of schemas as part of helping make your contracts more explicit, but also to help catch accidental breaking changes.
- Where possible, strive to make changes which are backwards compatible to ensure that independent deployability remains as possibility.
- If you do have to make backwards incompatible changes, find a way to allow consumers time to upgrade to avoid lock-step deployments.

Next, we need to address the fact that most people don't start with a microservice architecture, and look at how you can take an existing monolithic system and migrate it to a microservice architecture.

[1] *https://protolock.dev/*

[2] *https://github.com/OAI/OpenAPI-Specification/*

[3] Robinson, Ian, Jim Webber, and Savas Parastatidis, "REST in Practice: Hypermedia and Systems Architecture". O'Reilly 2010

[4] *https://graphql.org/*

[5] Stopford, Ben. *Designing Event-Driven Systems*. O'Reilly 2017.

[6] Narkhede, Neha, Gwen Shapira and Todd Palino. *Kafka: The Definitive Guide*. O'Reilly 2017

[7] *https://github.com/real-logic/simple-binary-encoding*

[8] *https://capnproto.org/*

[9] *https://google.github.io/flatbuffers/*

[10] Martin Fowler explores this in more detail in the context of schemaless datastorage: *https://martinfowler.com/articles/schemaless/*

[11] *https://www.asyncapi.com/*

[12] *https://cloudevents.io/*

[13] *https://github.com/nilslice/protolock*

[14] *https://www.npmjs.com/package/json-schema-diff-validator*

[15] Note that there are actually three different tools in this space with the same name! *https://github.com/Azure/openapi-diff* seems to get closest to a tool that actually passes or fails compatibility

[16] *https://github.com/confluentinc/schema-registry#documentation*