O'REILLY®

Compliments of confluent

Designing Event-Driven Systems

Concepts and Patterns for Streaming Services with Apache Kafka

Ben Stopford
Foreword by Sam Newman

Designing Event-Driven Systems

Concepts and Patterns for Streaming Services with Apache Kafka

Ben Stopford

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

Designing Event-Driven Systems
by Ben Stopford

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

Editor: Brian Foster
Production Editor: Justin Billing
Copyeditor: Rachel Monaghan
Proofreader: Amanda Kersey

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

April 2018: First Edition

Revision History for the First Edition
2018-03-28: First Release

This work is part of a collaboration between O'Reilly and Confluent. See our *statement of editorial independence*.

978-1-492-03822-1

[LSI]

Table of Contents

Part II. Designing Event-Driven Systems

Part III. Rethinking Architecture at Company Scales

Foreword

For as long as we've been talking about services, we've been talking about data. In fact, before we even had the word *microservices* in our lexicon, back when it was just good old-fashioned service-oriented architecture, we were talking about data: how to access it, where it lives, who "owns" it. Data is all-important—vital for the continued success of our business—but has also been seen as a massive constraint in how we design and evolve our systems.

My own journey into microservices began with work I was doing to help organizations ship software more quickly. This meant a lot of time was spent on things like cycle time analysis, build pipeline design, test automation, and infrastructure automation. The advent of the cloud was a huge boon to the work we were doing, as the improved automation made us even more productive. But I kept hitting some fundamental issues. All too often, the software wasn't designed in a way that made it easy to ship. And data was at the heart of the problem.

Back then, the most common pattern I saw for service-based systems was sharing a database among multiple services. The rationale was simple: the data I need is already in this other database, and accessing a database is easy, so I'll just reach in and grab what I need. This may allow for fast development of a new service, but over time it becomes a major constraint.

As I expanded upon in my book, *Building Microservices*, a shared database creates a huge coupling point in your architecture. It becomes difficult to understand what changes can be made to a schema shared by multiple services. David Parnas[1] showed us back in 1971 that the secret to creating software whose parts could be changed independently was to hide information between modules. But at a swoop, exposing a schema to multiple services prohibits our ability to independently evolve our codebases.

1 D. L. Parnas, *On the Criteria to Be Used in Decomposing Systems into Modules* (Pittsburgh, PA: Carnegie Mellon University, 1971).

As the needs and expectations of software changed, IT organizations changed with them. The shift from siloed IT toward business- or product-aligned teams helped improve the customer focus of those teams. This shift often happened in concert with the move to improve the autonomy of those teams, allowing them to develop new ideas, implement them, and then ship them, all while reducing the need for coordination with other parts of the organization. But highly coupled architectures require heavy coordination between systems and the teams that maintain them—they are the enemy of any organization that wants to optimize autonomy.

Amazon spotted this many years ago. It wanted to improve team autonomy to allow the company to evolve and ship software more quickly. To this end, Amazon created small, independent teams who would own the whole lifecycle of delivery. Steve Yegge, after leaving Amazon for Google, attempted to capture what it was that made those teams work so well in his infamous (in some circles) "Platform Rant". In it, he outlined the mandate from Amazon CEO Jeff Bezos regarding how teams should work together and how they should design systems. These points in particular resonate for me:

> 1) All teams will henceforth expose their data and functionality through service interfaces.
>
> 2) Teams must communicate with each other through these interfaces.
>
> 3) There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's datastore, no shared-memory model, no backdoors whatsoever. The only communication allowed is via service interface calls over the network.

In my own way, I came to the realization that how we store and share data is key to ensuring we develop loosely coupled architectures. Well-defined interfaces are key, as is hiding information. If we need to store data in a database, that database should be part of a service, and not accessed directly by other services. A well-defined interface should guide when and how that data is accessed and manipulated.

Much of my time over the past several years has been taken up with pushing this idea. But while people increasingly get it, challenges remain. The reality is that services do need to work together and do sometimes need to share data. How do you do that effectively? How do you ensure that this is done in a way that is sympathetic to your application's latency and load conditions? What happens when one service needs a lot of information from another?

Enter streams of events, specifically the kinds of streams that technology like Kafka makes possible. We're already using message brokers to exchange events, but Kafka's ability to make that event stream persistent allows us to consider a new way of storing and exchanging data without losing out on our ability to create loosely coupled autonomous architectures. In this book, Ben talks about the

idea of "turning the database inside out"—a concept that I suspect will get as many skeptical responses as I did back when I was suggesting moving away from giant shared databases. But after the last couple of years I've spent exploring these ideas with Ben, I can't help thinking that he and the other people working on these concepts and technology (and there is certainly lots of prior art here) really are on to something.

I'm hopeful that the ideas outlined in this book are another step forward in how we think about sharing and exchanging data, helping us change how we build microservice architecture. The ideas may well seem odd at first, but stick with them. Ben is about to take you on a very interesting journey.

—Sam Newman

Preface

In 2006 I was working at ThoughtWorks, in the UK. There was a certain energy to the office at that time, with lots of interesting things going on. The Agile movement was in full bloom, BDD (behavior-driven development) was flourishing, people were experimenting with Event Sourcing, and SOA (service-oriented architecture) was being adapted to smaller projects to deal with some of the issues we'd seen in larger implementations.

One project I worked on was led by Dave Farley, an energetic and cheerful fellow who managed to transfer his jovial bluster into pretty much everything we did. The project was a relatively standard, medium-sized enterprise application. It had a web portal where customers could request a variety of conveyancing services. The system would then run various synchronous and asynchronous processes to put the myriad of services they requested into action.

There were a number of interesting elements to that particular project, but the one that really stuck with me was the way the services communicated. It was the first system I'd worked on that was built solely from a collaboration of events. Having worked with a few different service-based systems before, all built with RPCs (remote procedure calls) or request-response messaging, I thought this one felt very different. There was something inherently spritely about the way you could plug new services right into the event stream, and something deeply satisfying about tailing the log of events and watching the "narrative" of the system whizz past.

A few years later, I was working at a large financial institution that wanted to build a data service at the heart of the company, somewhere applications could find the important datasets that made the bank work—trades, valuations, reference data, and the like. I find this sort of problem quite compelling: it was technically challenging and, although a number of banks and other large companies had taken this kind of approach before, it felt like the technology had moved on to a point where we could build something really interesting and transformative.

Yet getting the technology right was only the start of the problem. The system had to interface with every major department, and that meant a lot of stakeholders with a lot of requirements, a lot of different release schedules, and a lot of expectations around uptime. I remember discussing the practicalities of the project as we talked our design through in a two-week stakeholder kick-off meeting. It seemed a pretty tall order, not just technically, but organizationally, but it also seemed plausible.

So we pulled together a team, with a bunch of people from ThoughtWorks and Google and a few other places, and the resulting system had some pretty interesting properties. The datastore held queryable data in memory, spread over 35 machines per datacenter, so it could handle being hit from a compute grid. Writes went directly through the query layer into a messaging system, which formed (somewhat unusually for the time) the system of record. Both the query layer and the messaging layer were designed to be sharded so they could scale linearly. So every insert or update was also a published event, and there was no side-stepping it either; it was baked into the heart of the architecture.

The interesting thing about making messaging the system of record is you find yourself repurposing the data stream to do a whole variety of useful things: recording it on a filesystem for recovery, pushing it to another datacenter, hydrating a set of databases for reporting and analytics, and, of course, broadcasting it to anyone with the API who wants to listen.

But the real importance of using messaging as a system of record evaded me somewhat at the time. I remember speaking about the project at QCon, and there were more questions about the lone "messaging as a system of record" slide, which I'd largely glossed over, than there were about the fancy distributed join layer that the talk had focused on. So it slowly became apparent that, for all its features—the data-driven precaching that made joins fast, the SQL-over-Document interface, the immutable data model, and late-bound schema—what most customers needed was really subtly different, and somewhat simpler. While they would start off making use of the data service directly, as time passed, some requirement would often lead them to take a copy, store it independently, and do their own thing. But despite this, they still found the central dataset useful and would often take a subset, then later come back for more. So, on reflection, it seemed that a messaging system optimized to hold datasets would be more appropriate than a database optimized to publish them. A little while later Confluent formed, and Kafka seemed a perfect solution for this type of problem.

The interesting thing about these two experiences (the conveyancing application and the bank-wide data service) is that they are more closely related than they may initially appear. The conveyancing application had been wonderfully collaborative, yet pluggable. At the bank, a much larger set of applications and services integrated through events, but also leveraged a historic reference they could go

back to and query. So the contexts were quite different—the first was a single application, the second a company—but much of the elegance of both systems came from their use of events.

Streaming systems today are in many ways quite different from both of these examples, but the underlying patterns haven't really changed all that much. Nevertheless, the devil is in the details, and over the last few years we've seen clients take a variety of approaches to solving both of these kinds of problems, along with many others. Problems that both distributed logs and stream processing tools are well suited to, and I've tried to extract the key elements of these approaches in this short book.

How to Read This Book

The book is arranged into five sections. Part I sets the scene, with chapters that introduce Kafka and stream processing and should provide even seasoned practitioners with a useful overview of the base concepts. In Part II you'll find out how to build event-driven systems, how such systems relate to stateful stream processing, and how to apply patterns like Event Collaboration, Event Sourcing, and CQRS. Part III is more conceptual, building on the ideas from Part II, but applying them at the level of whole organizations. Here we question many of the common approaches used today, and dig into patterns like event streams as a source of truth. Part IV and Part V are more practical. Part V starts to dip into a little code, and there is an associated GitHub project to help you get started if you want to build fine-grained services with Kafka Streams.

The introduction given in Chapter 1 provides a high-level overview of the main concepts covered in this book, so it is a good place to start.

Acknowledgments

Many people contributed to this book, both directly and indirectly, but a special thanks to Jay Kreps, Sam Newman, Edward Ribeiro, Gwen Shapira, Steve Counsell, Martin Kleppmann, Yeva Byzek, Dan Hanley, Tim Bergland, and of course my ever-patient wife, Emily.

PART I
Setting the Stage

The truth is the log.

—Pat Helland, “Immutability Changes Everything,” 2015

CHAPTER 1
Introduction

While the main focus of this book is the building of event-driven systems of different sizes, there is a deeper focus on software that spans many teams. This is the realm of service-oriented architectures: an idea that arose around the start of the century, where a company reconfigures itself around shared services that do commonly useful things.

This idea became quite popular. Amazon famously banned all intersystem communications by anything that wasn't a service interface. Later, upstart Netflix went all in on microservices, and many other web-based startups followed suit. Enterprise companies did similar things, but often using messaging systems, which have a subtly different dynamic. Much was learned during this time, and there was significant progress made, but it wasn't straightforward.

One lesson learned, which was pretty ubiquitous at the time, was that service-based approaches significantly increased the probability of you getting paged at 3 a.m., when one or more services go down. In hindsight, this shouldn't have been surprising. If you take a set of largely independent applications and turn them into a web of highly connected ones, it doesn't take too much effort to imagine that one important but flaky service can have far-reaching implications, and in the worst case bring the whole system to a halt. As Steve Yegge put it in his famous Amazon/Google post, "Organizing into services taught teams not to trust each other in most of the same ways they're not supposed to trust external developers."

What did work well for Amazon, though, was the element of organizational change that came from being wholeheartedly service based. Service teams think of their software as being a cog in a far larger machine. As Ian Robinson put it, "Be of the web, not behind the web." This was a huge shift from the way people built applications previously, where intersystem communication was something teams reluctantly bolted on as an afterthought. But the services model made

interaction a first-class entity. Suddenly your users weren't just customers or businesspeople; they were other applications, and they really cared that your service was reliable. So applications became platforms, and building platforms is hard.

LinkedIn felt this pain as it evolved away from its original, monolithic Java application into 800–1,100 services. Complex dependencies led to instability, versioning issues caused painful lockstep releases, and early on, it wasn't clear that the new architecture was actually an improvement.

One difference in the way LinkedIn evolved its approach was its use of a messaging system built in-house: Kafka. Kafka added an asynchronous publish-subscribe model to the architecture that enabled trillions of messages a day to be transported around the organization. This was important for a company in hypergrowth, as it allowed new applications to be plugged in without disturbing the fragile web of synchronous interactions that drove the frontend.

But this idea of rearchitecting a system around events isn't new—event-driven architectures have been around for decades, and technologies like enterprise messaging are big business, particularly with (unsurprisingly) enterprise companies. Most enterprises have been around for a long time, and their systems have grown organically, over many iterations or through acquisition. Messaging systems naturally fit these complex and disconnected worlds for the same reasons observed at LinkedIn: events decouple, and this means different parts of the company can operate independently of one another. It also means it's easier to plug new systems into the real time stream of events.

A good example is the regulation that hit the finance industry in January 2018, which states that trading activity has to be reported to a regulator within one minute of it happening. A minute may seem like a long time in computing terms, but it takes only one batch-driven system, on the critical path in one business silo, for that to be unattainable. So the banks that had gone to the effort of installing real-time trade eventing, and plumbed it across all their product-aligned silos, made short work of these regulations. For the majority that hadn't it was a significant effort, typically resulting in half-hearted, hacky solutions.

So enterprise companies start out complex and disconnected: many separate, asynchronous islands—often with users of their own—operating independently of one another for the most part. Internet companies are different, starting life as simple, front-facing web applications where users click buttons and expect things to happen. Most start as monoliths and stay that way for some time (arguably for longer than they should). But as internet companies grow and their business gets more complex, they see a similar shift to asynchronicity. New teams and departments are introduced and they need to operate independently, freed from the synchronous bonds that tie the frontend. So ubiquitous desires for online utilities, like making a payment or updating a shopping basket, are slowly replaced by

a growing need for datasets that can be used, and evolved, without any specific application lock-in.

But messaging is no panacea. Enterprise service buses (ESBs), for example, have vocal detractors and traditional messaging systems have a number of issues of their own. They are often used to move data around an organization, but the absence of any notion of history limits their value. So, even though recent events typically have more value than old ones, business operations still need historical data—whether it's users wanting to query their account history, some service needing a list of customers, or analytics that need to be run for a management report.

On the other hand, data services with HTTP-fronted interfaces make lookups simple. Anyone can reach in and run a query. But they don't make it so easy to move data around. To extract a dataset you end up running a query, then periodically polling the service for changes. This is a bit of a hack, and typically the operators in charge of the service you're polling won't thank you for it.

But replayable logs, like Kafka, can play the role of an event store: a middle ground between a messaging system and a database. (If you don't know Kafka, don't worry—we dive into it in Chapter 4.) Replayable logs decouple services from one another, much like a messaging system does, but they also provide a central point of storage that is fault-tolerant and scalable—a shared source of truth that any application can fall back to.

A shared source of truth turns out to be a surprisingly useful thing. Microservices, for example, don't share their databases with one another (referred to as the IntegrationDatabase antipattern). There is a good reason for this: databases have very rich APIs that are wonderfully useful on their own, but when widely shared they make it hard to work out if and how one application is going to affect others, be it data couplings, contention, or load. But the business facts that services do choose to share are the most important facts of all. They are the truth that the rest of the business is built on. Pat Helland called out this distinction back in 2006, denoting it "data on the outside."

But a replayable log provides a far more suitable place to hold this kind of data because (somewhat counterintuitively) you can't query it! It is purely about storing data and pushing it to somewhere new. This idea of pure data movement is important, because data on the outside—the data services share—is the most tightly coupled of all, and the more services an ecosystem has, the more tightly coupled this data gets. The solution is to move data somewhere that is more loosely coupled, so that means moving it into *your* application where you can manipulate it to your heart's content. So data movement gives applications a level of operability and control that is unachievable with a direct, runtime dependency. This idea of retaining control turns out to be important—it's the same reason the shared database pattern doesn't work out well in practice.

So, this replayable log–based approach has two primary benefits. First, it makes it easy to react to events that are happening now, with a toolset specifically designed for manipulating them. Second, it provides a central repository that can push whole datasets to wherever they may be needed. This is pretty useful if you run a global business with datacenters spread around the world, need to bootstrap or prototype a new project quickly, do some ad hoc data exploration, or build a complex service ecosystem that can evolve freely and independently.

So there are some clear advantages to the event-driven approach (and there are of course advantages for the REST/RPC models too). But this is, in fact, only half the story. Streaming isn't simply an alternative to RPCs that happens to work better for highly connected use cases; it's a far more fundamental change in mindset that involves rethinking your business as an evolving stream of data, and your services as functions that transform these streams of data into something new.

This can feel unnatural. Many of us have been brought up with programming styles where we ask questions or issue commands and wait for answers. This is how procedural or object-oriented programs work, but the biggest culprit is probably the database. For nearly half a century databases have played a central role in system design, shaping—more than any other tool—the way we write (and think about) programs. This has been, in some ways, unfortunate.

As we move from chapter to chapter, this book builds up a subtly different approach to dealing with data, one where the database is taken apart, unbundled, deconstructed, and turned inside out. These concepts may sound strange or even novel, but they are, like many things in software, evolutions of older ideas that have arisen somewhat independently in various technology subcultures. For some time now, mainstream programmers have used event-driven architectures, Event Sourcing, and CQRS (Command Query Responsibility Segregation) as a means to break away from the pains of scaling database-centric systems. The big data space encountered similar issues as multiterabyte-sized datasets highlighted the inherent impracticalities of batch-driven data management, which in turn led to a pivot toward streaming. The functional world has sat aside, somewhat knowingly, periodically tugging at the imperative views of the masses.

But these disparate progressions—turning the database inside out, destructuring, CQRS, unbundling—all have one thing in common. They are all simple metaphors for the need to separate the conflation of concepts embedded into every database we use, to decouple them so that we can manage them separately and hence efficiently.

There are a number of reasons for wanting to do this, but maybe the most important of all is that it lets us build larger and more functionally diverse systems. So while a database-centric approach works wonderfully for individual applications, we don't live in a world of individual applications. We live in a

world of interconnected systems—individual components that, while all valuable in themselves, are really part of a much larger puzzle. We need a mechanism for sharing data that complements this complex, interconnected world. Events lead us to this. They constantly push data into our applications. These applications react, blending streams together, building views, changing state, and moving themselves forward. In the streaming model there is no shared database. The database is the event stream, and the application simply molds it into something new.

In fairness, streaming systems still have database-like attributes such as tables (for lookups) and transactions (for atomicity), but the approach has a radically different feel, more akin to functional or dataflow languages (and there is much cross-pollination between the streaming and functional programming communities).

So when it comes to data, we should be unequivocal about the shared facts of our system. They are the very essence of our business, after all. Facts may be evolved over time, applied in different ways, or even recast to different contexts, but they should always tie back to a single thread of irrevocable truth, one from which all others are derived—a central nervous system that underlies and drives every modern digital business.

This book looks quite specifically at the application of Apache Kafka to this problem. In Part I we introduce streaming and take a look at how Kafka works. Part II focuses on the patterns and techniques needed to build event-driven programs: Event Sourcing, Event Collaboration, CQRS, and more. Part III takes these ideas a step further, applying them in the context of multiteam systems, including microservices and SOA, with a focus on event streams as a source of truth and the aforementioned idea that both systems and companies can be reimagined as a database turned inside out. In the final part, we take a slightly more practical focus, building a small streaming system using Kafka Streams (and KSQL).

CHAPTER 2
The Origins of Streaming

This book is about building business systems with stream processing tools, so it is useful to have an appreciation for where stream processing came from. The maturation of this toolset, in the world of real-time analytics, has heavily influenced the way we build event-driven systems today.

Figure 2-1 shows a stream processing system used to ingest data from several hundred thousand mobile devices. Each device sends small JSON messages to denote applications on each mobile phone that are being opened, being closed, or crashing. This can be used to look for instability—that is, where the ratio of crashes to usage is comparatively high.

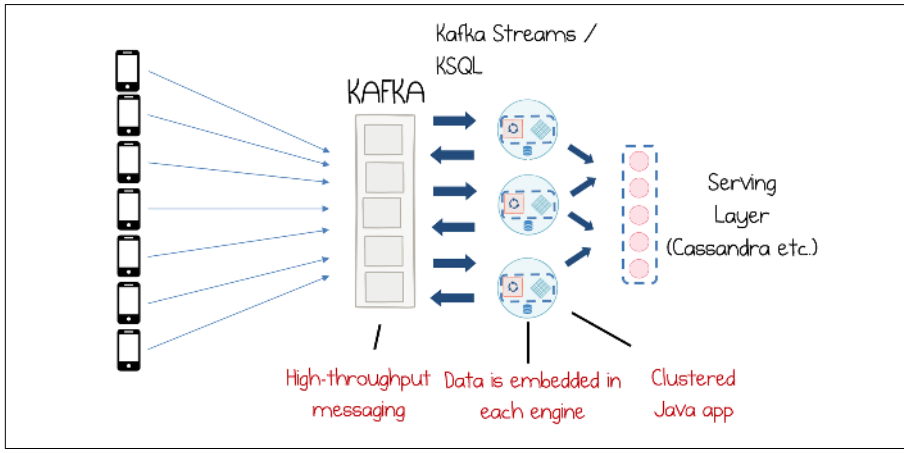


Figure 2-1. A typical streaming application that ingests data from mobile devices into Kafka, processes it in a streaming layer, and then pushes the result to a serving layer where it can be queried

The mobile devices land their data into Kafka, which buffers it until it can be extracted by the various applications that need to put it to further use. For this type of workload the cluster would be relatively large; as a ballpark figure Kafka ingests data at network speed, but the overhead of replication typically divides that by three (so a three-node 10 GbE cluster will ingest around 1 GB/s in practice).

To the right of Kafka in Figure 2-1 sits the stream processing layer. This is a clustered application, where queries are either defined up front via the Java DSL or sent dynamically via KSQL, Kafka's SQL-like stream processing language. Unlike in a traditional database, these queries compute continuously, so every time an input arrives in the stream processing layer, the query is recomputed, and a result is emitted if the value of the query has changed.

Once a new message has passed through all streaming computations, the result lands in a serving layer from which it can be queried. Cassandra is shown in Figure 2-1, but pushing to HDFS (Hadoop Distributed File System), pushing to another datastore, or querying directly from Kafka Streams using its interactive queries feature are all common approaches as well.

To understand streaming better, it helps to look at a typical query. Figure 2-2 shows one that computes the total number of app crashes per day. Every time a new message comes in, signifying that an application crashed, the count of total crashes for that application will be incremented. Note that this computation requires state: the count for the day so far (i.e., within the window duration) must be stored so that, should the stream processor crash/restart, the count will continue where it was before. Kafka Streams and KSQL manage this state internally, and that state is backed up to Kafka via a *changelog topic*. This is discussed in more detail in "Windows, Joins, Tables, and State Stores" on page 135 in Chapter 14.

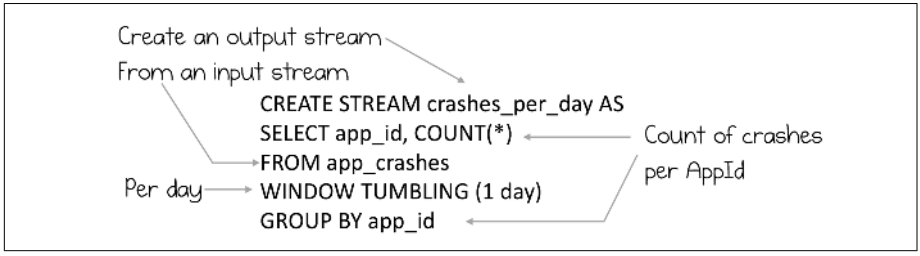


Figure 2-2. A simple KSQL query that evaluates crashes per day

Multiple queries of this type can be chained together in a pipeline. In Figure 2-3, we break the preceding problem into three steps chained over two stages. Queries (a) and (b) continuously compute apps opened per day and apps crashed per day, respectively. The two resulting output streams are combined together in the

final stage (c), which computes application stability by calculating the ratio between crashes and usage and comparing it to a fixed bound.

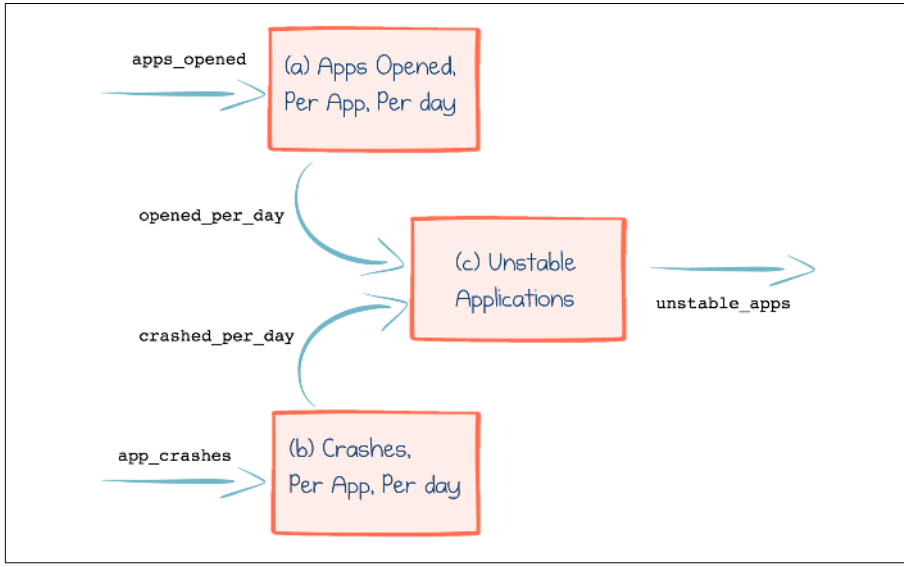


Figure 2-3. Two initial stream processing queries are pushed into a third to create a pipeline

There are a few other things to note about this streaming approach:

The streaming layer is fault-tolerant
: It runs as a cluster on all available nodes. If one node exits, another will pick up where it left off. Likewise, you can scale out the cluster by adding new processing nodes. Work, and any required state, will automatically be rerouted to make use of these new resources.

Each stream processor node can hold state of its own
: This is required for buffering as well as holding whole tables, for example, to do enrichments (streams and tables are discussed in more detail in "Windows, Joins, Tables, and State Stores" on page 135 in Chapter 14). This idea of local storage is important, as it lets the stream processor perform fast, message-at-a-time queries without crossing the network—a necessary feature for the high-velocity workloads seen in internet-scale use cases. But this ability to internalize state in local stores turns out to be useful for a number of business-related use cases too, as we discuss later in this book.

Each stream processor can write and store local state
: Making message-at-a-time network calls isn't a particularly good idea when you're handling a high-throughput event stream. For this reason stream processors write data locally (so writes and reads are fast) and back those writes

up to Kafka. So, for example, the aforementioned count requires a running total to be tracked so that, should a crash and restart occur, the computation resumes from its previous position and the count remains accurate. This ability to store data locally is very similar conceptually to the way you might interact with a database in a traditional application. But unlike in a traditional two-tier application, where interacting with the database means making a network call, in stream processing all the state is local (you might think of it as a kind of cache), so it is fast to access—no network calls needed. Because it is also flushed back to Kafka, it inherits Kafka's durability guarantees. We discuss this in more detail in "Scaling Concurrent Operations in Streaming Systems" on page 142 in Chapter 15.

CHAPTER 3

Is Kafka What You Think It Is?

There is an old parable about an elephant and a group of blind men. None of the men had come across an elephant before. One blind man approaches the leg and declares, "It's like a tree." Another man approaches the tail and declares, "It's like a rope." A third approaches the trunk and declares, "It's like a snake." So each blind man senses the elephant from his particular point of view, and comes to a subtly different conclusion as to what an elephant is. Of course the elephant is like all these things, but it is really just an elephant!

Likewise, when people learn about Kafka they often see it from a certain viewpoint. These perspectives are usually accurate, but highlight only some subsection of the whole platform. In this chapter we look at some common points of view.

Kafka Is Like REST but Asynchronous?

Kafka provides an asynchronous protocol for connecting programs together, but it is undoubtedly a bit different from, say, TCP (transmission control protocol), HTTP, or an RPC protocol. The difference is the presence of a *broker*. A broker is a separate piece of infrastructure that broadcasts messages to any programs that are interested in them, as well as storing them for as long as is needed. So it's perfect for streaming or fire-and-forget messaging.

Other use cases sit further from its home ground. A good example is request-response. Say you have a service for querying customer information. So you call a `getCustomer()` method, passing a `CustomerId`, and get a document describing a customer in the reply. You can build this type of request-response interaction with Kafka using two topics: one that transports the request and one that transports the response. People build systems like this, but in such cases the broker doesn't contribute all that much. There is no requirement for broadcast. There is

also no requirement for storage. So this leaves the question: would you be better off using a stateless protocol like HTTP?

So Kafka is a mechanism for programs to exchange information, but its home ground is event-based communication, where events are business facts that have value to more than one service and are worth keeping around.

Kafka Is Like a Service Bus?

If we consider Kafka as a messaging system—with its Connect interface, which pulls data from and pushes data to a wide range of interfaces and datastores, and streaming APIs that can manipulate data in flight—it does look a little like an ESB (enterprise service bus). The difference is that ESBs focus on the integration of legacy and off-the-shelf systems, using an ephemeral and comparably low-throughput messaging layer, which encourages request-response protocols (see the previous section).

Kafka, however, is a streaming platform, and as such puts emphasis on high-throughput events and stream processing. A Kafka cluster is a distributed system at heart, providing high availability, storage, and linear scale-out. This is quite different from traditional messaging systems, which are limited to a single machine, or if they do scale outward, those scalability properties do not stretch from end to end. Tools like Kafka Streams and KSQL allow you to write simple programs that manipulate events as they move and evolve. These make the processing capabilities of a database available in the application layer, via an API, and outside the confines of the shared broker. This is quite important.

ESBs are criticized in some circles. This criticism arises from the way the technology has been built up over the last 15 years, particularly where ESBs are controlled by central teams that dictate schemas, message flows, validation, and even transformation. In practice centralized approaches like this can constrain an organization, making it hard for individual applications and services to evolve at their own pace.

ThoughtWorks called this out recently, encouraging users to steer clear of recreating the issues seen in ESBs with Kafka. At the same time, the company encouraged users to investigate event streaming as a source of truth, which we discuss in Chapter 9. Both of these represent sensible advice.

So Kafka may look a little like an ESB, but as we'll see throughout this book, it is very different. It provides a far higher level of throughput, availability, and storage, and there are hundreds of companies routing their core facts through a single Kafka cluster. Beyond that, streaming encourages services to retain control, particularly of their data, rather than providing orchestration from a single, central team or platform. So while having one single Kafka cluster at the center of an organization is quite common, the pattern works because it is simple—nothing

more than data transfer and storage, provided at scale and high availability. This is emphasized by the core mantra of event-driven services: *Centralize an immutable stream of facts. Decentralize the freedom to act, adapt, and change.*

Kafka Is Like a Database?

Some people like to compare Kafka to a database. It certainly comes with similar features. It provides storage; production topics with hundreds of terabytes are not uncommon. It has a SQL interface that lets users define queries and execute them over the data held in the log. These can be piped into views that users can query directly. It also supports transactions. These are all things that sound quite "databasey" in nature!

So many of the elements of a traditional database are there, but if anything, Kafka is a database inside out (see "A Database Inside Out" on page 87 in Chapter 9), a tool for storing data, processing it in real time, and creating views. And while you are perfectly entitled to put a dataset in Kafka, run a KSQL query over it, and get an answer—much like you might in a traditional database—KSQL and Kafka Streams are optimized for continual computation rather than batch processing.

So while the analogy is not wholly inaccurate, it is a little off the mark. Kafka is designed to move data, operating on that data as it does so. It's about real-time processing first, long-term storage second.

What Is Kafka Really? A Streaming Platform

As Figure 3-1 illustrates, Kafka is a streaming platform. At its core sits a cluster of Kafka brokers (discussed in detail in Chapter 4). You can interact with the cluster through a wide range of client APIs in Go, Scala, Python, REST, and more.

There are two APIs for stream processing: Kafka Streams and KSQL (which we discuss in Chapter 14). These are database engines for data in flight, allowing users to filter streams, join them together, aggregate, store state, and run arbitrary functions over the evolving dataflow. These APIs can be stateful, which means they can hold data tables much like a regular database (see "Making Services Stateful" on page 47 in Chapter 6).

The third API is Connect. This has a whole ecosystem of connectors that interface with different types of database or other endpoints, both to pull data from and push data to Kafka. Finally there is a suite of utilities—such as Replicator and Mirror Maker, which tie disparate clusters together, and the Schema Registry, which validates and manages schemas—applied to messages passed through Kafka and a number of other tools in the Confluent platform.

A streaming platform brings these tools together with the purpose of turning data at rest into data that flows through an organization. The analogy of a central

nervous system is often used. The broker's ability to scale, store data, and run without interruption makes it a unique tool for connecting many disparate applications and services across a department or organization. The Connect interface makes it easy to evolve away from legacy systems, by unlocking hidden datasets and turning them into event streams. Stream processing lets applications and services embed logic directly over these resulting streams of events.

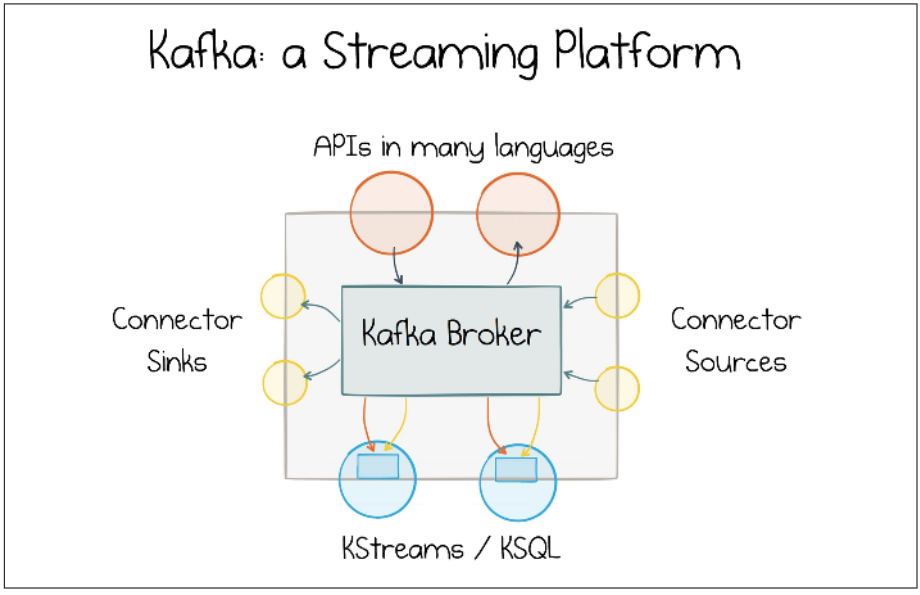


Figure 3-1. The core components of a streaming platform

CHAPTER 4

Beyond Messaging: An Overview of the Kafka Broker

A Kafka cluster is essentially a collection of files, filled with messages, spanning many different machines. Most of Kafka's code involves tying these various individual logs together, routing messages from producers to consumers reliably, replicating for fault tolerance, and handling failure gracefully. So it is a messaging system, at least of sorts, but it's quite different from the message brokers that preceded it. Like any technology, it comes with both pros and cons, and these shape the design of the systems we write. This chapter examines the Kafka broker (i.e., the server component) from the context of building business systems. We'll explore a little about how it works, as well as dipping into the less conventional use cases it supports like data storage, dynamic failover, and bandwidth protection.

Originally built to distribute the datasets created by large social networks, Kafka was predominantly shaped by a need to operate at scale, in the face of failure. Accordingly, its architecture inherits more from storage systems like HDFS, HBase, or Cassandra than it does from traditional messaging systems that implement JMS (Java Message Service) or AMQP (Advanced Message Queuing Protocol).

Like many good outcomes in computer science, this scalability comes largely from simplicity. The underlying abstraction is a partitioned log—essentially a set of append-only files spread over a number of machines—which encourages sequential access patterns that naturally flow with the grain of the underlying hardware.

A Kafka cluster is a distributed system, spreading data over many machines both for fault tolerance and for linear scale-out. The system is designed to handle a range of use cases, from high-throughput streaming, where only the latest mes-

sages matter, to mission-critical use cases where messages and their relative ordering must be preserved with the same guarantees as you'd expect from a DBMS (database management system) or storage system. The price paid for this scalability is a slightly simpler contract that lacks some of the obligations of JMS or AMQP, such as message selectors.

But this change of tack turns out to be quite important. Kafka's throughput properties make moving data from process to process faster and more practical than with previous technologies. Its ability to store datasets removes the queue-depth problems that plagued traditional messaging systems. Finally, its rich APIs, particularly Kafka Streams and KSQL, provide a unique mechanism for embedding data processing directly inside client programs. These attributes have led to its use as a message and storage backbone for service estates in a wide variety of companies that need all of these capabilities.

The Log: An Efficient Structure for Retaining and Distributing Messages

At the heart of the Kafka messaging system sits a partitioned, replayable log. The log-structured approach is itself a simple idea: a collection of messages, appended sequentially to a file. When a service wants to read messages from Kafka, it "seeks" to the position of the last message it read, then scans sequentially, reading messages in order while periodically recording its new position in the log (see Figure 4-1).

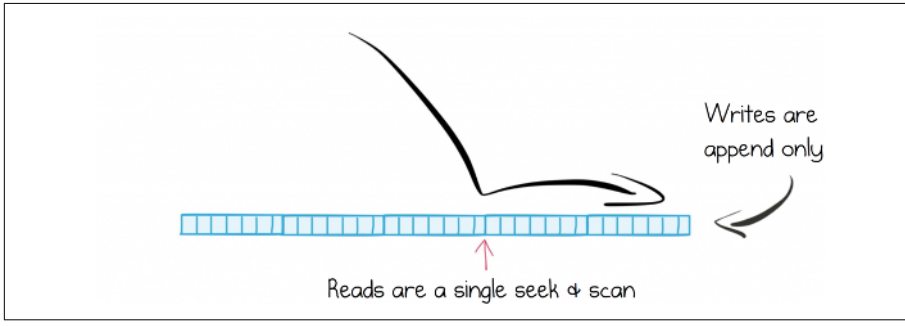


Figure 4-1. A log is an append-only journal

Taking a log-structured approach has an interesting side effect. Both reads and writes are sequential operations. This makes them sympathetic to the underlying media, leveraging prefetch, the various layers of caching, and naturally batching operations together. This in turn makes them efficient. In fact, when you read messages from Kafka, the server doesn't even import them into the JVM (Java virtual machine). Data is copied directly from the disk buffer to the network

buffer (zero copy)—an opportunity afforded by the simplicity of both the contract and the underlying data structure.

So batched, sequential operations help with overall performance. They also make the system well suited to storing messages longer term. Most traditional message brokers are built with index structures—hash tables or B-trees—used to manage acknowledgments, filter message headers, and remove messages when they have been read. But the downside is that these indexes must be maintained, and this comes at a cost. They must be kept in memory to get good performance, limiting retention significantly. But the log is O(1) when either reading or writing messages to a partition, so whether the data is on disk or cached in memory matters far less.

There are a few implications to this log-structured approach. If a service has some form of outage and doesn't read messages for a long time, the backlog won't cause the infrastructure to slow significantly (a common problem with traditional brokers, which have a tendency to slow down as they get full). Being log-structured also makes Kafka well suited to performing the role of an event store, for those who like to apply Event Sourcing within their services. This subject is discussed in depth in Chapter 7.

Partitions and Partitioning

Partitions are a fundamental concept for most distributed data systems. A partition is just a bucket that data is put into, much like buckets used to group data in a hash table. In Kafka's terminology each *log* is a *replica* of a *partition* held on a different machine. (So one partition might be replicated three times for high availability. Each replica is a separate log with the same data inside it.) What data goes into each partition is determined by a partitioner, coded into the Kafka producer. The partitioner will either spread data across the available partitions in a round-robin fashion or, if a key is provided with the message, use a hash of the key to determine the partition number. This latter point ensures that messages with the same key are always sent to the same partition and hence are strongly ordered.

Linear Scalability

As we've discussed, logs provide a hardware-sympathetic data structure for messaging workloads, but Kafka is really many logs, spanning many different machines. The system ties these together, routing messages reliably, replicating for fault tolerance, and handling failure gracefully.

While running on a single machine is possible, production clusters typically start at three machines with larger clusters in the hundreds. When you read and write

to a topic, you'll typically be reading and writing to all of them, partitioning your data over all the machines you have at your disposal. Scaling is thus a pretty simple affair: add new machines and rebalance. Consumption can also be performed in parallel, with messages in a topic being spread over several consumers in a consumer group (see Figure 4-2).

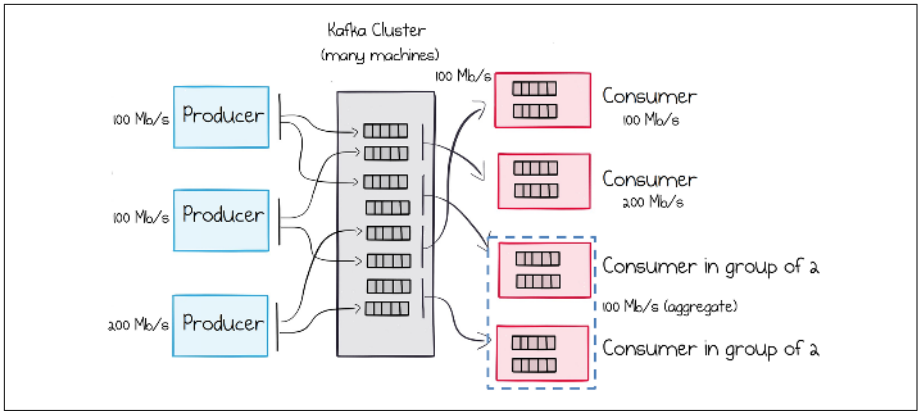


Figure 4-2. Producers spread messages over many partitions, on many machines, where each partition is a little queue; load-balanced consumers (denoted a consumer group) share the partitions between them; rate limits are applied to producers, consumers, and groups

The main advantage of this, from an architectural perspective, is that it takes the issue of scalability off the table. With Kafka, hitting a scalability wall is virtually impossible in the context of business systems. This can be quite empowering, especially when ecosystems grow, allowing implementers to pick patterns that are a little more footloose with bandwidth and data movement.

Scalability opens other opportunities too. Single clusters can grow to company scales, without the risk of workloads overpowering the infrastructure. For example, New Relic relies on a single cluster of around 100 nodes, spanning three datacenters, and processing 30 GB/s. In other, less data-intensive domains, 5- to 10-node clusters commonly support whole-company workloads. But it should be noted that not all companies take the "one big cluster" route. Netflix, for example, advises using several smaller clusters to reduce the operational overheads of running very large installations, but their largest installation is still around the 200-node mark.

To manage shared clusters, it's useful to carve bandwidth up, using the bandwidth segregation features that ship with Kafka. We'll discuss these next.

Segregating Load in Multiservice Ecosystems

Service architectures are by definition multitenant. A single cluster will be used by many different services. In fact, it's not uncommon for all services in a company to share a single production cluster. But doing so opens up the potential for inadvertent denial-of-service attacks, causing service degradation or instability.

To help with this, Kafka includes a throughput control feature, called quotas, that allows a defined amount of bandwidth to be allocated to specific services, ensuring that they operate within strictly enforced service-level agreements, or SLAs (see Figure 4-2). Greedy services are aggressively throttled, so a single cluster can be shared by any number of services without the fear of unexpected network contention. This feature can be applied to either individual service instances or load-balanced groups.

Maintaining Strong Ordering Guarantees

While it often isn't the case for analytics use cases, most business systems need strong ordering guarantees. Say a customer makes several updates to their customer information. The order in which these updates are processed is going to matter, or else the latest change might be overwritten with one of the older, out-of-date values.

There are a couple of things that need to be considered to ensure strong ordering guarantees. The first is that messages that require relative ordering need to be sent to the same partition. (Kafka provides ordering guarantees only within a partition.) This is managed for you: you supply the same key for all messages that require a relative order. So a stream of customer information updates would use the `CustomerId` as their partitioning key. All messages for the same customer would then be routed to the same partition, and hence be strongly ordered (see Figure 4-3).

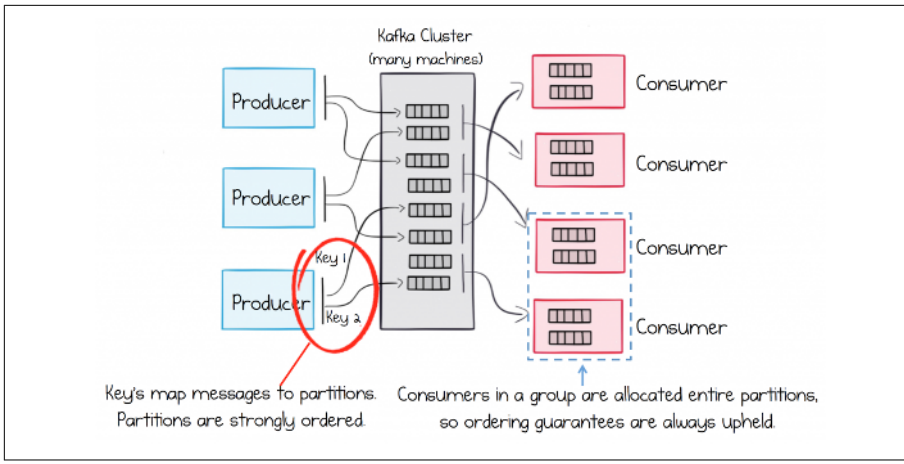


Figure 4-3. Ordering in Kafka is specified by producers using an ordering key

Sometimes key-based ordering isn't enough, and global ordering is required. This often comes up when you're migrating from legacy messaging systems where global ordering was an assumption of the original system's design. To maintain global ordering, use a single partition topic. Throughput will be limited to that of a single machine, but this is typically sufficient for use cases of this type.

The second thing to be aware of is retries. In almost all cases we want to enable retries in the producer so that if there is some network glitch, long-running garbage collection, failure, or the like, any messages that aren't successfully sent to the cluster will be retried. The subtlety is that messages are sent in batches, so we should be careful to send these batches one at a time, per destination machine, so there is no potential for a reordering of events when failures occur and batches are retried. This is simply something we configure.

Ensuring Messages Are Durable

Kafka provides durability through replication. This means messages are written to a configurable number of machines so that if one or more of those machines fail, the messages will not be lost. If you configure a replication factor of three, two machines can be lost without losing data.

To make best use of replication, for sensitive datasets like those seen in service-based applications, configure three replicas for each partition and configure the producer to wait for replication to complete before proceeding. Finally, as discussed earlier, configure retries in the producer.

Highly sensitive use cases may require that data be flushed to disk synchronously, but this approach should be used sparingly. It will have a significant

impact on throughput, particularly in highly concurrent environments. If you do take this approach, increase the producer batch size to increase the effectiveness of each disk flush on the machine (batches of messages are flushed together). This approach is useful for single machine deployments, too, where a single ZooKeeper node is run on the same machine and messages are flushed to disk synchronously for resilience.

Load-Balance Services and Make Them Highly Available

Event-driven services should always be run in a highly available (HA) configuration, unless there is genuinely no requirement for HA. The main reason for this is it's essentially a no-op. If we have one instance of a service, then start a second, load will naturally balance across the two. The same process provides high availability should one node crash (see Figure 4-4).

Say we have two instances of the orders service, reading messages from the Orders topic. Kafka would assign half of the partitions to each instance, so the load is spread over the two.

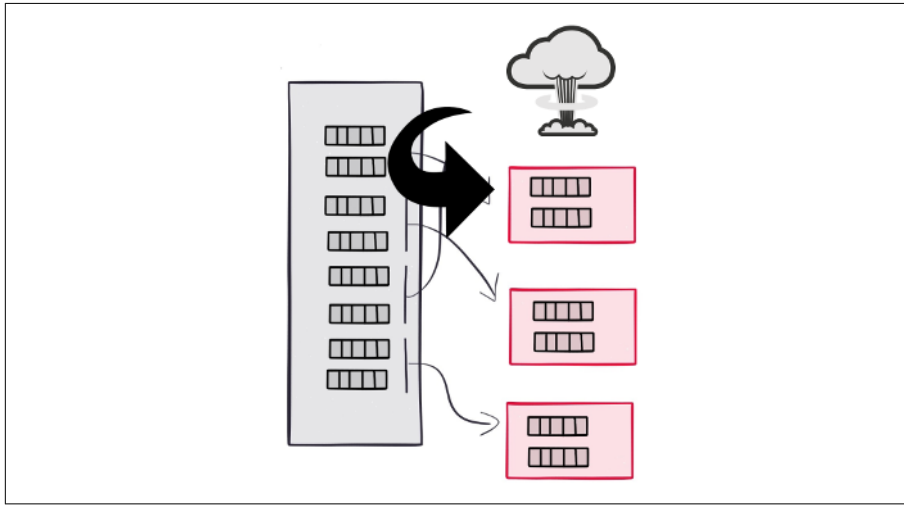


Figure 4-4. If an instance of a service dies, data is redirected and ordering guarantees are maintained

Should one of the services fail, Kafka will detect this failure and reroute messages from the failed service to the one that remains. If the failed service comes back online, load flips back again.

This process actually works by assigning whole partitions to different consumers. A strength of this approach is that a single partition can only ever be assigned to

a single service instance (consumer). This is an invariant, implying that ordering is guaranteed, even as services fail and restart.

So services inherit both high availability and load balancing, meaning they can scale out, handle unplanned outages, or perform rolling restarts without service downtime. In fact, Kafka releases are always backward-compatible with the previous version, so you are guaranteed to be able to release a new version without taking your system offline.

Compacted Topics

By default, topics in Kafka are retention-based: messages are retained for some configurable amount of time. Kafka also ships with a special type of topic that manages keyed datasets—that is, data that has a primary key (identifier) as you might have in a database table. These *compacted topics* retain only the most recent events, with any old events, for a certain key, being removed. They also support deletes (see "Deleting Data" on page 127 in Chapter 13).

Compacted topics work a bit like simple log-structure merge-trees (LSM trees). The topic is scanned periodically, and old messages are removed if they have been superseded (based on their key); see Figure 4-5. It's worth noting that this is an asynchronous process, so a compacted topic may contain some superseded messages, which are waiting to be compacted away.

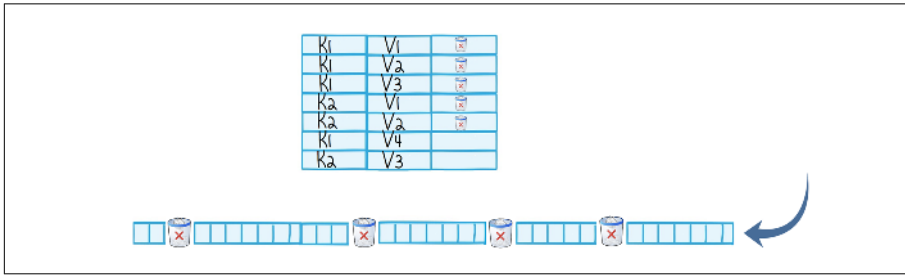


Figure 4-5. In a compacted topic, superseded messages that share the same key are removed. So, in this example, for key K2, messages V2 and V1 would eventually be compacted as they are superseded by V3.

Compacted topics let us make a couple of optimizations. First, they help us slow down a dataset's growth (by removing superseded events), but we do so in a data-specific way rather than, say, simply removing messages older than two weeks. Second, having smaller datasets makes it easier for us to move them from machine to machine.

This is important for stateful stream processing. Say a service uses the Kafka's Streams API to load the latest version of the product catalogue into a table (as discussed in "Windows, Joins, Tables, and State Stores" on page 135 in Chapter 14,

a table is a disk resident hash table held inside the API). If the product catalogue is stored in a compacted topic in Kafka, the load can be performed quicker and more efficiently if it doesn't have to load the whole versioned history as well (as would be the case with a regular topic).

Long-Term Data Storage

One of the bigger differences between Kafka and other messaging systems is that it can be used as a storage layer. In fact, it's not uncommon to see retention-based or compacted topics holding more than 100 TB of data. But Kafka isn't a database; it's a commit log offering no broad query functionality (and there are no plans for this to change). But its simple contract turns out to be quite useful for storing shared datasets in large systems or company architectures—for example, the use of events as a shared source of truth, as we discuss in Chapter 9.

Data can be stored in regular topics, which are great for audit or Event Sourcing, or compacted topics, which reduce the overall footprint. You can combine the two, getting the best of both worlds at the price of additional storage, by holding both and linking them together with a Kafka Streams job. This pattern is called the *latest-versioned pattern*.

Security

Kafka provides a number of enterprise-grade security features for both authentication and authorization. Client authentication is provided through either Kerberos or Transport Layer Security (TLS) client certificates, ensuring that the Kafka cluster knows who is making each request. There is also a Unix-like permissions system, which can be used to control which users can access which data. Network communication can be encrypted, allowing messages to be securely sent across untrusted networks. Finally, administrators can require authentication for communication between Kafka and ZooKeeper.

The quotas mechanism, discussed in the section "Segregating Load in Multiservice Ecosystems" on page 21, can be linked to this notion of identity, and Kafka's security features are extended across the different components of the Confluent platform (the Rest Proxy, Confluent Schema Registry, Replicator, etc.).

Summary

Kafka is a little different from your average messaging technology. Being designed as a distributed, scalable infrastructure component makes it an ideal backbone through which services can exchange and buffer events. There are obviously a number of elements unique to the technology itself, but the ones that

stand out are its abilities to scale, to run always on, and to retain datasets long-term.

We can use the patterns and features discussed in this chapter to build a wide variety of architectures, from fine-grained service-based systems right up to hulking corporate conglomerates. This is an approach that is safe, pragmatic, and tried and tested.

PART II

Designing Event-Driven Systems

Life is a series of natural and spontaneous changes. Don't resist them—that only creates sorrow. Let reality be reality. Let things flow naturally forward.

—Lao-Tzu, 6th–5th century BCE

CHAPTER 5

Events: A Basis for Collaboration

Service-based architectures, like microservices or SOA, are commonly built with synchronous request-response protocols. This approach is very natural. It is, after all, the way we write programs: we make calls to other code modules, await a response, and continue. It also fits closely with a lot of use cases we see each day: front-facing websites where users hit buttons and expect things to happen, then return.

But when we step into a world of many independent services, things start to change. As the number of services grows gradually, the web of synchronous interactions grows with them. Previously benign availability issues start to trigger far more widespread outages. Our ops engineers often end up as reluctant detectives, playing out distributed murder mysteries as they frantically run from service to service, piecing together snippets of secondhand information. (Who said what, to whom, and when?)

This is a well-known problem, and there are a number of solutions. One is to ensure each individual service has a significantly higher SLA than your system as a whole. Google provides a protocol for doing this. An alternative is to simply break down the synchronous ties that bind services together using (a) asynchronicity and (b) a message broker as an intermediary.

Say you are working in online retail. You would probably find that synchronous interfaces like `getImage()` or `processOrder()`—calls that expect an immediate response—feel natural and familiar. But when a user clicks Buy, they actually trigger a large, complex, and asynchronous process into action. This process takes a purchase and physically ships it to the user's door, way beyond the context of the original button click. So splitting software into asynchronous flows allows us to compartmentalize the different problems we need to solve and embrace a world that is itself inherently asynchronous.

In practice we tend to embrace this automatically. We've all found ourselves polling database tables for changes, or implementing some kind of scheduled cron job to churn through updates. These are simple ways to break the ties of synchronicity, but they always feel like a bit of a hack. There is a good reason for this: they probably are.

So we can condense all these issues into a single observation. The imperative programming model, where we command services to do our bidding, isn't a great fit for estates where services are operated independently.

In this chapter we're going to focus on the other side of the architecture coin: composing services not through chains of commands and queries, but rather through streams of events. This is an implementation pattern in its own right, and has been used in industry for many years, but it also forms a baseline for the more advanced patterns we'll be discussing in Part III and Part V, where we blend the ideas of event-driven processing with those seen in streaming platforms.

Commands, Events, and Queries

Before we go any further, consider that there are three distinct ways that programs can interact over a network: commands, events, and queries. If you've not considered the distinction between these three before, it's well worth doing so, as it provides an important reference for interprocess communication.

The three mechanisms through which services interact can be described as follows (see Table 5-1 and Figure 5-1):

Commands
: Commands are actions—requests for some operation to be performed by another service, something that will change the state of the system. Commands execute synchronously and typically indicate completion, although they may also include a result.[1]

  - Example: `processPayment()`, returning whether the payment succeeded.

1 The term *command* originally came from Bertrand Meyer's CQS (Command Query Separation) principle. A slightly different definition from Bertrand's is used here, leaving it optional as to whether a command should return a result or not. There is a reason for this: a command is a request for something specific to happen in the future. Sometimes it is desirable to have no return value; other times, a return value is important. Martin Fowler uses the example of popping a stack, while here we use the example of processing a payment, which simply returns whether the command succeeded. By leaving the command with an optional return type, the implementer can decide if it should return a result or not, and if not CQS/CQRS may be used. This saves the need for having another name for a command that *does* return a result. Finally, a command is never an event. A command has an explicit expectation that something (a state change or side effect) will happen in the future. Events come with no such future expectation. They are simply a statement that something happened.

- When to use: On operations that must complete synchronously, or when using orchestration or a process manager. Consider restricting the use of commands to inside a bounded context.

Events

Events are both a fact and a notification. They represent something that *happened* in the real world but include no expectation of any future action. They travel in only one direction and expect no response (sometimes called "fire and forget"), but one may be "synthesized" from a subsequent event.

- Example: `OrderCreated{`*`Widget`*`}`, `CustomerDetailsUpdated{`*`Customer`*`}`
- When to use: When loose coupling is important (e.g., in multiteam systems), where the event stream is useful to more than one service, or where data must be replicated from one application to another. Events also lend themselves to concurrent execution.

Queries

Queries are a request to look something up. Unlike events or commands, queries are free of side effects; they leave the state of the system unchanged.

- Example: `getOrder(ID=42) returns Order(42,…)`.
- When to use: For lightweight data retrieval across service boundaries, or heavyweight data retrieval within service boundaries.

Table 5-1. Differences between commands, events, and queries

	Behavior/state change	Includes a response
Command	Requested to happen	Maybe
Event	Just happened	Never
Query	None	Always

The beauty of events is they wear two hats: a notification hat that triggers services into action, but also a replication hat that copies data from one service to another. But from a services perspective, events lead to less coupling than commands and queries. Loose coupling is a desirable property where interactions cross deployment boundaries, as services with fewer dependencies are easier to change.

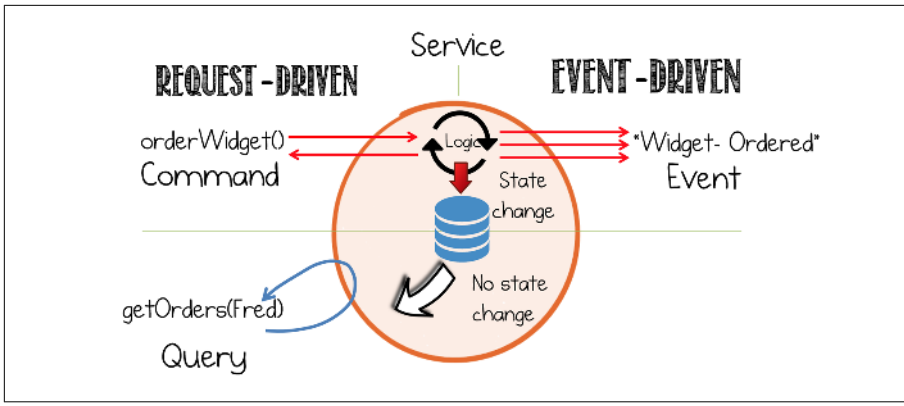


Figure 5-1. A visual summary of commands, events, and queries

Coupling and Message Brokers

The term *loose coupling* is used widely. It was originally a design heuristic for structuring programs, but when applied to network-attached services, particularly those run by different teams, it must be interpreted slightly differently. Here is a relatively recent definition from Frank Leymann:

> Loose coupling reduces the number of assumptions two parties make about one another when they exchange information.

These assumptions broadly relate to a combination of data, function, and operability. As it turns out, however, this isn't what most people mean when they use the term *loose coupling*. When people refer to a loosely coupled application, they usually mean something closer to *connascence*, defined as follows:[2]

> A measure of the impact a change to one component will have on others.

This captures the intuitive notion of coupling: that if two entities are coupled, then an action applied to one will result in some action applied to the other. But an important part of this definition of connascence is the word *change*, which implies a temporal element. Coupling isn't a static thing; it matters only in the very instant that we try to change our software. In fact, if we left our software alone, and never changed it, coupling wouldn't matter at all.

Is Loose Coupling Always Good?

There is a widely held sentiment in industry that tight coupling is bad and loose coupling is good. This is not wholly accurate. Both tight and loose coupling are

2 See *https://en.wikipedia.org/wiki/Connascence* and *http://wiki.cfcl.com/pub/Projects/Connascence/Resources/p147-page-jones.pdf*.

actually pretty useful in different situations. We might summarize the relationship as:

> Loose coupling lets components change independently of one another. Tight coupling lets components extract more value from one another.

The path to loose coupling is not to share. If you don't share anything, then other applications can't couple to you. Microservices, for example, are sometimes referred to as "shared nothing,"[3] encouraging different teams not to share data and not to share functionality (across service boundaries), as it impedes their ability to operate independently.[4]

Of course, the problem with not sharing is it's not very collaborative; you inevitably end up reinventing the wheel or forcing others to. So while it may be convenient for you, it's probably not so good for the department or company you work in. Somewhat unsurprisingly, sensible approaches strike a balance. Most business applications *have to* share data with one another, so there is always some level of coupling. Shared functionality, be it services like DNS or payment processing, can be valuable, as can shared code libraries. So tighter coupling can of course be a good thing, but we have to be aware that it is a tradeoff. Sharing always increases the coupling on whatever we decide to share.

Sharing always increases the coupling on whatever we decide to share.

As an example, in most traditional applications, you *couple tightly* to your database and your application will extract as much value as possible from the database's ability to perform data-intensive operations. There is little downside, as the application and database will change together, and you don't typically let other systems use your database. A different example is DNS, used widely across an organization. In this case its wide usage makes it deeply valuable, but also tightly coupled. But as it changes infrequently and has a thin interface, there is little practical downside.

So we can observe that the coupling of a single component is really a function of three factors, with an addendum:

---

3 "Shared nothing" is also used in the database world but to mean a slightly different thing.

4 As an anecdote, I once worked with a team that would encrypt sections of the information they published, not so it was secure, but so they could control who could couple to it (by explicitly giving the other party the encryption key). I wouldn't recommend this practice, but it makes the point that people really care about this problem.

- Interface surface area (functionality offered, breadth and quantity of data exposed)
- Number of users
- Operational stability and performance

The addendum: Frequency of change—that is, if a component doesn't change (be it data, function, or operation), then coupling (i.e., connascence) doesn't matter.

Messaging helps us build loosely coupled services because it moves pure data from a highly coupled place (the source) and puts it into a loosely coupled place (the subscriber). So any operations that need to be performed on that data are *not* done at source, but rather in each subscriber, and messaging technologies like Kafka take most of the operational stability/performance issues off the table.

On the other hand, request-driven approaches are more tightly coupled as functionality, data, and operational factors are concentrated in a single place. Later in this chapter we discuss the idea of a *bounded context*, which is a way of balancing these two: request-driven protocols used inside the bounded context, and messaging between them. We also discuss the wider consequences of coupling in some detail in Chapter 8.

Essential Data Coupling Is Unavoidable

All significantly sized businesses have core datasets that many programs need. If you are sending a user an email, you need their address; if you're building a sales report, you need sales figures; and so on. Data is something applications cannot do without, and there is no way to code around not having the data (while you might, for example, code around not having access to some piece of functionality). So pretty much all business systems, in larger organizations, need a base level of *essential data coupling*.

Functional couplings are optional. Core data couplings are essential.

Using Events for Notification

Most message brokers provide a publish-subscribe facility where the logic for how messages are routed is defined by the receivers rather than the senders; this process is known as *receiver-driven routing*. So the receiver retains control of their presence in the interaction, which makes the system pluggable (see Figure 5-2).

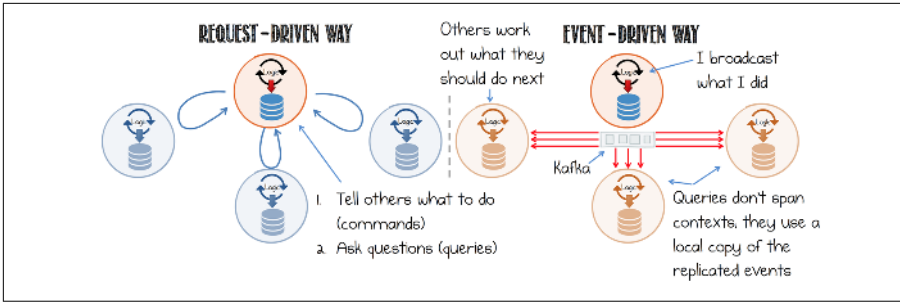


Figure 5-2. Comparison between the request-response and event-driven approaches demonstrating how event-driven approaches provide less coupling

Let's look at a simple example based on a customer ordering an iPad. The user clicks Buy, and an order is sent to the orders service. Three things then happen:

1. The shipping service is notified.
2. It looks up the address to send the iPad to.
3. It starts the shipping process.

In a REST- or RPC-based approach this might look like Figure 5-3.

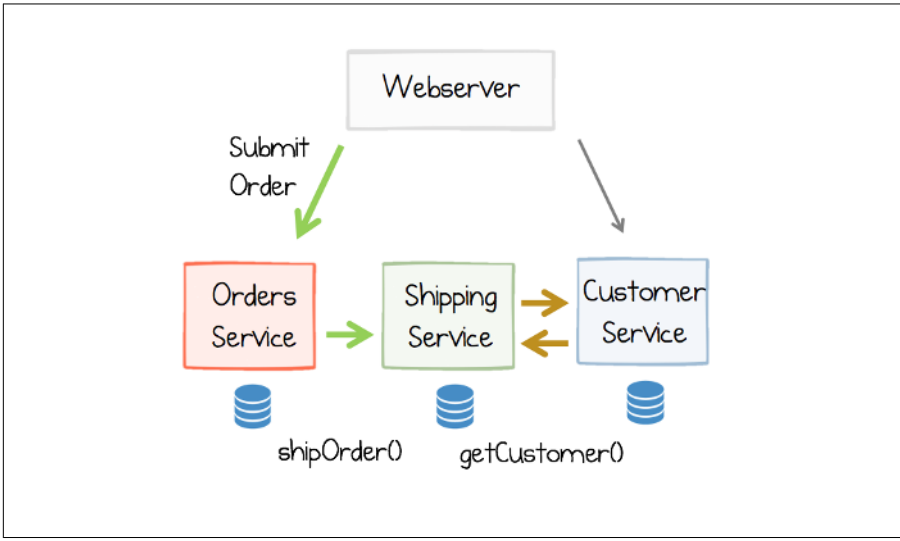


Figure 5-3. A request-driven order management system

The same flow can be built with an event-driven approach (Figure 5-4), where the orders service simply journals the event, "Order Created," which the shipping service then reacts to.

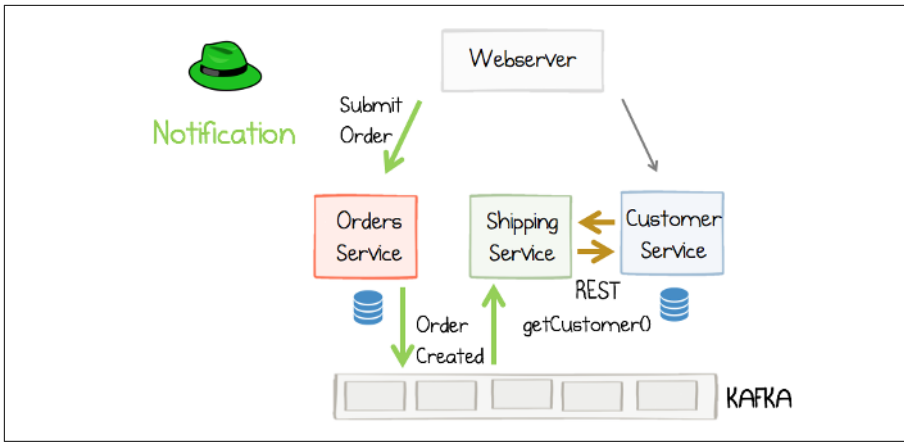


Figure 5-4. An event-driven version of the system described in Figure 5-3; in this configuration the events are used only as a means of notification: the orders service notifies the shipping service via Kafka

If we look closely at Figure 5-4, the interaction between the orders service and the shipping service hasn't changed all that much, other than that they communicate via events rather than calling each other directly. But there is an important change: the orders service has no knowledge that the shipping service exists. It just raises an event denoting that it did its job and an order was created. The shipping service now has control over whether it partakes in the interaction. This is an example of receiver-driven routing: logic for routing is located at the receiver of the events, rather than at the sender. The burden of responsibility is flipped! This reduces coupling and adds a useful level of pluggability to the system.

Pluggability becomes increasingly important as systems get more complex. Say we decide to extend our system by adding a repricing service, which updates the price of goods in real time, tweaking a product's price based on supply and demand (Figure 5-5). In a REST- or RPC-based approach we would need to introduce a `maybeUpdatePrice()` method, which is called by both the orders service and the payment service. But in the event-driven model, repricing is just a service that plugs into the event streams for orders and payments, sending out price updates when relevant criteria are met.

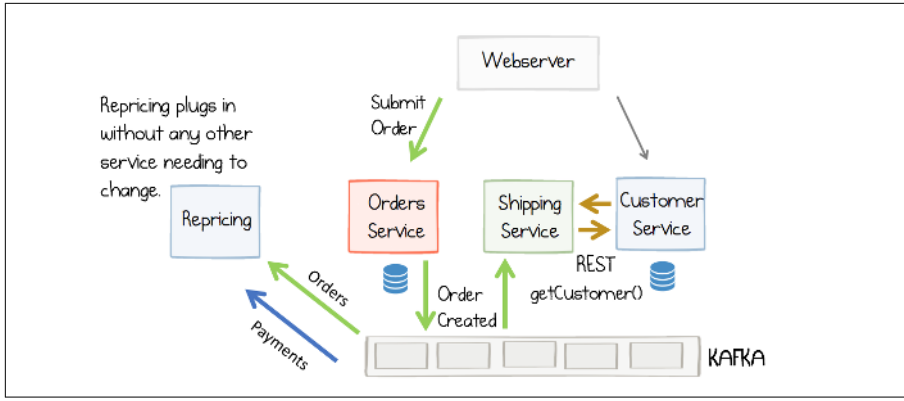


Figure 5-5. Extending the system described in Figure 5-4 by adding a repricing service to demonstrate the pluggability of the architecture

Using Events to Provide State Transfer

In Figure 5-5, we used events as a means of notification, but left the query for the customer's address as a REST/RPC call.

We can also use events as a type of state transfer so that, rather than sending the query to the customer service, we would use the event stream to replicate customer data from the customer service to the shipping service, where it can be queried locally (see Figure 5-6).

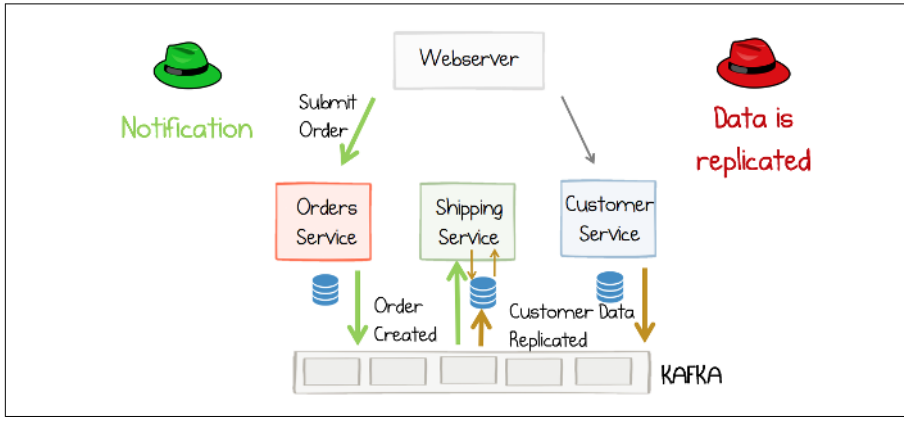


Figure 5-6. Extending the system described in Figure 5-4 to be fully event-driven; here events are used for notification (the orders service notifies the shipping service) as well as for data replication (data is replicated from the customer service to the shipping service, where it can be queried locally).

This makes use of the other property events have—their replication hat. (Formally this is termed event-carried state transfer, which is essentially a form of

data integration.) So the notification hat makes the architecture more pluggable, and the replication hat moves data from one service to another so queries can be executed locally. Replicating a dataset locally is advantageous in much the same way that caching is often advantageous, as it makes data access patterns faster.

Which Approach to Use

We can summarize the advantages of the pure "query by event-carried state transfer" approach as follows:

Better isolation and autonomy
: Isolation is required for autonomy. Keeping the data needed to drive queries isolated and local means it stays under the service's control.

Faster data access
: Local data is typically faster to access. This is particularly true when data from different services needs to be combined, or where the query spans geographies.

Where the data needs to be available offline
: In the case of a mobile device, ship, plane, train, or the like, replicating the dataset provides a mechanism for moving and resynchronizing when connected.

On the other hand, there are advantages to the REST/RPC approach:

Simplicity
: It's simpler to implement, as there are fewer moving parts and no state to manage.

Singleton
: State lives in only one place (inevitable caching aside!), meaning a value can be changed there and all users see it immediately. This is important for use cases that require synchronicity—for example, reading a previously updated account balance (we look at how to synthesize this property with events in "Collapsing CQRS with a Blocking Read" on page 142 in Chapter 15).

Centralized control
: Command-and-control workflows can be used to centralize business processes in a single controlling service. This makes it easier to reason about.

Of course, as we saw earlier, we can blend the two approaches together and, depending on which hat we emphasize, we get a solution that suits a differently sized architecture. If we're designing for a small, lightweight use case—like building an online application—we would put weight on the notification hat, as the weight of data replication might be considered an unnecessary burden. But in a larger and more complex architecture, we might place more emphasis on the

replication hat so that each service has greater autonomy over the data it queries. (This is discussed in more detail in Chapter 8.) Microservice applications tend to be larger and leverage both hats. Jonas Bonér puts this quite firmly:

> Communication *between* microservices needs to be based on asynchronous message passing (while logic *inside* each microservice is performed in a synchronous fashion).

Implementers should be careful to note that he directs this at a strict definition of microservices, one where services are independently deployable. Slacker interpretations, which are seen broadly in industry, may not qualify so strong an assertion.

The Event Collaboration Pattern

To build fine-grained services using events, a pattern called Event Collaboration is often used. This allows a set of services to collaborate around a single business workflow, with each service doing its bit by listening to events, then creating new ones. So, for example, we might start by creating an order, and then different services would evolve the workflow until the purchased item makes it to the user's door.

This might not sound too different from any other workflow, but what is special about Event Collaboration is that no single service owns the whole process; instead, each service owns a small part—some subset of state transitions—and these plug together through a chain of events. So each service does its work, then raises an event denoting what it did. If it processed a payment, it would raise a Payment Processed event. If it validated an order, it would raise Order Validated, and so on. These events trigger the next step in the chain (which could trigger that service again, or alternatively trigger another service).

In Figure 5-7 each circle represents an event. The color of the circle designates the topic it is in. A workflow evolves from Order Requested through to Order Completed. The three services (order, payment, shipping) handle the state transitions that pertain to their section of the workflow. Importantly, no service knows of the existence of any other service, and no service owns the entire workflow. For example, the payment service knows only that it must react to validated orders and create Payment Processed events, with the latter taking the workflow one step forward. So the currency of event collaboration is, unsurprisingly, events!

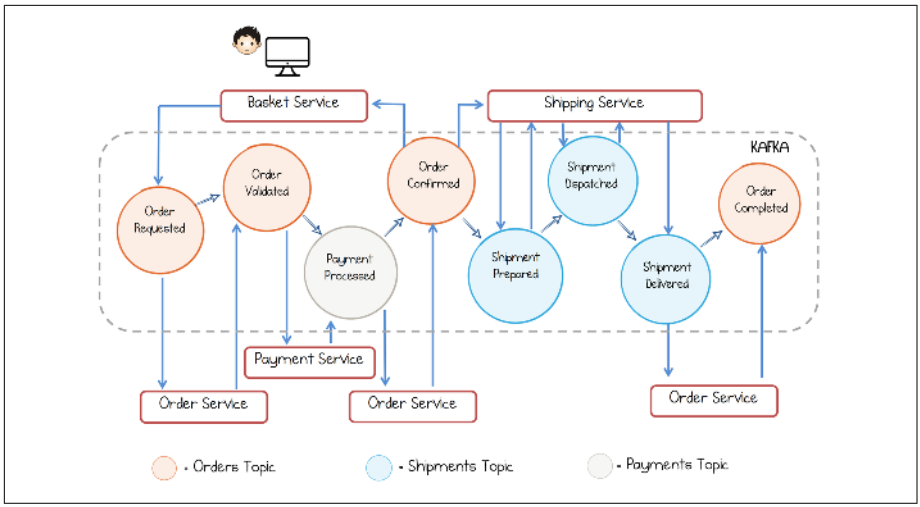


Figure 5-7. An example workflow implemented with Event Collaboration

The lack of any one point of central control means systems like these are often termed *choreographies*: each service handles some subset of state transitions, which, when put together, describe the whole business process. This can be contrasted with *orchestration*, where a single process commands and controls the whole workflow from one place—for example, via a process manager.[5] A process manager is implemented with request-response.

Choreographed systems have the advantage that they are pluggable. If the payment service decides to create three new event types for the payment part of the workflow, so long as the Payment Processed event remains, it can do so without affecting any other service. This is useful because it means if you're implementing a service, you can change the way you work and no other services need to know or care about it. By contrast, in an orchestrated system, where a single service dictates the workflow, all changes need to be made in the controller. Which of these approaches is best for you is quite dependent on use case, but the advantage of orchestration is that the whole workflow is written down, in code, in one place. That makes it easy to reason about the system. The downside is that the model is tightly coupled to the controller, so broadly speaking choreographed approaches better suit larger implementations (particularly those that span teams and hence change independently of one another).

5 See *http://www.enterpriseintegrationpatterns.com/patterns/messaging/ProcessManager.html* and *https://www.thoughtworks.com/insights/blog/scaling-microservices-event-stream*.

NOTE

The events service's share form a journal, or "shared narrative," describing exactly how your business evolved over time.

Relationship with Stream Processing

The notification and replication duality that events demonstrate maps cleanly to the concepts of stateless and stateful stream processing, respectively. The best way to understand this is to consider the shipping service example we discussed earlier in the chapter. If we changed the shipping service to use the Kafka Streams API, we could approach the problem in two ways (Figure 5-8):

Stateful approach
: Replicate the Customers table into the Kafka Streams API (denoted "KTable" in Figure 5-8). This makes use of the event-carried state transfer approach.

Stateless approach
: We process events and look up the appropriate customer with every order that is processed.

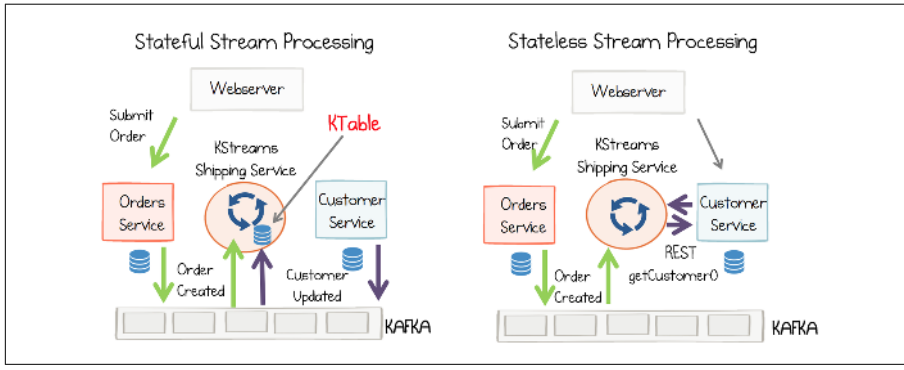


Figure 5-8. Stateful stream processing is similar to using events for both notification and state transfer (left), while stateless stream processing is similar to using events for notification (right)

So the use of event-carried state transfer, in stateful stream processing, differs in two important ways, when compared to the example we used earlier in this chapter:

- The dataset needs to be held, in its entirety, in Kafka. So if we are joining to a table of customers, all customer records must be stored in Kafka as events.

- The stream processor includes in-process, disk-resident storage to hold the table. There is no external database, and this makes the service stateful. Kafka Streams then applies a number of techniques to make managing this statefulness practical.

This topic is discussed in detail in Chapter 6.

Mixing Request- and Event-Driven Protocols

A common approach, particularly seen in smaller web-based systems, is to mix protocols, as shown in Figure 5-9. Online services interact directly with a user, say with REST, but also journal state changes to Kafka (see "Event Sourcing, Command Sourcing, and CQRS in a Nutshell" on page 55 in Chapter 7). Offline services (for billing, fulfillment, etc.) are built purely with events.

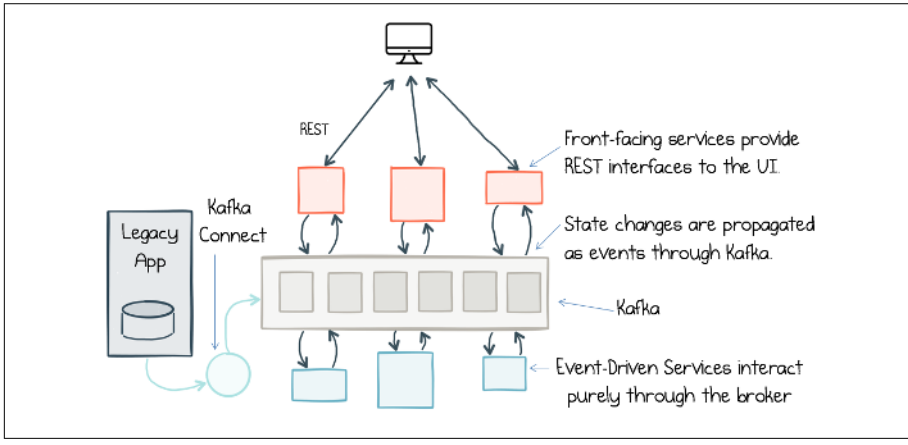


Figure 5-9. A very simple event-driven services example, data is imported from a legacy application via the Connect API; user-facing services provide REST APIs to the UI; state changes are journaled to Kafka as events. at the bottom, business processing is performed via Event Collaboration

In larger implementations, services tend to cluster together, for example within a department or team. They mix protocols inside one cluster, but rely on events to communicate between clusters (see Figure 5-10).

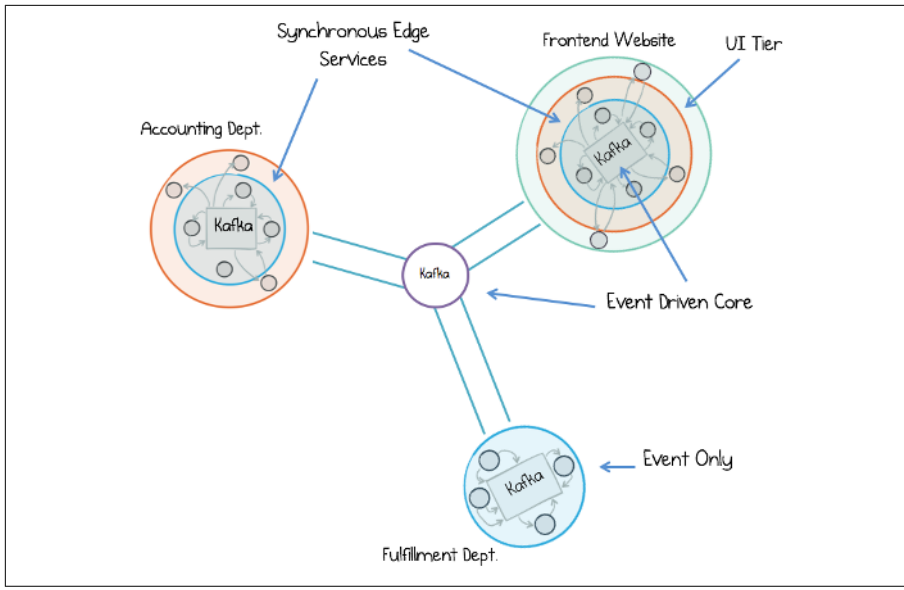


Figure 5-10. Clusters of services form bounded contexts within which functionality is shared. Contexts interact with one another only through events, spanning departments, geographies or clouds

In Figure 5-10 three departments communicate with one another *only* through events. Inside each department (the three larger circles), service interfaces are shared more freely and there are finer-grained event-driven flows that drive collaboration. Each department contains a number of internal bounded contexts—small groups of services that share a domain model, are usually deployed together, and collaborate closely. In practice, there is often a hierarchy of sharing. At the top of this hierarchy, departments are loosely coupled: the only thing they share is events. Inside a department, there will be many applications and those applications will interact with one another with both request-response and event-based mechanisms, as in Figure 5-9. Each application may itself be composed from several services, but these will typically be more tightly coupled to one another, sharing a domain model and having synchronized release schedules.

This approach, which confines reuse within a bounded context, is an idea that comes from domain-driven design, or DDD. One of the big ideas in DDD was that broad reuse could be counterproductive, and that a better approach was to create boundaries around areas of a business domain and model them separately. So within a bounded context the domain model is shared, and everything is available to everything else, but different bounded contexts don't share the same model, and typically interact through more restricted interfaces.

This idea was extended by microservice implementers, so a bounded context describes a set of closely related components or services that share code and are deployed together. Across bounded contexts there is less sharing (be it code, functionality, or data). In fact, as we noted earlier in this chapter, microservices are often termed "shared nothing" for this reason.[6]

Summary

Businesses are a collection of people, teams, and departments performing a wide range of functions, backed by technology. Teams need to work asynchronously with respect to one another to be efficient, and many business processes are inherently asynchronous—for example, shipping a parcel from a warehouse to a user's door. So we might start a project as a website, where the frontend makes synchronous calls to backend services, but as it grows the web of synchronous calls tightly couple services together at runtime. Event-based methods reverse this, decoupling systems in time and allowing them to evolve independently of one another.

In this chapter we noticed that events, in fact, have two separate roles: one for notification (a call for action), and the other a mechanism for state transfer (pushing data wherever it is needed). Events make the system pluggable, and for reasonably sized architectures it is sensible to blend request- and event-based protocols, but you must take care when using these two sides of the event duality: they lead to very different types of architecture. Finally, we looked at how to scale the two approaches by separating out different bounded contexts that collaborate only through events.

But with all this talk of events, we've talked little of replayable logs or stream processing. When we apply these patterns with Kafka, the toolset itself creates new opportunities. Retention in the broker becomes a tool we can design for, allowing us to embrace data on the outside with a central store of events that services can refer back to. So the ops engineers, whom we discussed in the opening section of this chapter, will still be playing detective, but hopefully not quite as often—and at least now the story comes with a script!

6 Neil Ford, Rebecca Parsons, and Pat Kua, *Building Evolutionary Architectures* (Sebastopol, CA: O'Reilly, 2017).

CHAPTER 6

Processing Events with Stateful Functions

Imperative styles of programming are some of the oldest of all, and their popularity persists for good reason. Procedures execute sequentially, spelling out a story on the page and altering the program's state as they do so.

As mainstream applications became distributed in the 1980s and 1990s, the same mindset was applied to this distributed domain. Approaches like Corba and EJB (Enterprise JavaBeans) raised the level of abstraction, making distributed programming more accessible. History has not always judged these so well. EJB, while touted as a panacea of its time, fell quickly by the wayside as systems creaked with the pains of tight coupling and the misguided notion that the network was something that should be abstracted away from the programmer.

In fairness, things have improved since then, with popular technologies like gRPC and Finagle adding elements of asynchronicity to the request-driven style. But the application of this mindset to the design of distributed systems isn't necessarily the most productive or resilient route to take. Two styles of programming that better suit distributed design, particularly in a services context, are the dataflow and functional styles.

You will have come across dataflow programming if you've used utilities like Sed or languages like Awk. These are used primarily for text processing; for example, a stream of lines might be pushed through a regex, one line at a time, with the output piped to the next command, chaining through stdin and stdout. This style of program is more like an assembly line, with each worker doing a specific task, as the products make their way along a conveyor belt. Since each worker is concerned only with the availability of data inputs, there have no "hidden state" to track. This is very similar to the way streaming systems work. Events accumulate in a stream processor waiting for a condition to be met, say, a join operation

between two different streams. When the correct events are present, the join operation completes and the pipeline continues to the next command. So Kafka provides the equivalent of a pipe in Unix shell, and stream processors provide the chained functions.

There is a similarly useful analogy with functional programming. As with the dataflow style, state is not mutated in place, but rather evolves from function to function, and this matches closely with the way stream processors operate. So most of the benefits of both functional and dataflow languages also apply to streaming systems. These can be broadly summarized as:

- Streaming has an inherent ability for parallelization.
- Streaming naturally lends itself to creating cached datasets and keeping them up to date. This makes it well suited to systems where data and code are separated by the network, notably data processing and GUIs.
- Streaming systems are more resilient than traditional approaches, as high availability is built into the runtime and programs execute in a lossless manner (see the discussion of Event Sourcing in Chapter 7).
- Streaming functions are typically easier to reason about than regular programs. Pure functions are free from side effects. Stateful functions are not, but do avoid shared mutable state.
- Streaming systems embrace a polyglot culture, be it via different programming languages or different datastores.
- Programs are written at a higher level of abstraction, making them more comprehensible.

But streaming approaches also inherit some of the downsides. Purely functional languages must negotiate an impedance mismatch when interacting with more procedural or stateful elements like filesystems or the network. In a similar vein, streaming systems must often translate to the request-response style of REST or RPCs and back again. This has led some implementers to build systems around a *functional core*, which processes events asynchronously, wrapped in an *imperative shell*, used to marshal to and from outward-facing request-response interfaces. The "functional core, imperative shell" pattern keeps the key elements of the system both flexible and scalable, encouraging services to avoid side effects and express their business logic as simple functions chained together through the log.

In the next section we'll look more closely at why statefulness, in the context of stream processing, matters.

Making Services Stateful

There is a well-held mantra that statelessness is good, and for good reason. Stateless services start instantly (no data load required) and can be scaled out linearly, cookie-cutter-style.

Web servers are a good example: to increase their capacity for generating dynamic content, we can scale a web tier horizontally, simply by adding new servers. So why would we want anything else? The rub is that most applications aren't really stateless. A web server needs to know what pages to render, what sessions are active, and more. It solves these problems by keeping the state in a database. So the database is stateful and the web server is stateless. The state problem has just been pushed down a layer. But as traffic to the website increases, it usually leads programmers to cache state locally, and local caching leads to cache invalidation strategies, and a spiral of coherence issues typically ensues.

Streaming platforms approach this problem of where state should live in a slightly different way. First, recall that events are also facts, converging toward the stream processor like conveyor belts on an assembly line. So, for many use cases, the events that trigger a process into action contain *all the data the program needs*, much like the dataflow programs just discussed. If you're validating the contents of an order, all you need is its event stream.

Sometimes this style of stateless processing happens naturally; other times implementers deliberately enrich events in advance, to ensure they have all the data they need for the job at hand. But enrichments inevitably mean looking things up, usually in a database.

Stateful stream processing engines, like Kafka's Streams API, go a step further: they ensure all the data a computation needs is loaded into the API ahead of time, be it events or any tables needed to do lookups or enrichments. In many cases this makes the API, and hence the application, stateful, and if it were restarted for some reason it would need to reacquire that state before it could proceed.

This should seem a bit counterintuitive. Why would you want to make a service stateful? Another way to look at this is as an advanced form of caching that better suits data-intensive workloads. To make this clearer, let's look at three examples —one that uses database lookups, one that is event-driven but stateless, and one that is event-driven but stateful.

The Event-Driven Approach

Say we have an email service that listens to an event stream of orders and then sends confirmation emails to users once they complete a purchase. This requires information about both the order as well as the associated payment. Such an email service might be created in a number of different ways. Let's start by

assuming it's a simple event-driven service (i.e., no use of a streaming API, as in Figure 6-1). It might react to order events, then look up the corresponding payment. Or it might do the reverse: reacting to payments, then looking up the corresponding order. Let's assume the former.

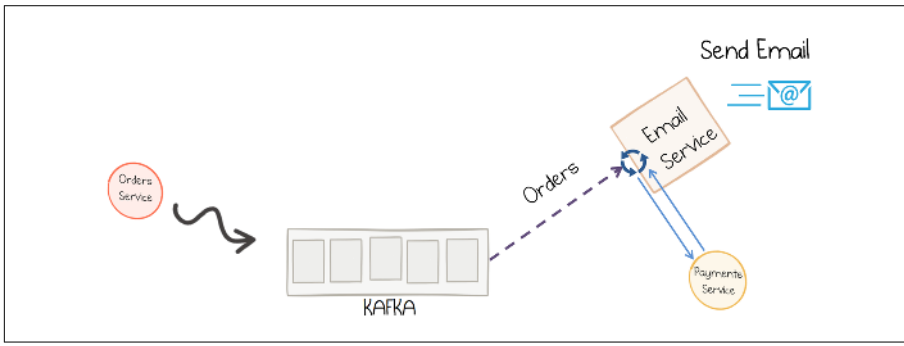


Figure 6-1. A simple event-driven service that looks up the data it needs as it processes messages

So a single event stream is processed, and lookups that pull in any required data are performed inline. The solution suffers from two problems:

- The constant need to look things up, one message at a time.
- The payment and order are created at about the same time, so one might arrive before the other. This means that if the order arrives in the email service before the payment is available in the database, then we'd have to either block and poll until it becomes available or, worse, skip the email processing completely.

The Pure (Stateless) Streaming Approach

A streaming system comes at this problem from a slightly different angle. The streams are buffered until both events arrive, and can be joined together (Figure 6-2).

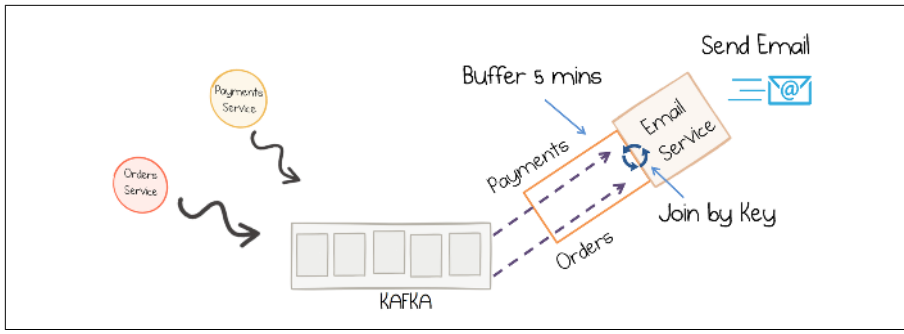


Figure 6-2. A stateless streaming service that joins two streams at runtime

This solves the two aforementioned issues with the event-driven approach. There are no remote lookups, addressing the first point. It also no longer matters what order events arrive in, addressing the second point.

The second point turns out to be particularly important. When you're working with asynchronous channels there is no easy way to ensure relative ordering across several of them. So even if we know that the order is always created before the payment, it may well be delayed, arriving the other way around.

Finally, note that this approach isn't, strictly speaking, stateless. The buffer actually makes the email service stateful, albeit just a little. When Kafka Streams restarts, before it does any processing, it will reload the contents of each buffer. This is important for achieving deterministic results. For example, the output of a join operation is dependent on the contents of the opposing buffer when a message arrives.

The Stateful Streaming Approach

Alas, the data flowing through the various event streams isn't always enough—sometimes you need lookups or enrichments. For example, the email service would need access to the customer's email address. There will be no recent event for this (unless you happened to be very lucky and the customer just updated their details). So you'd have to look up the email address in the customer service via, say, a REST call (Figure 6-3).

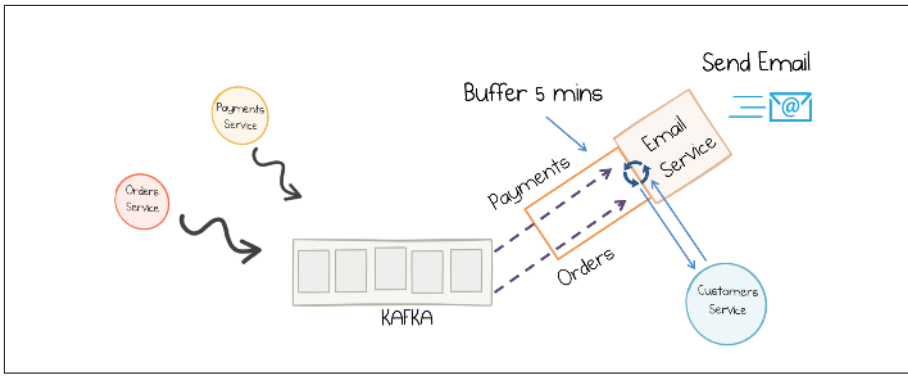


Figure 6-3. A stateless streaming service that looks up reference data in another service at runtime

This is of course a perfectly valid approach (in fact, many production systems do exactly this), but a stateful stream processing system can make a further optimization. It uses the same process of local buffering used to handle potential delays in the orders and payments topics, but instead of buffering for just a few minutes, it preloads the whole customer event stream from Kafka into the email service, where it can be used to look up historic values (Figure 6-4).

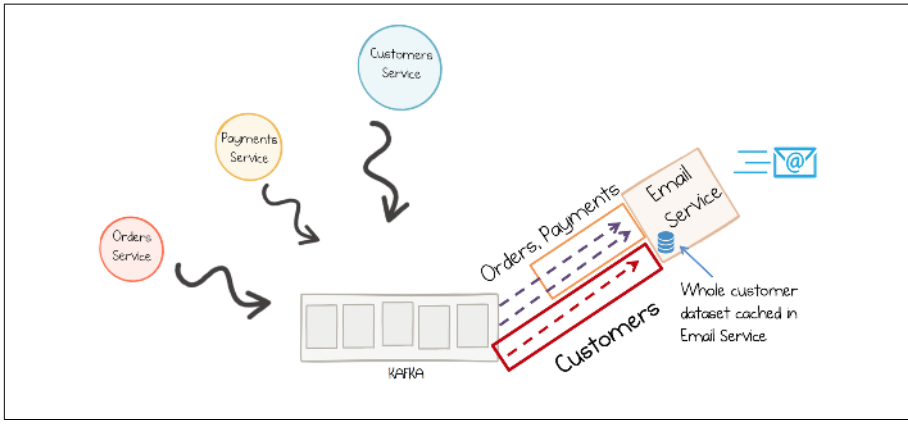


Figure 6-4. A stateful streaming service that replicates the Customers topic into a local table, held inside the Kafka Streams API

So now the email service is both buffering recent events, as well as creating a local lookup table. (The mechanics of this are discussed in more detail in Chapter 14.)

This final, fully stateful approach comes with disadvantages:

- The service is now stateful, meaning for an instance of the email service to operate it needs the relevant customer data to be present. This means, in the worst case, loading the full dataset on startup.

as well as advantages:

- The service is no longer dependent on the worst-case performance or liveness of the customer service.
- The service can process events faster, as each operation is executed without making a network call.
- The service is free to perform more data-centric operations on the data it holds.

This final point is particularly important for the increasingly data-centric systems we build today. As an example, imagine we have a GUI that allows users to browse order, payment, and customer information in a scrollable grid. The grid lets the user scroll up and down through the items it displays.

In a traditional, stateless model, each row on the screen would require a call to all three services. This would be sluggish in practice, so caching would likely be added, along with some hand-crafted polling mechanism to keep the cache up to date.

But in the streaming approach, data is constantly pushed into the UI (Figure 6-5). So you might define a query for the data displayed in the grid, something like `select * from orders, payments, customers where…`. The API executes it over the incoming event streams, stores the result locally, and keeps it up to date. So streaming behaves a bit like a decoratively defined cache.

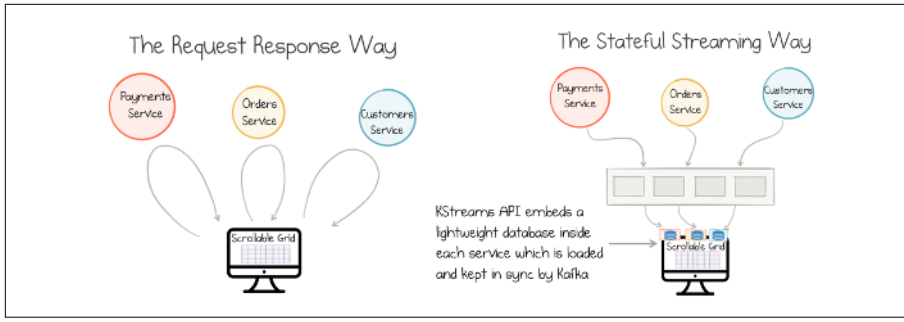


Figure 6-5. Stateful stream processing is used to materialize data inside a web server so that the UI can access it locally for better performance, in this case via a scrollable grid

The Practicalities of Being Stateful

Being stateful comes with some challenges: when a new node starts, it must load all stateful components (i.e., state stores) before it can start processing messages, and in the worst case this reload can take some time. To help with this problem, Kafka Streams provides three mechanisms for making being stateful a bit more practical:

- It uses a technique called standby replicas, which ensure that for every table or state store on one node, there is a replica kept up to date on another. So, if any node fails, it will immediately fail over to its backup node without interrupting processing unduly.
- Disk checkpoints are created periodically so that, should a node fail and restart, it can load its previous checkpoint, then top up the few messages it missed when it was offline from the log.
- Finally, compacted topics are used to keep the dataset as small as possible. This acts to reduce the load time for a complete rebuild should one be necessary.

Kafka Streams uses intermediary topics, which can be reset and rederived using the Streams Reset tool.[1]

An *event-driven application* uses a single input stream to drive its work. A *streaming application* blends one or more input streams into one or more output streams. A *stateful streaming application* also recasts streams to tables (used to do enrichments) and stores intermediary state in the log, so it internalizes all the data it needs.

Summary

This chapter covers three different ways of doing event-based processing: the simple event-driven approach, where you process a single event stream one message at a time; the streaming approach, which joins different event streams together; and finally, the stateful streaming approach, which turns streams into tables and stores data in the log.

So instead of pushing the state problem down a layer into a database, stateful stream processors, like Kafka's Streams API, are proudly stateful. They make data available wherever it is required. This increases performance and autonomy. No remote calls needed!

1 See *http://bit.ly/2GaCRZO* and *http://bit.ly/2IUPHJa*.

Of course, being stateful comes with its downsides, but it is optional, and real-world streaming systems blend together all three approaches. We go into the detail of how these streaming operations work in Chapter 14.

CHAPTER 7

Event Sourcing, CQRS, and Other Stateful Patterns

In Chapter 5 we introduced the Event Collaboration pattern, where events describe an evolving business process—like processing an online purchase or booking and settling a trade—and several services collaborate to push that workflow forward.

This leads to a log of every state change the system makes, held immutably, which in turn leads to two related patterns, Command Query Response Segregation (CQRS) and Event Sourcing,[1] designed to help systems scale and be less prone to corruption. This chapter explores what these concepts mean, how they can be implemented, and when they should be applied.

Event Sourcing, Command Sourcing, and CQRS in a Nutshell

At a high level, Event Sourcing is just the observation that events (i.e., state changes) are a core element of any system. So, if they are stored, immutably, in the order they were created in, the resulting event log provides a comprehensive audit of exactly what the system did. What's more, we can always rederive the current state of the system by rewinding the log and replaying the events in order.

CQRS is a natural progression from this. As a simple example, you might write events to Kafka (write model), read them back, and then push them into a database (read model). In this case Kafka maps the read model onto the write model

1 See *https://martinfowler.com/eaaDev/EventSourcing.html* and *http://bit.ly/2pLKRFF*.

asynchronously, decoupling the two in time so the two parts can be optimized independently.

Command Sourcing is essentially a variant of Event Sourcing but applied to events that come *into* a service, rather than via the events it creates.

That's all a bit abstract, so let's walk through the example in Figure 7-1. We'll use one similar to the one used in the previous chapter, where a user makes an online purchase and the resulting order is validated and returned.

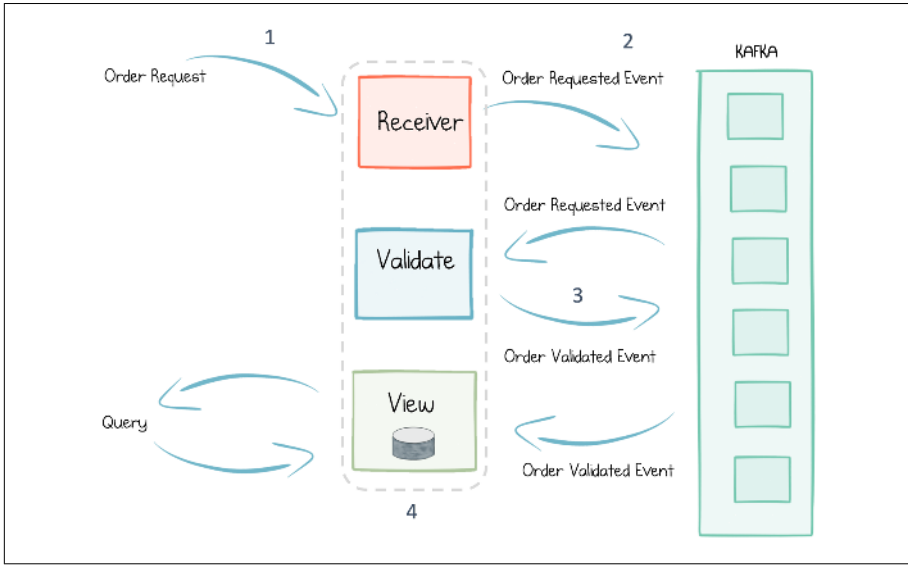


Figure 7-1. A simple order validation workflow

When a purchase is made (1), *Command Sourcing* dictates that the order request be immediately stored as an event in the log, before anything happens (2). That way, should anything go wrong, the service can be rewound and replayed—for example, to recover from a corruption.

Next, the order is validated, and another event is stored in the log to reflect the resulting change in state (3). In contrast to an update-in-place persistence model like CRUD (create, read, update, delete), the validated order is represented as an entirely new event, being appended to the log rather than overwriting the existing order. This is the essence of Event Sourcing.

Finally, to query orders, a database is attached to the resulting event stream, deriving an *event-sourced view* that represents the current state of orders in the system (4). So (1) and (4) provide the Command and Query sides of CQRS.

These patterns have a number of benefits, which we will examine in detail in the subsequent sections.

Version Control for Your Data

When you store events in a log, it behaves a bit like a version control system for your data. Consider the situation illustrated in Figure 7-2. If a programmatic bug is introduced—let's say a timestamp field was interpreted with the wrong time zone—you would get a data corruption. The corruption would make its way into the database. It would also make it into interactions the service makes with other services, making the corruption more widespread and harder to fix.

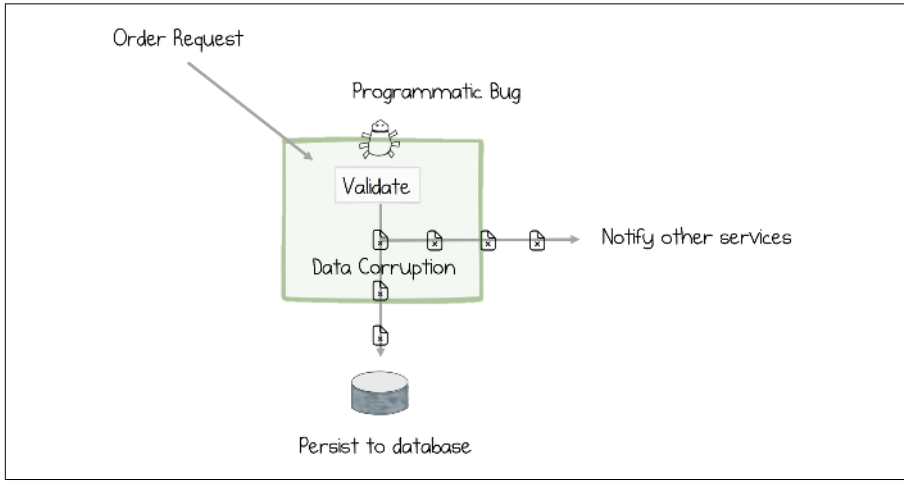


Figure 7-2. A programmatic bug can lead to data corruption, both in the service's own database as well as in data it exposes to other services

Recovering from this situation is tricky for a couple of reasons. First, the original inputs to the system haven't been recorded exactly, so we only have the corrupted version of the order. We will have to uncorrupt it manually. Second, unlike a version control system, which can travel back in time, a database is mutated in place, meaning the previous state of the system is lost forever. So there is no easy way for this service to undo the damage the corruption did.

To fix this, the programmer would need to go through a series of steps: applying a fix to the software, running a database script to fix the corrupted timestamps in the database, and finally, working out some way of resending any corrupted data previously sent to other services. At best this will involve some custom code that pulls data out of the database, fixes it up, and makes new service calls to redistribute the corrected data. But because the database is lossy—as values are overwritten—this may not be enough. (If rather than the release being fixed, it was rolled back to a previous version after some time running as the new version, the data migration process would likely be even more complex.)

NOTE

A replayable log turns ephemeral messaging into messaging that remembers.

Switching to an Event/Command Sourcing approach, where both inputs and state changes are recorded, might look something like Figure 7-3.

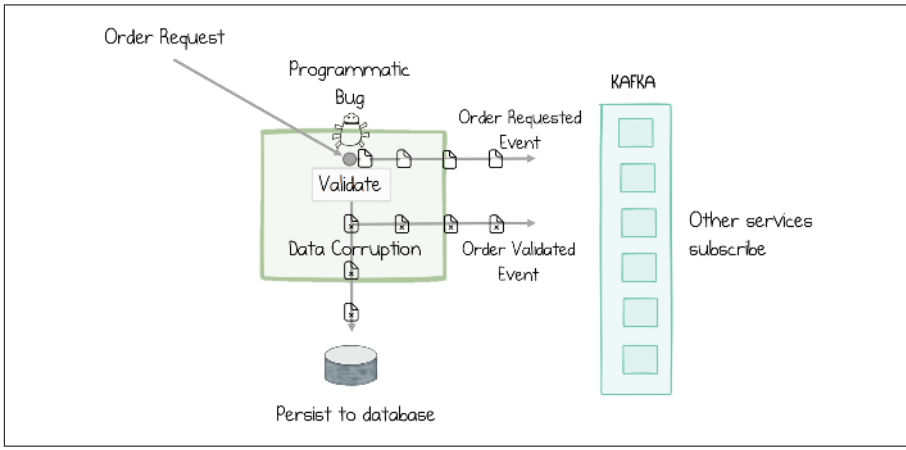


Figure 7-3. Adding Kafka and an Event Sourcing approach to the system described in Figure 7-2 ensures that the original events are preserved before the code, and bug, execute

As Kafka can store events for as long as we need (as discussed in "Long-Term Data Storage" on page 25 in Chapter 4), correcting the timestamp corruption is now a relatively simple affair. First the bug is fixed, then the log is rewound to before the bug was introduced, and the system is replayed from the stream of order requests. The database is automatically overwritten with corrected timestamps, and new events are published downstream, correcting the previous corrupted ones. This ability to store inputs, rewind, and replay makes the system far better at recovering from corruptions and bugs.

So Command Sourcing lets us record our inputs, which means the system can always be rewound and replayed. Event Sourcing records our state changes, which ensures we know exactly what happened during our system's execution, and we can always regenerate our current state (in this case the contents of the database) from this log of state changes (Figure 7-4).

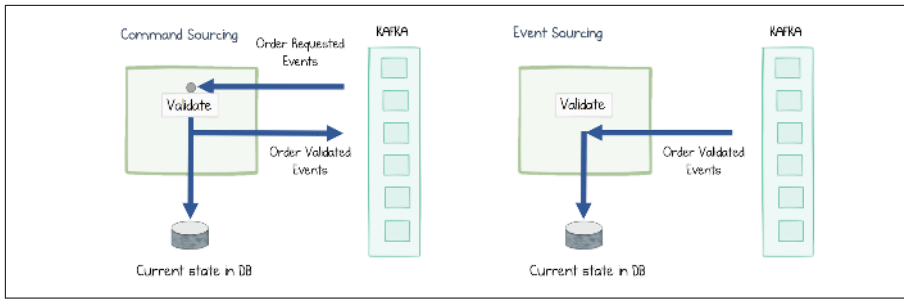


Figure 7-4. Command Sourcing provides straight-through reprocessing from the original commands, while Event Sourcing rederives the service's current state from just the post-processed events in the log

Being able to store an ordered journal of state changes is useful for debugging and traceability purposes, too, answering retrospective questions like "Why did this order mysteriously get rejected?" or "Why is this balance suddenly in the red?"—questions that are harder to answer with mutable data storage.

NOTE

Event Sourcing ensures every state change in a system is recorded, much like a version control system. As the saying goes, "Accountants don't use erasers."

It is also worth mentioning that there are other well-established database patterns that provide some of these properties. Staging tables can be used to hold unvalidated inputs, triggers can be applied in many relational databases to create audit tables, and Bitemporal databases also provide an auditable data structure. These are all useful techniques, but none of them lends itself to "rewind and replay" functionality without a significant amount of effort on the programmer's part. By contrast, with the Event Sourcing approach, there is little or no additional code to write or test. The primary execution path is used both for runtime execution as well as for recovery.

Making Events the Source of Truth

One side effect of the previous example is that the application of Event Sourcing means the event, not the database record, is the source of truth. Making events first-class entities in this way leads to some interesting implications.

If we consider the order request example again, there are two versions of the order: one in the database and one in the notification. This leads to a couple of different issues. The first is that both the notification and database update must be made atomically. Imagine if a failure happens midway between the database

being updated and the notification being sent (Figure 7-5). The best-case scenario is the creation of duplicate notifications. The worst-case scenario is that the notification might not be made at all. This issue can worsen in complex systems, as we discuss at length in Chapter 12, where we look at Kafka's transactions feature.

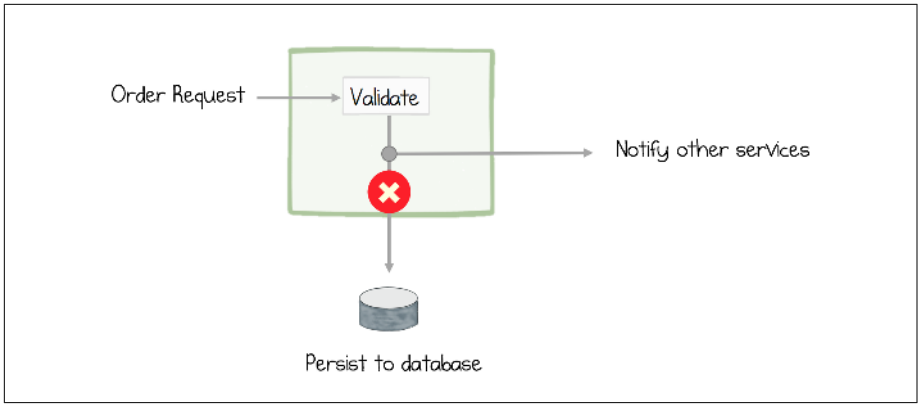


Figure 7-5. The order request is validated and a notification is sent to other services, but the service fails before the data is persisted to the database

The second problem is that, in practice, it's quite easy for the data in the database and the data in the notification to diverge as code churns and features are implemented. The implication here is that, while the database may well be correct, if the notifications don't quite match, then the data quality of the system as a whole suffers. (See "The Data Divergence Problem" on page 95 in Chapter 10.)

Event Sourcing addresses both of these problems by making the event stream the primary source of truth (Figure 7-6). Where data needs to be queried, a read model or event-sourced view is *derived* directly from the stream.

NOTE

Event Sourcing ensures that the state a service communicates and the state a service saves internally are the same.

This actually makes a lot of sense. In a traditional system the database is the source of truth. This is sensible from an *internal* perspective. But if you consider it from the point of view of other services, they don't care what is stored internally; it's the data everyone else sees that is important. So the event being the source of truth makes a lot of sense from their perspective. This leads us to CQRS.

Command Query Responsibility Segregation

As discussed earlier, CQRS (Command Query Responsibility Segregation) separates the write path from the read path and links them with an asynchronous channel (Figure 7-6). This idea isn't limited to application design—it comes up in a number of other fields too. Databases, for example, implement the idea of a write-ahead log. Inserts and updates are immediately journaled sequentially to disk, as soon as they arrive. This makes them durable, so the database can reply back to the user in the knowledge that the data is safe, but without having to wait for the slow process of updating the various concurrent data structures like tables, indexes, and so on.[2] The point is that (a) should something go wrong, the internal state of the database can be recovered from the log, and (b) writes and reads can be optimized independently.

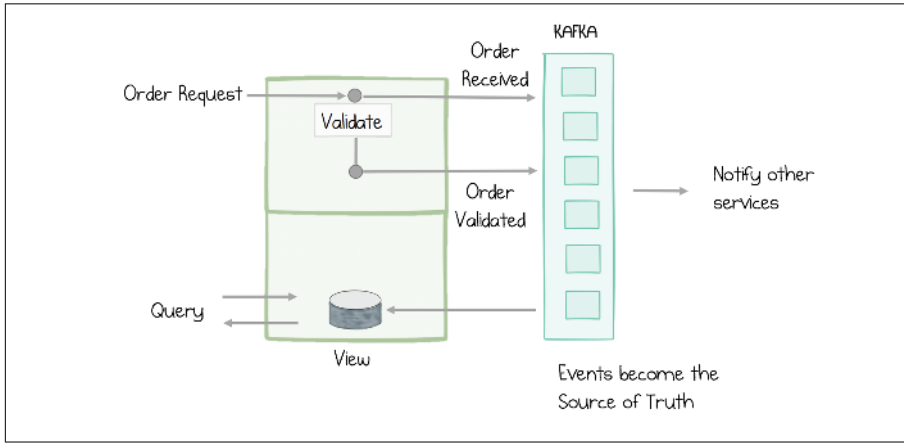


Figure 7-6. When we make the event stream the source of truth, the notification and the database update come from the same event, which is stored immutably and durably; when we split the read and write model, the system is an implementation of the CQRS design pattern

When we apply CQRS in our applications, we do it for very similar reasons. Inputs are journaled to Kafka, which is much faster than writing to a database. Segregating the read model and updating it asynchronously means that the expensive maintenance of update-in-place data structures, like indexes, can be batched together so they are more efficient. This means CQRS will display better overall read and write performance when compared to an equivalent, more traditional, CRUD-based system.

2 Some databases—for example, DRUID—make this separation quite concrete. Other databases block until indexes have been updated.

Of course there is no free lunch. The catch is that, because the read model is updated asynchronously, it will run slightly behind the write model in time. So if a user performs a write, then immediately performs a read, it is possible that the entry they wrote originally has not had time to propagate, so they can't "read their own writes." As we will see in the example in "Collapsing CQRS with a Blocking Read" on page 142 in Chapter 15, there are strategies for addressing this problem.

Materialized Views

There is a close link between the query side of CQRS and a materialized view in a relational database. A materialized view is a table that contains the results of some predefined query, with the view being updated every time any of the underlying tables change.

Materialized views are used as a performance optimization so, instead of a query being computed when a user needs data, the query is precomputed and stored. For example, if we wanted to display how many active users there are on each page of a website, this might involve us scanning a database table of user visits, which would be relatively expensive to compute. But if we were to precompute the query, the summary of active users that results will be comparatively small and hence fast to retrieve. Thus, it is a good candidate to be precomputed.

We can create exactly the same construct with CQRS using Kafka. Writes go into Kafka on the command side (rather than updating a database table directly). We can transform the event stream in a way that suits our use case, typically using Kafka Streams or KSQL, then materialize it as a precomputed query or materialized view. As Kafka is publish-subscribe, we can have many such views, precomputed to match the various use cases we have (Figure 7-7). But unlike with materialized views in a relational database, the underlying events are decoupled from the view. This means (a) they can be scaled independently, and (b) the writing process (so whatever process records user visits) doesn't have to wait for the view to be computed before it returns.

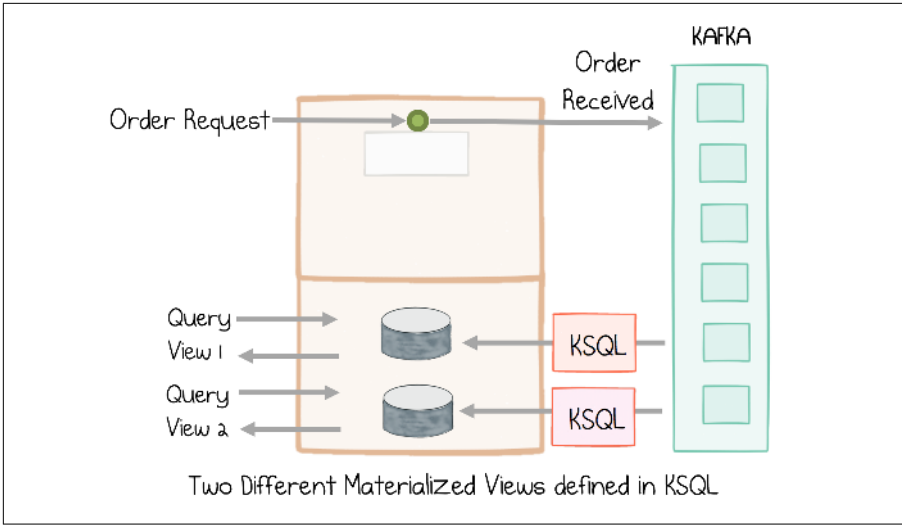


Figure 7-7. CQRS allows multiple read models, which may be optimized for a specific use case, much like a materialized view, or may use a different storage technology

This idea of storing data in a log and creating many derived views is taken further when we discuss "Event Streams as a Shared Source of Truth" in Chapter 9.

NOTE

If an event stream is the source of truth, you can have as many different views in as many different shapes, sizes, or technologies as you may need. Each is focused on the use case at hand.

Polyglot Views

Whatever sized data problem you have, be it free-text search, analytic aggregation, fast key/value lookups, or a host of others, there is a database available today that is just right for your use case. But this also means there is no "one-size-fits-all" approach to databases, at least not anymore. A supplementary benefit of using CQRS is that a single write model can push data into many read models or materialized views. So your read model can be in any database, or even a range of different databases.

A replayable log makes it easy to bootstrap such *polyglot views* from the same data, each tuned to different use cases (Figure 7-7). A common example of this is to use a fast key/value store to service queries from a website, but then use a search engine like Elasticsearch or Solr to support the free-text-search use case.

Whole Fact or Delta?

One question that arises in event-driven—particularly event-sourced—programs, is whether the events should be modeled as whole facts (a whole order, in its entirety) or as deltas that must be recombined (first a whole order message, followed by messages denoting only what changed: "amount updated to $5," "Order cancelled," etc.).

As an analogy, imagine you are building a version control system like SVN or Git. When a user commits a file for the first time, the system saves the whole file to disk. Subsequent commits, reflecting changes to that file, might save only the *delta*—that is, just the lines that were added, changed, or removed. Then, when the user checks out a certain version, the system opens the version-0 file and applies all subsequent deltas, in order, to derive the version the user asked for.

The alternate approach is to simply store the whole file, exactly as it was at the time it was changed, for every single commit. This will obviously take more storage, but it means that checking out a specific version from the history is a quick and easy file retrieval. However, if the user wanted to compare different versions, the system would have to use a "diff" function.

These two approaches apply equally to data we keep in the log. So to take a more business-oriented example, an order is typically a set of line items (i.e., you often order several different items in a single purchase). When implementing a system that processes purchases, you might wonder: should the order be modeled as a single order event with all the line items inside it, or should each line item be a separate event with the order being recomposed by scanning the various independent line items? In domain-driven design, an order of this latter type is termed an *aggregate* (as it is an aggregate of line items) with the wrapping entity —that is, the order—being termed an *aggregate root*.

As with many things in software design, there are a host of different opinions on which approach is best for a certain use case. There are a few rules of thumb that can help, though. The most important one is *journal the whole fact as it arrived*. So when a user creates an order, if that order turns up with all line items inside it, we'd typically record it as a single entity.

But what happens when a user cancels a single line item? The simple solution is to just journal the whole thing again, as another aggregate but cancelled. But what if for some reason the order is not available, and all we get is a single canceled line item? Then there would be the temptation to look up the original order internally (say from a database), and combine it with the cancellation to create a new Cancelled Order with all its line items embedded inside it. This typically isn't a good idea, because (a) we're not recording exactly what we received, and (b) having to look up the order in the database erodes the performance benefits

of CQRS. The rule of thumb is record what you receive, so if only one line item arrives, record that. The process of combining can be done on read.

Conversely, breaking events up into subevents as they arrive often isn't good practice, either, for similar reasons. So, in summary, the rule of thumb is *record exactly what you receive, immutably*.

Implementing Event Sourcing and CQRS with Kafka

Kafka ships with two different APIs that make it easier to build and manage CQRS-styled views derived from events stored in Kafka. The Kafka Connect API and associated Connector ecosystem provides an out-of-the-box mechanism to push data into, or pull data from, a variety of databases, data sources, and data sinks. In addition, the Kafka Streams API ships with a simple embedded database, called a state store, built into the API (see "Windows, Joins, Tables, and State Stores" on page 135 in Chapter 14).

In the rest of this section we cover some useful patterns for implementing Event Sourcing and CQRS with these tools.

Build In-Process Views with Tables and State Stores in Kafka Streams

Kafka's Streams API provides one of the simplest mechanisms for implementing Event Sourcing and CQRS because it lets you implement a view natively, right inside the Kafka Streams API—no external database needed!

At its simplest this involves turning a stream of events in Kafka into a table that can be queried locally. For example, turning a stream of `Customer` events into a table of `Customers` that can be queried by `CustomerId` takes only a single line of code:

```
KTable<CustomerId, Customer> customerTable = builder.table("customer-topic");
```

This single line of code does several things:

- It subscribes to events in the customer topic.
- It resets to the earliest offset and loads all `Customer` events into the Kafka Streams API. That means it loads the data from Kafka into your service. (Typically a compacted topic is used to reduce the initial/worst-case load time.)
- It pushes those events into a state store (Figure 7-8), a local, disk-resident hash table, located inside the Kafka Streams API. This results in a local, disk-resident table of `Customers` that can be queried by key or by range scan.

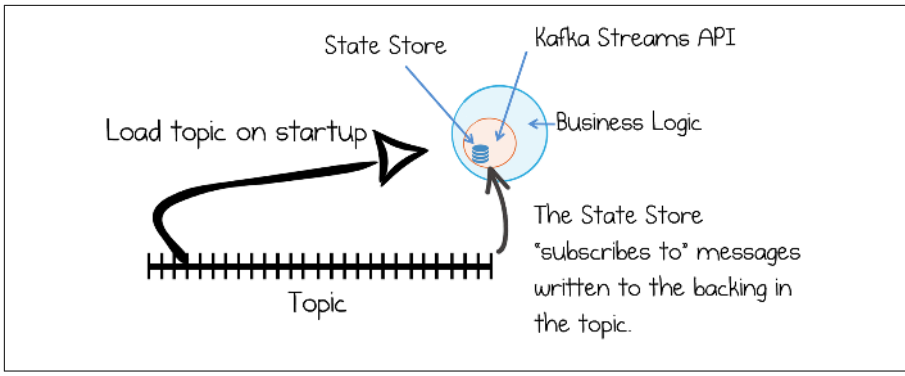


Figure 7-8. State stores in Kafka Streams can be used to create use-case-specific views right inside the service

In Chapter 15 we walk through a set of richer code examples that create different types of views using tables and state stores, along with discussing how this approach can be scaled.

Writing Through a Database into a Kafka Topic with Kafka Connect

One way to get events into Kafka is to write *through* a database table into a Kafka topic. Strictly speaking, this isn't an Event Sourcing– or CQRS-based pattern, but it's useful nonetheless.

In Figure 7-9, the orders service writes orders to a database. The writes are converted into an event stream by Kafka's Connect API. This triggers downstream processing, which validates the order. When the "Order Validated" event returns to the orders service, the database is updated with the final state of the order, before the call returns to the user.

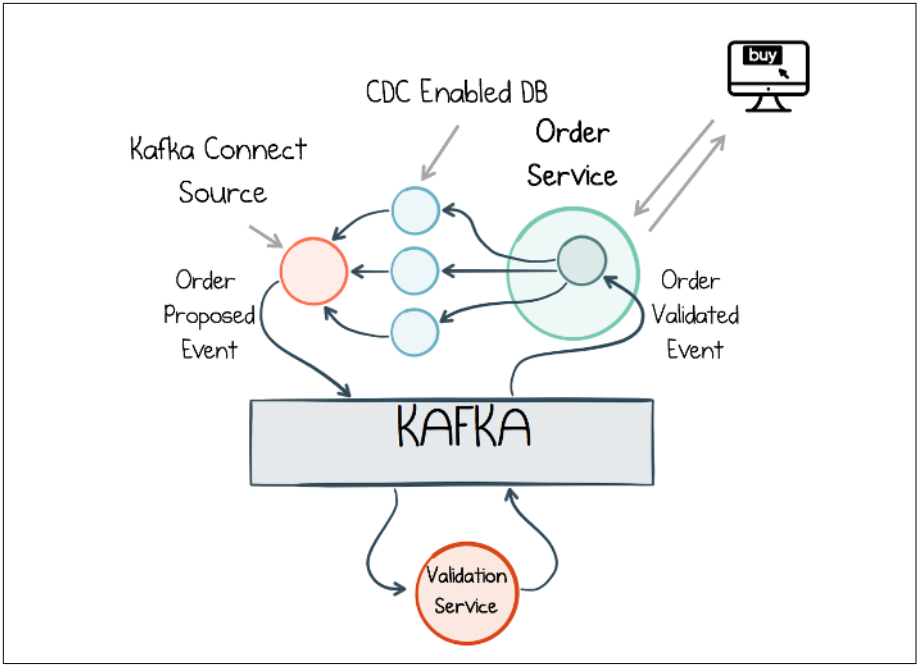


Figure 7-9. An example of writing through a database to an event stream

The most reliable and efficient way to achieve this is using a technique called change data capture (CDC). Most databases write every modification operation to a write-ahead log, so, should the database encounter an error, it can recover its state from there. Many also provide some mechanism for capturing modification operations that were committed. Connectors that implement CDC repurpose these, translating database operations into events that are exposed in a messaging system like Kafka. Because CDC makes use of a native "eventing" interface it is (a) very efficient, as the connector is monitoring a file or being triggered directly when changes occur, rather than issuing queries through the database's main API, and (b) very accurate, as issuing queries through the database's main API will often create an opportunity for operations to be missed if several arrive, for the same row, within a polling period.

In the Kafka ecosystem CDC isn't available for every database, but the ecosystem is growing. Some popular databases with CDC support in Kafka Connect are MySQL, Postgres, MongoDB, and Cassandra. There are also proprietary CDC connectors for Oracle, IBM, SQL Server, and more. The full list of connectors is available on the Connect home page.

The advantage of this database-fronted approach is that it provides a consistency point: you write through it into Kafka, meaning you can always read your own writes.

Writing Through a State Store to a Kafka Topic in Kafka Streams

The same pattern of writing through a database into a Kafka topic can be achieved inside Kafka Streams, where the database is replaced with a Kafka Streams state store (Figure 7-10). This comes with all the benefits of writing through a database with CDC, but has a few additional advantages:

- The database is local, making it faster to access.
- Because the state store is wrapped by Kafka Streams, it can partake in transactions, so events published by the service and writes to the state store are atomic.
- There is less configuration, as it's a single API (no external database, and no CDC connector to configure).

We discuss this use of state stores for holding application-level state in the section "Windows, Joins, Tables, and State Stores" on page 135 in Chapter 14.

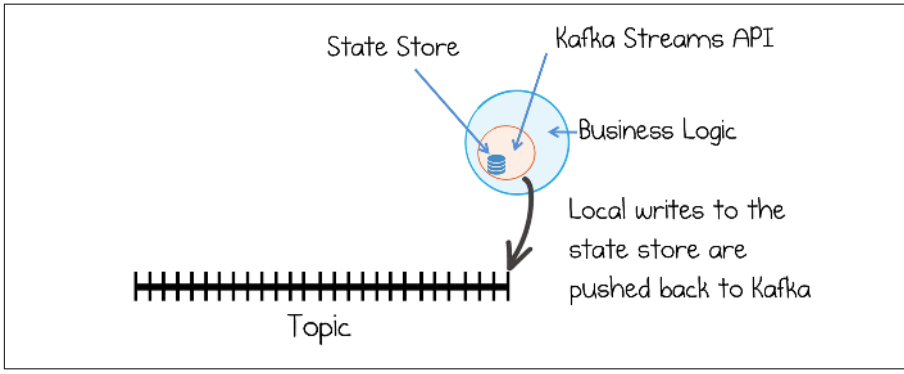


Figure 7-10. Applying the write-through pattern with Kafka Streams and a state store

Unlocking Legacy Systems with CDC

In reality, most projects have some element of legacy and renewal, and while there is a place for big-bang redesigns, incremental change is typically an easier pill to swallow.

The problem with legacy is that there is usually a good reason for moving away from it: the most common being that it is hard to change. But most business operations in legacy applications will converge on their database. This means that, no matter how creepy the legacy code is, the database provides a coherent seam to latch into the existing business workflow, from where we can extract events via CDC. Once we have the event stream, we can plug in new event-driven services that allow us to evolve away from the past, incrementally (Figure 7-11).

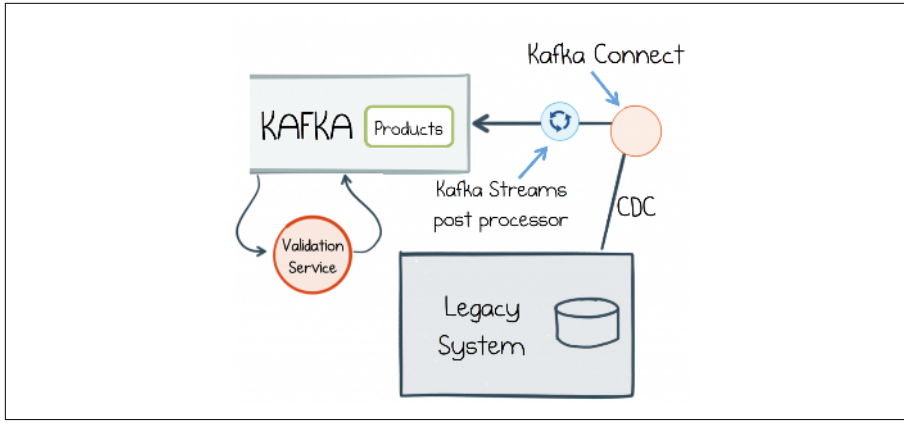


Figure 7-11. Unlocking legacy data using Kafka's Connect API

So part of our legacy system might allow admins to manage and update the product catalog. We might retain this functionality by importing the dataset into Kafka from the legacy system's database. Then that product catalog can be reused in the validation service, or any other.

An issue with attaching to legacy, or any externally sourced dataset, is that the data is not always well formed. If this is a problem, consider adding a post-processing stage. Kafka Connect's single message transforms are useful for this type of operation (for example, adding simple adjustments or enrichments), while Kafka's Streams API is ideal for simple to very complex manipulations and for precomputing views that other services need.

Query a Read-Optimized View Created in a Database

Another common pattern is to use the Connect API to create a read-optimized, event-sourced view, built inside a database. Such views can be created quickly and easily in any number of different databases using the sink connectors available for Kafka Connect. As we discussed in the previous section, these are often termed *polyglot views*, and they open up the architecture to a wide range of data storage technologies.

In the example in Figure 7-12, we use Elasticsearch for its rich indexing and query capabilities, but whatever shape of problem you have, these days there is a database that fits. Another common pattern is to precompute the contents as a materialized view using Kafka Streams, KSQL, or Kafka Connect's single message transforms feature (see "Materialized Views" on page 62 earlier in this chapter).

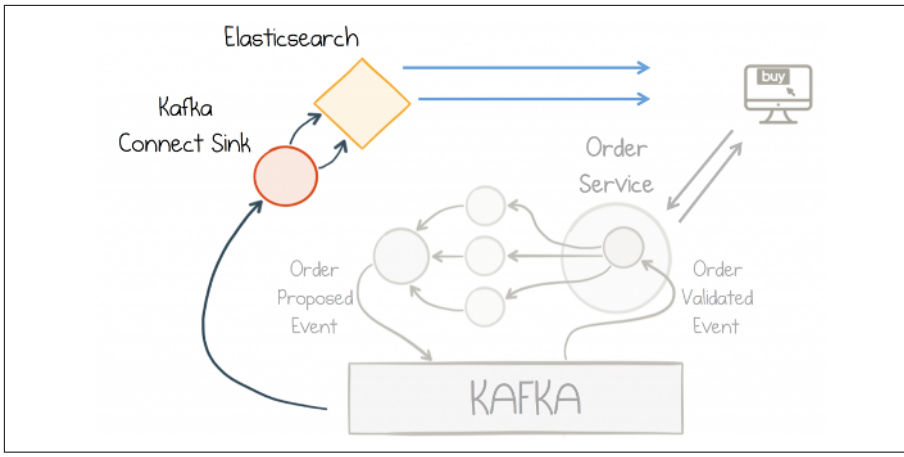


Figure 7-12. Full-text search is added via an Elasticsearch database connected through Kafka's Connect API

Memory Images/Prepopulated Caches

Finally, we should mention a pattern called MemoryImage (Figure 7-13). This is just a fancy term, coined by Martin Fowler, for caching a whole dataset into memory—where it can be queried—rather than making use of an external database.

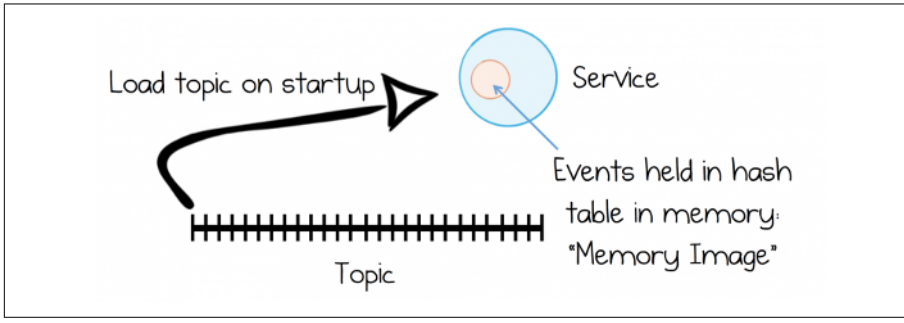


Figure 7-13. A MemoryImage is a simple "cache" of an entire topic loaded into a service so it can be referenced locally

MemoryImages provide a simple and efficient model for datasets that (a) fit in memory and (b) can be loaded in a reasonable amount of time. To reduce the load time issue, it's common to keep a snapshot of the event log using a compacted topic (which represents the latest set of events, without any of the version history). The MemoryImage pattern can be hand-crafted, or it can be implemented with Kafka Streams using in-memory state stores. The pattern suits high-performance use cases that don't need to overflow to disk.

The Event-Sourced View

Throughout the rest of this book we will use the term *event-sourced view* (Figure 7-14) to refer to a query resource (database, memory image, etc.) created in one service from data authored by (and hence owned) by another.

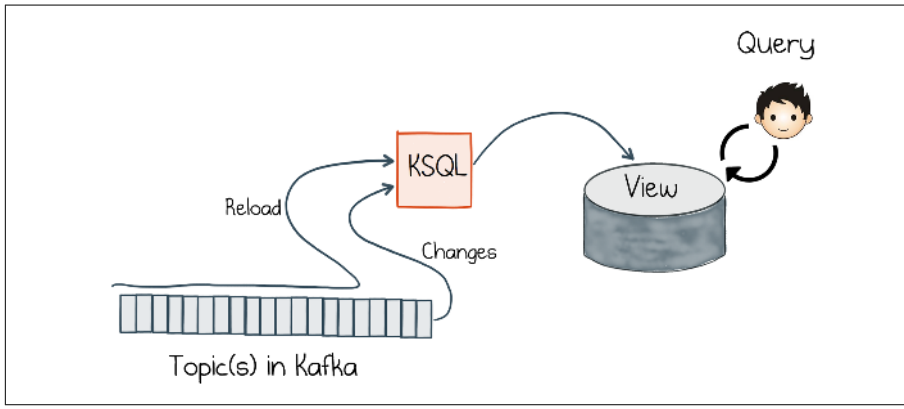


Figure 7-14. An event-sourced view implemented with KSQL and a database; if built with Kafka Streams, the query and view both exist in the same layer

What differentiates an event-sourced view from a typical database, cache, and the like is that, while it can represent data in any form the user requires, its data is sourced directly from the log and can be regenerated at any time.

For example, we might create a view of orders, payments, and customer information, filtering anything that doesn't ship within the US. This would be an event-sourced view if, when we change the view definition—say to include orders that ship to Canada—we can automatically recreate the view in its entirety from the log.

An event-sourced view is equivalent to a projection in Event Sourcing parlance.

Summary

In this chapter we looked at how an event can be more than just a mechanism for notification, or state transfer. Event Sourcing is about saving state using the exact same medium we use to communicate it, in a way that ensures that every change is recorded immutably. As we noted in the section "Version Control for Your Data" on page 57, this makes recovery from failure or corruption simpler and more efficient when compared to traditional methods of application design.

CQRS goes a step further by turning these raw events into an event-sourced view—a queryable endpoint that derives itself (and can be rederived) directly from the log. The importance of CQRS is that it scales, by optimizing read and write models independently of one another.

We then looked at various patterns for getting events into the log, as well as building views using Kafka Streams and Kafka's Connect interface and our database of choice.

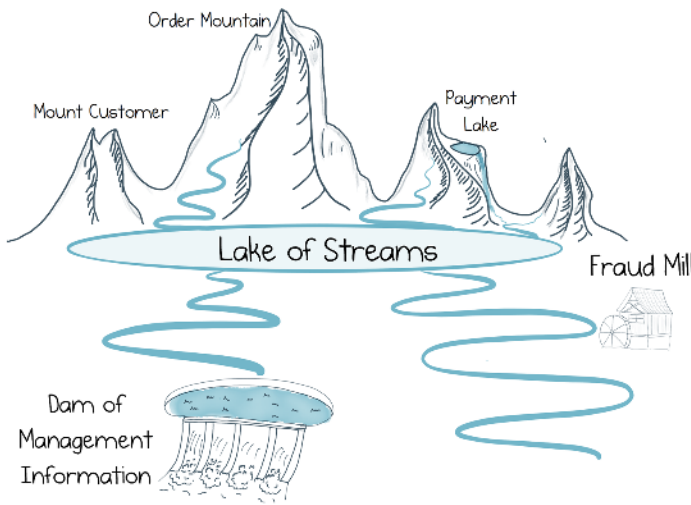
Ultimately, from the perspective of an architect or programmer, switching to this event-sourced approach will have a significant effect on an application's design. Event Sourcing and CQRS make events first-class citizens. This allows systems to relate the data that lives inside a service directly to the data it shares with others. Later we'll see how we can tie these together with Kafka's Transactions API. We will also extend the ideas introduced in this chapter by applying them to inter-team contexts, with the "Event Streaming as a Shared Source of Truth" approach discussed in Chapter 9.

PART III

Rethinking Architecture at Company Scales

If you squint a bit, you can see the whole of your organization's systems and data flows as a single distributed database.

—Jay Kreps, 2013



CHAPTER 8

Sharing Data and Services Across an Organization

When we build software, our main focus is, quite rightly, aimed at solving some real-world problem. It might be a new web page, a report of sales features, an analytics program searching for fraudulent behavior, or an almost infinite set of options that provide clear and physical benefits to our users. These are all very tangible goals—goals that serve our business today.

But when we build software we also consider the future—not by staring into a crystal ball in some vain attempt to predict what our company will need next year, but rather by facing up to the fact that whatever does happen, our software will need to change. We do this without really thinking about it. We carefully modularize our code so it is comprehensible and reusable. We write tests, run continuous integration, and maybe even do continuous deployment. These things take effort, yet they bear little resemblance to anything a user might ask for directly. We do these things because they make our code last, and that doesn't mean sitting on some filesystem the far side of `git push`. It means providing for a codebase that is changed, evolved, refactored, and repurposed. Aging in software isn't a function of time; it is a function of *how we choose to change it*.

But when we design systems, we are less likely to think about how they will age. We are far *more* likely to ask questions like: Will the system scale as our user base increases? Will response times be fast enough to keep users happy? Will it promote reuse? In fact, you might even wonder what a system designed to last a long time looks like.

If we look to history to answer this question, it would point us to mainframe applications for payroll, big-name client programs like Excel or Safari, or even operating systems like Windows or Linux. But these are all complex, individual programs that have been hugely valuable to society. They have also all been diffi-

cult to evolve, particularly with regard to organizing a large engineering effort around a single codebase. So if it's hard to build large but individual software programs, how do we build the software that runs a company? This is the question we address in this particular section: how do we design systems that age well at company scales and keep our businesses nimble?

As it happens, many companies sensibly start their lives with a single system, which becomes monolithic as it slowly turns into the proverbial big ball of mud. The most common response to this today is to break the monolith into a range of different applications and services. In Chapter 1 we talked about companies like Amazon, LinkedIn, and Netflix, which take a service-based approach to this. This is no panacea; in fact, many implementations of the microservices pattern suffer from the misconceived notion that modularizing software over the network will somehow improve its sustainability. This of course isn't what microservices are really about. But regardless of your interpretation, breaking a monolith, alone, will do little to improve sustainability. There is a very good reason for this too. When we design systems at company scales, those systems become far more about people than they are about software.

As a company grows it forms into teams, and those teams have different responsibilities and need to be able to make progress without extensive interaction with one another. The larger the company, the more of this autonomy they need. This is the basis of management theories like Slack.[1]

In stark contrast to this, total independence won't work either. Different teams or departments need some level of interaction, or at least a shared sense of purpose. In fact, dividing sociological groups is a tactic deployed in both politics and war as a mechanism for reducing the capabilities of an opponent. The point here is that a balance must be struck, organizationally, in terms of the way people, responsibility, and communication structures are arranged in a company, and this applies as acutely to software as it does to people, because beyond the confines of a single application, people factors invariably dominate.

Some companies tackle this at an organizational level using approaches like the Inverse Conway Maneuver, which applies the idea that, if the shape of software and the shape of organizations are intrinsically linked (as Conway argued), then it's often easier to change the organization and let the software follow suit than it is to do the reverse. But regardless of the approach taken, when we design software systems where components are operated and evolved independently, the problem we face has three distinct parts—organization, software, and data—which are all intrinsically linked. To complicate matters further, what really dif-

1 Tom DeMarco, *Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency* (New York: Broadway Books, 2001).

ferentiates the good systems from the bad is their ability to manage these three factors as they evolve, independently, over time.

While this may seem a little abstract, you have no doubt felt the interplay between these forces before. Say you're working on a project, but to finish it you need three other teams to complete work on their side first—you know intuitively that it's going to take longer to build, schedule, and release. If someone from another team asks if their application can pull some data out of your database, you know that's probably going to lead to pain in the long run, as you're left wondering if your latest release will break the dependency they have on you. Finally, while it might seem trivial to call a couple of REST services to populate your user interface, you know that an outage on their side is going to mean you get called at 3 a.m., and the more complex the dependencies get, the harder it's going to be to figure out why the system doesn't work. These are all examples of problems we face when an organization, its software, and its data evolve slowly.

The Microservices pattern is unusually opinionated in this regard. It comes down hard on independence in organization, software, and data. Microservices are run by different teams, have different deployment cycles, don't share code, and don't share databases.[2] The problem is that replacing this with a web of RPC/REST calls isn't generally a preferable solution. This leads to an important tension: we want to promote reuse to develop software quickly, but at the same time the more dependencies we have, the harder it is to change.

Reuse can be a bad thing. Reuse lets us develop software quickly and succinctly, but the more we reuse a component, the more dependencies that component has, and the harder it is to change.

To better understand this tension, we need to question some core principles of software design—principles that work wonderfully when we're building a single application, but fare less well when we build software that spans many teams.

Encapsulation Isn't Always Your Friend

As software engineers we're taught to encapsulate. If you're building a library for other people to use, you'll carefully pick a contract that suits the functionality you want to expose. If you're building a service-based system, you might be inclined to follow a similar process. This works well if we can cleanly separate responsibilities between the different services. A single sign-on (SSO) service, for example, has a well-defined role, which is cleanly separated from the roles other

2 Sam Newman, *Building Microservices* (Sebastopol, CA: O'Reilly, 2014).

services play (Figure 8-1). This clean separation means that, even in the face of rapid requirement churn, it's unlikely the SSO service will need to change. It exists in a tightly bounded context.

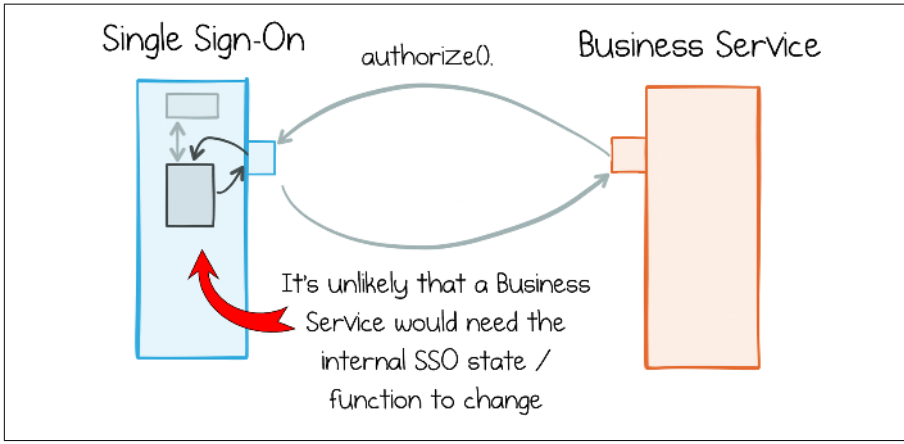


Figure 8-1. An SSO service provides a good example of encapsulation and reuse

The problem is that, in the real world, business services can't typically retain the same clean separation of concerns, meaning new requirements will inevitably crosscut service boundaries and several services will need to change at once. This can be measured.[3] So if one team needs to implement a feature, and that requires another team to make a code change, we end up having to make changes to both services at around the same time. In a monolithic system this is pretty straightforward—you make the change and then do a release—but it's considerably more painful where independent services must synchronize. The coordination between teams and release cycles erodes agility.

This problem isn't actually restricted to services. Shared libraries suffer from the same problem. If you work in retail, it might seem sensible to create a library that models how customers, orders, payments, and the like all relate to one another. You could include common logic for standard operations like returns and refunds. Lots of people did this in the early days of object orientation, but it turned out to be quite painful because suddenly the most sensitive part of your system was coupled to many different programs, making it really fiddly to change and release. This is why microservices typically don't share a single domain model. But some library reuse is of course OK. Take a logging library, for example—much like the earlier SSO example, you're unlikely to have a business requirement that needs the logging library to change.

3 See *https://www.infoq.com/news/2017/04/tornhill-prioritise-tech-debt* and *http://bit.ly/2pKa2rR*.

But in reality, of course, library reuse comes with a get-out clause: the code can be implemented anywhere. Say you did use the aforementioned shared retail domain model. If it becomes too painful to use, you could always just write the code yourself! (Whether that is actually a good idea is a different discussion.) But when we consider different applications or services that share data with one another, there is no such solution: *if you don't have the data, there is literally nothing you can do*.

This leads to two fundamental differences between services and shared libraries:

- A service is run and operated by someone else.
- A service typically has data of its own, whereas a library (or database) requires you to input any data it needs.

Data sits at the very heart of this problem: most business services inevitably rely heavily on one another's data. If you're an online retailer, the stream of orders, the product catalog, or the customer information will find its way into the requirements of many of your services. Each of these services needs broad access to these datasets to do its work, and there is no temporary workaround for not having the data you need. So you need access to shared datasets, but you also want to stay loosely coupled. This turns out to be a pretty hard bargain to strike.

The Data Dichotomy

Encapsulation encourages us to hide data, but data systems have little to do with encapsulation. In fact, quite the opposite: databases do everything they can to expose the data they hold (Figure 8-2). They come with wonderfully powerful, declarative interfaces that can contort the data they hold into pretty much any shape you might desire. That's exactly what a data scientist needs for an exploratory investigation, but it's not so great for managing the spiral of interservice dependencies in a burgeoning service estate.

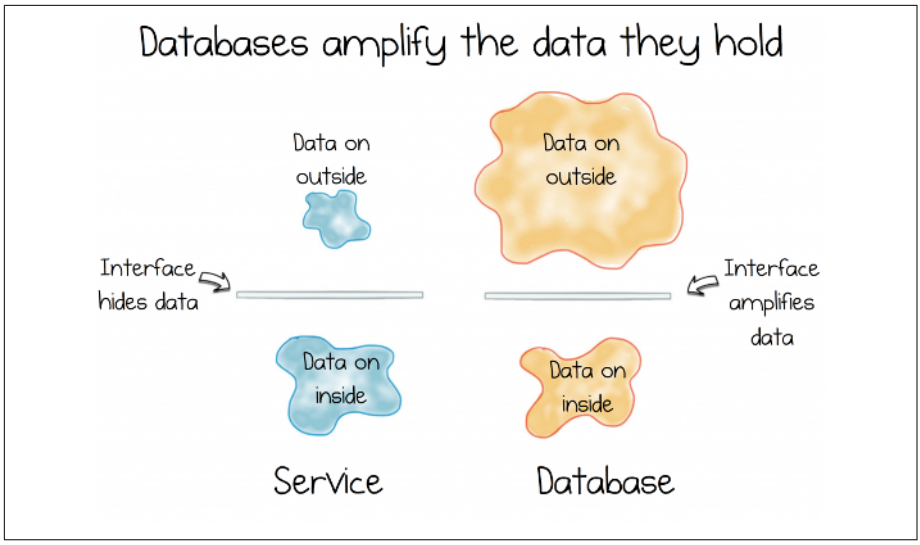


Figure 8-2. Services encapsulate the data they hold to reduce coupling and aid reuse; databases amplify the data they hold to provide greater utility to their user

So we find ourselves faced with a conundrum, a dichotomy: databases are about exposing data and making it useful. Services are about hiding it so they can stay decoupled. These two forces are fundamental. They underlie much of what we do, subtly jostling for supremacy in the systems we build.

What Happens to Systems as They Evolve?

As systems evolve and grow we see the effects of this data dichotomy play out in a couple of different ways.

The God Service Problem

As data services grow they inevitably expose an increasing set of functions, to the point where they start to look like some form of kooky, homegrown database (Figure 8-3).

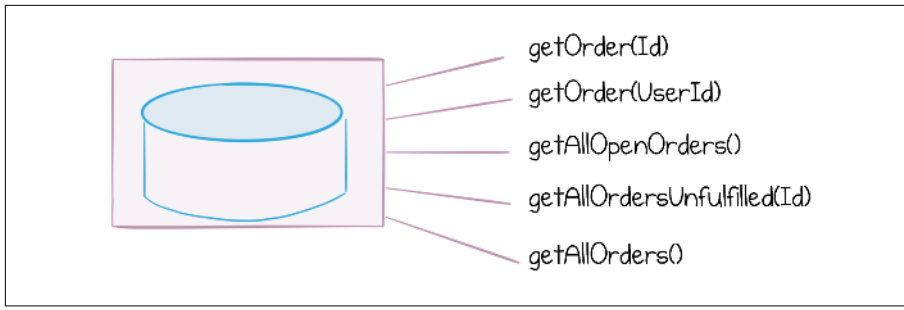


Figure 8-3. Service interfaces inevitably grow over time

Now creating something that looks like a kooky, shared database can lead to a set of issues of its own. The more functionality, data, and users data services have, the more tightly coupled they become and the harder (and more expensive) they are to operate and evolve.

The REST-to-ETL Problem

A second, often more common, issue when you're faced with a data service is that it actually becomes preferable to suck the data out so it can be held and manipulated locally (Figure 8-4). There are lots of reasons for this to happen in practice, but some of the main ones are:

- The data needs to be combined with some other dataset.
- The data needs to be closer, either due to geography or to be used offline (e.g., on a mobile).
- The data service displays operational issues, which cause outages downstream.
- The data service doesn't provide the functionality the client needs and/or can't change quick enough.

But to extract data from some service, then keep that data up to date, you need some kind of polling mechanism. While this is not altogether terrible, it isn't ideal either.

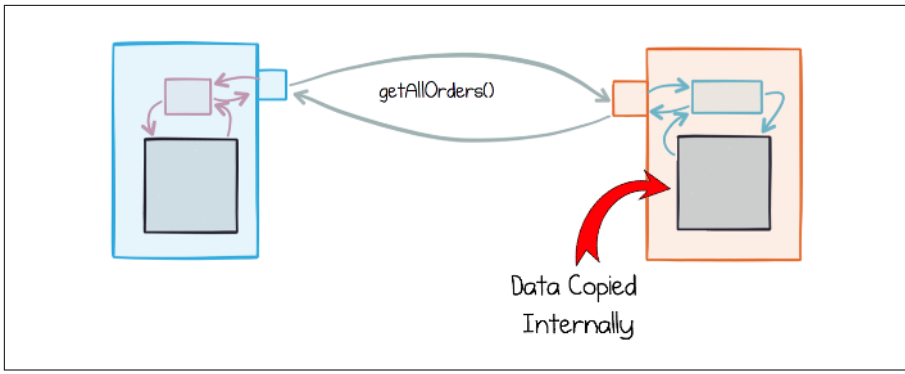


Figure 8-4. Data is moved from service to service en masse

What's more, as this happens again and again in larger architectures, with data being extracted and moved from service to service, little errors or idiosyncrasies often creep in. Over time these typically worsen and the data quality of the whole ecosystem starts to suffer. The more mutable copies, the more data will diverge over time.

Making matters worse, divergent datasets are very hard to fix in retrospect. (Techniques like master data management are in many ways a Band-aid over this.) In fact, some of the most intractable technology problems that businesses encounter arise from divergent datasets proliferating from application to application. This issue is discussed in more detail in Chapter 10.

So a cyclical pattern of behavior emerges between (a) the drive to centralize datasets to keep them accurate and (b) the temptation (or need) to extract datasets and go it alone—an endless cycle of data inadequacy (Figure 8-5).

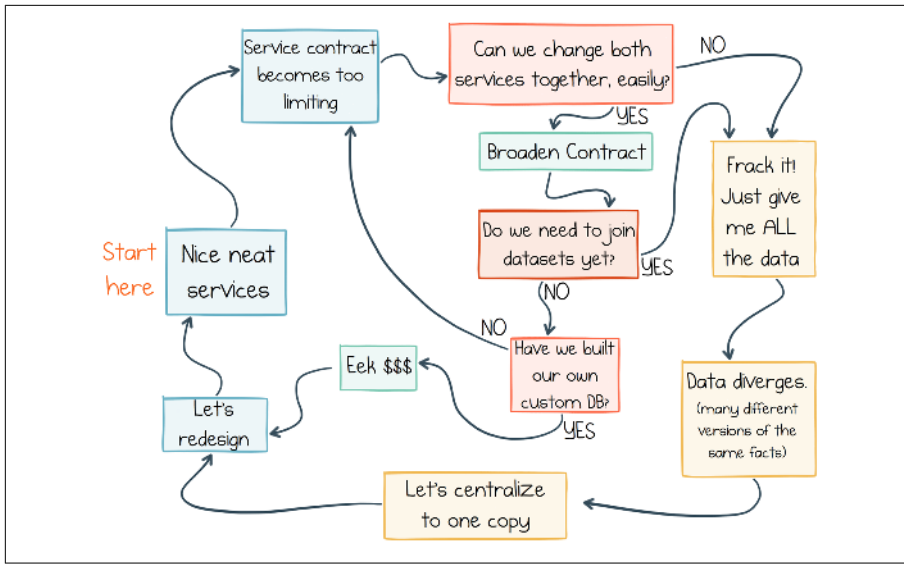


Figure 8-5. The cycle of data inadequacy

Make Data on the Outside a First-Class Citizen

To address these various issues, we need to think about shared data in a slightly different way. We need to consider it a first-class citizen of the architectures we build. Pat Helland makes this distinction in his paper "Data on the Inside and Data on the Outside."

One of the key insights he makes is that the data services share needs to be treated differently from the data they hold internally. Data on the outside is hard to change, because many programs depend upon it. But, for this very reason, data on the outside is the most important data of all.

A second important insight is that service teams need to adopt an openly outward-facing role: one designed to serve, and be an integral part of, the wider ecosystem. This is very different from the way traditional applications are built: written to operate in isolation, with methods for exposing their data bolted on later as an afterthought.

With these points in mind it becomes clear that data on the outside—the data services share—needs to be carefully curated and nurtured, but to keep our freedom to iterate we need to turn it into data on the inside so that we can make it our own.

The problem is that none of the approaches available today—service interfaces, messaging, or a shared database—provide a good solution for dealing with this transition (Figure 8-6), for the following reasons:

- Service interfaces form tight point-to-point couplings, make it hard to share data at any level of scale, and leave the unanswered question: how do you join the many islands of state back together?
- Shared databases concentrate use cases into a single place, and this stifles progress.
- Messaging moves data from a tightly coupled place (the originating service) to a loosely coupled place (the service that is using the data). This means datasets can be brought together, enriched, and manipulated as required. Moving data locally typically improves performance, as well as decoupling sender and receiver. Unfortunately, messaging systems provide no historical reference, which means it's harder to bootstrap new applications, and this can lead to data quality issues over time (discussed in "The Data Divergence Problem" on page 95 in Chapter 10).

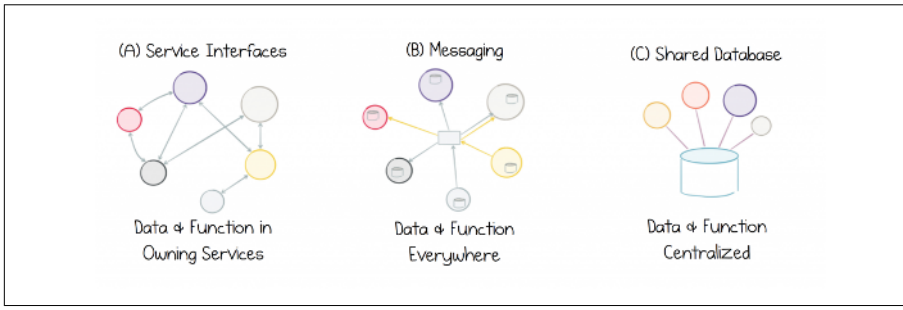


Figure 8-6. Tradeoff between service interfaces, messaging, and a shared database

A better solution is to use a replayable log like Kafka. This works like a kind of event store: part messaging system, part database.

NOTE

Messaging turns highly coupled, shared datasets (data on the outside) into data a service can own and control (data on the inside). Replayable logs go a step further by adding a central reference.

Don't Be Afraid to Evolve

When you start a new project, form a new department, or launch a new company, you don't need to get everything right from the start. Most projects evolve. They start life as monoliths, and later they add distributed components, evolve into microservices, and add event streaming. The important point is when the approach becomes constraining, you change it. But experienced architects know where the tipping point for this lies. Leave it too late, and change can become too

costly to schedule. This is closely linked with the concept of fitness functions in evolutionary architectures.[4]

Summary

Patterns like microservices are opinionated when it comes to services being independent: services are run by different teams, have different deployment cycles, don't share code, and don't share databases. The problem is that replacing this with a web of RPC calls fails to address the question: how do services get access to these islands of data for anything beyond trivial lookups?

The data dichotomy highlights this question, underlining the tension between the need for services to stay decoupled and their need to control, enrich, and combine data in their own time.

This leads to three core conclusions: (1) as architectures grow and systems become more data-centric, moving datasets from service to service becomes an inevitable part of how systems evolve; (2) data on the outside—the data services share—becomes an important entity in its own right; (3) sharing a database is not a sensible solution to data on the outside, but sharing a replayable log better balances these concerns, as it can hold datasets long-term, and it facilitates event-driven programming, reacting to the now.

This approach can keep data across many services in sync, through a loosely coupled interface, giving them the freedom to slice, dice, enrich, and evolve data locally.

4 Neil Ford, Rebecca Parsons, and Pat Kua, *Building Evolutionary Architectures* (Sebastopol, CA: O'Reilly, 2017).

CHAPTER 9

Event Streams as a Shared Source of Truth

As we saw in Part II of this book, events are a useful tool for system design, providing notification, state transfer, and decoupling. For a couple of decades now, messaging systems have leveraged these properties, moving events from system to system, but only in the last few years have messaging systems started to be used as a storage layer, retaining the datasets that flow through them. This creates an interesting architectural pattern. A company's core datasets are stored as centralized event streams, with all the decoupling effects of a message broker built in. But unlike traditional brokers, which delete messages once they have been read, historic data is stored and made available to any team that needs it. This links closely with the ideas developed in Event Sourcing (see Chapter 7) and Pat Helland's concept of data on the outside. ThoughtWorks calls this pattern event streaming as the source of truth.

A Database Inside Out

"Turning the database inside out" was a phrase coined by Martin Kleppmann. Essentially it is the idea that a database has a number of core components—a commit log, a query engine, indexes, and caching—and rather than conflating these concerns inside a single black-box technology like a database does, we can split them into separate parts using stream processing tools and these parts can exist in different places, joined together by the log. So Kafka plays the role of the commit log, a stream processor like Kafka Streams is used to create indexes or views, and these views behave like a form of continuously updated cache, living inside or close to your application (Figure 9-1).

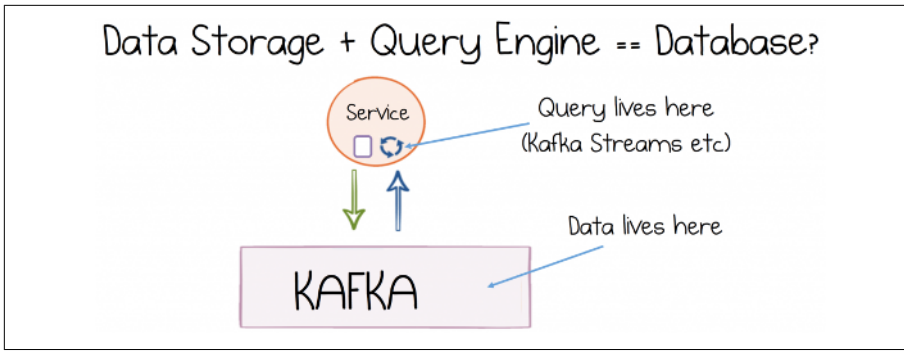


Figure 9-1. A streaming engine and a replayable log have the core components of a database

As an example we might consider this pattern in the context of a simple GUI application that lets users browse order, payment, and customer information in a scrollable grid. Because the user can scroll the grid quickly up and down, the data would likely need to be cached locally. But in a streaming model, rather than periodically polling the database and then caching the result, we would define a view that represents the exact dataset needed in the scrollable grid, and the stream processor would take care of materializing it for us. So rather than querying data in a database, then layering caching over the top, we explicitly push data to where it is needed and process it there (i.e., it's inside the GUI, right next to our code).

But while we call "turning the database inside out" a pattern, it would probably be more accurate to call it an analogy: a different way of explaining what stream processing is. It is a powerful one, though. One reason that it seems to resonate with people is we have a deep-seated notion that pushing business logic into a database is a bad idea. But the reverse—pushing data into your code—opens up a wealth of opportunities for blending our data and our code together. So stream processing flips the traditional approach to data and code on its head, encouraging us to bring data into the application layer—to create tables, views, and indexes exactly where we need them.

NOTE

"The database inside out" is an analogy for stream processing where the same components we find in a database—a commit log, views, indexes, caches—are not confined to a single place, but instead can be made available wherever they are needed.

This idea actually comes up in a number of other areas too. The Clojure community talks about deconstructing the database. There are overlaps with Event Sourcing and polyglot persistence as we discussed in Chapter 7. But the idea was

originally proposed by Jay Kreps back in 2013, where he calls it "unbundling" but frames it in a slightly different context, and this turns out to be quite important:

> There is an analogy here between the role a log serves for data flow inside a distributed database and the role it serves for data integration in a larger organization…if you squint a bit, you can see the whole of your organization's systems and data flows as a single distributed database. You can view all the individual query-oriented systems (Redis, SOLR, Hive tables, and so on) as just particular indexes on your data. You can view the stream processing systems like Storm or Samza as just a very well-developed trigger and view materialization mechanism. Classical database people, I have noticed, like this view very much because it finally explains to them what on earth people are doing with all these different data systems—they are just different index types!

What is interesting about Jay's description is he casts the analogy in the context of a whole company. The key insight is essentially the same: stream processing segregates responsibility for data storage away from the mechanism used to query it. So there might be one shared event stream, say for payments, but many specialized views, in different parts of the company (Figure 9-2). But this provides an important alternative to traditional mechanisms for data integration. More specifically:

- The log makes data available centrally as a shared source of truth but with the simplest possible contract. This keeps applications loosely coupled.
- Query functionality is not shared; it is private to each service, allowing teams to move quickly by retaining control of the datasets they use.

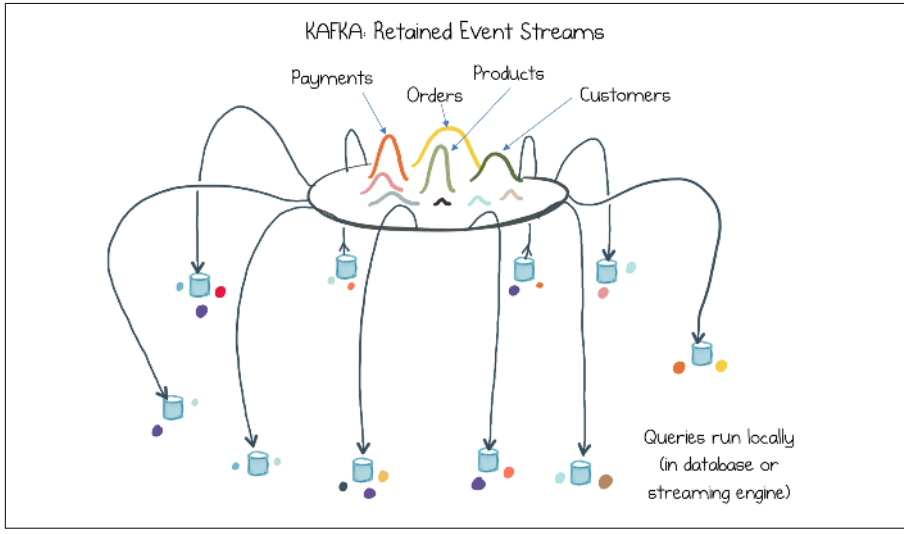


Figure 9-2. A number of different applications and services, each with its own views, derived from the company's core datasets held in Kafka; the views can be optimized for each use case

NOTE

The "database inside out" idea is important because a replayable log, combined with a set of views, is a far more malleable entity than a single shared database. The different views can be tuned to different people's needs, but are all derived from the same source of truth: the log.

As we'll see in Chapter 10, this leads to some further optimizations where each view is optimized to target a specific use case, in much the same way that materialized views are used in relational databases to create read-optimized, use-case-focused datasets. Of course, unlike in a relational database, the view is decoupled from the underlying data and can be regenerated from the log should it need to be changed. (See "Materialized Views" on page 62 in Chapter 7.)

Summary

This chapter introduced the analogy that stream processing can be viewed as a database turned inside out, or unbundled. In this analogy, responsibility for data storage (the log) is segregated from the mechanism used to query it (the Stream Processing API). This makes it possible to create views and embed them exactly where they are needed—in another application, in another geography, or on another platform. There are two main drivers for pushing data to code in this way:

- As a performance optimization, by making data local
- To decouple the data in an organization, but keep it close to a single shared source of truth

So at an organizational level, the pattern forms a kind of database of databases where a single repository of event data is used to feed many views, and each view can flex with the needs of that particular team.

CHAPTER 10
Lean Data

Lean data is a simple idea: rather than collecting and curating large datasets, applications carefully select small, lean ones—just the data they need at a point in time—which are pushed from a central event store into caches, or stores they control. The resulting lightweight views are propped up by operational processes that make rederiving those views practical.

If Messaging Remembers, Databases Don't Have To

One interesting consequence of using event streams as a source of truth (see Chapter 9) is that any data you extract from the log doesn't need to be stored reliably. If it is lost you can always go back for more. So if the messaging layer remembers, downstream databases don't have to (Figure 10-1). This means you can, if you choose, regenerate a database or event-sourced view completely from the log. This might seem a bit strange. Why might you want to do that?

In the context of traditional messaging, ETL (extract, transform, load) pipelines, and the like, the messaging layer is ephemeral, and users write all messages to a database once the messages have been read. After all, they may never see them again. There are a few problems that result from this. Architectures become comparably heavyweight, with a large number of applications and services retaining copies of a large proportion of the company's data. At a macro level, as time passes, all these copies tend to diverge from one another and data quality issues start to creep in.

Data quality issues are common and often worse than people suspect. They arise for a great many reasons. One of these is linked to our use of databases as long-lived resources that are tweaked and tuned over time, leading to inadvertently introduced errors. A database is essentially a file, and that file will be as old as the system it lives in. It will have been copied from environment to environment,

and the data in the database will have been subject to many operational fixes over its lifetime. So it is unsurprising that errors and inaccuracies creep in.

In stream processing, files aren't copied around in this way. If a stream processor creates a view, then does a release that changes the shape of that view, it typically throws the original view away, resets to offset 0, and derives a new one from the log.

Looking to other areas of our industry—DevOps and friends—we see similar patterns. There was a time when system administrators would individually tweak, tune, and mutate the computers they managed. Those computers would end up being subtly different from one another, and when things went wrong it was often hard to work out why.

Today, issues like these have been largely solved within as-a-service cultures that favor immutability through infrastructure as code. This approach comes with some clear benefits: deployments become deterministic, builds are identical, and rebuilds are easy. Suddenly ops engineers transform into happy people empowered by the predictability of the infrastructure they wield, and comforted by the certainty that their software will do exactly what it did in test.

Streaming encourages a similar approach, but for data. Event-sourced views are kept lean and can be rederived from the log in a deterministic way. The view could be a cache, a Kafka Streams state store, or a full-blown database. But for this to work, we need to deal with a problem. Loading data can be quite slow. In the next section we look at ways to keep this manageable.

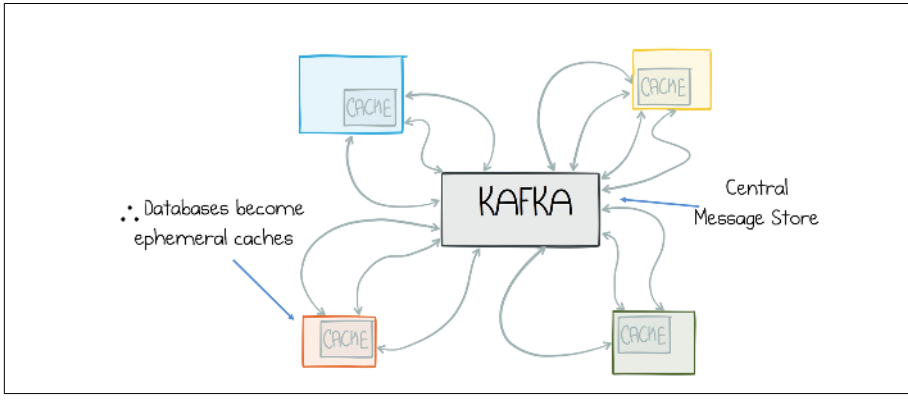


Figure 10-1. If the messaging system can store data, then the views or databases it feeds don't have to

Take Only the Data You Need, Nothing More

If datasets are stored in Kafka, when you pull data into your service you can pick out just the pieces you need. This minimizes the size of the resulting views and

allows them to be reshaped so that they are read-optimized. This is analogous to the way materialized views are used in relational databases to optimize for reads except, unlike in a relational database, writes and reads are decoupled. (See "Materialized Views" on page 62 in Chapter 7.)

The inventory service, discussed in Chapter 15, makes a good example. It reads inventory messages that include lots of information about the various products stored in the warehouse. When the service reads each message it throws away the vast majority of the document, stripping it back to just two fields: the product ID and the number of items in stock.

Reducing the breadth of the view keeps the dataset small and loosely coupled. By keeping the dataset small you can store more rows in the available memory or disk, and performance will typically improve as a result. Coupling is reduced too, since should the schema change, it is less likely that the service will store affected fields.

If messaging remembers, derived views can be refined to contain only the data that is absolutely necessary. You can always go back for more.

The approach is simple to implement in either the Kafka's Streams DSL or KSQL, or by using the Connect API's single message transforms feature.

Rebuilding Event-Sourced Views

The obvious drawback of lean data is that, should you need more data, you need to go back to the log. The cleanest way to do this is to drop the view and rebuild it from scratch. If you're using an in-memory data structure, this will happen by default, but if you're using Kafka Streams or a database, there are other factors to consider.

Kafka Streams

If you create event-sourced views with Kafka Streams, view regeneration is par for the course. Views are either tables, which are a direct materialization of a Kafka topic, or state stores, which are populated with the result of some declarative transformation, defined in JVM code or KSQL. Both of these are automatically rebuilt if the disk within the service is lost (or removed) or if the Streams Reset tool is invoked.[1] We discussed how Kafka Streams manages statefulness in

1 See *http://bit.ly/2GaCRZO* and *http://bit.ly/2IUPHJa*.

more detail in the section "The Practicalities of Being Stateful" on page 52 in Chapter 6.

Databases and Caches

In today's world there are many different types of databases with a wide range of performance tradeoffs. While regenerating a 50 TB Oracle database from scratch would likely be impractical, regenerating the event-sourced views used in business services is often quite workable with careful technology choice.

Because worst-case regeneration time is the limiting factor, it helps to pick a write-optimized database or cache. There are a great many options, but sensible choices include:

- An in-memory database/cache like Redis, MemSQL, or Hazelcast
- A memory-optimized database like Couchbase or one that lets you disable journaling like MongoDB
- A write/disk optimized, log-structured database like Cassandra or RocksDB

Handling the Impracticalities of Data Movement

Rebuilding an event-sourced view may still be a relatively long-winded process (minutes or even hours![2]). Because of this lead time, when releasing new software, developers typically regenerate views in advance (or in parallel), with the switch from old view to new view happening when the view is fully caught up. This is essentially the same approach taken by *stateful* stream processing applications that use Kafka Streams.

The pattern works well for simple services that need small- to medium-sized datasets, say, to drive rules engines. Working with larger datasets means slower load times. If you need to rebuild terabyte-sized datasets in a single instance of a highly indexed, disk-resident database, this load time could well be prohibitively long. But in many common cases, memory-based solutions that have fast write times, or horizontal scaling, will keep ingestion fast.

Automation and Schema Migration

Operators typically lose trust in tools that are not regularly used. This is even more true when those scripts operate on data (database rollback scripts are noto-

2 As a yardstick, RocksDB (which Kafka Streams uses) will bulk-load ~10M 500 KB objects per minute (roughly GbE speed). Badger will do ~4M × 1K objects a minute per SSD. Postgres will bulk-load ~1M rows per minute.

rious). So when you move from environment to environment, as part of your development workflow, it's often best to recreate views directly from the log rather than copying database files from environment to environment as you might in a traditional database workflow (Figure 10-2).

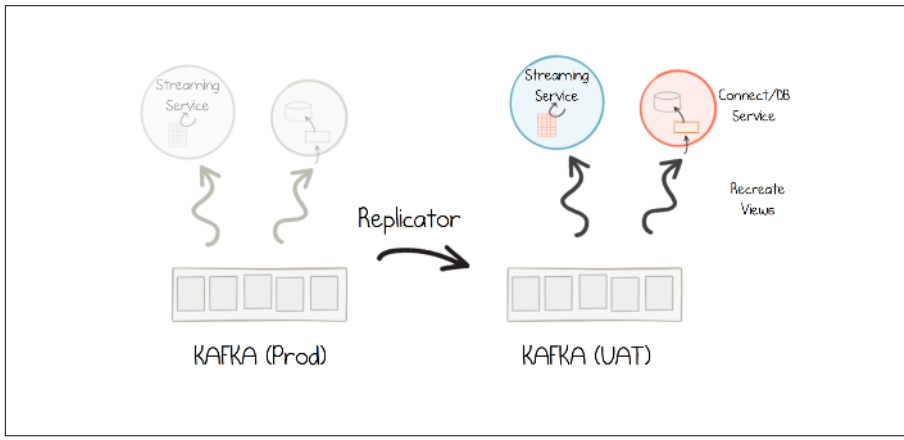


Figure 10-2. Data is replicated to a UAT environment where views are regenerated from source

A good example of this is when schemas change. If you have used traditional messaging approaches before to integrate data from one system into another, you may have encountered a time when the message schema changed in a non-backward-compatible way. For example, if you were importing customer information from a messaging system into a database when the customer message schema undergoes a breaking change, you would typically craft a database script to migrate the data forward, then subscribe to the new topic of messages.

If using Kafka to hold datasets in full,[3] instead of performing a schema migration, you can simply drop and regenerate the view.

The Data Divergence Problem

In practice, all large companies start to have problems with data quality as they grow. This is actually a surprisingly deep and complex subject, with the issues that result being painstaking, laborious, and expensive to fix.

The roots of these issues come from a variety of places. Consider a typical business application that loads data from several other systems and stores that data in a database. There is a data model used on the wire (e.g., JSON), an internal

3 We assume the datasets have been migrated forward using a technique like dual-schema upgrade window, discussed in "Handling Schema Change and Breaking Backward Compatibility" on page 124 in Chapter 13.

domain model (e.g., an object model), a data model in the database (e.g., DDL), and finally various schemas for any outbound communication. Code needs to be written for each of these translations, and this code needs to be evolved as the various schemas change. These layers, and the understanding required for each one, introduce opportunities for misinterpretation.

Semantic issues are even trickier to address, often arising where teams, departments, or companies meet. Consider two companies going through a merger. There will be a host of equivalent datasets that were modeled differently by each side. You might think of this as a simple transformation problem, but typically there are far deeper semantic conflicts: Is a supplier a customer? Is a contractor an employee? This opens up more opportunity for misinterpretation.

So as data is moved from application to application and from service to service, these interactions behave a bit like a game of telephone (a.k.a. Chinese whispers): everything starts well, but as time passes the original message gets misinterpreted and transforms into something quite different.

Some of the resulting issues can be serious: a bank whose Risk and Finance departments disagree on the bank's position, or a retailer—with a particularly protracted workflow—taking a week to answer customer questions. All large companies have stories like these of one form or other. So this isn't a problem you typically face building a small web application, but it's a problem faced by many larger, more mature architectures.

There are tried-and-tested methods for addressing these concerns. Some companies create reconciliation systems that can turn into small cottage industries of their own. In fact, there is a whole industry dedicated to combating this problem —master data management—along with a whole suite of tools for data wrangling such issues toward a shared common ground.

Streaming platforms help address these problems in a slightly different way. First, they form a kind of central nervous system that connects all applications to a single shared source of truth, reducing the telephone/Chinese whispers effect. Secondly, because data is retained immutably in the log, it's easier to track down when an error was introduced, and it's easier to apply fixes with the original data on hand. So while we will always make mistakes and misinterpretations, techniques like event streams as a source of truth, Command Sourcing, and lean data allow individual teams to develop the operational maturity needed to avoid mistakes in the first place, or repair the effects of them once they happen.

Summary

So when it comes to data, we should be unequivocal about the shared facts of our system. They are the very essence of our business, after all. Lean practices encourage us to stay close to these shared facts, a process where we manufacture

views that are specifically optimized to the problem space at hand. This keeps them lightweight and easier to regenerate, leveraging a similar mindset to that developed in Command Sourcing and Event Sourcing, which we discussed in Chapter 7. So while these facts may be evolved over time, applied in different ways, or recast to different contexts, they will always tie back to a single source of truth.

PART IV

Consistency, Concurrency, and Evolution

Trust is built with consistency.

—Lincoln Chafee

CHAPTER 11

Consistency and Concurrency in Event-Driven Systems

The term *consistency* is quite overused in our industry, with several different meanings applied in a range of contexts. Consistency in CAP theorem differs from consistency in ACID transactions, and there is a whole spectrum of subtly different guarantees, including strong consistency and eventual consistency, among others. The lack of consistent terminology around this word may seem a little ironic, but it is really a reflection of the complexity of a subject that goes way beyond the scope of this book.[1]

But despite these many subtleties, most people have an intuitive notion of what consistency is, one often formed from writing single-threaded programs[2] or making use of a database. This typically equates to some general notions about the transactional guarantees a database provides. When you write a record, it stays written. When you read a record, you read the most recently written value. If you perform multiple operations in a transaction, they all become visible at once, and you don't need to be concerned with what other users may be doing at the same

1 For a full treatment, see Martin Kleppmann's encyclopedic *Designing Data-Intensive Applications* (Sebastopol, CA: O'Reilly, 2017).

2 The various consistency models really reflect optimizations on the concept of in-order execution against a single copy of data. These optimizations are necessary in practice, and most users would prefer to trade a slightly weaker guarantee for the better performance (or availability) characteristics that typically come with them. So implementers come up with different ways to slacken the simple "in-order execution" guarantee. These various optimizations lead to different consistency models, and because there are many dimensions to optimize, particularly in distributed systems, there are many resulting models. When we discuss "Scaling Concurrent Operations in Streaming Systems" on page 142 in Chapter 15, we'll see how streaming systems achieve strong guarantees by partitioning relevant data into different stream threads, then wrapping those operations in a transaction, which ensures that, for that operation, we have in-order execution on a single copy of data (i.e., a strong consistency model).

time. We might call this idea *intuitive consistency* (which is closest in technical terms to the famous ACID properties).

A common approach to building business systems is to take this intuitive notion and apply it directly. If you build traditional three-tier applications (i.e., client, server, and database), this is often what you would do. The database manages concurrent changes, isolated from other users, and everyone observes exactly the same view at any one point in time. But groups of services generally don't have such strong guarantees. A set of microservices might call one another synchronously, but as data moves from service to service it will often become visible to users at different times, unless all services coordinate around a single database and force the use of a single global consistency model.

But in a world where applications are distributed (across geographies, devices, etc.), it isn't always desirable to have a single, global consistency model. If you create data on a mobile device, it can only be consistent with data on a backend server if the two are connected. When disconnected, they will be, by definition, inconsistent (at least in that moment) and will synchronize at some later point, *eventually* becoming consistent. But designing systems that handle periods of inconsistency is important. For a mobile device, being able to function offline is a desirable feature, as is resynchronizing with the backend server when it reconnects, converging to consistency as it does so. But the usefulness of this mode of operation depends on the specific work that needs to be done. A mobile shopping application might let you select your weekly groceries while you're offline, but it can't work out whether those items will be available, or let you physically buy anything until you come back online again. So these are use cases where global strong consistency is undesirable.

Business systems often don't need to work offline in this way, but there are still benefits to avoiding global strong consistency and distributed transactions: they are difficult and expensive to scale, don't work well across geographies, and are often relatively slow. In fact, experience with distributed transactions that span different systems, using techniques like XA, led the majority of implementers to design around the need for such expensive coordination points.

But on the other hand, business systems typically want strong consistency to reduce the potential for errors, which is why there are vocal proponents who consider stronger safety properties valuable. There is also an argument for wanting a bit of both worlds. This middle ground is where event-driven systems sit, often with some form of eventual consistency.

Eventual Consistency

The previous section refers to an intuitive notion of consistency: the idea that business operations execute sequentially on a single copy of data. It's quite easy

to build a system that has this property. Services call one another through RPCs, just like methods in a single-threaded program: a set of sequential operations. Data is passed by reference, using an ID. Each service looks up the data it needs in the database. When it needs to change it, it changes it in the database. Such a system might look something like Figure 11-1.

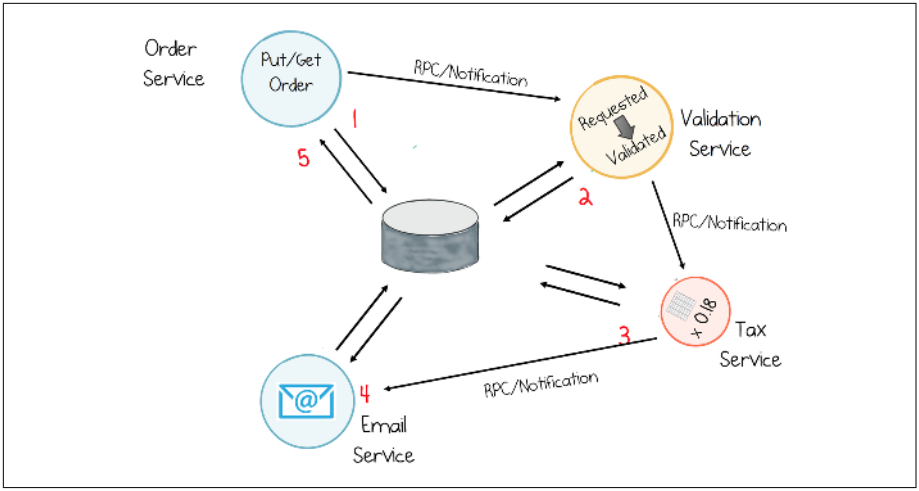


Figure 11-1. A set of services that notify one another and share data via a database

This approach provides a very intuitive model as everything progresses sequentially, but as services scale out and more services are added, it can get hard to scale and operate systems that follow this approach.

Event-driven systems aren't typically built in this way. Instead, they leverage asynchronous broadcast, deliberately removing the need for global state and avoiding synchronous execution. (We went through the issues with global shared state in "What Happens to Systems as They Evolve?" on page 80 in Chapter 8.) Such systems are often referred to as being "eventually consistent."

There are two consequences of eventual consistency in this context:

Timeliness
: If two services process the same event stream, they will process them at different rates, so one might lag behind the other. If a business operation consults *both* services for any reason, this could lead to an inconsistency.

Collisions
: If different services make changes to the same entity in the same event stream, if that data is later coalesced—say in a database—some changes might be lost.

Let's dig into these with an example that continues our theme of online retail systems. In Figure 11-2 an order is accepted in the orders service (1). This is picked

up by the validation service, where it is validated (2). Sales tax is added (3). An email is sent (4). The updated order goes back to the orders service (5), where it can be queried via the orders view (this is an implementation of CQRS, as we discussed in Chapter 7). After being sent a confirmation email (6), the user can click through to the order (7).

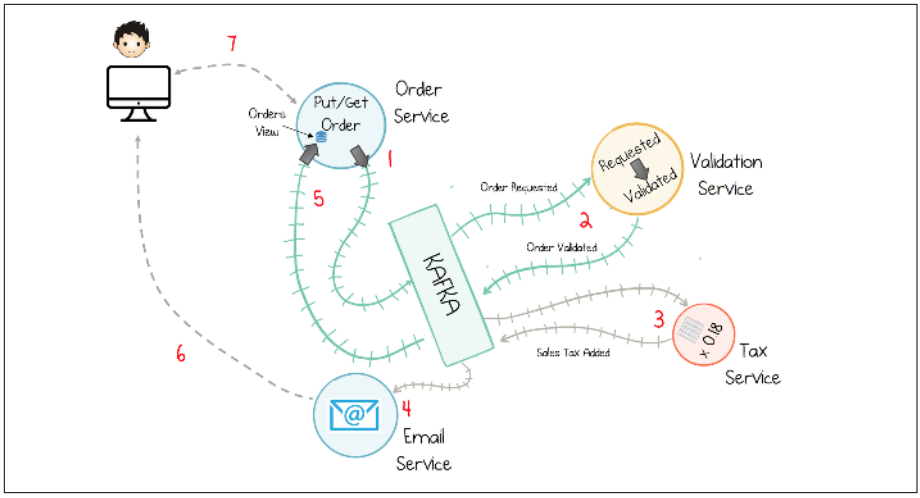


Figure 11-2. An event-driven system connected via a log

Timeliness

Consider the email service (4) and orders view (5). Both subscribe to the same event stream (Validated Orders) and process them concurrently. Executing concurrently means one will lag slightly behind the other. Of course, if we stopped writes to the system, then both the orders view and the email service would *eventually converge* on the same state, but in normal operation they will be at slightly different positions in the event stream. So they lack timeliness with respect to one another. This could cause an issue for a user, as there is an indirect connection between the email service and the orders service. If the user clicks the link in the confirmation email, but the view in the orders service is lagging, the link would either fail or return an incorrect state (7).

So a lack of timeliness (i.e., lag) can cause problems if services are linked in some way, but in larger ecosystems it is beneficial for the services to be decoupled, as it allows them to do their work concurrently and in isolation, and the issues of timeliness can usually be managed (this relates closely to the discussion around CQRS in Chapter 7).

But what if this behavior is unacceptable, as this email example demonstrates? Well, we can always add serial execution back in. The call to the orders service might block until the view is updated (this is the approach taken in the worked

example in Chapter 15). Alternatively, we might have the orders service raise a View Updated event, used to trigger the email service after the view has been updated. Both of these synthesize serial execution where it is necessary.

Collisions and Merging

Collisions occur if two services update the same entity at the same time. If we design the system to execute serially, this won't happen, but if we allow concurrent execution it can.

Consider the validation service and tax service in Figure 11-2. To make them run serially, we might force the tax service to execute first (by programming the service to react to Order Requested events), then force the validation service to execute next (by programming the service to react to events that have had sales tax added). This linearizes execution for each order and means that the final event will have all the information in (i.e., it is both validated and has sales tax added). Of course, making events run serially in this way increases the end-to-end latency.

Alternatively, we can let the validation service and the tax service execute concurrently, but we'd end up with two events with important information in each: one validated order and one order with sales tax added. This means that, to get the correct order, with both validation and sales tax applied, we would have to *merge* these two messages. (So, in this case, the merge would have to happen in both the email service and in the orders view.)

In some situations this ability to make changes to the same entity in different processes at the same time, and merge them later, can be extremely powerful (e.g., an online whiteboarding tool). But in others it can be error-prone. Typically, when building business systems, particularly ones involving money and the like, we tend to err on the side of caution. There is a formal technique for merging data in this way that has guaranteed integrity; it is called a conflict-free replicated data type, or CRDT. CRDTs essentially restrict what operations you can perform to ensure that, when data is changed and later merged, you don't lose information. The downside is that the dialect is relatively limited.

A good compromise for large business systems is to keep the lack of timeliness (which allows us to have lots of replicas of the same state, available read-only) but remove the opportunity for collisions altogether (by disallowing concurrent mutations). We do this by allocating a single writer to each type of data (topic) or alternatively to each state transition. We'll talk about this next.

The Single Writer Principle

A useful way to generify these ideas is to isolate consistency concerns into owning services using the *single writer principle*. Martin Thompson used this term in

response to the wide-scale use of locks in concurrent environments, and the subsequent efficiencies that we can often gain by consolidating writes to a single thread. The core idea closely relates to the Actor model,[3] several ideas in database research,[4] and anecdotally to system design. From a services perspective, it also marries with the idea that services should have a single responsibility.

At its heart it's a simple concept: *responsibility for propagating events of a specific type is assigned to a single service—a single writer*. So the inventory service owns how the stock inventory progresses over time, the orders service owns the progression of orders, and so on.

Conflating writes into a single service makes it easier to manage consistency efficiently. But this principle has worth that goes beyond correctness or concurrency properties. For example:

- It allows versioning (e.g., applying a version number) and consistency checks (e.g., checking a version number; see "Identity and Concurrency Control" on page 108) to be applied in a single place.
- It isolates the logic for evolving each business entity, in time, to a single service, making it easier to reason about and to change (for example, rolling out a schema change, as discussed in "Handling Schema Change and Breaking Backward Compatibility" on page 124 in Chapter 13).
- It dedicates ownership of a dataset to a single team, allowing that team to specialize. One antipattern observed in industry, when Enterprise Messaging or an Enterprise Service Bus was applied, was that centralized schemas and business logic could become a barrier to progress.[5] The single writer principle encourages service teams with clearly defined ownership of shared datasets, putting focus on data on the outside as well as allocating clear responsibility for it. This becomes important in domains that have complex business rules associated with different types of data. So, for example, in finance, where products require rich domain knowledge to model and evolve, isolating responsibility for data evolution to a single service and team is often considered an advantage.

When the single writer principle is applied in conjunction with Event Collaboration (discussed in Chapter 5), each writer evolves part of a single business workflow through a set of successive events. So in Figure 11-3, which shows a larger online retail workflow, several services collaborate around the order process as an order moves from inception, through payment processing and shipping, to

3 *https://dspace.mit.edu/handle/1721.1/6952* and *https://en.wikipedia.org/wiki/Actor_model*.

4 See *http://bit.ly/deds-end-of-an-era* and *http://bit.ly/deds-volt*.

5 See *https://thght.works/2IUS1zS* and *https://thght.works/2GdFYMq*.

completion. There are separate topics for order, payment, and shipment. The order, payment, and shipping services take control of all state changes made in their respective topics.

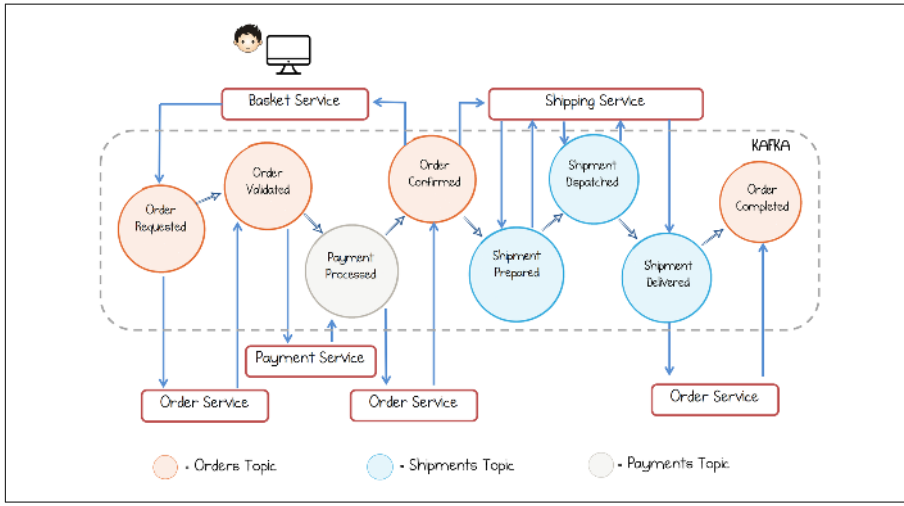


Figure 11-3. Here each circle represents an event; the color of the circle designates the topic it is in; a workflow evolves from Order Requested through to Order Completed; on the way, four services perform different state transitions in topics they are "single writer" to; the overall workflow spans them all

So, instead of sharing a global consistency model (e.g., via a database), we use the single writer principle to create local points of consistency that are connected via the event stream. There are a couple of variants on this pattern, which we will discuss in the next two sections.

As we'll see in "Scaling Concurrent Operations in Streaming Systems" on page 142 in Chapter 15, single writers can be scaled out linearly through partitioning, if we use Kafka's Streams API.

Command Topic

A common variant on this pattern uses two topics per entity, often named Command and Entity. This is logically identical to the base pattern, but the Command topic can be written to by any process and is used only for the initiating event. The Entity topic can be written to only by the owning service: the single writer. Splitting these two allows administrators to enforce the single writer principle strictly by configuring topic permissions. So, for example, we might break order events into two topics, shown in Table 11-1.

Table 11-1. A Command Topic is used to separate the initial command from subsequent events

Topic	`OrderCommandTopic`	`OrdersTopic`
Event types	`OrderRequest(ed)`	`OrderValidated, OrderCompleted`
Writer	Any service	Orders service

Single Writer Per Transition

A less stringent variant of the single writer principle involves services owning individual transitions rather than all transitions in a topic (see Table 11-2). So, for example, the payment service might not use a Payment topic at all. It might simply add extra payment information to the existing order message (so there would be a Payment section of the Order schema). The payment service then owns just that one transition and the orders service owns the others.

Table 11-2. The order service and payment services both write to the orders topic, but each service is responsible for a different state transition

Service	Orders service	Payment service
Topic	`OrdersTopic`	`OrdersTopic`
Writable transition	`OrderRequested->OrderValidated` `PaymentReceived->OrderConfirmed`	`OrderValidated->PaymentReceived`

Atomicity with Transactions

Kafka provides a transactions feature with two guarantees:

- Messages sent to different topics, within a transaction, will either all be written or none at all.
- Messages sent to a single topic, in a transaction, will never be subject to duplicates, even on failure.

But transactions provide a very interesting and powerful feature to Kafka Streams: they join writes to state stores and writes to output topics together, atomically. Kafka's transactions are covered in full in Chapter 12.

Identity and Concurrency Control

The notion of identity is hugely important in business systems, yet it is often overlooked. For example, to detect duplicates, messages need to be uniquely identified, as we discussed in Chapter 12. Identity is also important for handling the potential for updates to be made at the same time, by implementing optimistic concurrency control.

The basic premise of identity is that it should correlate with the real world: an order has an `OrderId`, a payment has a `PaymentId`, and so on. If that entity is logically mutable (for example, an order that has several states, `Created`, `Validated`, etc., or a customer whose email address might be updated), then it should have a version identifier also:

```
"Customer"{
          "CustomerId": "1234"
          "Source": "ConfluentWebPortal"
          "Version": "1"
          ...
}
```

The version identifier can then be used to handle concurrency. As an example (see Figure 11-4), say a user named Bob opens his customer details in one browser window (reading version 1), then opens the same page in a second browser window (also version 1). He then changes his address and submits in the second window, so the server increments the version to 2. If Bob goes back to the first window and changes his phone number, the update should be rejected due to a version comparison check on the server.

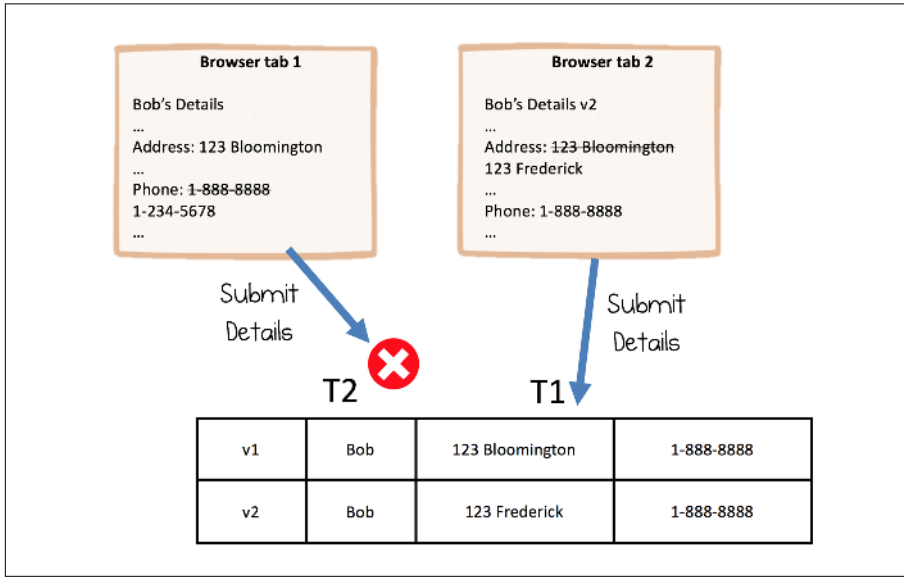


Figure 11-4. An example of optimistic concurrency control. The write fails at T2 because the data in the browser is now stale and the server performs a version number comparison before permitting the write.

The optimistic concurrency control technique can be implemented in synchronous or asynchronous systems equivalently.

Limitations

The single writer principle and other related patterns discussed in this chapter are exactly that: patterns, which are useful in common cases, but don't provide general solutions. There will always be exceptions, particularly in environments where the implementation is constrained, for example, by legacy systems.

Summary

In this chapter we looked at why global consistency can be problematic and why eventual consistency can be useful. We adapted eventual consistency with the single writer principle, keeping its lack of timeliness but avoiding collisions. Finally, we looked at implementing identity and concurrency control in event-driven systems.

CHAPTER 12

Transactions, but Not as We Know Them

Kafka ships with built-in transactions, in much the same way that most relational databases do. The implementation is quite different, as we will see, but the goal is similar: to ensure that our programs create predictable and repeatable results, even when things fail.

Transactions do three important things in a services context:

- They remove duplicates, which cause many streaming operations to get incorrect results (even something as simple as a count).
- They allow groups of messages to be sent, atomically, to different topics—for example, Order Confirmed and Decrease Stock Level, which would leave the system in an inconsistent state if only one of the two succeeded.
- Because Kafka Streams uses state stores, and state stores are backed by a Kafka topic, when we save data to the state store, then send a message to another service, we can wrap the whole thing in a transaction. This property turns out to be particularly useful.

In this chapter we delve into transactions, looking at the problems they solve, how we should make use of them, and how they actually work under the covers.

The Duplicates Problem

Any service-based architecture is itself a distributed system, a field renowned for being difficult, particularly when things go wrong. Thought experiments like the Two Generals' Problem and proofs like FLP highlight these inherent difficulties. But in practice the problem seems less complex. If you make a call to a service and it's not running for whatever reason, you retry, and eventually the call will complete.

One issue with this is that retries can result in duplicate processing, and this can cause very real problems. Taking a payment twice from someone's account will lead to an incorrect balance (Figure 12-1). Adding duplicate tweets to a user's feed will lead to a poor user experience. The list goes on.

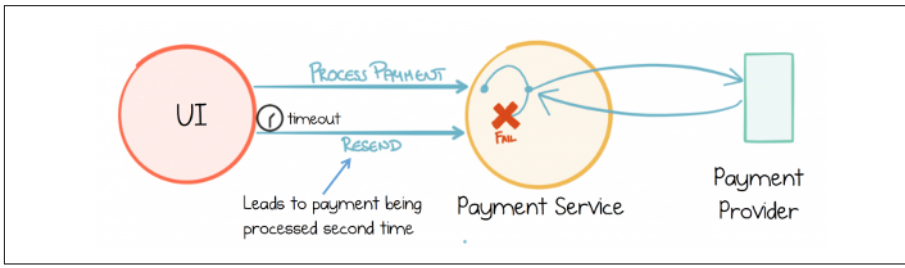


Figure 12-1. The UI makes a call to the payment service, which calls an external payment provider; the payment service fails before returning to the UI; as the UI did not get a response, it eventually times out and retries the call; the user's account could be debited twice

In reality we handle these duplicate issues automatically in the majority of systems we build, as many systems simply push data to a database, which will automatically deduplicate based on the primary key. Such processes are naturally idempotent. So if a customer updates their address and we are saving that data in a database, we don't care if we create duplicates, as the worst-case scenario is that the database table that holds customer addresses gets updated twice, which is no big deal. This applies to the payment example also, so long as each one has a unique ID. As long as deduplication happens at the end of each use case, then, it doesn't matter how many duplicate calls are made in between. This is an old idea, dating back to the early days of TCP (Transmission Control Protocol). It's called the end-to-end principle.

The rub is this—for this natural deduplication to work, every network call needs to:

- Have an appropriate key that defines its identity.
- Be deduplicated in a database that holds an extensive history of these keys. Or, duplicates have to be constantly considered in the business logic we write, which increases the cognitive overhead of this task.

Event-driven systems attempt to move away from this database-centric style of processing, instead executing business logic, communicating the results of that processing, and moving on.

The result of this is that most event-driven systems end up deduplicating on every message received, before it is processed, and every message sent out has a

carefully chosen ID so it can be deduplicated downstream. This is at best a bit of a hassle. At worst it's a breeding ground for errors.

But if you think about it, this is no more an application layer concern than ordering of messages, arranging redelivery, or any of the other benefits that come with TCP. We choose TCP over UDP (User Datagram Protocol) because we want to program at a higher level of abstraction, where delivery, ordering, and so on are handled for us. So we're left wondering why these issues of duplication have leaked up into the application layer. Isn't this something our infrastructure should solve for us?

Transactions in Kafka allow the creation of long chains of services, where the processing of each step in the chain is wrapped in exactly-once guarantees. This reduces duplicates, which means services are easier to program and, as we'll see later in this chapter, transactions let us tie streams and state together when we implement storage either through Kafka Streams state stores or using the Event Sourcing design pattern. All this happens automatically if you are using the Kafka Streams API.

The bad news is that this isn't some magic fairy dust that sprinkles exactly-onceness over your entire system. Your system will involve many different parts, some based on Kafka, some based on other technologies, the latter of which won't be covered by the guarantee.

But it *does* sprinkle exactly-onceness over the Kafka bits, the interactions between your services (Figure 12-2). This frees services from the need to deduplicate data coming in and pick appropriate keys for data going out. So we can happily chain services together, inside an event-driven workflow, without these additional concerns. This turns out to be quite empowering.

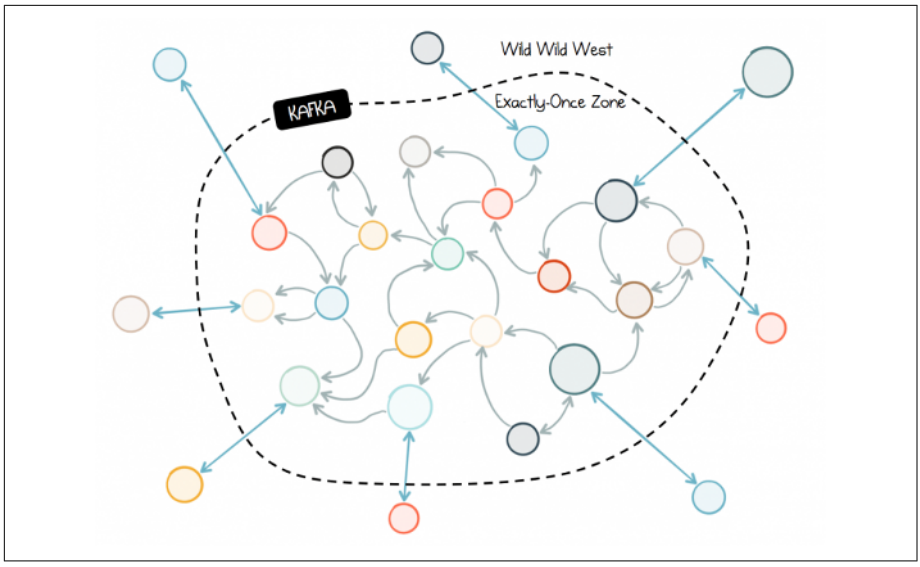


Figure 12-2. Kafka's transactions provide guarantees for communication performed through Kafka, but not beyond it

Using the Transactions API to Remove Duplicates

As a simple example, imagine we have an account validation service. It takes deposits in, validates them, and then sends a new message back to Kafka marking the deposit as validated.

Kafka records the progress that each consumer makes by storing an offset in a special topic, called `consumer_offsets`. So to validate each deposit exactly once, we need to perform the final two actions—(a) send the "Deposit Validated" message back to Kafka, and (b) commit the appropriate offset to the `consumer_offsets` topic—as a single atomic unit (Figure 12-3). The code for this would look something like the following:

```
//Read and validate deposits
validatedDeposits = validate(consumer.poll(0))

//Send validated deposits & commit offsets atomically
producer.beginTransaction()
producer.send(validatedDeposits)
producer.sendOffsetsToTransaction(offsets(consumer))
producer.endTransaction()
```

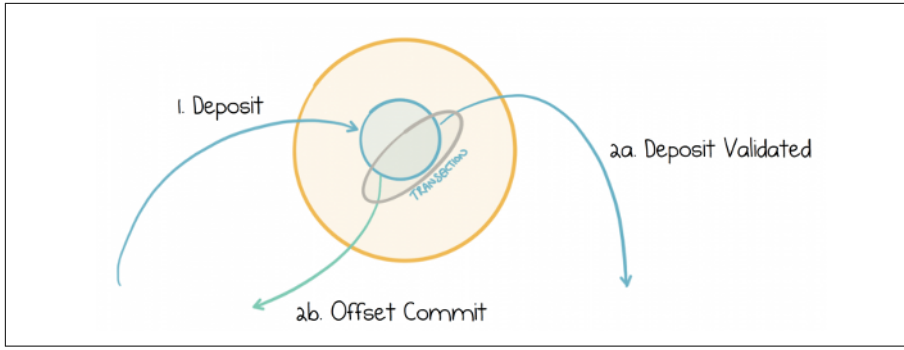


Figure 12-3. A single message operation is in fact two operations: a send and an acknowledge, which must be performed atomically to avoid duplication

If you are using the Kafka Streams API, no extra code is required. You simply enable the feature.

Exactly Once Is Both Idempotence and Atomic Commit

As Kafka is a broker, there are actually two opportunities for duplication. Sending a message to Kafka might fail before an acknowledgment is sent back to the client, with a subsequent retry potentially resulting in a duplicate message. On the other side, the process reading from Kafka might fail before offsets are committed, meaning that the same message might be read a second time when the process restarts (Figure 12-4).

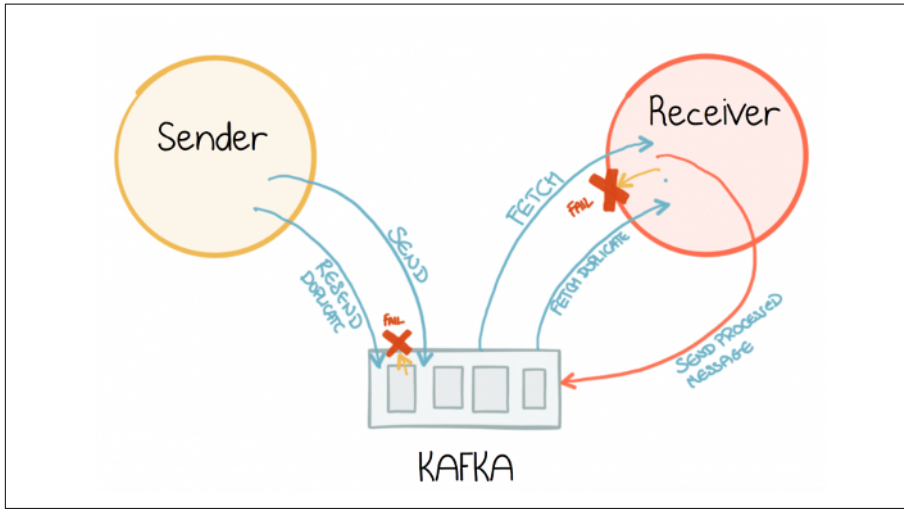


Figure 12-4. Message brokers provide two opportunities for failure—one when sending to the broker, and one when reading from it

So idempotence is required in the broker to ensure duplicates cannot be created in the log. Idempotence, in this context, is just deduplication. Each producer is given an identifier, and each message is given a sequence number. The combination of the two uniquely defines each batch of messages sent. The broker uses this unique sequence number to work out if a message is already in the log and discards it if it is. This is a significantly more efficient approach than storing every key you've ever seen in a database.

On the read side, we might simply deduplicate (e.g., in a database). But Kafka's transactions actually provide a broader guarantee, more akin to transactions in a database, tying all messages sent together in a single *atomic commit*. So idempotence is built into the broker, and then an atomic commit is layered on top.

How Kafka's Transactions Work Under the Covers

Looking at the code example in the previous section, you might notice that Kafka's transactions implementation looks a lot like transactions in a database. You start a transaction, write messages to Kafka, then commit or abort. But the whole model is actually pretty different, because of course it's designed for streaming.

One key difference is the use of marker messages that make their way through the various streams. Marker messages are an idea first introduced by Chandy and Lamport almost 30 years ago in a method called the Snapshot Marker Model. Kafka's transactions are an adaptation of this idea, albeit with a subtly different goal.

While this approach to transactional messaging is complex to implement, conceptually it's quite easy to understand (Figure 12-5). Take our previous example, where two messages were written to two different topics atomically. One message goes to the `Deposits` topic, the other to the `committed_offsets` topic.

Begin markers are sent down both.[1] We then send our messages. Finally, when we're done, we flush each topic with a Commit (or Abort) marker, which concludes the transaction.

Now the aim of a transaction is to ensure only "committed" data is seen by downstream programs. To make this work, when a consumer sees a Begin marker it starts buffering internally. Messages are held up until the Commit marker arrives. Then, and only then, are the messages presented to the consuming program. This buffering ensures that consumers only ever read committed data.

1 In practice a clever optimization is used to move buffering from the consumer to the broker, reducing memory pressure. Begin markers are also optimized out.

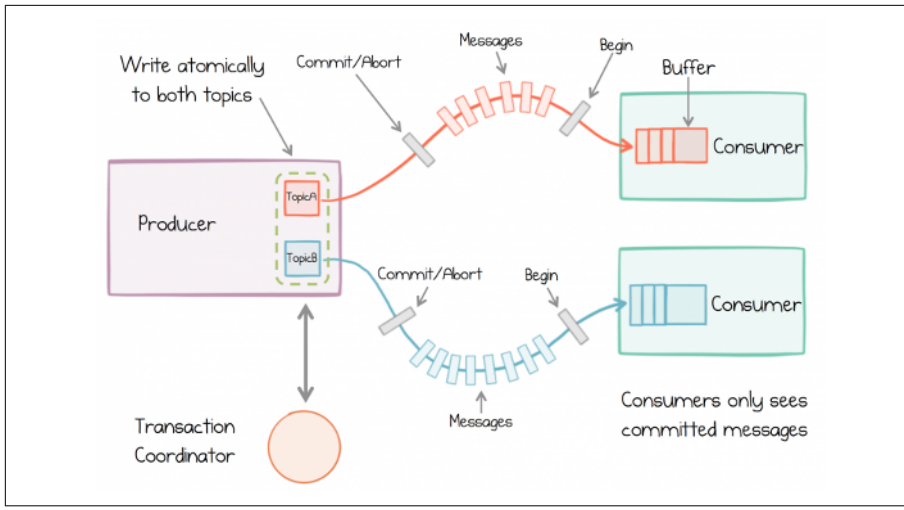


Figure 12-5. Conceptual model of transactions in Kafka

To ensure each transaction is atomic, sending the Commit markers involves the use of a transaction coordinator. There will be many of these spread throughout the cluster, so there is no single point of failure, but each transaction uses just one.

The transaction coordinator is the ultimate arbiter that marks a transaction committed atomically, and maintains a transaction log to back this up (this step implements two-phase commit).

For those that worry about performance, there is of course an overhead that comes with this feature, and if you were required to commit after every message, the performance degradation would be noticeable. But in practice there is no need for that, as the overhead is dispersed among whole batches of messages, allowing us to balance transactional overhead with worst-case latency. For example, batches that commit every 100 ms, with a 1 KB message size, have a 3% overhead when compared to in-order, at-least-once delivery. You can test this out yourself with the performance test scripts that ship with Kafka.

In reality, there are many subtle details to this implementation, particularly around recovering from failure, fencing zombie processes, and correctly allocating IDs, but what we have covered here is enough to provide a high-level understanding of how this feature works. For a comprehensive explanation of how transactions work, see the post "Transactions in Apache Kafka" by Apurva Mehta and Jason Gustafson.

Store State and Send Events Atomically

As we saw in Chapter 7, Kafka can be used to store data in the log, with the most common means being a state store (a disk-resident hash table, held inside the API, and backed by a Kafka topic) in Kafka Streams. As a state store gets its durability from a Kafka topic, we can use transactions to tie writes to the state store and writes to other output topics together. This turns out to be an extremely powerful pattern because it mimics the tying of messaging and databases together atomically, something that traditionally required painfully slow protocols like XA.

NOTE

The database used by Kafka Streams is a state store. Because state stores are backed by Kafka topics, transactions let us tie messages we send and state we save in state stores together, atomically.

Imagine we extend the previous example so our validation service keeps track of the balance as money is deposited. So if the balance is currently $50, and we deposit $5 more, then the balance should go to $55. We record that $5 was deposited, but we also store this current balance, $55, by writing it to a state store (or directly to a compacted topic). See Figure 12-6.

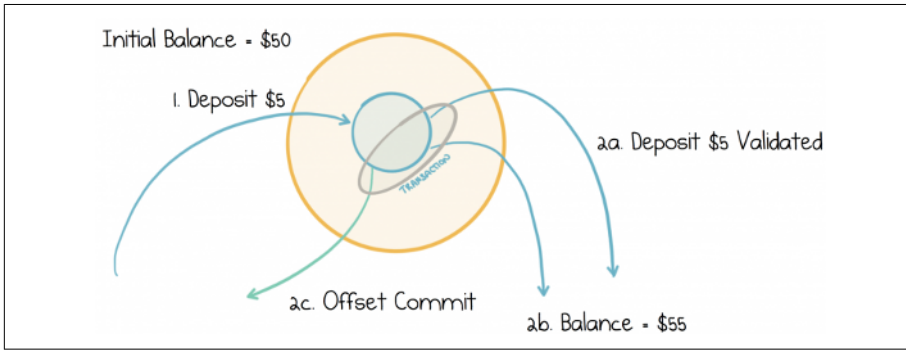


Figure 12-6. Three messages are sent atomically: a deposit, a balance update, and the acknowledgment

If transactions are enabled in Kafka Streams, all these operations will be wrapped in a transaction automatically, ensuring the balance will always be atomically in sync with deposits. You can achieve the same process with the product and consumer by wrapping the calls manually in your code, and the current account balance can be reread on startup.

What's powerful about this example is that it blends concepts of both messaging and state management. We listen to events, act, and create new events, but we

also manage state, the current balance, in Kafka—all wrapped in the same transaction.

Do We Need Transactions? Can We Do All This with Idempotence?

People have been building both event- and request-driven systems for decades, simply by making their processes idempotent with identifiers and databases. But implementing idempotence comes with some challenges. While defining the ID of an order is relatively obvious, not all streams of events have such a clear concept of identity. If we had a stream of events representing the average account balance per region per hour, we could come up with a suitable key, but you can imagine it would be a lot more brittle and error-prone.

Also, transactions encapsulate the concept of deduplication entirely inside your service. You don't muddy the waters seen by other services downstream with any duplicates you might create. This makes the contract of each service clean and encapsulated. Idempotence, on the other hand, relies on every service that sits downstream *correctly* implementing deduplication, which clearly makes their contract more complex and error-prone.

What Can't Transactions Do?

There are a few limitations or potential misunderstandings of transactions that are worth noting. First, they work only in situations where both the input and the output go through Kafka. If you are calling an *external* service (e.g., via HTTP), updating a database, writing to stdout, or anything other than writing *to* and *from* the Kafka broker, transactional guarantees won't apply and calls can be duplicated. So, much like using a transactional database, transactions work only when you are using Kafka.

Also akin to accessing a database, transactions commit when messages are sent, so once they are committed there is no way to roll them back, even if a subsequent transaction downstream fails. So if the UI sends a transactional message to the orders service and the orders service fails while sending messages of its own, any messages the orders service sent would be rolled back, but there is no way to roll back the transaction in the UI. If you need multiservice transactions, consider implementing sagas.

Transactions commit atomically in the broker (just like a transaction would commit in a database), but there are no guarantees regarding when an arbitrary consumer will read those messages. This may seem obvious, but it is sometimes a point of confusion. Say we send a message to the Orders topic and a message to the Payments topic, inside a transaction there is no way to know when a con-

sumer will read one or the other, or that they might read them together. But again note that this is identical to the contract offered by a transactional database.

Finally, in the examples here we use the producer and consumer APIs to demonstrate how transactions work. But the Kafka's Streams API actually requires no extra coding whatsoever. All you do is set a configuration and exactly-once processing is enabled automatically.

But while there is full support for individual producers and consumers, transactions are not currently supported for consumer groups (although this will change). If you have this requirement, use the Kafka Streams API, where consumer groups are supported in full.

Making Use of Transactions in Your Services

In Chapter 5 we described a design pattern known as Event Collaboration. In this pattern messages move from service to service, creating a workflow. It's initiated with an Order Requested event and it ends with Order Complete. In between, several different services get involved, moving the workflow forward.

Transactions are important in complex workflows like this because the end-to-end principle is hard to apply. Without them, deduplication would need to happen in every service. Moreover, building a reliable streaming application without transactions turns out to be pretty tough. There are a couple of reasons for this: (a) Streams applications make use of many intermediary topics, and deduplicating them after each step is a burden (and would be near impossible in KSQL), (b) the DSL provides a range of one-to-many operations (e.g., `flatMap()`), which are hard to manage idempotently without the transactions API. Kafka's transactions feature resolves these issues, along with atomically tying stream processing with the storing of intermediary state in state stores.

Summary

Transactions affect the way we build services in a number of specific ways:

- They take idempotence right off the table for services interconnected with Kafka. So when we build services that follow the pattern "read, process, (save), send," we don't need to worry about deduplicating inputs or constructing keys for outputs.
- We no longer need to worry about ensuring there are appropriate unique keys on the messages we send. This typically applies less to topics containing business events, which often have good keys already. But it's useful when we're managing derivative/intermediary data—for example, when we're remapping events, creating aggregate events, or using the Streams API.

- Where Kafka is used for persistence, we can wrap both messages we send to other services and state we need internally in a single transaction that will commit or fail. This makes it easier to build simple stateful apps and services.

So, to put it simply, when you are building event-based systems, Kafka's transactions free you from the worries of failure and retries in a distributed world—worries that really should be a concern of the infrastructure, not of your code. This raises the level of abstraction, making it easier to get accurate, repeatable results from large estates of fine-grained services.

Having said all that, we should also be careful. Transactions remove just one of the issues that come with distributed systems, but there are many more. Coarse-grained services still have their place. But in a world where we want to be fast and nimble, streaming platforms raise the bar, allowing us to build finer-grained services that behave as predictably in complex chains as they would standing alone.

CHAPTER 13

Evolving Schemas and Data over Time

Schemas are the APIs used by event-driven services, so a publisher and subscriber need to agree on exactly how a message is formatted. This creates a logical coupling between sender and receiver based on the schema they both share. In the same way that request-driven services make use of service discovery technology to discover APIs, event-driven technologies need some mechanism to discover what topics are available, and what data (i.e., schema) they provide.

There are a fair few options available for schema management: Protobuf and JSON Schema are both popular, but most projects in the Kafka space use Avro. For central schema management and verification, Confluent has an open source Schema Registry that provides a central repository for Avro schemas.

Using Schemas to Manage the Evolution of Data in Time

Schemas provide a contract that defines what a message should look like. This is pretty intuitive. Importantly, though, most schema technologies provide a mechanism for validating whether a message in a new schema is backward-compatible with previous versions (or vice versa). This property is essential. (Don't use Java serialization or any non-evolvable format across services that change independently.)

Say you added a "return code" field to the schema for an order; this would be a backward-compatible change. Programs running with the old schema would still be able to read messages, but they wouldn't see the "return code" field (termed *forward compatibility*). Programs with the new schema would be able to read the whole message, with the "return code" field included (termed *backward compatibility*).

Unfortunately, you can't move or remove fields from a schema in a compatible way, although it's typically possible to synthesize a move with a clone. The data will be duplicated in two places until such time as a breaking change can be released.

This ability to evolve a schema with additive changes that don't break old programs is how most shared messaging models are managed over time.

The Confluent Schema Registry can be used to police this approach. The Schema Registry provides a mapping between topics in Kafka and the schema they use (Figure 13-1). It also enforces compatibility rules before messages are added. So the Schema Registry will check every message sent to Kafka for Avro compatibility, ensuring that incompatible messages will fail on publication.

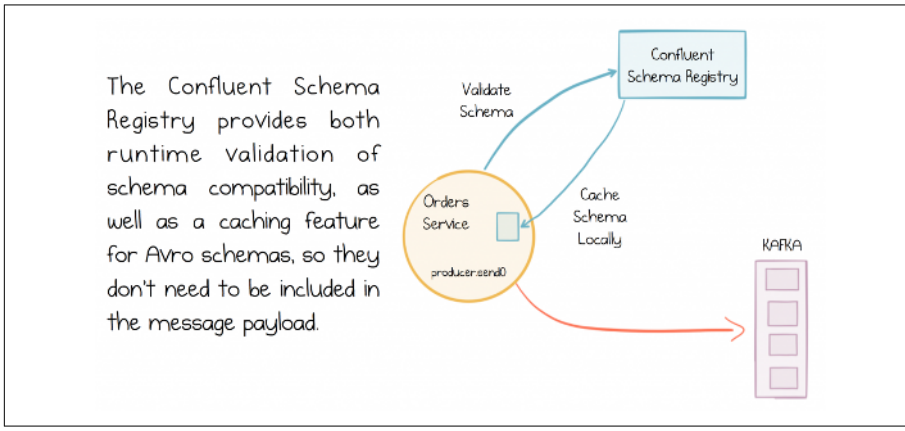


Figure 13-1. Calling out to the Schema Registry to validate schema compatibility when reading and writing orders in the orders service

Handling Schema Change and Breaking Backward Compatibility

The pain of long schema migrations is one of the telltale criticisms of the relational era. But the reality is that evolving schemas are a fundamental attribute of the way data ages. The main difference between then and now is that late-bound/schema-on-read approaches allow many incompatible data schemas to exist in the same table, topic, or the like at the same time. This pushes the problem of translating the format—from old to new—into the application layer, hence the name "schema on read."

Schema on read turns out to be useful in a couple of ways. In many cases recent data is more valuable than older data, so programs can move forward without migrating older data they don't really care about. This is a useful, pragmatic solution used broadly in practice, particularly with messaging. But schema on read

can also be simple to implement if the parsing code for the previous schemas already exists in the codebase (which is often the case in practice).

However, whichever approach you take, unless you do a single holistic big-bang release, you will end up handling the schema-evolution problem, be it by physically migrating datasets forward or by having different application-layer routines. Kafka is no different.

As we discussed in the previous section, most of the time backward compatibility between schemas can be maintained through additive changes (i.e., new fields, but not moves or deletes). But periodically schemas will need upgrading in a non-backward-compatible way. The most common approach for this is Dual Schema Upgrade Window, where we create two topics, orders-v1 and orders-v2, for messages with the old and new schemas, respectively. Assuming orders are mastered by the orders service, this gives you a few options:

- The orders service can dual-publish in both schemas at the same time, to two topics, using Kafka's transactions API to make the publication atomic. (This approach doesn't solve back-population so isn't appropriate for topics used for long-term storage.)
- The orders service can be repointed to write to orders-v2. A Kafka Streams job is added to down-convert from the orders-v2 topic to the orders-v1 for backward compatibility. (This also doesn't solve back-population.) See Figure 13-2.
- The orders service continues to write to orders-v1. A Kafka Streams job is added that up-converts from orders-v1 topic to orders-v2 topic until all clients have upgraded, at which point the orders service is repointed to orders-v2. (This approach handles back-population.)
- The orders service can migrate its dataset internally, in its own database, then republish the whole view into the log in the orders-v2 topic. It then continues to write to both orders-v1 and orders-v2 using the appropriate formats. (This approach handles back-population.)

All four approaches achieve the same goal: to give services a window in which they can upgrade. The last two options make it easier to port historic messages from the v1 to the v2 topics, as the Kafka Streams job will do this automatically if it is started from offset 0. This makes it better suited to long-retention topics such as those used in Event Sourcing use cases.

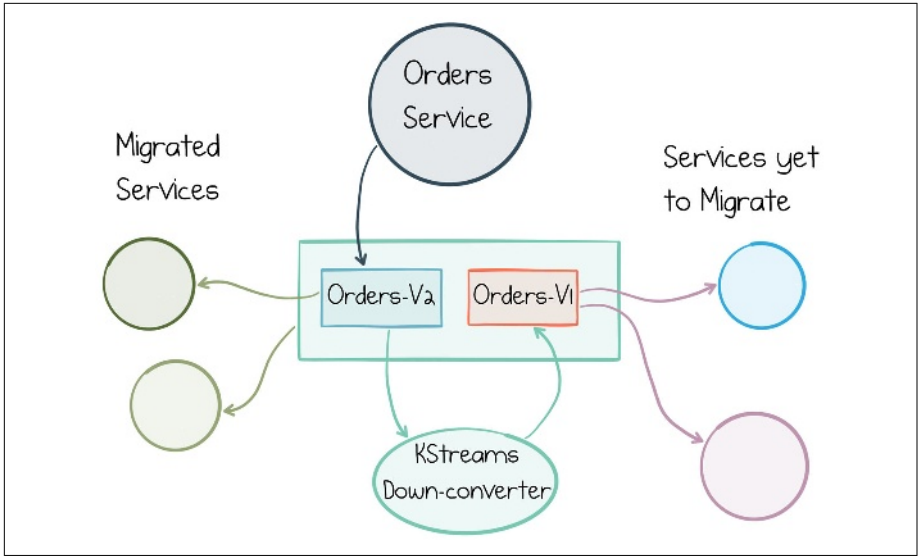


Figure 13-2. Dual Schema Upgrade Window: the same data coexists in two topics, with different schemas, so there is a window during which services can upgrade

Services continue in this dual-topic mode until fully migrated to the v2 topic, at which point the v1 topic can be archived or deleted as appropriate.

As an aside, we discussed the single writer principle in Chapter 11. One of the reasons for applying this approach is that it makes schema upgrades simpler. If we had three different services writing orders, it would be much harder to schedule a non-backward-compatible upgrade without a conjoined release.

Collaborating over Schema Change

In the previous section we discussed how to roll out a non-backward-compatible schema change. However, before such a process ensues, or even before we make a minor change to a schema, there is usually some form of team-to-team collaboration that takes place to work out whether the change is appropriate. This can take many forms. Sending an email to the affected teams, telling them what the new schema is and when it's going live, is pretty common, as is having a central team that manages the process. Neither of these approaches works particularly well in practice, though. The email method lacks structure and accountability. The central team approach stifles progress, because you have to wait for the central team to make the change and then arrange some form of sign-off.

The best approach I've seen for this is to use GitHub. This works well because (a) schemas are code and should be version-controlled for all the same reasons code is, and (b) GitHub lets implementers propose a change and raise a pull request (PR), which they can code against while they build and test their system. Other

interested parties can review, comment, and approve. Once consensus is reached, the PR can be merged and the new schema can be rolled out. It is this process for reliably reaching (and auditing) consensus on a change, without impeding the progress of the implementer unduly, that makes this approach the most useful option.

Handling Unreadable Messages

Schemas aren't always enough to ensure downstream applications will work. There is nothing to prevent a semantic error—for example, an unexpected character, invalid country code, negative quantity, or even invalid bytes (say due to corruption)—from causing a process to stall. Such errors will typically hold up processing until the issue is fixed, which can be unacceptable in some environments.

Traditional messaging systems often include a related concept called a *dead letter queue*,[1] which is used to hold messages that can't be sent, for example, because they cannot be routed to a destination queue. The concept doesn't apply in the same way to Kafka, but it is possible for consumers to not be able to read a message, either for semantic reasons or due to the message payload being invalid (e.g., the CRC check failing, on read, as the message has become corrupted).

Some implementers choose to create a type of dead letter queue of their own in a separate topic. If a consumer cannot read a message for whatever reason, it is placed on this error queue so processing can continue. Later the error queue can be reprocessed.

Deleting Data

When you keep datasets in the log for longer periods of time, or even indefinitely, there are times you need to delete messages, correct errors or corrupted data, or redact sensitive sections. A good example of this is recent regulations like General Data Protection Regulation (GDPR), which, among other things, gives users the right to be forgotten.

The simplest way to remove messages from Kafka is to simply let them expire. By default, Kafka will keep data for two weeks, and you can tune this to an arbitrarily large (or small) period of time. There is also an Admin API that lets you delete messages explicitly if they are older than some specified time or offset. When using Kafka for Event Sourcing or as a source of truth, you typically don't need delete. Instead, removal of a record is performed with a null value (or delete

1 See *https://www.rabbitmq.com/dlx.html* and *https://ibm.co/2ui3rKO*.

marker as appropriate). This ensures the fully versioned history is held intact, and most Connect sinks are built with delete markers in mind.

But for regulatory requirements like GDPR, adding a delete marker isn't enough, as all data needs to be physically removed from the system. There are a variety of approaches to this problem. Some people favor a security-based approach such as crypto shredding, but for most people, compacted topics are the tool of choice, as they allow messages to be explicitly deleted or replaced via their key.

But data isn't removed from compacted topics in the same way as in a relational database. Instead, Kafka uses a mechanism closer to those used by Cassandra and HBase, where records are marked for removal and then later deleted when the compaction process runs. Deleting a message from a compacted topic is as simple as writing a new message to the topic with the key you want to delete and a null value. When compaction runs, the message will be deleted forever.

If the key of the topic is something other than the `CustomerId`, then you need some process to map the two. For example, if you have a topic of Orders, then you need a mapping of customer to `OrderId` held somewhere. Then, to "forget" a customer, simply look up their orders and either explicitly delete them from Kafka, or alternatively redact any customer information they contain. You might roll this into a process of your own, or you might do it using Kafka Streams if you are so inclined.

There is a less common case, which is worth mentioning, where the key (which Kafka uses for ordering) is completely different from the key you want to be able to delete by. Let's say that you need to key your orders by `ProductId`. This choice of key won't let you delete orders for individual customers, so the simple method just described wouldn't work. You can still achieve this by using a key that is a composite of the two: make the key `[ProductId][CustomerId]`, then use a custom partitioner in the producer that extracts the `ProductId` and partitions only on that value. Then you can delete messages using the mechanism discussed earlier using the `[ProductId][CustomerId]` pair as the key.

Triggering Downstream Deletes

Quite often you'll be in a pipeline where Kafka is moving data from one database to another using Kafka connectors. In this case, you need to delete the record in the originating database and have that propagate through Kafka to any Connect sinks you have downstream. If you're using CDC this will just work: the delete will be picked up by the source connector, propagated through Kafka, and deleted in the sinks. If you're not using a CDC-enabled connector, you'll need some custom mechanism for managing deletes.

Segregating Public and Private Topics

When using Kafka for Event Sourcing or stream processing, in the same cluster through which different services communicate, we typically want to segregate private, internal topics from shared, business topics.

Some teams prefer to do this by convention, but you can apply a stricter segregation using the authorization interface. Essentially you assign read/write permissions, for your internal topics, only to the services that own them. This can be implemented through simple runtime validation, or alternatively fully secured via TLS or SASL.

Summary

In this chapter we looked at a collection of somewhat disparate issues that affect event-driven systems. We considered the problem of schema change: something that is inevitable in the real-world. Often this can be managed simply by evolving the schema with a format like Avro or Protobuf that supports backward compatibility. At other times evolution will not be possible and the system will have to undergo a non-backward-compatible change. The dual-schema upgrade window is one way to handle this.

Then we briefly looked at handling unreadable messages as well as how data can be deleted. For many users deleting data won't be an issue–it will simply age out of the log–but for those that keep data for longer periods this typically becomes important.

PART V

Implementing Streaming Services with Kafka

A reactive system does not compute or perform a function, it maintains a certain ongoing relationship with its environment.

—David Harel and Amir Pnueli, "On the Development of Reactive Systems," 1985

CHAPTER 14

Kafka Streams and KSQL

When it comes to building event-driven services, the Kafka Streams API provides the most complete toolset for handling a distributed, asynchronous world. Kafka Streams is designed to perform streaming computations. We discussed a simple example of such a use case, where we processed app open/close events emitted from mobile phones in Chapter 2. We also touched on its stateful elements in Chapter 6. This led us to three types of services we can build: event-driven, streaming, and stateful streaming.

In this chapter we look more closely at this unique tool for stateful stream processing, along with its powerful declarative interface: KSQL.

A Simple Email Service Built with Kafka Streams and KSQL

Kafka Streams is the core API for stream processing on the JVM (Java, Scala, Clojure, etc.). It is based on a DSL (domain-specific language) that provides a declaratively styled interface where streams can be joined, filtered, grouped, or aggregated via the DSL itself. It also provides functionally styled mechanisms (`map`, `flatMap`, `transform`, `peek`, etc.) for adding bespoke processing of messages one at a time. Importantly, you can blend these two approaches together in the services you build, with the declarative interface providing a high-level abstraction for SQL-like operations and the more functional methods adding the freedom to branch out into any arbitrary code you may wish to write.

But what if you're not running on the JVM? In this case you'd use KSQL. KSQL provides a simple, interactive SQL-like wrapper for the Kafka Streams API. It can be run standalone, for example, via the Sidecar pattern, and called remotely. As KSQL utilizes the Kafka Streams API under the hood, we can use it to do the same kind of declarative slicing and dicing. We can also apply custom processing

either by implementing a user-defined function (UDF) directly or, more commonly, by pushing the output to a Kafka topic and using a native Kafka client, in whatever language our service is built in, to process the manipulated streams one message at a time. Whichever approach we take, these tools let us model business operations in an asynchronous, nonblocking, and coordination-free manner.

Let's consider something more concrete. Imagine we have a service that sends emails to platinum-level clients (Figure 14-1). We can break this problem into two parts. First, we prepare by joining a stream of orders to a table of customers and filtering for the "platinum" clients. Second, we need code to construct and send the email itself. We would do the former in the DSL and the latter with a per-message function:

```
//Join customers and orders
orders.join(customers, Tuple::new)
 //Consider confirmed orders for platinum customers
 .filter((k, tuple) -> tuple.customer.level().equals(PLATINUM)
 && tuple.order.state().equals(CONFIRMED))
 //Send email for each customer/order pair
 .peek((k, tuple) -> emailer.sendMail(tuple));
```

NOTE

The code for this is available on GitHub.

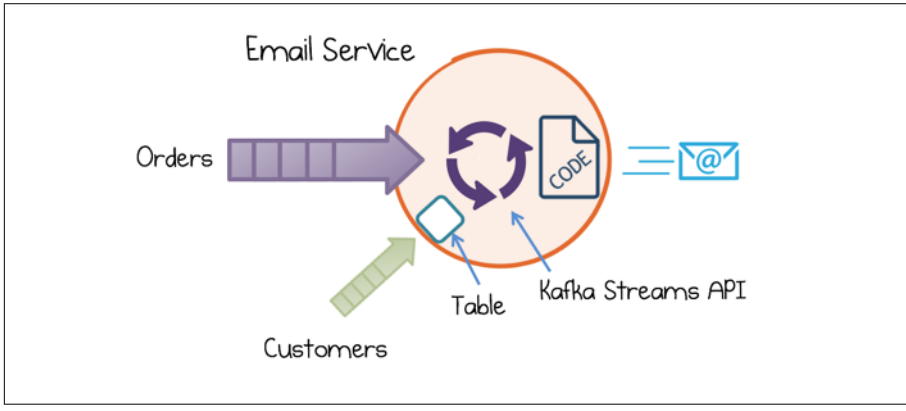


Figure 14-1. An example email service that joins orders and customers, then sends an email

We can perform the same operation using KSQL (Figure 14-2). The pattern is the same; the event stream is dissected with a declarative statement, then processed one record at a time:

```
//Create a stream of confirmed orders for platinum customers
ksql> CREATE STREAM platinum_emails AS SELECT * FROM orders
  WHERE client_level == 'PLATINUM' AND state == 'CONFIRMED';
```

Then we implement the emailer as a simple consumer using Kafka's Node.js API (though a wide number of languages are supported) with KSQL running as a sidecar.

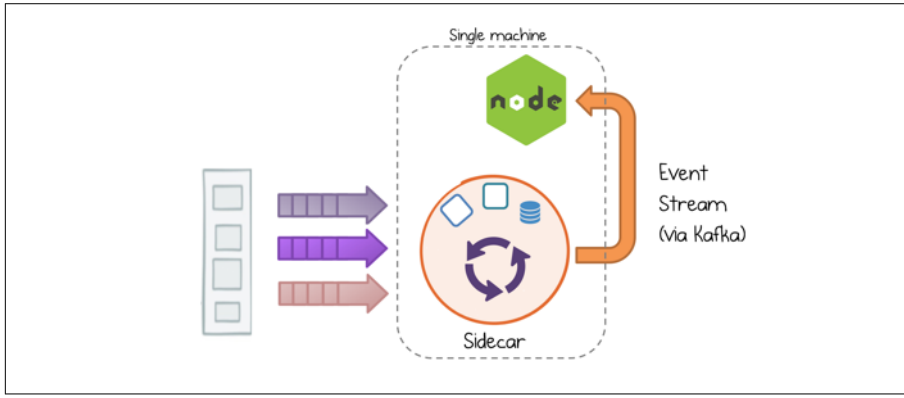


Figure 14-2. Executing a streaming operation as a sidecar, with the resulting stream being processed by a Node.js client

Windows, Joins, Tables, and State Stores

Chapter 6 introduced the notion of holding whole tables inside the Kafka Streams API, making services stateful. Here we look a little more closely at how both streams and tables are implemented, along with some of the other core features.

Let's revisit the email service example once again, where an email is sent to confirm payment of a new order, as pictured in Figure 14-3. We apply a stream-stream join, which waits for corresponding Order and Payment events to both be present in the email service before triggering the email code. The join behaves much like a logical AND.

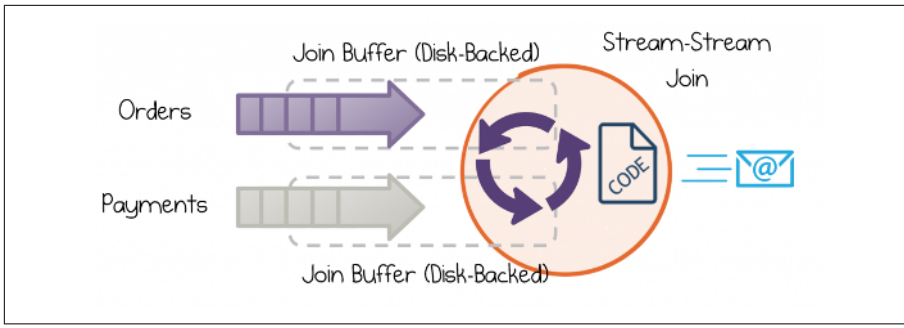


Figure 14-3. A stream-stream join between orders and payments

Incoming event streams are buffered for a defined period of time (denoted retention). But to avoid doing all of this buffering in memory, state stores—disk-backed hash tables—overflow the buffered streams to disk. Thus, regardless of which event turns up later, the corresponding event can be quickly retrieved from the buffer so the join operation can complete.

Kafka Streams also manages whole tables. Tables are a local manifestation of a complete topic—usually compacted—held in a state store by key. (You might also think of them as a stream with infinite retention.) In a services context, such tables are often used for enrichments. So to look up the customer's email, we might use a table loaded from the Customers topic in Kafka.

The nice thing about using a table is that it behaves a lot like tables in a database. So when we join a stream of orders to a table of customers, there is no need to worry about retention periods, windows, or other such complexities. Figure 14-4 shows a three-way join between orders, payments, and customers, where customers are represented as a table.

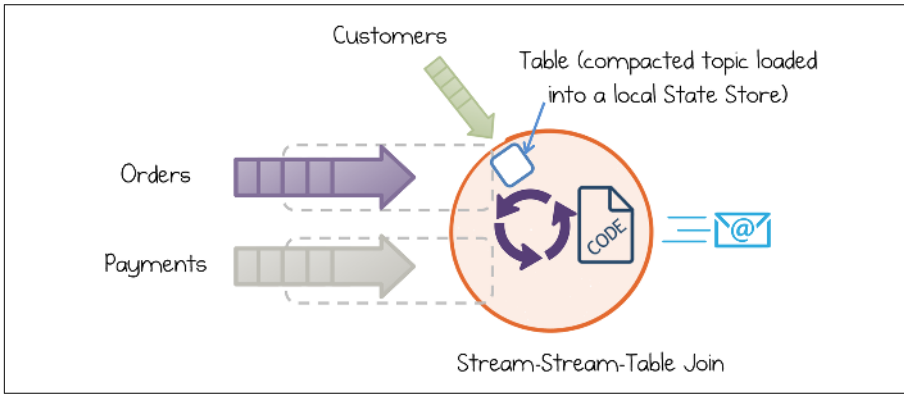


Figure 14-4. A three-way join between two streams and a table

There are actually two types of table in Kafka Streams: KTables and Global KTables. With just one instance of a service running, these behave equivalently. However, if we scaled our service out—so it had four instances running in parallel—we'd see slightly different behaviors. This is because Global KTables are broadcast: each service instance gets a complete copy of the entire table. Regular KTables are partitioned: the dataset is spread over all service instances.

Whether a table is broadcast or partitioned affects the way it can perform joins. With a Global KTable, because the whole table exists on every node, we can join to any attribute we wish, much like a foreign key join in a database. This is not true in a KTable. Because it is partitioned, it can be joined only by its primary key, just like you have to use the primary key when you join two streams. So to join a KTable or stream by an attribute that is not its primary key, we must perform a repartition. This is discussed in "Rekey to Join" on page 145 in Chapter 15.

So, in short, Global KTables work well as lookup tables or star joins but take up more space on disk because they are broadcast. KTables let you scale your services out when the dataset is larger, but may require that data be rekeyed.[1]

The final use of the state store is to save information, just like we might write data to a regular database (Figure 14-5). Anything we save can be read back again later, say after a restart. So we might expose an Admin interface to our email service that provides statistics on emails that have been sent. We could store, these stats in a state store and they'll be saved locally as well as being backed up to Kafka, using what's called a *changelog topic*, inheriting all of Kafka's durability guarantees.

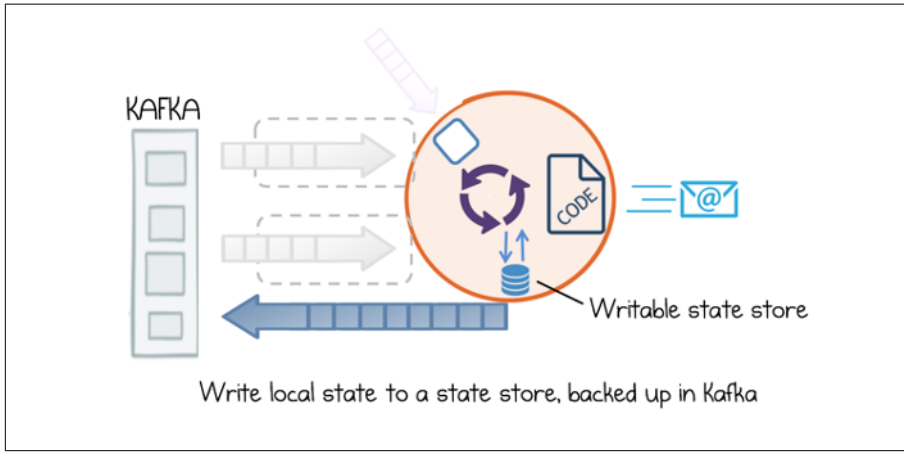


Figure 14-5. Using a state store to keep application-specific state within the Kafka Streams API as well as backed up in Kafka

1 The difference between these two is actually slightly subtler.

Summary

This chapter provided a brief introduction to streams, tables, and state stores: three of the most important elements of a streaming application. Streams are infinite and we process them a record at a time. Tables represent a whole dataset, materialized locally, which we can join to much like a database table. State stores behave like dedicated databases which we can read and write to directly with any information we might wish to store. These features are of course just the tip of the iceberg, and both Kafka Streams and KSQL provide a far broader set of features, some of which we explore in Chapter 15, but they all build on these base concepts.

CHAPTER 15

Building Streaming Services

An Order Validation Ecosystem

Having developed a basic understanding of Kafka Streams, now let's look at the techniques needed to build a small streaming services application. We will base this chapter around a simple order processing workflow that validates and processes orders in response to HTTP requests, mapping the synchronous world of a standard REST interface to the asynchronous world of events, and back again.

Download the code for this example from GitHub.

Starting from the lefthand side of the Figure 15-1, the REST interface provides methods to POST and GET orders. Posting an order creates an Order Created event in Kafka. Three validation engines (Fraud, Inventory, Order Details) subscribe to these events and execute in parallel, emitting a `PASS` or `FAIL` based on whether each validation succeeds. The result of these validations is pushed through a separate topic, Order Validations, so that we retain the single writer relationship between the orders service and the Orders topic.[1] The results of the various validation checks are aggregated back in the orders service, which then moves the order to a Validated or Failed state, based on the combined result. Validated orders accumulate in the Orders view, where they can be queried historically. This is an implementation of the CQRS design pattern (see "Command

1 In this case we choose to use a separate topic, Order Validations, but we might also choose to update the Orders topic directly using the single-writer-per-transition approach discussed in Chapter 11.

Query Responsibility Segregation" on page 61 in Chapter 7). The email service sends confirmation emails.

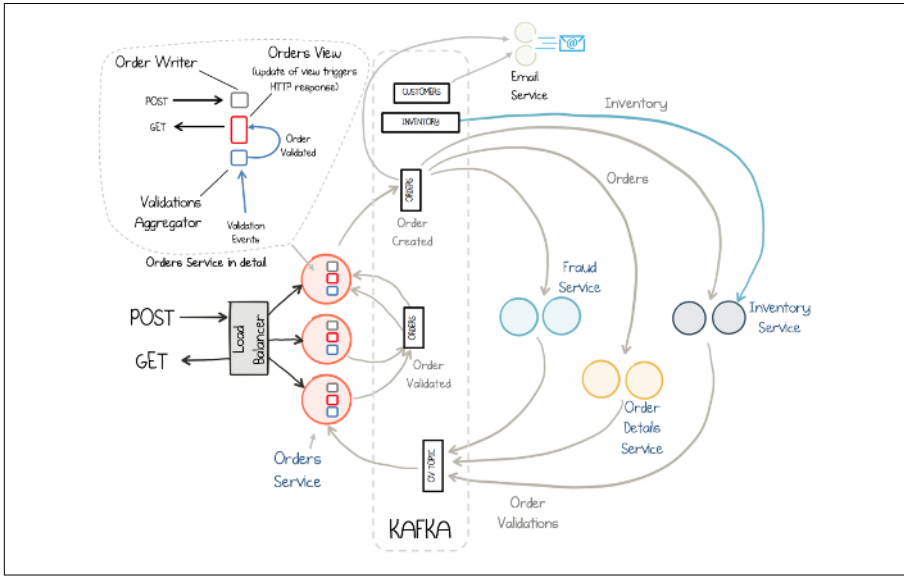


Figure 15-1. An order processing system implemented as streaming services

The inventory service both validates orders and reserves inventory for the purchase—an interesting problem, as it involves tying reads and writes together atomically. We look at this in detail later in this chapter.

Join-Filter-Process

Most streaming systems implement the same broad pattern where a set of streams is prepared, and then work is performed one event at a time. This involves three steps:

1. **Join**. The DSL is used to join a set of streams and tables emitted by other services.
2. **Filter**. Anything that isn't required is filtered. Aggregations are often used here too.
3. **Process**. The join result is passed to a function where business logic executes. The output of this business logic is pushed into another stream.

This pattern is seen in most services but is probably best demonstrated by the email service, which joins orders, payments, and customers, forwarding the result to a function that sends an email. The pattern can be implemented in either Kafka Streams or KSQL equivalently.

Event-Sourced Views in Kafka Streams

To allow users to perform a HTTP GET, and potentially retrieve historical orders, the orders service creates a queryable event-sourced view. (See "The Event-Sourced View" on page 71 in Chapter 7.) This works by pushing orders into a set of state stores partitioned over the three instances of the Orders view, allowing load and storage to be spread between them.

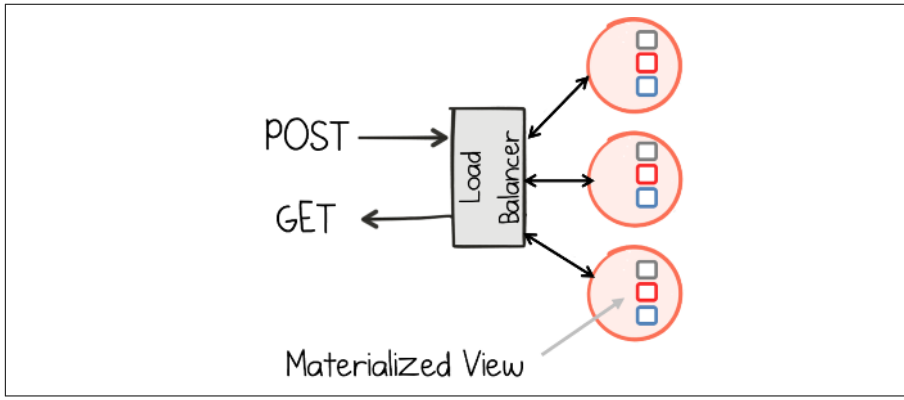


Figure 15-2. Close-up of the Orders Service, from Figure 15-1, demonstrating the materialized view it creates which can be accessed via an HTTP GET; the view represents the Query-side of the CQRS pattern and is spread over all three instances of the Orders Service

Because data is partitioned it can be scaled out horizontally (Kafka Streams supports dynamic load rebalancing), but it also means GET requests must be routed to the right node—the one that has the partition for the key being requested. This is handled automatically via the interactive queries functionality in Kafka Streams.[2]

There are actually two parts to this. The first is the query, which defines what data goes into the view. In this case we are grouping orders by their key (so new orders overwrite old orders), with the result written to a state store where it can be queried. We might implement this with the Kafka Streams DSL like so:

```
builder.stream(ORDERS.name(), serializer)
       .groupByKey(groupSerializer)
       .reduce((agg, newVal) -> newVal, getStateStore())
```

2 It is also common practice to implement such event-sourced views via Kafka Connect and your database of choice, as we discussed in "Query a Read-Optimized View Created in a Database" on page 69 in Chapter 7. Use this method when you need a richer query model or greater storage capacity.

The second part is to expose the state store(s) over an HTTP endpoint, which is simple enough, but when running with multiple instances requests must be routed to the correct partition and instance for a certain key. Kafka Streams includes a metadata service that does this for you.

Collapsing CQRS with a Blocking Read

The orders service implements a blocking HTTP GET so that clients can read their own writes. This technique is used to collapse the asynchronous nature of the CQRS pattern. So, for example, if a client wants to perform a write operation, immediately followed by a read, the event might not have propagated to the view, meaning they would either get an error or an incorrect value.

One solution is to block the GET operation until the event arrives (or a configured timeout passes), collapsing the asynchronicity of the CQRS pattern so that it appears synchronous to the client. This technique is essentially long polling. The orders service, in the example code, implements this technique using nonblocking IO.

Scaling Concurrent Operations in Streaming Systems

The inventory service is interesting because it needs to implement several specialist techniques to ensure it works accurately and consistently. The service performs a simple operation: when a user purchases an iPad, it makes sure there are enough iPads available for the order to be fulfilled, then physically reserves a number of them so no other process can take them (Figure 15-3). This is a little trickier than it may seem initially, as the operation involves managing atomic state across topics. Specifically:

1. Validate whether there are enough iPads in stock (inventory in warehouse minus items reserved).
2. Update the table of "reserved items" to reserve the iPad so no one else can take it.
3. Send out a message that validates the order.

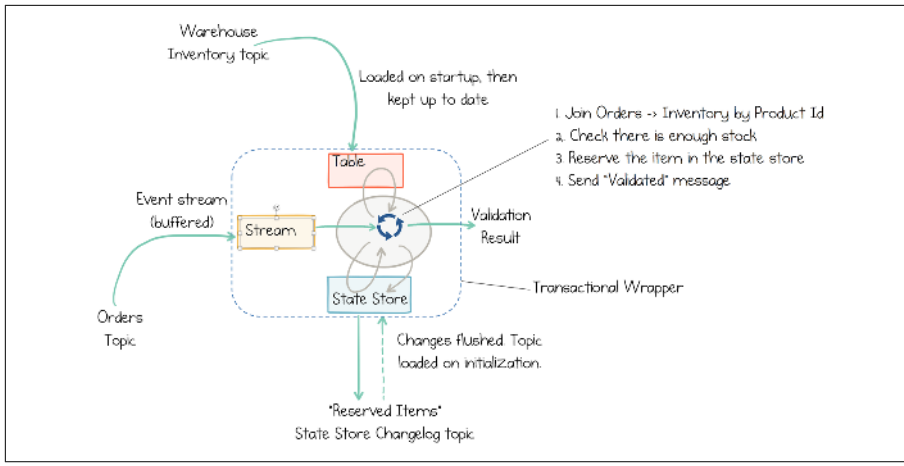


Figure 15-3. The inventory service validates orders by ensuring there is enough inventory in stock, then reserving items using a state store, which is backed by Kafka; all operations are wrapped in a transaction

This will work reliably only if we:

- Enable Kafka's transactions feature.
- Ensure that data is partitioned by `ProductId` before this operation is performed.

The first point should be pretty obvious: if we fail and we're not wrapped in a transaction, we have no idea what state the system will be in. But the second point should be a little less clear, because for it to make sense we need to think about this particular operation being scaled out linearly over several different threads or machines.

Stateful stream processing systems like Kafka Streams have a novel and high-performance mechanism for managing stateful problems like these concurrently. We have a single critical section:

1. Read the number of unreserved iPads currently in stock.
2. Reserve the iPads requested on the order.

Let's first consider how a traditional (i.e., not stateful) streaming system might work (Figure 15-4). If we scale the operation to run over two parallel processes, we would run the critical section inside a transaction in a (shared) database. So both instances would bottleneck on the same database instance.

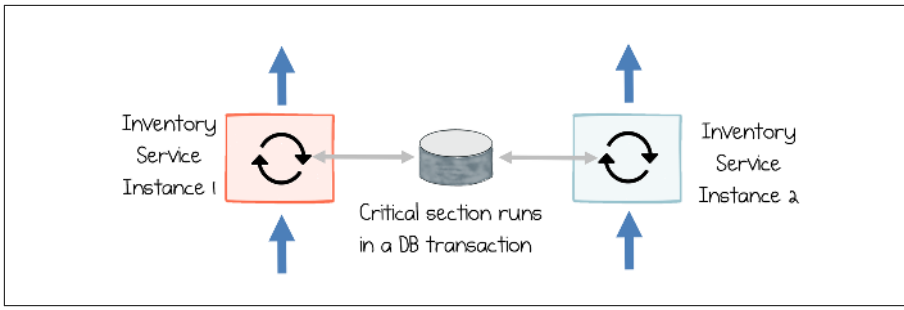


Figure 15-4. Two instances of a service manage concurrent operations via a shared database

Stateful stream processing systems like Kafka Streams avoid remote transactions or cross-process coordination. They do this by *partitioning* the problem over a set of threads or processes using a chosen business *key*. ("Partitions and Partitioning" was discussed in Chapter 4.) This provides the key (no pun intended) to scaling these systems horizontally.

Partitioning in Kafka Streams works by rerouting messages so that all the state required for one particular computation is sent to a single thread, where the computation can be performed.[3] The approach is inherently parallel, which is how streaming systems achieve such high message-at-a-time processing rates (for example, in the use case discussed in Chapter 2). But the approach works only if there is a logical key that cleanly segregates all operations: both state that they need, and state they operate on.

So splitting (i.e., partitioning) the problem by `ProductId` ensures that all operations for one `ProductId` will be sequentially executed on the same thread. That means all iPads will be processed on one thread, all iWatches will be processed on one (potentially different) thread, and the two will require no coordination between each other to perform the critical section (Figure 15-5). The resulting operation is atomic (thanks to Kafka's transactions), can be scaled out horizontally, and requires no expensive cross-network coordination. (This is similar to the Map phase in MapReduce systems.)

3 As an aside, one of the nice things about this feature is that it is managed by Kafka, not Kafka Streams. Kafka's Consumer Group Protocol lets any group of consumers control how partitions are distributed across the group.

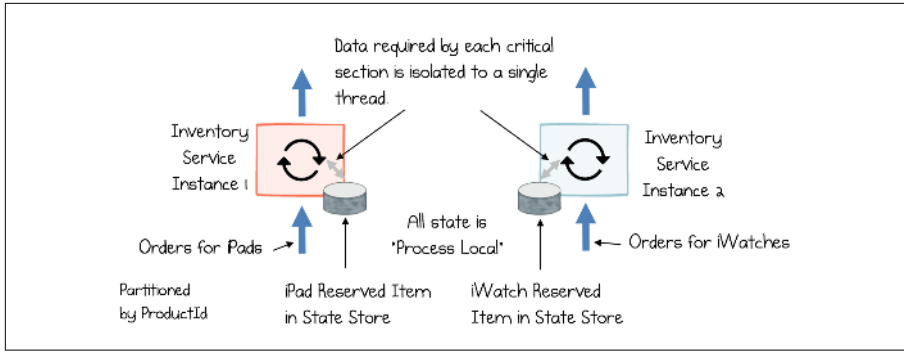


Figure 15-5. Services using the Kafka Streams API partition both event streams and stored state across the various services, which means all data required to run the critical section exists locally and is accessed by a single thread

The inventory service must rearrange orders so they are processed by `ProductId`. This is done with an operation called a *rekey*, which pushes orders into a new intermediary topic in Kafka, this time keyed by `ProductId`, and then back out to the inventory service. The code is very simple:

```
orders.selectKey((id, order) -> order.getProduct())//rekey by ProductId
```

Part 2 of the critical section is a state mutation: inventory must be reserved. The inventory service does this with a Kafka Streams state store (a local, disk-resident hash table, backed by a Kafka topic). So each thread executing will have a state store for "reserved stock" for some subset of the products. You can program with these state stores much like you would program with a hash map or key/value store, but with the benefit that all the data is persisted to Kafka and restored if the process restarts. A state store can be created in a single line of code:

```
KeyValueStore<Product, Long> store = context.getStateStore(RESERVED);
```

Then we make use of it, much like a regular hash table:

```
//Get the current reserved stock for this product
Long reserved = store.get(order.getProduct());
//Add the quantity for this order and submit it back
 store.put(order.getProduct(), reserved + order.getQuantity())
```

Writing to the store also partakes in Kafka's transactions, discussed in Chapter 12.

Rekey to Join

We can apply exactly the same technique used in the previous section, for partitioning writes, to partitioning reads (e.g., to do a join). Say we want to join a stream of orders (keyed by `OrderId`) to a table of warehouse inventory (keyed by `ProductId`), as we do in Figure 15-3. The join will have to use the `ProductId`.

This is what would be termed a foreign key relationship in relational parlance, mapping from `WarehouseInventory.ProductId` (its primary key) onto `Order.ProductId` (which isn't its primary key). To do this, we need to shuffle orders across the different nodes so that the orders end up being processed in the same thread that has the corresponding warehouse inventory assigned.

As mentioned earlier, this data redistribution step is called a *rekey*, and data arranged in this way is termed co-partitioned. Once rekeyed, the join condition can be performed without any additional network access required. For example, in Figure 15-6, inventory with `productId=5` is collocated with orders for `productId=5`.

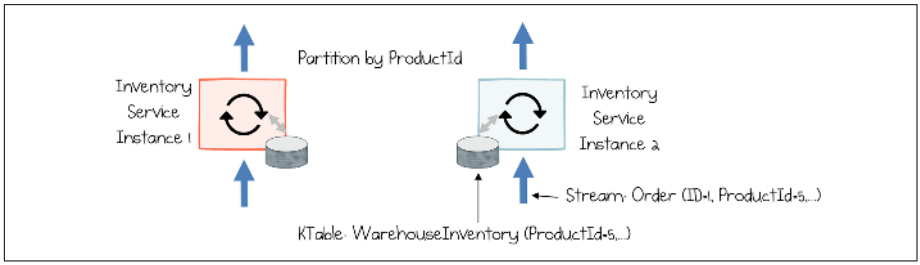


Figure 15-6. To perform a join between orders and warehouse inventory by ProductId, orders are repartitioned by ProductId, ensuring that for each product all corresponding orders will be on the same instance

Repartitioning and Staged Execution

Real-world systems are often more complex. One minute we're performing a join, the next we're aggregating by customer or materializing data in a view, with each operation requiring a different data distribution profile. Different operations like these chain together in a pipeline. The inventory service provides a good example of this. It uses a rekey operation to distribute data by `ProductId`. Once complete, it has to be rekeyed back to `OrderId` so it can be added to the Orders view (Figure 15-7). (The Orders view is destructive—that is, old versions of an order will be replaced by newer ones—so it's important that the stream be keyed by `OrderId` so that no data is lost.)

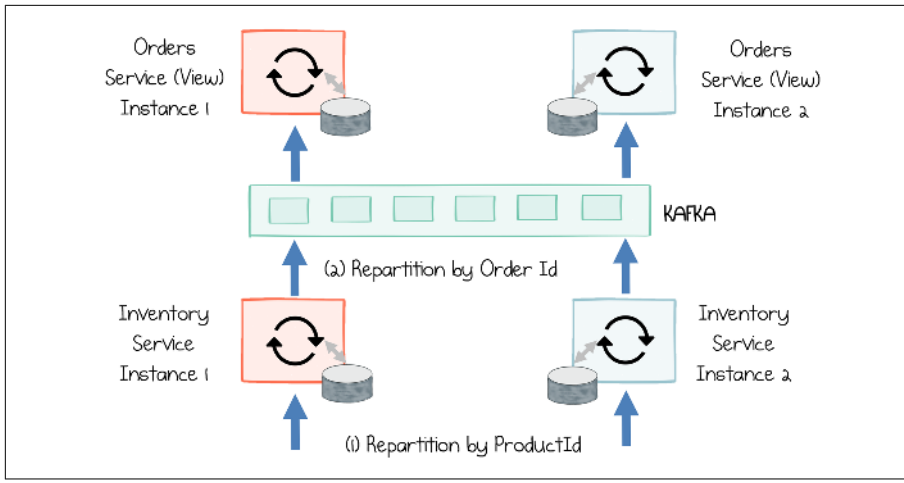


Figure 15-7. Two stages, which require joins based on different keys, are chained together via a rekey operation that changes the key from ProductId to OrderId

There are limitations to this approach, though. *The keys used to partition the event streams must be invariant if ordering is to be guaranteed*. So in this particular case it means the keys, `ProductId` and `OrderId`, on each order must remain fixed across all messages that relate to that order. Typically, this is a fairly easy thing to manage at a domain level (for example, by enforcing that, should a user want to change the product they are purchasing, a brand new order must be created).

Waiting for N Events

Another relatively common use case in business systems is to wait for *N* events to occur. This is trivial if each event is located in a different topic—it's simply a three-way join—but if events arrive on a single topic, it requires a little more thought.

The orders service, in the example discussed earlier in this chapter (Figure 15-1), waits for validation results from each of the three validation services, all sent via the same topic. Validation succeeds holistically only if all three return a `PASS`. Assuming you are counting messages with a certain key, the solution takes the form:

1. Group by the key.
2. Count occurrences of each key (using an aggregator executed with a window).
3. Filter the output for the required count.

Reflecting on the Design

Any distributed system comes with a baseline cost. This should go without saying. The solution described here provides good scalability and resiliency properties, but will always be more complex to implement and run than a simple, single-process application designed to perform the same logic. You should always carefully weigh the tradeoff between better nonfunctional properties and simplicity when designing a system. Having said that, a real system will inevitably be more complex, with more moving parts, so the pluggability and extensibility of this style of system can provide a worthy return against the initial upfront cost.

A More Holistic Streaming Ecosystem

In this final section we take a brief look at a larger ecosystem (Figure 15-8) that pulls together some of the main elements discussed in this book thus far, outlining how each service contributes, and the implementation patterns each might use:

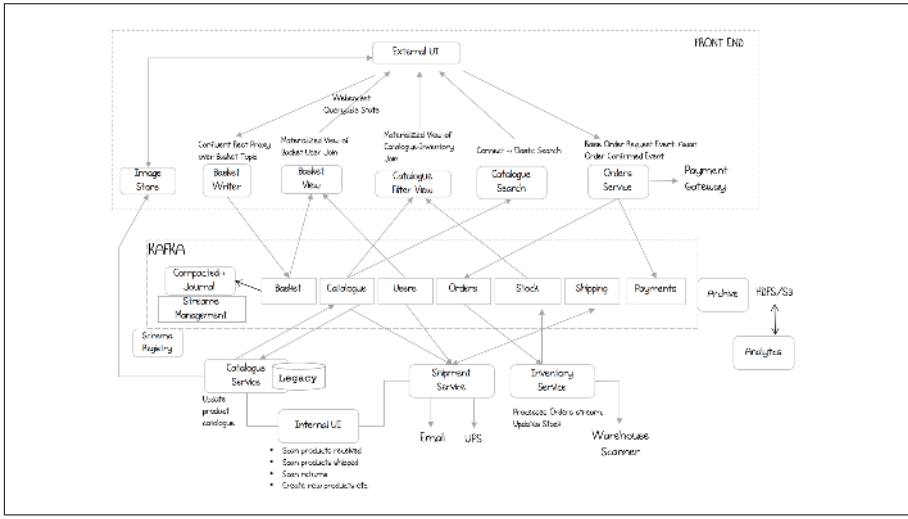


Figure 15-8. A more holistic streaming ecosystem

Basket writer/view

These represent an implementation of CQRS, as discussed in "Command Query Responsibility Segregation" on page 61 in Chapter 7. The Basket writer proxies HTTP requests, forwarding them to the Basket topic in Kafka when a user adds a new item. The Confluent REST proxy (which ships with the Confluent distribution of Kafka) is used for this. The Basket view is an event-sourced view, implemented in Kafka Streams, with the contents of its

state stores exposed over a REST interface in a manner similar to the orders service in the example discussed earlier in this chapter (Kafka Connect and a database could be substituted also). The view represents a join between User and Basket topics, but much of the information is thrown away, retaining only the bare minimum: `userId → List[product]`. This minimizes the view's footprint.

The Catalogue Filter view
: This is another event-sourced view but requires richer support for pagination, so the implementation uses Kafka Connect and Cassandra.

Catalogue search
: A third event-sourced view; this one uses Solr for its full-text search capabilities.

Orders service
: Orders are validated and saved to Kafka. This could be implemented either as a single service or a small ecosystem like the one detailed earlier in this chapter.

Catalog service
: A legacy codebase that manages changes made to the product catalog, initiated from an internal UI. This has comparatively fewer users, and an existing codebase. Events are picked up from the legacy Postgres database using a CDC connector to push them into Kafka. The single-message transforms feature reformats the messages before they are made public. Images are saved to a distributed filesystem for access by the web tier.

Shipping service
: A streaming service leveraging the Kafka Streams API. This service reacts to orders as they are created, updating the Shipping topic as notifications are received from the delivery company.

Inventory service
: Another streaming service leveraging the Kafka Streams API. This service updates inventory levels as products enter and leave the warehouse.

Archive
: All events are archived to HDFS, including two, fixed T-1 and T-10 point-in-time snapshots for recovery purposes. This uses Kafka Connect and its HDFS connector.

Streams management
: A set of stream processors manages creating latest/versioned topics where relevant (see the Latest-Versioned pattern in "Long-Term Data Storage" on page 25 in Chapter 3). This layer also manages the swing topics used when non-backward-compatible schema changes need to be rolled out. (See "Han-

dling Schema Change and Breaking Backward Compatibility" on page 124 in Chapter 13.)

Schema Registry
: The Confluent Schema Registry provides runtime validation of schemas and their compatibility.

Summary

When we build services using a streaming platform, some will be stateless—simple functions that take an input, perform a business operation, and produce an output. Some will be stateful, but read only, as in event-sourced views. Others will need to both read and write state, either entirely inside the Kafka ecosystem (and hence wrapped in Kafka's transactional guarantees), or by calling out to other services or databases. One of the most attractive properties of a stateful stream processing API is that all of these options are available, allowing us to trade the operational ease of stateless approaches for the data processing capabilities of stateful ones.

But there are of course drawbacks to this approach. While standby replicas, checkpoints, and compacted topics all mitigate the risks of pushing data to code, there is always a worst-case scenario where service-resident datasets must be rebuilt, and this should be considered as part of any system design.

There is also a mindset shift that comes with the streaming model, one that is inherently more asynchronous and adopts a more functional and data-centric style, when compared to the more procedural nature of traditional service interfaces. But this is—in the opinion of this author—an investment worth making.

In this chapter we looked at a very simple system that processes orders. We did this with a set of small streaming microservices that implement the Event Collaboration pattern we discussed in Chapter 5. Finally, we looked at how we can create a larger architecture using the broader range of patterns discussed in this book.

About the Author

Ben Stopford is a technologist working in the Office of the CTO at Confluent, Inc. (the company behind Apache Kafka), where he has worked on a wide range of projects, from implementing the latest version of Kafka's replication protocol through to developing strategies for streaming applications. Before Confluent, Ben led the design and build of a company-wide data platform for a large financial institution, as well as working on a number of early service-oriented systems, both in finance and at ThoughtWorks.

Ben is a regular conference speaker, blogger, and keen observer of the data-technology space. He believes that we are entering an interesting and formative period where data-engineering, software engineering, and the lifecycle of organisations become ever more closely intertwined.