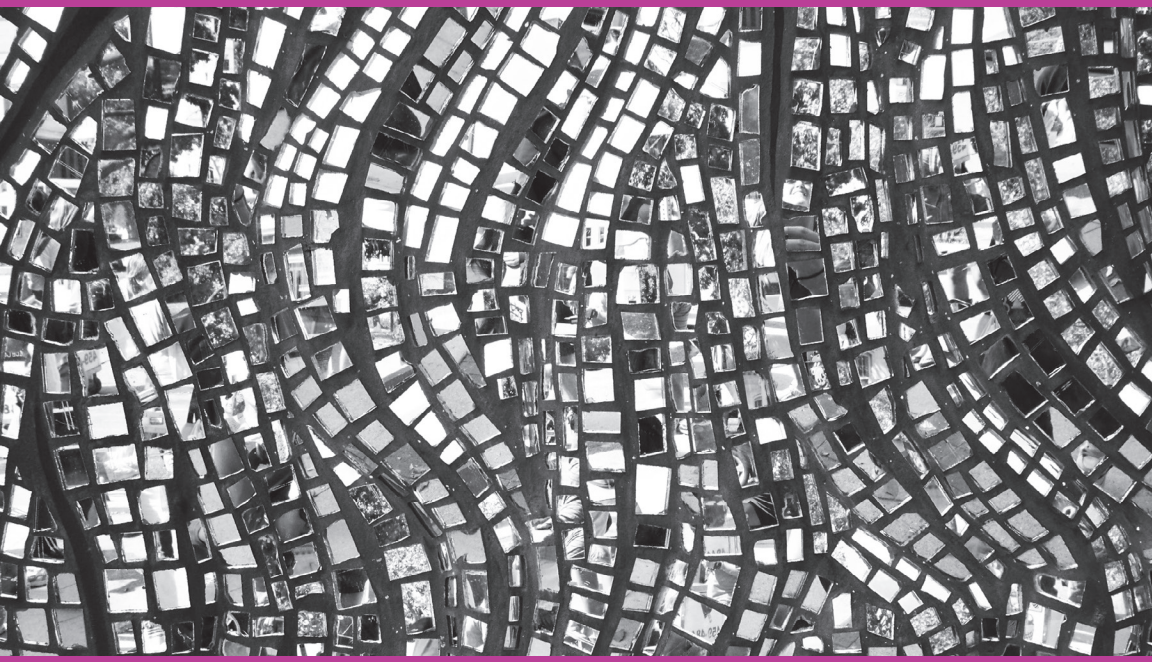
O'REILLY®

Migrating to Microservice Databases

From Relational Monolith to Distributed Data



Edson Yanaga

Migrating to Microservice Databases

From Relational Monolith to Distributed Data

Edson Yanaga

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

Migrating to Microservice Databases
by Edson Yanaga

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*.

Editors: Nan Barber and Susan Conant
Production Editor: Melanie Yarbrough
Copyeditor: Octal Publishing, Inc.
Proofreader: Eliahu Sussman

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

February 2017: First Edition

Revision History for the First Edition
2017-01-25: First Release
2017-03-31: Second Release

978-1-491-97186-4

[LSI]

You can sell your time, but you can never buy it back. So the price of everything in life is the amount of time you spend on it.

To my family: Edna, my wife, and Felipe and Guilherme, my two dear sons. This book was very expensive to me, but I hope that it will help many developers to create better software. And with it, change the world for the better for all of you.

To my dear late friend: Daniel deOliveira. Daniel was a DFJUG leader and founding Java Champion. He helped thousands of Java developers worldwide and was one of those rare people who demonstrated how passion can truly transform the world in which we live for the better. I admired him for demonstrating what a Java Champion must be.

To Emmanuel Bernard, Randall Hauch, and Steve Suehring. Thanks for all the valuable insight provided by your technical feedback. The content of this book is much better, thanks to you.

Table of Contents

Foreword

To say that data is important is an understatement. Does your code outlive your data, or vice versa? QED. The most recent example of this adage involves Artificial Intelligence (AI). Algorithms are important. Computational power is important. But the key to AI is collecting a massive amount of data. Regardless of your algorithm, no data means no hope. That is why you see such a race to collect data by the tech giants in very diverse fields—automotive, voice, writing, behavior, and so on.

And despite the critical importance of data, this subject is often barely touched or even ignored when discussing microservices. In microservices style, you should write *stateless* applications. But useful applications are not without state, so what you end up doing is moving the state out of your app and into *data services*. You've just shifted the problem. I can't blame anyone; properly implementing the full elasticity of a data service is so much more difficult than doing this for stateless code. Most of the patterns and platforms supporting the microservices architecture style have left the data problem for later. The good news is that this is changing. Some platforms, like Kubernetes, are now addressing this issue head on.

After you tackle the elasticity problem, you reach a second and more pernicious one: the evolution of your data. Like code, data structure evolves, whether for new business needs, or to reshape the actual structure to cope better with performance or address more use cases. In a microservices architecture, this problem is particularly acute because although data needs to flow from one service to the other, you do not want to interlock your microservices and force synchronized releases. That would defeat the whole purpose!

This is why Edson's book makes me happy. Not only does he discuss data in a microservices architecture, but he also discusses *evolution* of this data. And he does all of this in a very pragmatic and practical manner. You'll be ready to use these evolution strategies as soon as you close the book. Whether you fully embrace microservices or just want to bring more agility to your IT system, expect more and more discussions on these subjects within your teams—be prepared.

— Emmanuel Bernard
Hibernate Team and Red Hat
Middleware's data platform
architect

CHAPTER 1
Introduction

Microservices certainly aren't a panacea, but they're a good solution if you have the right problem. And each solution also comes with its own set of problems. Most of the attention when approaching the microservice solution is focused on the architecture around the code artifacts, but no application lives without its data. And when distributing data between different microservices, we have the challenge of integrating them.

In the sections that follow, we'll explore some of the reasons you might want to consider microservices for your application. If you understand why you need them, we'll be able to help you figure out how to distribute and integrate your persistent data in relational databases.

The Feedback Loop

The feedback loop is one of the most important processes in human development. We need to constantly assess the way that we do things to ensure that we're on the right track. Even the classic Plan-Do-Check-Act (PDCA) process is a variation of the feedback loop.

In software—as with everything we do in life—the longer the feedback loop, the worse the results are. And this happens because we have a limited amount of capacity for holding information in our brains, both in terms of volume and duration.

Remember the old days when all we had as a tool to code was a text editor with black background and green fonts? We needed to com-

pile our code to check if the syntax was correct. Sometimes the compilation took minutes, and when it was finished we already had lost the context of what we were doing before. The *lead time*[1] in this case was too long. We improved when our IDEs featured on-the-fly syntax highlighting and compilation.

We can say the same thing for testing. We used to have a dedicated team for manual testing, and the lead time between committing something and knowing if we broke anything was days or weeks. Today, we have automated testing tools for unit testing, integration testing, acceptance testing, and so on. We improved because now we can simply run a build on our own machines and check if we broke code somewhere else in the application.

These are some of the numerous examples of how reducing the lead time generated better results in the software development process. In fact, we might consider that all the major improvements we had with respect to process and tools over the past 40 years were targeting the improvement of the feedback loop in one way or another.

The current improvement areas that we're discussing for the feedback loop are DevOps and microservices.

DevOps

You can find thousands of different definitions regarding DevOps. Most of them talk about culture, processes, and tools. And they're not wrong. They're all part of this bigger transformation that is DevOps.

The purpose of DevOps is to make software development teams reclaim the ownership of their work. As we all know, bad things happen when we separate people from the consequences of their jobs. The entire team, Dev and Ops, must be responsible for the outcomes of the application.

There's no bigger frustration for developers than watching their code stay idle in a repository for months before entering into production. We need to regain that bright gleam in our eyes from delivering something and seeing the difference that it makes in people's lives.

1 The amount of time between the beginning of a task and its completion.

We need to deliver software faster—and safer. But what are the excuses that we lean on to prevent us from delivering it?

After visiting hundreds of different development teams, from small to big, and from financial institutions to ecommerce companies, I can testify that the number one excuse is bugs.

We don't deliver software faster because each one of our software releases creates a lot of bugs in production.

The next question is: what causes bugs in production?

This one might be easy to answer. The cause of bugs in production in each one of our releases is *change*: both changes in code and in the environment. When we change things, they tend to fall apart. But we can't use this as an excuse for not changing! Change is part of our lives. In the end, it's the only certainty we have.

Let's try to make a very simple correlation between changes and bugs. The more changes we have in each one of our releases, the more bugs we have in production. Doesn't it make sense? The more we mix the things in our codebase, the more likely it is something gets screwed up somewhere.

The traditional way of trying to solve this problem is to have more time for testing. If we delivered code every week, now we need two weeks—because we need to test more. If we delivered code every month, now we need two months, and so on. It isn't difficult to imagine that sooner or later some teams are going to deploy software into production only on anniversaries.

This approach sounds anti-economical. The economic approach for delivering software in order to have fewer bugs in production is the opposite: we need to deliver more often. And when we deliver more often, we're also reducing the amount of things that change between one release and the next. So the fewer things we change between releases, the less likely it is for the new version to cause bugs in production.

And even if we still have bugs in production, if we only changed a few dozen lines of code, where can the source of these bugs possibly be? The smaller the changes, the easier it is to spot the source of the bugs. And it's easier to fix them, too.

The technical term used in DevOps to characterize the amount of changes that we have between each release of software is called *batch*

size. So, if we had to coin just one principle for DevOps success, it would be this:

> Reduce your batch size to the minimum allowable size you can handle.

To achieve that, you need a fully automated software deployment pipeline. That's where the processes and tools fit together in the big picture. But you're doing all of that in order to reduce your batch size.

Bugs Caused by Environment Differences Are the Worst

When we're dealing with bugs, we usually have log statements, a stacktrace, a debugger, and so on. But even with all of that, we still find ourselves shouting: "but it works on my machine!"

This horrible scenario—code that works on your machine but doesn't in production—is caused by differences in your environments. You have different operating systems, different kernel versions, different dependency versions, different database drivers, and so forth. In fact, it's a surprise things ever do work well in production.

You need to develop, test, and run your applications in development environments that are as close as possible in configuration to your production environment. Maybe you can't have an Oracle RAC and multiple Xeon servers to run in your development environment. But you might be able to run the same Oracle version, the same kernel version, and the same application server version in a virtual machine (VM) on your own development machine.

Infrastructure-as-code tools such as Ansible, Puppet, and Chef really shine, automating the configuration of infrastructure in multiple environments. We strongly advocate that you use them, and you should commit their scripts in the same source repository as your application code.[2] There's usually a match between the environment configuration and your application code. Why can't they be versioned together?

Container technologies offer many advantages, but they are particularly useful at solving the problem of different environment con-

2 Just make sure to follow the tool's best practices and do not store sensitive information, such as passwords, in a way that unauthorized users might have access to it.

figurations by packaging application *and* environment into a single containment unit—the *container*. More specifically, the result of packaging application and environment in a single unit is called a *virtual appliance*. You can set up virtual appliances through VMs, but they tend to be big and slow to start. Containers take virtual appliances one level further by minimizing the virtual appliance size and startup time, and by providing an easy way for distributing and consuming container images.

Another popular tool is Vagrant. Vagrant currently does much more than that, but it was created as a provisioning tool with which you can easily set up a development environment that closely mimics as your production environment. You literally just need a `Vagrantfile`, some configuration scripts, and with a simple `vagrant up` command, you can have a full-featured VM or container with your development dependencies ready to run.

Why Microservices?

Some might think that the discussion around microservices is about scalability. Most likely it's not. Certainly we always read great things about the microservices architectures implemented by companies like Netflix or Amazon. So let me ask a question: how many companies in the world can be Netflix and Amazon? And following this question, another one: how many companies in the world need to deal with the same scalability requirements as Netflix or Amazon?

The answer is that the great majority of developers worldwide are dealing with enterprise application software. Now, I don't want to underestimate Netflix's or Amazon's domain model, but an enterprise domain model is a completely wild beast to deal with.

So, for the majority of us developers, microservices is usually not about scalability; it's all about again improving our lead time and reducing the batch size of our releases.

But we have DevOps that shares the same goals, so why are we even discussing microservices to achieve this? Maybe your development team is so big and your codebase is so huge that it's just too difficult to change anything without messing up a dozen different points in your application. It's difficult to coordinate work between people in a huge, tightly coupled, and entangled codebase.

With microservices, we're trying to split a piece of this huge monolithic codebase into a smaller, well-defined, cohesive, and loosely coupled artifact. And we'll call this piece a microservice. If we can identify some pieces of our codebase that naturally change together and apart from the rest, we can separate them into another artifact that can be released independently from the other artifacts. We'll improve our lead time and batch size because we won't need to wait for the other pieces to be "ready"; thus, we can deploy our microservice into production.

You Need to Be This Tall to Use Microservices

Microservices architectures encompasses multiple artifacts, each of which must be deployed into production. If you still have issues deploying one single monolith into production, what makes you think that you'll have fewer problems with multiple artifacts? A very mature software deployment pipeline is an absolute requirement for any microservices architecture. Some indicators that you can use to assess pipeline maturity are the amount of manual intervention required, the amount of automated tests, the automatic provisioning of environments, and monitoring.

Distributed systems are difficult. So are people. When we're dealing with microservices, we must be aware that we'll need to face an entire new set of problems that distributed systems bring to the table. Tracing, monitoring, log aggregation, and resilience are some of problems that you don't need to deal with when you work on a monolith.

Microservices architectures come with a high toll, which is worth paying if the problems with your monolithic approaches cost you more. Monoliths and microservices are different architectures, and architectures are all about trade-off.

Strangler Pattern

Martin Fowler wrote a nice article regarding the monolith-first approach. Let me quote two interesting points of his article:

- Almost all the successful microservice stories have started with a monolith that grew too big and was broken up.

- Almost all the cases I've heard of a system that was built as a microservice system from scratch, it has ended up in serious trouble.

For all of us enterprise application software developers, maybe we're lucky—we don't need to throw everything away and start from scratch (if anybody even considered this approach). We would end up in serious trouble. But the real lucky part is that we already have a monolith to maintain in production.

The monolith-first is also called the *strangler pattern* because it resembles the development of a tree called the *strangler fig*. The strangler fig starts small in the top of a host tree. Its roots then start to grow toward the ground. Once its roots reach the ground, it grows stronger and stronger, and the fig tree begins to grow around the host tree. Eventually the fig tree becomes bigger than the host tree, and sometimes it even kills the host. Maybe it's the perfect analogy, as we all have somewhere hidden in our hearts the deep desire of killing that monolith beast.

Having a stable monolith is a good starting point because one of the hardest things in software is the identification of boundaries between the domain model—things that change together, and things that change apart. Create wrong boundaries and you'll be doomed with the consequences of cascading changes and bugs. And boundary identification is usually something that we mature over time. We refactor and restructure our system to accommodate the acquired boundary knowledge. And it's much easier to do that when you have a single codebase to deal with, for which our modern IDEs will be able to refactor and move things automatically. Later you'll be able to use these established boundaries for your microservices. That's why we really enjoy the strangler pattern: you start small with microservices and grow around a monolith. It sounds like the wisest and safest approach for evolving enterprise application software.

The usual candidates for the first microservices in your new architecture are new features of your system or changing features that are peripheral to the application's core. In time, your microservices architecture will grow just like a strangler fig tree, but we believe that the reality of most companies will still be one, two, or maybe even up to half-dozen microservices coexisting around a monolith.

The challenge of choosing which piece of software is a good candidate for a microservice requires a bit of Domain-Driven Design knowledge, which we'll cover in the next section.

Domain-Driven Design

It's interesting how some methodologies and techniques take years to "mature" or to gain awareness among the general public. And Domain-Driven Design (DDD) is one of these very useful techniques that is becoming almost essential in any discussion about microservices. Why now? Historically we've always been trying to achieve two synergic properties in software design: *high cohesion* and *low coupling*. We aim for the ability to create boundaries between entities in our model so that they work well together and don't propagate changes to other entities beyond the boundary. Unfortunately, we're usually especially bad at that.

DDD is an approach to software development that tackles complex systems by mapping activities, tasks, events, and data from a business domain to software artifacts. One of the most important concepts of DDD is the *bounded context*, which is a cohesive and well-defined unit within the business model in which you define the boundaries of your software artifacts.

From a domain model perspective, microservices are all about boundaries: we're splitting a specific piece of our domain model that can be turned into an independently releasable artifact. With a badly defined boundary, we will create an artifact that depends too much on information confined in another microservice. We will also create another operational pain: whenever we make modifications in one artifact, we will need to synchronize these changes with another artifact.

We advocate for the monolith-first approach because it allows you to mature your knowledge around your business domain model first. DDD is such a useful technique for identifying the bounded contexts of your domain model: things that are grouped together and achieve high cohesion and low coupling. From the beginning, it's very difficult to guess which parts of the system change together and which ones change separately. However, after months, or more likely years, developers and business analysts should have a better picture of the evolution cycle of each one of the bounded contexts.

These are the ideal candidates for microservices extraction, and that will be the starting point for the strangling of our monolith.

To learn more about DDD, check out Eric Evan's book, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, and Vaughn Vernon's book, *Implementing Domain-Driven Design*.

Microservices Characteristics

James Lewis and Martin Fowler provided a reasonable common set of characteristics that fit most of the microservices architectures:

- Componentization via services
- Organized around business capabilities
- Products not projects
- Smart endpoints and dumb pipes
- Decentralized governance
- Decentralized data management
- Infrastructure automation
- Design for failure
- Evolutionary design

All of the aforementioned characteristics certainly deserve their own careful attention. But after researching, coding, and talking about microservices architectures for a couple of years, I have to admit that the most common question that arises is this:

> How do I evolve my monolithic legacy database?

This question provoked some thoughts with respect to how enterprise application developers could break their monoliths more effectively. So the main characteristic that we'll be discussing throughout this book is *Decentralized Data Management*. Trying to simplify it to a single-sentence concept, we might be able to state that:

> Each microservice should have its own separate database.

This statement comes with its own challenges. Even if we think about greenfield projects, there are many different scenarios in which we require information that will be provided by another ser-

vice. Experience has taught us that relying on remote calls (either some kind of Remote Procedure Call [RPC] or REST over HTTP) usually is not performant enough for data-intensive use cases, both in terms of throughput and latency.

This book is all about strategies for dealing with your relational database. Chapter 2 addresses the architectures associated with deployment. The zero downtime migrations presented in Chapter 3 are not exclusive to microservices, but they're even more important in the context of distributed systems. Because we're dealing with distributed systems with information scattered through different artifacts interconnected via a network, we'll also need to deal with how this information will converge. Chapter 4 describes the difference between consistency models: Create, Read, Update, and Delete (CRUD); and Command and Query Responsibility Segregation (CQRS). The final topic, which is covered in Chapter 5, looks at how we can integrate the information between the nodes of a microservices architecture.

What About NoSQL Databases?

Discussing microservices and database types different than relational ones seems natural. If each microservice must have is own separate database, what prevents you from choosing other types of technology? Perhaps some kinds of data will be better handled through key-value stores, or document stores, or even flat files and git repositories.

There are many different success stories about using NoSQL databases in different contexts, and some of these contexts might fit your current enterprise context, as well. But even if it does, we still recommend that you begin your microservices journey on the safe side: using a relational database. First, make it work using your existing relational database. Once you have successfully finished implementing and integrating your first microservice, you can decide whether you (or) your project will be better served by another type of database technology.

The microservices journey is difficult and as with any change, you'll have better chances if you struggle with one problem at a time. It doesn't help having to simultaneously deal with a new thing such as microservices *and* new unexpected problems caused by a different database technology.

CHAPTER 2
Zero Downtime

Any improvement that you can make toward the reduction of your batch size that consequently leads to a faster feedback loop is important. When you begin this continuous improvement, sooner or later you will reach a point at which you can no longer reduce the time between releases due to your *maintenance window*—that short timeframe during which you are allowed to drop the users from your system and perform a software release.

Maintenance windows are usually scheduled for the hours of the day when you have the least concern disrupting users who are accessing your application. This implies that you will mostly need to perform your software releases late at night or on weekends. That's not what we, as the people responsible for owning it in production, would consider sustainable. We want to reclaim our lives, and if we are now supposed to release software even more often, certainly it's not sustainable to do it every night of the week.

Zero downtime is the property of your software deployment pipeline by which you release a new version of your software to your users without disrupting their current activities—or at least minimizing the extent of potential disruptions.

In a deployment pipeline, zero downtime is the feature that will enable you to eliminate the maintenance window. Instead of having a strict timeframe with in which you can deploy your releases, you might have the freedom to deploy new releases of software at any time of the day. Most companies have a maintenance window that occurs once a day (usually at night), making your smallest release

cycle a single day. With zero downtime, you will have the ability to deploy multiple times per day, possibly with increasingly smaller batches of change.

Zero Downtime and Microservices

Just as we saw in "Why Microservices?" on page 5, we're choosing microservices as a strategy to release faster and more frequently. Thus, we can't be tied to a specific maintenance window.

If you have only a specific timeframe in which you can release all of your production artifacts, maybe you don't need microservices at all; you can keep the same release pace by using your old-and-gold monolith.

But zero downtime is not only about releasing at any time of day. In a distributed system with multiple moving parts, you can't allow the unavailability caused by a deployment in a single artifact to bring down your entire system. You're not allowed to have downtime for this reason.

Deployment Architectures

Traditional deployment architectures have the clients issuing requests directly to your server deployment, as pictured in Figure 2-1.

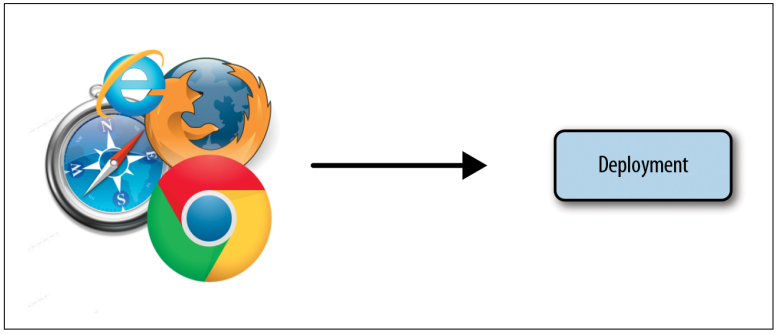


Figure 2-1. Traditional deployment architecture

Unless your platform provides you with some sort of "hot deployment," you'll need to undeploy your application's current version and then deploy the new version to your running system. This will result in an undesirable amount of downtime. More often than not,

it adds up to the time you need to wait for your application server to reboot, as most of us do that anyway in order to clean up anything that might have been left by the previous version.

To allow our deployment architecture to have zero downtime, we need to add another component to it. For a typical web application, this means that instead of allowing users to directly connect to your application's process servicing requests, we'll now have another process receiving the user's requests and forwarding them to your application. This new addition to the architecture is usually called a *proxy* or a *load balancer*, as shown in Figure 2-2.

If your application receives a small amount of requests per second, this new process will mostly be acting as a proxy. However, if you have a large amount of incoming requests per second, you will likely have more than one instance of your application running at the same time. In this scenario, you'll need something to balance the load between these instances—hence a load balancer.

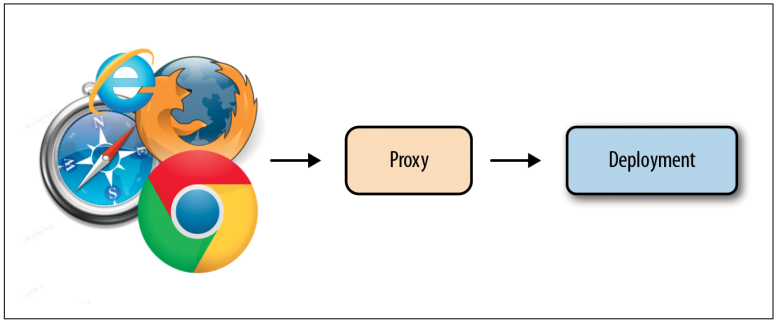


Figure 2-2. Deployment architecture with a proxy

Some common examples of software products that are used today as proxies or load balancers are `haproxy` and `nginx`, even though you could easily configure your old and well-known `Apache` web server to perform these activities to a certain extent.

After you have modified your architecture to accommodate the proxy or load balancer, you can upgrade it so that you can create *blue/green deployments* of your software releases.

Blue/Green Deployment

Blue/green deployment is a very interesting deployment architecture that consists of two different releases of your application running

concurrently. This means that you'll require two identical environments: one for the production stage, and one for your development platform, each being capable of handling 100% of your requests on its own. You will need the *current* version and the *new* version running in production during a deployment process. This is represented by the *blue* deployment and the *green* deployment, respectively, as depicted in Figure 2-3.

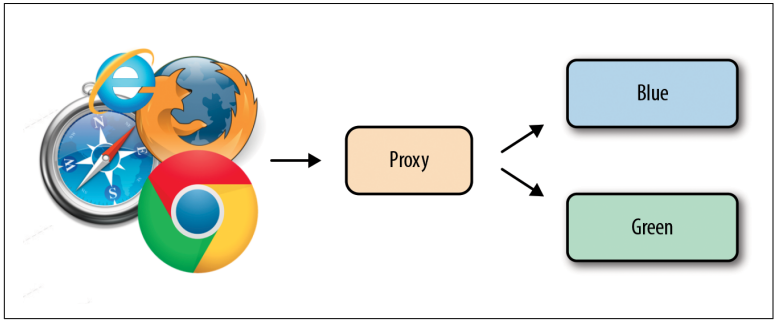


Figure 2-3. A blue/green deployment architecture

NOTE

Blue/Green Naming Convention

Throughout this book, we will always consider the blue deployment as the current running version, and the green deployment as the new version of your artifact. It's not an industry-standard coloring; it was chosen at the discretion of the author.

In a usual production scenario, your proxy will be forwarding to your blue deployment. After you start and finish the deployment of the new version in the green deployment, you can manually (or even automatically) configure your proxy to stop forwarding your requests to the blue deployment and start forwarding them to the green one. This must be made as an on-the-fly change so that no incoming requests will be lost between the changes from blue deployment to green.

This deployment architecture greatly reduces the risk of your software deployment process. If there is anything wrong with the new version, you can simply change your proxy to forward your requests to the previous version—without the implication of having to wait for it to be deployed again and then warmed up (and experience

tells us that this process can take a terrifyingly long amount of time when things go wrong).

Compatibility Between Releases

One very important issue that arises when using a blue/green deployment strategy is that your software releases must be forward *and* backward compatible to be able to consistently coexist at the same time running in production. From a code perspective, it usually implies that changes in exposed APIs must retain compatibility. And from the state perspective (data), it implies that eventual changes that you execute in the structure of the information must allow both versions to read and write successfully in a consistent state. We'll cover more of this topic in Chapter 3.

Canary Deployment

The idea of routing 100% of the users to a new version all at once might scare some developers. If anything goes wrong, 100% of your users will be affected. Instead, we could try an approach that gradually increases user traffic to a new version and keeps monitoring it for problems. In the event of a problem, you roll back 100% of the requests to the current version.

This is known as a canary deployment, the name borrowed from a technique employed by coal miners many years ago, before the advent of modern sensor safety equipment. A common issue with coal mines is the build up of toxic gases, not all of which even have an odor. To alert themselves to the presence of dangerous gases, miners would bring caged canaries with them into the mines. In addition to their cheerful singing, canaries are highly susceptible to toxic gases. If the canary died, it was time for the miners to get out fast, before they ended up like the canary.

Canary development draws on this analogy, with the gradual deployment and monitoring playing the role of the canary: if problems with the new version are detected, you have the ability to revert to the previous version and avert potential disaster.

We can make another distinction even within canary deployments. A standard canary deployment can be handled by infrastructure alone, as you route a certain percentage of all the requests to your

new version. On the other hand, a *smart* canary requires the presence of a *smart router* or a *feature-toggle framework*.

Smart Routers and Feature-Toggle Frameworks

A *smart router* is a piece of software dedicated to routing requests to backend endpoints based on business logic. One popular implementation in the Java world for this kind of software is Netflix's OSS Zuul.

For example, in a smart router, you can choose to route only the iOS users first to the new deployment—because they're the users having issues with the current version. You don't want to risk breaking the Android users. Or else you might want to check the log messages on the new version only for the iOS users.

Feature-toggle frameworks allow you to choose which part of your code will be executed, depending on some configurable toggles. Popular frameworks in the Java space are FF4J and Togglz.

The toggles are usually Boolean values that are stored in an external data source. They can be changed online in order to modify the behavior of the application dynamically.

Think of feature toggles as an if/else framework configured externally through the toggles. It's an over-simplification of the concept, but it might give you a notion of how it works.

The interesting thing about feature toggles is that you can separate the concept of a deployment from the release of a feature. When you flip the toggle to expose your new feature to users, the codebase has already been deployed for a long time. And if anything goes wrong, you can always flip it back and hide it from your users.

Feature toggles also come with many downsides, so be careful when choosing to use them. The new code and the old code will be maintained in the same codebase until you do a cleanup. Verifiability also becomes very difficult with feature toggles because knowing in which state the toggles were at a given point in time becomes tricky. If you work in a field governed by regulations, it's also difficult to audit whether certain pieces of the code are correctly executed on your production system.

A/B Testing

A/B testing is not related directly to the deployment process. It's an advanced scenario in which you can use two different and separate production environments to test a business hypothesis.

When we think about blue/green deployment, we're always releasing a new version whose purpose is to supersede the previous one.

In A/B testing, there's no relation of current/new version, because both versions can be different branches of source code. We're running two separate production environments to determine which one performs better in terms of business value.

We can even have two production environments, A and B, with each of them implementing a blue/green deployment architecture.

One strong requirement for using an A/B testing strategy is that you have an advanced monitoring platform that is tied to business results instead of just infrastructure statistics.

After we have measured them long enough and compared both to a standard baseline, we get to choose which version (A or B) performed better and then kill the other one.

Application State

Any journeyman who follows the DevOps path sooner or later will come to the conclusion that with all of the tools, techniques, and culture that are available, creating a software deployment pipeline is not that difficult when you talk about code, because code is stateless. The real problem is the application state.

From the state perspective, the application has two types of state: *ephemeral* and *persistent*. Ephemeral state is usually stored in memory through the use of HTTP sessions in the application server. In some cases, you might even prefer to not deal with the ephemeral state when releasing a new version. In a worst-case scenario, the user will need to authenticate again and restart the task he was executing. Of course, he won't exactly be happy if he loses that 200-line form he was filling in, but you get the point.

To prevent ephemeral state loss during deployments, we must externalize this state to another datastore. One usual approach is to store the HTTP session state in in-memory, key-value solutions such as

Infinispan, Memcached, or Redis. This way, even if you restart your application server, you'll have your ephemeral state available in the external datastore.

It's much more difficult when it comes to persistent state. For enterprise applications, the number one choice for persistent state is undoubtedly a relational database. We're not allowed to lose any information from persistent data, so we need some special techniques to be able to deal with the upgrade of this data. We cover these in Chapter 3.

CHAPTER 3
Evolving Your Relational Database

> Code is easy; state is hard.
>
> —Edson Yanaga

The preceding statement is a bold one.[1] However, code is not easy. Maybe bad code is easy to write, but *good code is always difficult*. Yet, even if good code is tricky to write, managing persistent state is tougher.

From a very simple point of view, a relational database comprises *tables* with multiple *columns* and *rows*, and *relationships* between them. The collection of database objects' definitions associated within a certain namespace is called a *schema*. You can also consider a schema to be the definition of your data structures within your database.

Just as our data changes over time with Data Manipulation Language (DML) statements, so does our schema. We need to add more tables, add and remove columns, and so on. The process of evolving our database structure over time is called *schema evolution*.

Schema evolution uses Data Definition Language (DDL) statements to transition the database structure from one version to the other. The set of statements used in each one of these transitions is called *database migrations*, or simply *migrations*.

1 If it got your attention, the statement's mission was accomplished!

It's not unusual to have teams applying database migrations manually between releases of software. Nor is it unusual to have someone sending an email to the Database Administrator (DBA) with the migrations to be applied. Unfortunately, it's also not unusual for those instructions to get lost among hundreds of other emails.

Database migrations need to be a part of our software deployment process. Database migrations *are* code, and they must be treated as such. They need to be committed in the same code repository as your application code. They must be versioned along with your application code. Isn't your database schema tied to a specific application version, and vice versa? There's no better way to assure this match between versions than to keep them in the same code repository.

We also need an automated software deployment pipeline and tools that automate these database migration steps. We'll cover some of them in the next section.

Popular Tools

Some of the most popular tools for schema evolution are Liquibase and Flyway. Opinions might vary, but the current set of features that both offer almost match each other. Choosing one instead of the other is a matter of preference and familiarity.

Both tools allow you to perform the schema evolution of your relational database during the startup phase of your application. You will likely want to avoid this, because this strategy is only feasible when you can guarantee that you will have only a single instance of your application starting up at a given moment. That might not be the case if you are running your instances in a Platform as a Service (PaaS) or container orchestration environment.

Our recommended approach is to tie the execution of the schema evolution to your software deployment pipeline so that you can assure that the tool will be run only once for each deployment, and that your application will have the required schema already upgraded when it starts up.

In their latest versions, both Liquibase and Flyway provide locking mechanisms to prevent multiple concurrent processes updating the database. We still prefer to not tie database migrations to application startup: we want to stay on the safe side.

Zero Downtime Migrations

As pointed out in the section "Application State" on page 19, you can achieve zero downtime for ephemeral state by externalizing the state data in a storage external to the application. From a relational database perspective, zero downtime on a blue/green deployment requires that both your new and old schemas' versions continue to work correctly at the same time.

Schema versions between consecutive releases must be mutually *compatible*. It also means that we can't create database migrations that are *destructive*. Destructive here means that we can't afford to lose any data, so we can't issue any statement that can *potentially* cause the loss of data.

Suppose that we needed to rename a column in our database schema. The traditional approach would be to issue this kind of DDL statement:

```
ALTER TABLE customers RENAME COLUMN wrong TO correct;
```

But in the context of zero downtime migrations, this statement is not allowable for three reasons:

- It is destructive: you're losing the information that was present in the old column.[2]
- It is not compatible with the current version of your software. Only the new version knows how to manipulate the new column.
- It can take a long time to execute: some database management systems (DBMS) might lock the entire table to execute this statement, leading to application downtime.

Instead of just issuing a single statement to achieve a single column rename, we'll need to get used to breaking these *big* changes into multiple smaller changes. We're again using the concept of *baby steps* to improve the quality of our software deployment pipeline.

2 You might argue that we're just *moving* information, but if you try to access the old column, it's not there anymore!

The previous DDL statement can be refactored to the following smaller steps, each one being executed in multiple sequential versions of your software:

```
ALTER TABLE customers ADD COLUMN correct VARCHAR(20);
UPDATE customers SET correct = wrong
  WHERE id BETWEEN 1 AND 100;
UPDATE customers SET correct = wrong
  WHERE id BETWEEN 101 AND 200;
ALTER TABLE customers DELETE COLUMN wrong;
```

The first impression is that now you're going to have a lot of work even for some of the simplest database refactorings! It might *seem* like a lot of work, but it's work that is possible to automate. Luckily, we have software that can handle this for us, and all of the automated mechanisms will be executed within our software deployment pipeline.

Because we're never issuing any destructive statement, *you can always roll back to the previous version*. You can check application state after running a database migration, and if any data doesn't look right to you, you can always keep the current version instead of promoting the new one.

Avoid Locks by Using Sharding

Sharding in the context of databases is the process of splitting very large databases into smaller parts, or *shards*. As experience can tell us, some statements that we issue to our database can take a considerable amount of time to execute. During these statements' execution, the database becomes locked and unavailable for the application. This means that we are introducing a period of downtime to our users.

We can't control the amount of time that an `ALTER TABLE` statement is going to take. But at least on some of the most popular DBMSs available in the market, issuing an `ALTER TABLE ADD COLUMN` statement won't lead to locking. Regarding the `UPDATE` statements that we issue to our database during our migrations, we can definitely address the locking time.

It is probably safe to assume that the execution time for an `UPDATE` statement is directly proportional to the amount of data being updated and the number of rows in the table. The more rows and the more data that you choose to update in a single statement, the

longer it's going to take to execute. To minimize the lock time in each one of these statements, we must split our updates into smaller shards.

Suppose that our `Account` table has 1,000,000 rows and its `number` column is indexed and sequential to all rows in the table. A traditional `UPDATE` statement to increase the `amount` column by 10% would be as follows:

```
UPDATE Account SET amount = amount * 1.1;
```

Suppose that this statement is going to take 10 seconds, and that 10 seconds is not a reasonable amount of downtime for our users. However, two seconds might be acceptable. We could achieve this two-second downtime by splitting the dataset of the statement into five smaller shards.[3] Then we would have the following set of `UPDATE` statements:

```
UPDATE Account SET amount = amount * 1.1
  WHERE number BETWEEN 1 AND 200000;
UPDATE Account SET amount = amount * 1.1
  WHERE number BETWEEN 200001 AND 400000;
UPDATE Account SET amount = amount * 1.1
  WHERE number BETWEEN 400001 AND 600000;
UPDATE Account SET amount = amount * 1.1
  WHERE number BETWEEN 600001 AND 800000;
UPDATE Account SET amount = amount * 1.1
  WHERE number BETWEEN 800001 AND 1000000;
```

That's the reasoning behind using shards: minimize application downtime caused by database locking in `UPDATE` statements. You might argue that if there's any kind of locking, it's not real "zero" downtime. However the true purpose of zero downtime is to achieve zero *disruption* to our users. Your business scenario will dictate the maximum period of time that you can allow for database locking.

How can you *know* the amount of time that your `UPDATE` statements are going to take into production? The truth is that you can't. But we can make safer bets by constantly rehearsing the migrations that we release before going into production.

3 It might be an oversimplification for the execution time calculation, but it's a fair bet for instructional purposes.

Rehearse Your Migrations Up to Exhaustion

We cannot emphasize enough the fact that we must *rehearse* our migrations up to *exhaustion* in multiple steps of your software deployment pipeline. Migrations manipulate persistent data, and sometimes wrong statements can lead to catastrophic consequences in production environments.

Your Ops team will probably have a backup in hand just in case something happens, but that's a situation you want to avoid at all costs. First, it leads to application unavailability—which means downtime. Second, not all mistakes are detected early enough so that you can just replace your data with a backup. Sometimes it can take hours or days for you to realize that your data is in an inconsistent state, and by then it's already too late to just recover everything from the last backup.

Migration rehearsal should start in your own development machine and then be repeated multiple times in each one of your software deployment pipeline stages.

Check Your Data Between Migration Steps

We want to play on the safe side. Always. Even though we rehearsed our migrations up to exhaustion, we still want to check that we didn't blow anything up in production.

After each one of your releases, you should check if your application is behaving correctly. This includes not only checking it per se, but also checking the data in your database. Open your database's command-line interface (CLI), issue multiple `SELECT` statements, and ensure that everything is OK before proceeding to the next version.

Add a Column Migration

Adding a column is probably the simplest migration we can apply to our schema, and we'll start our zero downtime migrations journey with this. The following list is an overview of the needed steps:

`ALTER TABLE ADD COLUMN`
: Add the column to your table. Be aware to not add a `NOT NULL` constraint to your column at this step, even if your model

requires it, because it will break the `INSERT`/`UPDATE` statements from your current version—the current version still doesn't provide a value for this newly added column.

Code computes the read value and writes to new column
: Your new version should be writing to the new column, but it can't assume that a value will be present when reading from it. When your new version reads an absent value, you have the choice of either using a default value or computing an alternative value based on other information that you have in the application.

Update data using shards
: Issue `UPDATE` statements to assign values to the new column.

Code reads and writes from the new column
: Finally, use the new column for read and writes in your application.

`NOT NULL` Constraints

Any `NOT NULL` constraint must be applied only after a successful execution of all the migration steps. It can be the final step of any of the zero downtime migrations presented in this book.

Rename a Column Migration

Renaming a column requires more steps to successfully execute the migration because we already have data in our table and we need to migrate this information from one column to the other. Here is a list of these steps:

`ALTER TABLE ADD COLUMN`
: Add the column to your table. Be careful to not add a `NOT NULL` constraint to your column at this step, even if your model requires it, because it will break the `INSERT`/`UPDATE` statements from your current version—the current version still doesn't provide a value for this newly added column.

Code reads from the old column and writes to both
: Your new version will read values from the old column and write to both. This will guarantee that all new rows will have both columns populated with correct values.

Copy data using small shards
: Your current version is still reading values from the old column and guaranteeing that new rows are being added with both columns populated. However, you need to copy the data from the old column to the new column so that all of your current rows also have both columns populated.

Code reads from the new column and writes to both
: Your new version will read values from the new column but still keep writing to both old and new columns. This step is necessary to guarantee that if something goes wrong with your new version, the current version will still be able to work properly with the old column.

Code reads and writes from the new column
: After some period of monitoring and automated or manual checking of the values, you can release a new version of your code that finally reads and writes from your new column. At this point you have already successfully refactored your column name.

Delete the old column (later)
: Deleting a column is a destructive operation because data will be lost and no longer recoverable. You should never delete a column in a directly subsequent release of your code. You might realize that you made a mistake in the refactoring a few days or even weeks later. As a precaution, you should delete only after a quarantine period. Following the quarantine, you can use your routine maintenance period to delete all the columns in your database that are no longer used by your code. Of course, you should be using some tracking system to ensure that nothing is left behind, because we all know that human memory is less than ideally suited for that task.

Change Type/Format of a Column Migration

The migrations steps required for changing the type or format of a column are not different from those used in renaming a column. This is a good thing: it's easier to get used to something when we're repeating the same steps, no matter which kind of change we're making. The following list is an overview of the process:

`ALTER TABLE ADD COLUMN`
: Add the column to your table. Be careful to not add a `NOT NULL` constraint to your column at this step, even if your model requires it, because it will break the `INSERT`/`UPDATE` statements from your current version—the current version still doesn't provide a value for this newly added column.

Code reads from the old column and writes to both
: Your new version will read values from the old column and write to both. This will guarantee that all new rows will have both columns populated with correct values.

Copy data using small shards
: Your current version is still reading values from the old column and guaranteeing that new rows are being added with both columns populated. However, you need to copy the data from the old column to the new column so that all of your current rows also have both columns populated.

Code reads from the new column and writes to both
: Your new version will read values from the new column but still keep writing to both old and new columns. This step is necessary to guarantee that if something goes wrong with your new version, the current version will still be able to work properly with the old column.

Code reads and writes from the new column
: After some period of monitoring and automated or manual checking of the values, you can release a new version of your code that finally reads and writes from your new column. At this point you have already successfully refactored your column name.

Delete the old column (later)
: As pointed out earlier, deleting a column is a destructive operation because data will be lost and no longer recoverable. You should never delete a column in a directly subsequent release of your code. You might realize that you made a mistake in the refactoring a few days or even weeks later. As a precaution, you should delete only after a quarantine period. Following the quarantine, you can use your routine maintenance period to delete all the columns in your database that are no longer used by your code. Of course, you should be using some tracking sys-

tem to ensure that nothing is left behind, because we all know that human memory is less than ideally suited for that task.

Delete a Column Migration

Sometimes we just don't need a column anymore. You're already aware that deleting a column is a destructive operation, but sometimes we need to do it. We collected some steps to make this migration a bit safer:

DON'T
: *Never* delete a column in your database when you're releasing a new version. It's not the first time we mention it, but we can't emphasize this point enough. Deleting a column is a destructive operation, and you *always* want to guarantee that your current version will still be running smoothly in production even if your new release messes up something.

Stop using the read value but keep writing to the column
: Instead of deleting the column, you should just stop using the value in your code for read operations. However, write operations should still be writing to the column in case you need it to keep your current version running.

Stop writing to the column (optional)
: Keep the column with the data there for safety reasons. If you decide to execute this step, make sure to drop any `NOT NULL` constraint or else you will prevent your code from inserting new rows.

Delete the column (later)
: Quarantine your column to be deleted later. After the quarantine period, you can use your routine maintenance period to delete all the columns in your database that are no longer used by your code. Of course, you should be using some tracking system to ensure that nothing is left behind because…that's right, human memory is less than ideally suited for this task.

Referential Integrity Constraints

Referential integrity constraints are one of the features of DBMSs we become acquainted with. We use them because they make us feel comfortable. These constraints guarantee that you will never mistakenly insert a value that doesn't exist in the referenced column. It also prevents you from deleting a row whose column is referenced on the other side of the constraint.

If you have seconds thoughts about using referential integrity constraints, their use is not tied to business requirements for your running software. They act as safety nets, preventing your code from sustaining mistaken behavior and beginning to write uncorrelated data to your database. Because we are applying zero-downtime migrations to our schema and none of our migrations are destructive, you still have another safety net: you never lose any data, and you are always capable of deploying your previous version into production without the fear of losing or corrupting anything. In this sense, we believe that it is safe to simply drop your referential integrity constraints before applying your series of migrations. Later, when everything is deployed into production and you know that you are done with the refactorings, you will recreate the constraints using the new columns.[4]

4 It can sound terrifying to suggest disabling the safety net exactly when things are more likely to break, as this defeats the purpose of using a safety net. But then again, it's all about trade-offs. Sometimes you need to break some walls to make room for improvement.

CHAPTER 4
CRUD and CQRS

Two of the most popular patterns for dealing with data manipulation are Create, Read, Update, and Delete (CRUD) and Command and Query Responsibility Segregation (CQRS). Most developers are familiar with CRUD because the majority of the material and tools available try to support this pattern in one way or another. Any tool or framework that is promoted as a fast way to deliver your software to market provides some sort of scaffolding or dynamic generation of CRUD operations.

Things start to get blurry when we talk about CQRS. Certainly the subject of microservices will usually invoke CQRS in many different discussions between people at conferences and among members of development teams, but personal experience shows that we still have plenty of room for clarification. If you look for the term "CQRS" on a search engine, you will find many good definitions. But even after reading those, it might be difficult to grasp exactly the "why" or even the "how" of CQRS.

This chapter will try to present clear distinction between and motivation behind using both CRUD and CQRS patterns. And any discussion about CQRS won't be complete if we do not understand the different consistency models that are involved in distributed systems —how these systems handle read and write operations on the data state in different nodes. We'll start our explanation with these concepts.

Consistency Models

When we're talking about *consistency* in distributed systems, we are referring to the concept that you will have some data distributed in different nodes of your system, and each one of those might have a copy of your data. If it's a read-only dataset, any client connecting to any of the nodes will always receive the same data, so there is no consistency problem. When it comes to read-write datasets, some conflicts can arise. Each one of the nodes can update its own copy of the data, so if a client connects to different nodes in your system, it might receive different values for the same data.

The way that we deal with updates on different nodes and how we propagate the information between them leads to different consistency models. The description presented in the next sections about *eventual consistency* and *strong consistency* is an over-simplification of the concepts, but it should paint a sufficiently complete picture of them within the context of information integration between microservices and relational databases.

Eventual Consistency

Eventual consistency is a model in distributed computing that guarantees that given an update to a data item in your dataset, *eventually*, at a point in the future, all access to this data item in any node will return the same value. Because each one of the nodes can update its own copy of the data item, if two or more nodes modify the same data item, you will have a conflict. *Conflict resolution algorithms* are then required to achieve convergence.[1] One example of a conflict resolution algorithm is *last write wins*. If we are able to add a synchronized timestamp or counter to all of our updates, the last update always wins the conflict.

One special case for eventual consistency is when you have your data distributed in multiple nodes, but only one of them is allowed to make updates to the data. If one node is the canonical source of information for the updates, you won't have conflicts in the other nodes as long as they are able to apply the updates in the exact same order as the canonical information source. You add the possibility of

1 Convergence is the state in which all the nodes of the system have eventually achieved consistency.

write unavailability, but that's a bearable trade-off for some business use cases. We explore this subject in greater detail in "Event Sourcing" on page 56.

Strong Consistency

Strong consistency is a model that is most familiar to database developers, given that it resembles the traditional transaction model with its Atomicity, Consistency, Isolation, and Durability (ACID) properties. In this model, any update in any node requires that all nodes agree on the new value before making it visible for client reads. It sounds naively simple, but it also introduces the requirement of blocking all the nodes until they converge. It might be especially problematic depending on network latency and throughput.

Applicability

There are always exceptions to any rule, but eventual consistency tends to be favored for scenarios in which high throughput and availability are more important requirements than immediate consistency. Keep in mind that most real-world business use cases are already eventual consistency. When you read a web page or receive a spreadsheet or report through email, you are already looking at information as it was some seconds, minutes, or even hours ago. Eventually all information converges, but we're used to this eventuality in our lives. Shouldn't we also be used to it when developing our applications?

CRUD

CRUD architectures are certainly the most common architectures in traditional data manipulation applications. In this scenario, we use the same data model for both read and write operations, as shown in Figure 4-1. One of the key advantages of this architecture is its simplicity—you use the same common set of classes for all operations. Tools can easily generate the scaffolding code to automate its implementation.

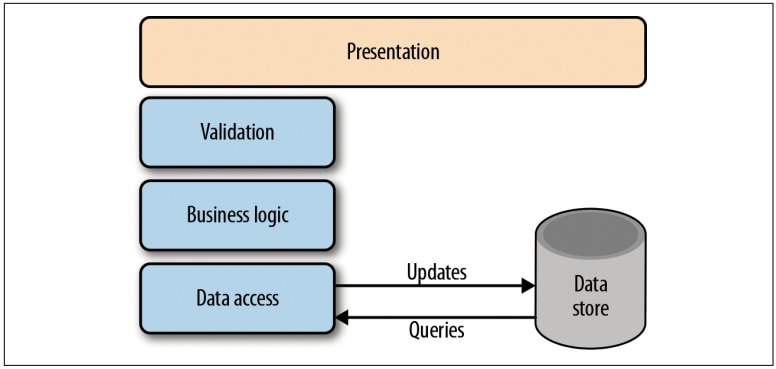


Figure 4-1. A traditional CRUD architecture (Source)

If we implemented a `Customer` class in our project and wanted to change the `Customer`'s name, we would retrieve a `Customer` entity from our data store, change the property, and then issue an update statement to our data store to have the representation persisted.

It's a very common practice nowadays to see a CRUD architecture being implemented through REST endpoints exposed over HTTP.

Probably the majority of basic data manipulation operations will be best served using a CRUD architecture, and in fact that's our recommended approach for that. However, when complexity begins to arise in your use cases, you might want to consider something different. One of the things missing in a CRUD architecture is *intent*.

Suppose that we wanted to change the `Customer`'s address. In a CRUD architecture, we just update the property and issue an update statement. We can't figure out if the change was motivated by an incorrect value or if the customer moved to another city. Maybe we have a business scenario in which we were required to trigger a notification to an external system in case of a relocation. In this case, a CQRS architecture might be a better fit.

CQRS

CQRS is a fancy name for an architecture that uses different data models to represent read and write operations.

Let's look again at the scenario of changing the `Customer`'s address. In a CQRS architecture (see Figure 4-2), we could model our write operations as `Commands`. In this case, we can implement a `WrongAd`

`dressUpdateCommand` and a `RelocationUpdateCommand`. Internally, both would be changing the `Customer` data store. But now that we know the intent of the update, we can fire the notification to the external system only in the `RelocationUpdateCommand` code.

For the read operations, we will be invoking some query methods on a read interface, such as `CustomerQueryService`, and we would be returning Data Transfer Objects (DTOs)[2] to be exposed to our presentation layer.

Things start to become even more interesting when we realize that we don't need to use the same data store for both models. We can use a simple CQRS architecture, such as the one represented in Figure 4-2, or we can split the read and write data stores into separate tables, schemas, or even database instances or technologies, as demonstrated in Figure 4-3. The possibility of using data stores in different nodes with different technologies sounds tempting for a microservices architecture.

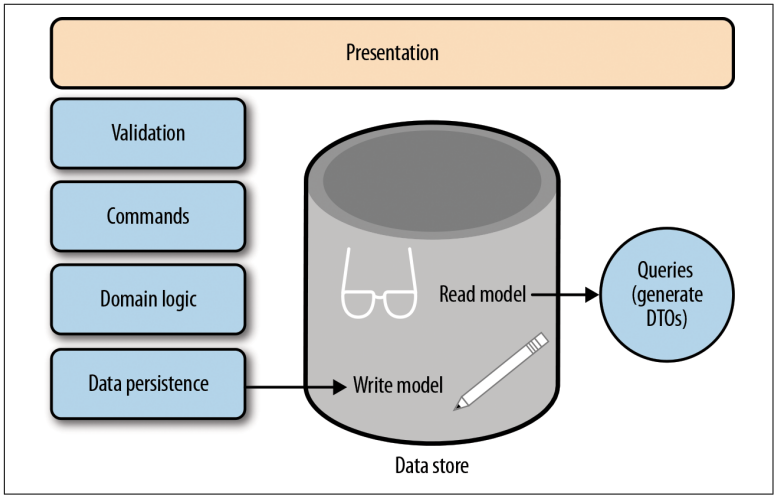


Figure 4-2. A basic CQRS architecture (Source)

2 We're using the DTO term here not in the sense of an anemic domain model, but to address its sole purpose of being the container of information.

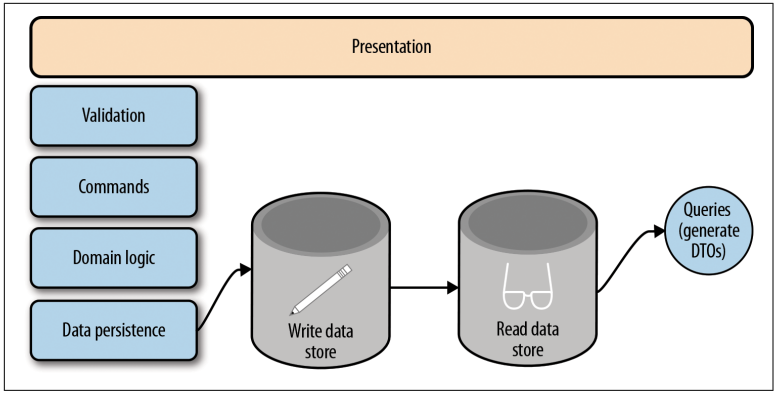


Figure 4-3. A CQRS architecture with separate read and write stores (Source)

Some motivations for using separate data stores for read and write operations are performance and distribution. Your write operations might generate a lot of contention on the data store. Or your read operations may be so intensive that the write operations degrade significantly. You also might need to consolidate the information of your model using information provided by other data stores. This can be time consuming and won't perform well if you try to update the read model together with your write model. You might want to consider doing that asynchronously. Your read operations could be implemented in a separate service (remember microservices?), so you would need to issue the update request to the read model in a remote data store.

NOTE

CQRS and Multiple Read Data Stores

It's not unusual for an application to use the same information in a number of different ways. Sometimes you need one subset of the information; at other times, you might need another subset of the information. Sometimes different features of your application consume different aggregate information from the same data store. Different requirements also allow you to have different consistency models between read data stores, even though the write data store might be the same.

When you design your application with CQRS, you're not tied to having a single read data store in your model. If you have different requirements for different features of your application, you can create more than one read data store, each one with a read model optimized for the specific use case being implemented.

Whenever we're issuing the update requests to another component of our system, we are creating an *event*. It's not a strict requirement, but when events come to play, that's the moment we strongly consider adding a *message broker* to our architecture. You could be storing the events in a separate table and keep polling for new records, but for most implementations, a message broker will be the wisest choice because it favors a decoupled architecture. And if you're using a Java Platform, Enterprise Edition (Java EE) environment, you already have that for free anyway.

Event Sourcing

Sometimes one thing leads to another, and now that we've raised the concept of events, we will also want to consider the concept of *event sourcing*.

Event sourcing is commonly used together with CQRS. Even though neither one implies the use of the other, they fit well together and complement each other in interesting ways. Traditional CRUD and CQRS architectures store only the current state of the data in the data stores. It's probably OK for most situations, but this approach also has its limitations:

- Using a single data store for both operations can limit scalability due to performance problems.
- In a concurrent system with multiple users, you might run into data update conflicts.
- Without an additional auditing mechanism, you have neither the history of updates nor its source.

NOTE

Other Auditing Approaches

Event sourcing is just one of the possible solutions when auditing is a requirement for your application. Some database patterns implemented by your application or through triggers can keep an audit trail of what has changed and when, creating some sort of event sourcing in the sense that the change events are recorded.

You can add triggers to the database in order to create these change events, or you can use an open source tool like Hibernate Envers to achieve that if you're already using Hibernate in your project. With Hibernate Envers, you even have the added benefit of built-in query capabilities for you versioned entities.

To solve this limitation in event sourcing, we model the state of the data as a sequence of events. Each one of these events is stored in an append-only data store. In this case, the canonical source of information is the sequence of events, not a single entity stored in the data store.

We'll use the classic example of the amount of money stored in a bank account. Without using event sourcing, a `credit()` operation to the amount of a bank account would need to do the following:

1. Query the bank account to get the current value
2. Add the supplied amount to the current value of the entity
3. Issue an update to the data store with the new value

All of the preceding operations would probably be executed inside a single method within the scope of a single transaction.

With event sourcing, there's no single entity responsible for the amount of money stored in the bank account. Instead of updating the bank account directly, we would need to create another entity that represents the `credit()` operation. We can call it `CreditOperation` and it contains the value to be credited. Then we would be required to do the following:

1. Create a `CreditOperation` with the amount to be credited
2. Issue an insert to the data store with the `CreditOperation`

Assuming that all bank accounts start with a zero balance, it's just a matter of sequentially applying all the `credit()` and `debit()` operations to compute the current amount of money. You probably already noticed that this is not a computational-intensive operation if the number of operations is small, but the process tends to become very slow as the size of the dataset grows. That's when CQRS comes to the assistance of event sourcing.

With CQRS and event sourcing, you can store the `credit()` and `debit()` operations (the write operations) in a data store and then store the consolidated current amount of money in another data store (for the read operations). The canonical source of information will still be the set of `credit()` and `debit()` operations, but the read data store is created for performance reasons. You can update the read data store synchronously or asynchronously. In a synchronous operation, you can achieve strong or eventual consistency; in an asynchronous operation, you will always have eventual consistency. There are many different strategies for populating the read data store, which we cover in Chapter 5.

Notice that when you combine CQRS and event sourcing you get auditing for free: in any given moment of time, you can replay all the `credit()` and `debit()` operations to check whether the amount of money in the read data store is correct. You also get a free time machine: you can check the state of your bank account at any given moment in the past.

Synchronous or Asynchronous Updates?

Synchronously updating the read model sounds like the obvious choice at first glance but in fact, it turns out that in the real world asynchronous updates are generally more flexible and powerful, and they have added benefits. Let's take another look at the banking example.

Real banking systems update the read models asynchronously because they have a lot more control with their procedures and policies. Banks record operations as they occur, but they reconcile those operations at night and often in multiple passes. For example, reconciliation often first applies all credit operations and then all debit operations, which eliminates any improper ordering in the actual recording of the transactions due to technical reasons (like a store or ATM couldn't post its transactions for a few hours) or user error (like depositing money into an account after an expensive purchase to prevent overdrawing the account).

CHAPTER 5
Integration Strategies

Using the *Decentralized Data Management* characteristic of microservices architectures, each one of our microservices should have its own separate database—which could possibly, again, be a relational database or not. However, a legacy monolithic relational database is very unlikely to simply migrate the tables and the data from your current schema to a new separate schema or database instance, or even a new database technology.

We want to evolve our architecture as smoothly as possible: it requires baby steps and carefully considered migrations in each one of these steps to minimize disruption and, consequently, downtime. Moreover, a microservice is not an island; it requires information provided by other services, and also provides information required by other services. Therefore, we need to integrate the data between at least two separate services: one can be your old monolithic application and the other one will be your new microservice artifact.

In this chapter, we will present a set of different integration strategies collected from personal experience and from successful integrations delivered by companies worldwide. Most of these strategies assume that you will be also using a relational database in your new microservice. And you should not be surprised with this choice of technology: relational databases are a solid, mature, battle-tested technology. I would even suggest it as your first choice when dealing with breaking monolithic database into a microservices database. We can play on the safe side and we will have many more integration options for the journey between your current legacy, mono-

lithic, tightly coupled, and entangled database to your decoupled microservices database. Later on, when you have already successfully decoupled the data from each one of the endpoints, you will be free to explore and use another technology, which might be a better fit for your specific use case.

The following sections will present a brief description and set of considerations when using each one of these integration strategies:

- Shared Tables
- Database View
- Database Materialized View
- Database Trigger
- Transactional Code
- ETL Tools
- Data Virtualization
- Event Sourcing
- Change Data Capture

Some of these strategies require features that might or might not be implemented in your current choice of database management system. We'll leave you to check the features and restrictions imposed by each product and version.

Shared Tables

Shared tables is a database integration technique that makes two or more artifacts in your system communicate through reading and writing to the same set of tables in a shared database. This certainly sounds like a bad idea at first. And you are probably right. Even in the end, it probably will still be a bad idea. We can consider this to be in the quick-and-dirty category of solutions, but we can't discard it completely due to its popularity. It has been used for a long time and is probably also the most common integration strategy used when you need to integrate different applications and artifacts that require the same information.

Sam Newman did a great job explaining the downsides of this approach in his book *Building Microservices*. We'll list some of them later in this section.

Shared Tables Applicability

Shared tables strategy (or technique) is suitable for very simple cases and is the fastest to implement an integration approach. Sometimes your project schedule makes you consider adding some technical debt in order to deliver value into production in time. If you're using shared tables consciously for a quick hack and then plan to pay this debt later, you'll be greatly served by this integration strategy before you can plan and implement a more robust strategy.

Shared Tables Considerations

Here is a list of some of the elements of shared tablets that you should consider:

Fastest data integration
: Shared tables is by far the most common form of data integration because it is probably the fastest and easiest to implement. Just put everything inside the same database schema!

Strong consistency
: All of the data will be modified in the same database schema using transactions. You'll achieve strong consistency semantics.

Low cohesion and high coupling
: Remember some desirable properties behind good software, such as high cohesion and low coupling? With shared tables you have none. Everything is accessible and modifiable by all the artifacts sharing the data. You control neither behavior nor the semantics. Schema migrations tend to become so difficult that they will be used as an excuse for not changing at all.

Database View

Database views are a concept that can be interpreted in at least two different ways. The first interpretation is that a view is just a `Result Set` for a stored `Query`.[1] The second interpretation is that a view is a logical representation of one or more tables—these are called *base tables*. You can use views to provide a subset or superset of the data that is stored in tables.

1 Represented by a `SELECT` statement.

Database View Applicability

A database view is a better approach than shared tables for simple cases because it allows you to create another representation of your model suited to your specific artifact and use case. It can help to reduce the coupling between the artifacts so that later you can more easily choose a more robust integration strategy.

You might not be able to use this strategy if you have write requirements and your DBMS implementation doesn't support it, or if the query backing your view is too costly to be run in the frequency required by your artifact.

Database View Considerations

Here are some things to keep in mind with respect to database views:

Easiest strategy to implement
: You can create database views via a simple `CREATE VIEW` statement in which you just need to supply a single `SELECT` statement. We can see the `CREATE VIEW` statement as a way for telling the DBMS to store the supplied query to be run later, as it is required.

Largest support from DBMS vendors
: As of this writing, even embeddable database engines such as H2 or SQLite support the `CREATE VIEW` statement, so it is safe to consider that all DBMSs used by enterprise applications have this feature.

Possible performance issues
: Performance issues might arise, depending on the stored query and the frequency of view accesses. It's a common performance optimization scenario in which you'll need to apply your standard query optimization techniques to achieve the desired performance.

Strong consistency
: All operations on a view are executed against the base tables, so when we're updating any row on a database view, in fact, you're applying these statements to the underlying base tables. You'll be using the standard transactions and Atomicity, Consistency,

Isolation, and Durability (ACID) behavior provided by your DBMS to ensure strong consistency between your models.

One database must be reachable by the other
: The schema on which you are creating the database view must be able to read the base tables. Maybe you are using a different schema inside a common database instance. You might also be able to use tables on remote database instances, provided that your DBMS has this feature.[2]

Updatable depending on DBMS support
: Traditionally, database views were read-only structures against which you issued your `SELECT` statements. Again, depending on your DBMS, you might be able to issue update statements against your views. Some requirements for this to work on your DBMS might include that your view have all the primary keys of the base tables and that you must reference all of them on your update statement being executed on the database view.

Database Materialized View

A database materialized view is a database object that stores the results of a query. It's a tool that is usually only familiar to database administrators (DBAs) because most developers tend to specialize in coding. From the developer's perspective, there is no difference between a materialized view and a view or a table, because you can query both in the same way that you have always been issuing your `SELECT` statements. Indeed, some DBMSs implement materialized views as true physical tables. Their structure only differs from tables in the sense that they are used as replication storage for other local or even remote tables using a synchronization or *snapshotting* mechanism.

The data synchronization between the *master tables*[3] and the materialized view can usually be triggered on demand, based on an interval timer by a transaction commit.

2 Like Oracle's `DBLink` feature.

3 The tables that are the source of the information being replicated.

Database Materialized View Applicability

Database materialized views are probably already used in most enterprise applications for aggregation and reporting features as cached read data stores. This fact makes it the perfect candidate when the DBAs in your organization are already familiar with this tool and willing to collaborate on new tools. Database materialized views have the benefits of a plain database view without the performance implications. It's usually a better alternative when you have multiple `JOIN`s or aggregations and you can deal with eventual consistency.

Database Materialized View Considerations

What follows is a synopsis of database materialized views:

Better performance
: Database materialized views are often implemented as true physical tables, so data is already stored in the same format as the query (and often in a denormalized state). Because you don't have any joins to execute, performance can improve significantly. And you also have the possibility of optimizing your database materialized views with indexes for your queries.

Strong or eventual consistency
: If you're able to configure your DBMS to update your materialized views in each commit inside the same transaction, you'll achieve strong consistency. Or you'll have eventual consistency when updating your materialized view on demand or with an interval timer trigger.

One database must be reachable by the other
: The source of information for materialized views are the base tables, so they must be reachable by your schema for them to be created successfully. The base tables can be local or remote databases, and the reachability depends on DBMS features.

Updatable depending on DBMS support
: Your DBMS might support updatable materialized views. Some restrictions might apply for this feature to be available, and you might need to build your materialized view with all the primary keys of the base tables that you want to update.

Database Trigger

Database triggers are code that is automatically executed in response to database events. Some common database events in which we might be interested are `AFTER INSERT`, `AFTER UPDATE`, and `AFTER DELETE`, but there are many more events we can respond to. We can use code within a trigger to update the integrated tables in a very similar way to that of a database materialized view.

Database Trigger Applicability

Database triggers are good candidates if the type of data being integrated is simple and if you already have the legacy of maintaining other database triggers in your application. They will quickly become impractical if your integration requires multiple `JOIN`s or aggregations.

We don't advise adding triggers as an integration strategy to your application if you're not already using them for other purposes.

Database Trigger Considerations

Here are some things to consider regarding database triggers:

Depends on DBMS support
: Even though triggers are a well-known feature, not all available DMBSs support them as of this writing.

Strong consistency
: Trigger code is executed within the same transaction of the source table update, so you'll always achieve strong consistency semantics.

One database must be reachable by the other
: Trigger code is executed against default database structures, so they must be reachable by your schema for them to be created successfully. The structures can be local or remote, and the reachability depends on DBMS features.

Transactional Code

Integration can always be implemented in our own source code instead of relying on software provided by third parties. In the same way that a database trigger or a materialized view can update our

target tables in response to an event, we can code this logic in our update code.

Sometimes the business logic resides in a database stored procedure: in this case, the code is not much different from the code that would be implemented in a trigger. We just need to ensure that everything is run within the same transaction to guarantee data integrity.

If we are using a platform such as Java to code our business logic, we can achieve similar results using distributed transactions to guarantee data integrity.

Transactional Code Applicability

Using transactional code for data integration is much more difficult than expected. It might be feasible for some simple use cases, but it can quickly become impractical if more than two artifacts need to share the same data. The synchronous requirement and network issues like latency and unavailability also minimize the applicability of this strategy.

Transactional Code Considerations

Here are some things to keep in mind about transactional code:

Any code: usually stored procedures or distributed transactions
: If you're not relying on database views or materialized views, you will probably be dealing with stored procedures or other technology that supports distributed transactions to guarantee data integrity.

Strong consistency
: The usage of transactions with ACID semantics guarantees that you will have strong consistency in both ends of the integration.

Possible cohesion/coupling issues
: Any technology can be used correctly or incorrectly, but experience shows that most of the times when we're distributing transactional code between different artifacts, we're also coupling both sides with a shared domain model. This can lead to a maintenance nightmare known as *cascading changes*. Any change to your code in one artifact leads to a change in the other artifact. Cascading changes in a microservices architecture also leads to another anti-pattern called *synchronized relea-*

ses, as the artifacts usually will only work properly with specific versions on each side.

Possible performance issues
: Transactions with ACID semantics on distributed endpoints might require fine-tuning optimizations to not affect operational performance.

Updatable depending on how it is implemented
: Because it's just code, you can implement bidirectional updates on both artifacts if your requirements demand it. There are no restrictions on the technology side of this integration strategy.

Extract, Transform, and Load Tools

Extract, Transform, and Load (ETL) tools are popular at the management level, where they are used to generate business reports and sometimes dashboards that provide consolidated strategic information for decision making. That's one of the reasons why many of these tools are marketed as *business intelligence* tools.

Some examples of open source implementations are Pentaho and Dashbuilder. Most of the available tools will allow you to use plain old SQL to extract the information you want from different tables and schemas of your database transform and consolidate this information, and then load them to the result tables. Based on the result tables, you can later generate spreadsheet or HTML reports, or even present this information in a beautiful visual representation such as the one depicted in Figure 5-1.

In the ETL cycle you *extract* your information from a data source (likely to be your production database), *transform* your information to the format that you desire (aggregating the results or correlating them), and finally, you *load* the information on a target data source. The target data source usually contains highly denormalized information to be consumed by reporting engines.

ETL Tools Applicability

ETL tools are a good candidate when you are already using them for other purposes. ETL can handle more complex scenarios with multiple `JOIN`s and aggregations, and it's feasible if the latency of the eventual consistency is bearable for the business use case.

If the long-term plan is to support more integrations between other systems and microservices, alternative approaches such as event sourcing, data virtualization, or change data capture might be better solutions.

Figure 5-1. Dashbuilder panel displaying consolidated information (Source)

ETL Tools Considerations

Here are some ETL tools considerations:

Many available tools
: Dozens of open source projects and commercially supported products make ETL tools the most widely available integration strategy. The options for solutions can range from free to very expensive. Success on ETL tools usage depends much more on the implementation project than the tool, per se, even though vendor-specific features might help this process.

Requires external trigger (usually time-based)
: The ETL cycle can take a long time to execute, so in most cases it's unusual to keep it running continuously. ETL tools provide mechanisms to trigger the start of the ETL cycle on demand or through time-based triggers with `cron` semantics.

Can aggregate from multiple data sources
: You are not restricted to a single data source. The most common use case will be the aggregation of information from multiple schemas in the same database instance through SQL queries.

Eventual consistency
: Because the update of the information on your ETL tool is usually triggered on demand or on a time-based schedule, the information is potentially always outdated. You can achieve eventual consistency only with this integration strategy. It's also worth noting that even though the information is outdated, it can be consistent if the reading of the information was all done in the same transaction (not all products support this feature).

Read-only integration
: The ETL process is not designed to allow updates to the data source. Information flows only one way to the target data source.

Data Virtualization

Data virtualization is a strategy that allows you to create an abstraction layer over different data sources. The data source types can be as heterogeneous as flat files, relational databases, and nonrelational databases.

With a data virtualization platform, you can create a Virtual Database (VDB) that provides real-time access to data in multiple heterogeneous data sources. Unlike ETL tools that copy the data into a different data structure, VDBs access the original data sources in real time and do not need to make copies of any data.

You can also create multiple VDBs with different data models on top of the same data sources. For example, each client application (which again might be a microservice) might want its own VDB with data structured specifically for what that client needs. Data virtualization is a powerful tool for an integration scenario in which you have multiple artifacts consuming the same data in different ways.

The VDB abstraction also allows you to create multiple VDBs with different data models from the same data sources. It's a powerful tool in an integration scenario in which you have multiple different artifacts consuming the same data but in different ways.

One open source data virtualization platform is Teiid. Figure 5-2 illustrates Teiid's architecture, but it is also a good representation of the general concept of VDBs in a data virtualization platform.

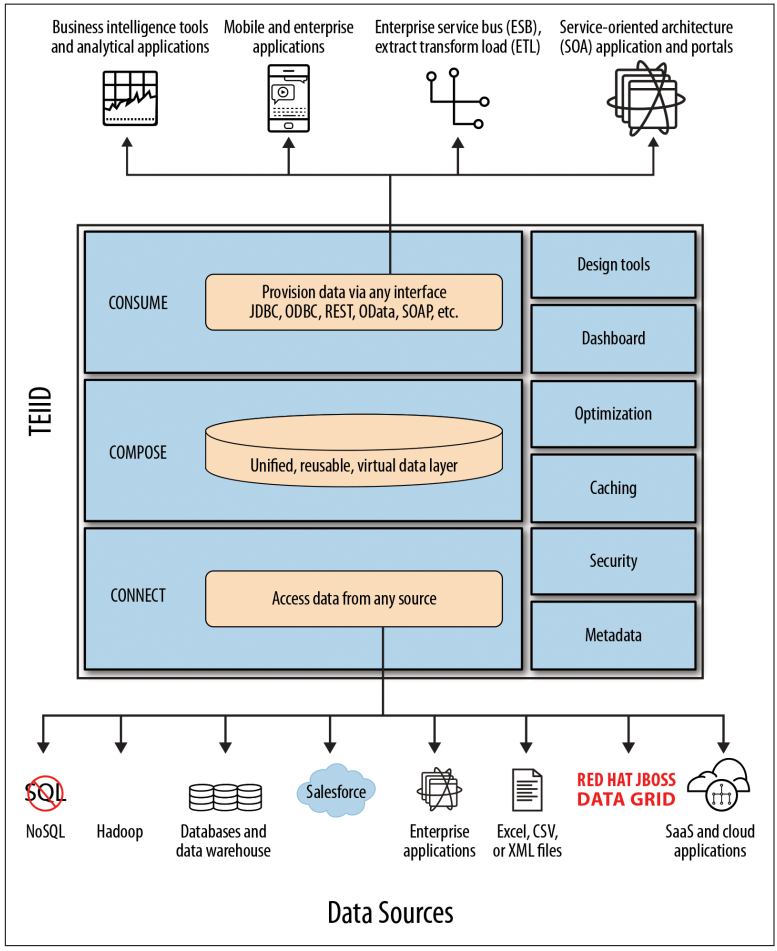


Figure 5-2. Teiid's data virtualization architecture (Source)

Data Virtualization Applicability

Data virtualization is the most flexible integration strategy covered in this book. It allows you to create VDBs from any data source and to accommodate any model that your application requires. It's also a convenient strategy for developers because it allows them to use their existing and traditional development skills to craft an artifact that will simply be accessing a database (in this case, a virtual one).

If you're not ready to jump on the event sourcing bandwagon, or your business use case does not require that you use events, data virtualization is probably the best integration strategy for your artifact. It can handle very complex scenarios with `JOIN`s and aggregations, even with different database instances or technologies.

It's also very useful for monolithic applications, and it can provide the same benefits of database materialized views for your application, including caching. If you first decide to use data virtualization in your monolith, the path to future microservices will also be easier —even when you decide to use a different database technology.

Data Virtualization Considerations

Here are some factors to consider regarding data virtualization:

Real-time access option
: Because you're not moving the data from the data sources and you have direct access to them, the information available on a VDB is available with real-time access.[4] Keep in mind that even with the access being online, you still might have performance issues depending on the tuning, scale, and type of the data sources. The real-time access property doesn't hold true when your VDB is defined to materialize and/or cache data. One of the reasons for doing that is when the data source is too expensive to process. In this case, your client application can still access the materialized/cached data anytime without hitting the actual data sources.

Strong or eventual consistency
: If you're directly accessing the data sources in real time, there is no data replication: you achieve strong consistency with this strategy. On the other hand, if you choose to define your VDB with materialized or cached data, you will achieve only eventual consistency.

Can aggregate from multiple datasources
: Data aggregation from multiple heterogeneus data sources is one of the premises of data virtualization platforms.

4 Note that *real-time access* here means that information is consumed online, not in the sense of systems with strictly defined response times as real-time systems.

Updatable depending on data virtualization platform
: Depending on the features of your data virtualization platform, your VDB might provide a read-write data source for you to consume.

Event Sourcing

We covered this in "Event Sourcing" on page 39, and it is of special interest in the scope of distributed systems such as microservices architectures—particularly the pattern of using event sourcing and Command Query Responsibility Segregation (CQRS) with different read and write data stores. If we're able to model our write data store as a stream of events, we can use a *message bus* to propagate them. The message bus clients can then consume these messages and build their own read data store to be used as a local replica.

Event Sourcing Applicability

Event sourcing is a good candidate for your integration technique given some requirements:

- You already have experience with event sourcing
- Your domain model is already modeled as events
- You're already using a message broker to distributed messages through the system
- You're already using asynchronous processing in some routines of your application
- Your domain model is consolidated enough to not expect new events in the foreseeable future

If you need to modify your existing application to fit most if not all of the aforementioned requirements, it's probably safer to choose another integration strategy instead of event sourcing.

Event Sourcing Is Harder Than It Seems

It's never enough to warn any developer that software architecture is all about trade-offs. There's no such thing as a free lunch in this area. Event sourcing might seems like the best approach for decoupling and scaling your system, but it certainly also has its drawbacks.

Properly designing event sourcing is *hard*. More often than not, you'll find badly modeled event sourcing, which leads to increased coupling instead of the desired low coupling. The central point of integration in an event-sourcing architecture are the *events*. You'll probably have more than one type of event in your system. Each one of these events carry information, and this information has a type composed of a schema (it has a structure to hold the data) and a meaning (semantics). This type is the coupling between the systems. Adding a new event potentially means a cascade of changes to the consumers.

Event Sourcing Considerations

Here are some issues about event sourcing you might want to consider:

State of data is a stream of events
: If the state of the write data store is already modeled as a stream of events, it becomes even simpler to get these same events and propagate them throughout our distributed system via a message bus.

Eases auditing
: Because the current state of the data is the result of applying the event stream one after another, we can easily check the correctness of the model by reapplying all the events in the same order since the initial state. It also allows easy reconstruction of the read model based on the same principle.

Eventual consistency
: Distributing the events through a message bus means that the events are going to be processed asynchronously. This leads to eventual consistency semantics.

Usually combined with a message bus
: Events are naturally modeled as messages propagated and consumed through a message bus.

High scalability
: The asynchronous nature of a message bus makes this strategy highly scalable. We don't need to handle throttling because the message consumers can handle the messages at their own pace. It eliminates the possibility of a producer overwhelming the consumer by sending a high volume of messages in a short period of time.

Change Data Capture

Another approach is to use Change Data Capture (CDC), which is a data integration strategy that captures the individual changes being committed to a data source and makes them available as a sequence of events. We might even consider CDC to be a *poor man's event sourcing with CQRS*. In this approach, the read data store is updated through a stream of events as in event sourcing, but the write data store is not a stream of events.

It lacks some of the characteristics of true event sourcing, as covered in "Event Sourcing" on page 39, but most of the CDC tools offer pluggable mechanisms that prevent you from changing the domain model or code of your connected application. It's an especially valuable feature when dealing with legacy systems because you don't want to mess with the old tables and code, which could have been written in an ancient language or platform. Not having to change either the source code or the database makes CDC one of the first and most likely candidates for you to get started in breaking your monolith into smaller parts.

CDC Applicability

If your DBMS is supported by the CDC tool, this is the least intrusive integration strategy available. You don't need to change the structure of your existing data or your legacy code. And because the CDC events are already modeled as change events such as Create, Update, and Delete, it's unlikely that you'll need to implement newer types of events later—minimizing coupling. This is our favorite integration strategy when dealing with legacy monolithic applications for nontrivial use cases.

CDC Considerations

Keep the following in mind when implementing CDC:

Read data source is updated through a stream of events
: Just as in "Event Sourcing" on page 56, we'll be consuming the update events in our read data stores, but without the need to change our domain model to be represented as a stream of events. It's one the best recommended approaches when dealing with legacy systems.

Eventual consistency
: Distributing the events through a message bus means that the events are going to be processed asynchronously. This leads to eventual consistency semantics.

Usually combined with a message bus
: Events are naturally modeled as messages propagated and consumed through a message bus.

High scalability
: The asynchronous nature of a message bus makes this strategy highly scalable. We don't need to handle throttling because the message consumers can handle the messages at their own pace. It eliminates the possibility of a producer overwhelming the consumer by sending a high volume of messages in a short period of time.

Debezium

Debezium is a new open source project that implements CDC. As of this writing, it supports pluggable connectors for MySQL and MongoDB for version 0.3; PostgreSQL support is coming for version 0.4. Designed to persist and distribute the stream of events to CDC clients, i's built on top of well-known and popular technologies such as Apache Kafka to persist and distribute the stream of events to CDC clients.

Debezium fits very well in data replication scenarios such as those used in microservices architectures. You can plug the Debezium connector into your current database, configure it to listen for changes in a set of tables, and then stream it to a Kafka topic.

Debezium messages have an extensive amount of information, including the structure of the data, the new state of the data that was

modified, and whenever possible, the prior state of the data before it was modified. If we're watching the stream for changes to a "Customers" table, we might see a message payload that contains the information shown in Example 5-1 when an `UPDATE` statement is committed to change the value of the `first_name` field from `"Anne"` to `"Anne Marie"` for row with `id` 1004.

Example 5-1. Payload information in a Debezium message

```
{
  "payload": {
    "before": {
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": {
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "name": "dbserver1",
      "server_id": 223344,
      "ts_sec": 1480505,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 1086,
      "row": 0,
      "snapshot": null
    },
    "op": "u",
    "ts_ms": 1480505875755
  }
}
```

It's not hard to imagine that we can consume this message and populate our local read data store in our microservice as a local replica. The information broadcast in the message can be huge depending on the data that was changed, but the best approach is to process and store only the information that is relevant to your microservice.

About the Author

Edson Yanaga, Red Hat's director of developer experience, is a Java Champion and a Microsoft MVP. He is also a published author and a frequent speaker at international conferences, where he discusses Java, microservices, cloud computing, DevOps, and software craftsmanship.

Yanaga considers himself a software craftsman and is convinced that we can all create a better world for people with better software. His life's purpose is to deliver and help developers worldwide to deliver better software, faster and more safely—and he feels lucky to also call that a job!