With examples in Java

Microservices Patterns

Chris Richardson

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Microservices Patterns
With examples in Java
Version 7

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Microservices Patterns: With examples in Java*. I am very excited to see the first few chapters be released and I am looking forward to completing the book. This is an intermediate level book designed for enterprise developers and architects who want to adopt the microservice architecture.

I don't believe in blindly advocating for any particular technology. The microservice architecture is not a silver bullet. It has both benefits and drawbacks. Moreover, there are numerous issues that you must address when using the microservice architecture. This book captures this philosophy. Most of the content is organized around patterns, which are a great way to describe the trade-offs of using a particular approach.

I am initially releasing the first two chapters. Chapter 1 envisions the state of software development at Food to Go, Inc., which is the fictitious company from my first book *POJOs in Action*. After ten years they are in what I call monolithic hell. All aspects of software development and deployment are slow and painful. Sadly, the odds are high that you are in a similar situation. In this chapter, you will learn how to escape monolithic hell. I describe the microservice architecture, it's benefits and drawbacks. You will learn about the microservices pattern language, which is a collection of patterns that solve the problems that you face when using the microservice architecture.

Chapter 2 describes the key decision that you must make when using the microservice architecture: how to compose an application into a set of services. You will learn about the important of software architecture. I describe how the microservice architecture is what is known as an architectural style. You will learn about two main decomposition strategies.

Chapter 3 looks at how inter-process communication (IPC) plays much more critical role in a microservice architecture than it does in a monolithic application. You will learn about the various IPC options including messaging and REST.I describe why asynchronous messaging is preferred approach. You will learn how to send messages as part of a database transaction and why it's important.

Looking ahead, later chapters dig deeper into the microservice architecture. I'll describe key architectural issues including inter-process communication and transaction management in the microservice architecture. The latter is especially challenging since each service has its own database and traditional distributed transaction are not a viable

option for modern applications. After that I will cover numerous other topics including deployment, testing and monitoring patterns. You will also learn how to refactor an existing monolithic application into a microservice architecture.

As you are reading *Microservices patterns*, I hope you'll take advantage of the Author Online forum. I will be reading your comments and responding. I appreciate any feedback, as it will help me write a better book.

Thanks again!

— Chris Richardson

brief contents

1 Escaping monolithic hell

This chapter covers:

- Describes what is monolithic hell and how to escape it by adopting the microservice architecture
- Defines the microservice architecture as an architectural style
- Explains the benefits and drawbacks of microservices
- Describes the microservices pattern language and why you should use it
- Discusses why modern successful software development of large, complex applications requires three things: the microservice architecture; DevOps; small, autonomous teams

It was only Monday lunchtime but Mary, the CTO of Food to Go, Inc (FTGO) was already feeling frustrated. Her day had started off really well. She had spent the previous week with other software architects and developers at an excellent conference learning about the latest software development techniques including continuous deployment and the microservice architecture. The conference had left her feeling empowered and eager to improve how FTGO developed software.

Unfortunately, that feeling had quickly evaporated. She had just spent the first morning back in the office in yet another painful meeting with senior engineering and business people. They had spent two hours discussing why the development team was going to miss another critical release date. Sadly, this kind of meeting had becoming increasingly common over the past few years. Despite adopting agile, the pace of development was slowing down, making it next to impossible to meet the business's goals. And, to make matters worse there didn't seem to be a simple solution.

The conference had made Mary realize that FTGO was suffering from a case of monolithic hell and that the cure was to adopt the microservice architecture. However,

the microservice architecture and the associated state of the art software development practices described at the conference felt like an elusive dream. It was not clear to Mary how she could fight today's fires while simultaneously improving how software was developed at FTGO. Fortunately, as you will learn by reading this book there is a way. But first, let's look at the problems that FTGO is facing and how they got there.

1.1 About FTGO

Since its launch in late 2005, FTGO had grown by leaps and bounds. Today, it was one of the leading online food delivery companies in the US. The business even plans to expand overseas although those plans are in jeopardy because of delays in implementing the necessary features.

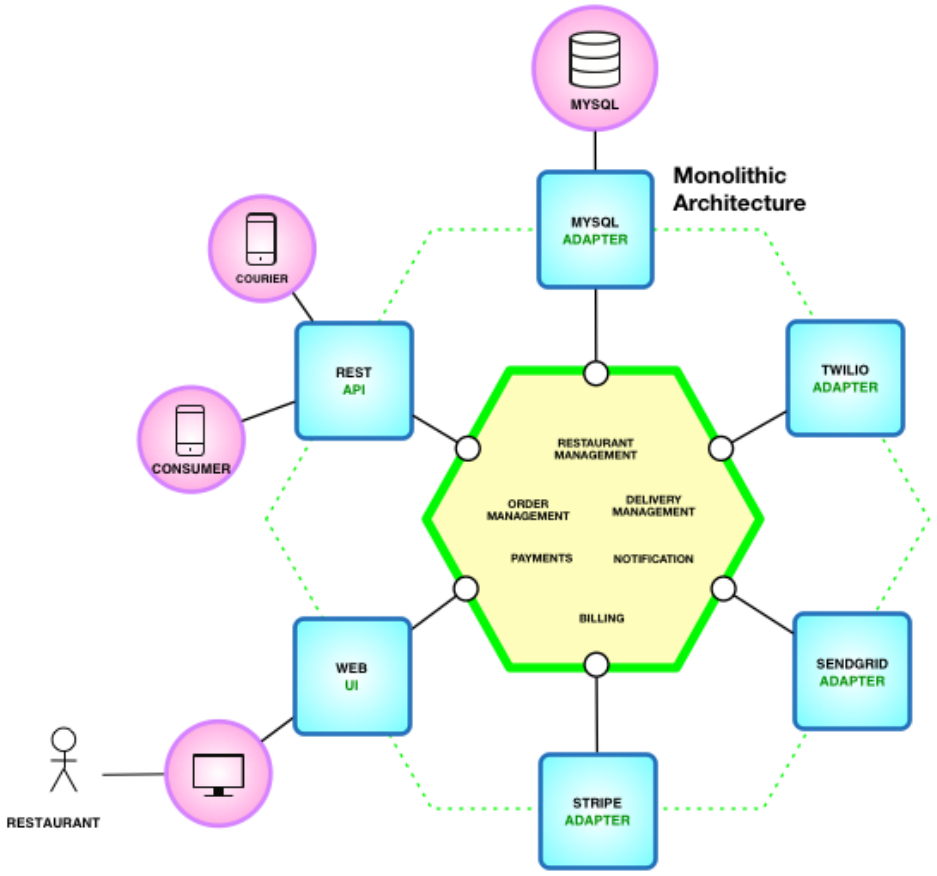
1.1.1 What the FTGO application does

At its core, the FTGO application is quite simple. Consumers use the FTGO website or mobile application to place food orders at local restaurants. FTGO coordinates a network of couriers who deliver the orders. It is also responsible for paying couriers and restaurants. Restaurants use the FTGO website to edit their menus and manage orders. The application uses various web services including Stripe for payments, Twilio for messaging, and Amazon SES for email.

1.1.2 The FTGO architecture

FTGO is a typical enterprise Java application and has a layered, modular architecture. The core of the application are the business logic components. Surrounding the business logic are various adapters that implemented UIs and integrate with external systems. Figure 1.1 shows the application's architecture.

Figure 1.1. The FTGO currently has a monolithic architecture



The business logic consists of modules that are comprised of services and domain objects. Examples of the modules include `Order Management`, `Delivery Management`, `Billing` and `Payments`. There are several adapters that interface with the external systems including database access components, messaging components, web applications and REST APIs.

Despite having a logically modular architecture, the FTGO application is packaged as a single WAR file and deployed on Tomcat. It is an example of the widely used **monolithic** style of software architecture, which structures a system as a single executable or deployable component. If the FTGO application was written in GoLang, it would be a single executable. A Ruby or NodeJS version of the application would be a single directory hierarchy of source code.

1.1.3 The benefits of the monolithic architecture

In the early days of FTGO, the application's monolithic architecture had lots of benefits. The application was simple to develop. IDEs and other developer tools are

focused on building a single application. The FTGO application was also relatively straightforward to test. The developers wrote end-to-end tests that simply launched the application and tested UI with Selenium.

Deploying the FTGO application was also straightforward. All a developer had to do was copy the WAR file to a server that had Tomcat installed. Scaling was also easy. FTGO ran multiple instances of the application behind a load balancer.

1.1.4 Monolithic hell

Unfortunately, as the FTGO developers have discovered, the monolithic architecture has a huge limitation. Successful applications, like the FTGO application, have a habit of outgrowing the monolithic architecture. Each sprint, the FTGO development team implements a few more stories, which, of course, makes the code base larger. Moreover, as the company became more successful, the size of the development team steadily grew. Not only did this increase the growth rate of code base but it also increased the management overhead.

The once small, simple FTGO application, which was developed by small team, grew over the past 10 years into a monstrous monolith developed by a large team. As a result of outgrowing its architecture, FTGO is in monolithic hell. Development is slow and painful. Agile development and deployment is impossible. Lets look at the reasons.

1.1.5 Overwhelming complexity intimidates developers

A major problem with the FTGO application is that it is too complex. It is simply too large for any developer to fully understand. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming. Deadlines are missed.

To make matters worse, this overwhelming complexity tends to be a downwards spiral. If the codebase is difficult to understand then a developer won't make changes correctly. Each change makes the codebase incrementally more complex, and more difficult to understand. The clean, modular architecture shown earlier in figure 1.1 doesn't reflect reality. FTGO is gradually becoming a monstrous, incomprehensible big ball of mud.

Mary remembers recently attending a conference where she met a developer who was writing a tool to analyze the dependencies between the thousands of JARs in their multi-million LOC application. At the time, that tool seemed like something FTGO could use. Now she is not so sure.

1.1.6 Slow day to day development

As well having to fight overwhelming complexity, FTGO developers find day to day development tasks slow. The large application overloads and slows down a developer's IDE. Building the FTGO application takes a long time. Moreover, because it is so large, the application takes a long time to startup. As a result, the edit-build-run-test loop takes a long time, which badly impacts productivity.

1.1.7 An obstacle to agile development and deployment

Another problem with the FTGO application is that deploying changes into production is a long and painful process. The team deploys typically updates production once a month, usually late Friday or Saturday night. Mary keeps reading that the state of the art for SaaS applications is continuous deployment: deploying changes to production many times a day. Apparently, as of 2011 Amazon.com deployed a change into production every 11.6 seconds without ever impacting the user! For the FTGO developers, updating production more than once a month seems like a distance dream. And adopting continuous deployment is next to impossible.

FTGO has partially adopted agile. The engineering team is divided up into squads and uses two week sprints. Unfortunately, the journey from code complete to running in production is long and arduous. One problem with so many developers committing to the same code base is that the build is frequently in an unreleasable state. When the FTGO developers attempted to solve this problem by using feature branches that resulted in lengthy, painful merges. Consequently, once a team completes their sprint, there is a long period of testing and code stabilization.

Another reason it takes so long to get changes into production is that testing takes a long time. Because the code base is so complex and the impact of a change is not well understood, developers and the CI server must run the entire test suite. There are even some parts of the system that require manual testing. It also takes a while to diagnose and fix the cause of a test failure. As a result, it takes a couple of days to complete a testing cycle.

1.1.8 Scaling the application can be challenging

The FTGO team also has problems scaling their application. That is because different application modules have conflicting resource requirements. The restaurant data, for example, is stored in an large in-memory database, which ideally deployed on servers with lots of memory. In contrast, the image processing module, is CPU intensive and best deployed on servers with lots of CPU. Since these modules are part of the same application, FTGO must compromise on the server configuration.

1.1.9 Reliability

Another problem with the FTGO application is reliability. Because all modules are running within the same process, a bug in one module sometimes causes the entire application to crash. Every so often, for example, a memory leak in a relatively unimportant module crashes all instances of the application one by one. The FTGO developers don't enjoy being paged in the middle of the night because of a production outage.

1.1.10 Requires long-term commitment to a technology stack

The final aspect of monolithic hell experienced by the FTGO team is that the architecture forces them to use a single technology stack. The monolithic architecture makes it difficult to adopt new frameworks, and languages. It is extremely expensive

and risky to rewrite the entire monolithic application to use a new and presumably better technology. Consequently, developers are stuck with the technology choices they made at the start of the project. Sometimes that means maintaining an application written using an increasingly obsolete technology stack.

The Spring framework has continued to evolve while being backwards compatible so in theory FTGO might have been able to upgrade. Unfortunately, the FTGO application uses versions of frameworks that are incompatible with newer versions of Spring. The development team has never found the time to upgrade those frameworks. As a result, major parts of the application are written using increasingly out of date frameworks. What's more, the FTGO developers would like to experiment with non-JVM languages such as GoLang and NodeJS. Sadly, this is not possible with a monolithic application.

1.2 The microservice architecture to the rescue

FTGO is in trouble because the application has outgrown its monolithic architecture. The monolithic architecture worked well initially. But as the application became grew the development team encountered numerous issues. They have slowly marched towards and arrived at monolithic hell. All aspects of development are slow and painful.

Interestingly, software architecture has very little to do with functional requirements. You can implement a set of use cases - an application's functional requirements - with any architecture, even a big ball of mud. Architecture matters because of how it affects the so-called non-functional requirements, aka. quality attributes or "-ilities". As the FTGO application grew, various quality attributes have suffered, most notably those that impact the velocity of software delivery: maintainability, extensibility, and testability.

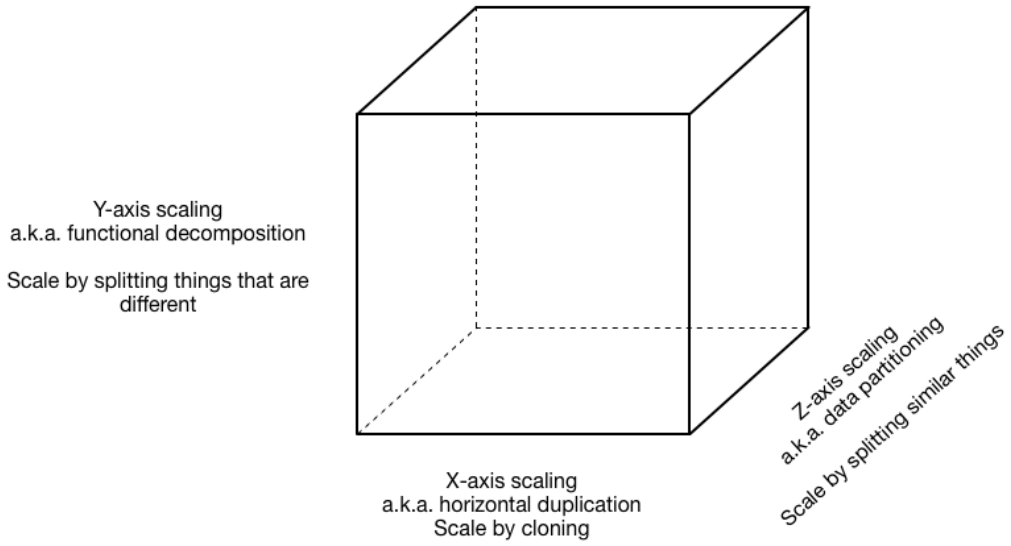
On the one hand, a disciplined team can slow down the pace of their descent towards monolithic hell. They can work hard to maintain the modularity of their application. They can write comprehensive automated tests. But on the other hand, they cannot avoid the issues of a large team working on a single monolithic application. Nor can they solve the problem of an increasingly obsolete technology stack. The best that a team can do is delay the inevitable. In order to escape monolithic hell they must migrate to a new architecture: the Microservice architecture.

Today, there is growing consensus that if you are building a large, complex application then you should consider using the microservice architecture. But what are microservices exactly? Unfortunately, the name doesn't help since it over emphasizes size. There are numerous definitions of the microservice architecture. Some take the name too literally and claim that a service should be tiny, e.g. 100 LOC. Others claim that a service should be two weeks of work. Adrian Cockcroft, formerly of Netflix, defines a microservice architecture as a service-oriented architecture composed of loosely coupled elements that have bounded contexts. That is not bad a definition but it is a little dense. Lets see if we can do better.

1.2.1 Scale cube and microservices

My definition of the microservice architecture is inspired by the excellent book **The art of scalability**. This book describes a really useful, three dimension scalability model: the scale cube, which is shown in Figure 1.2.

Figure 1.2. The scale cube defines three separate ways to scale an application.

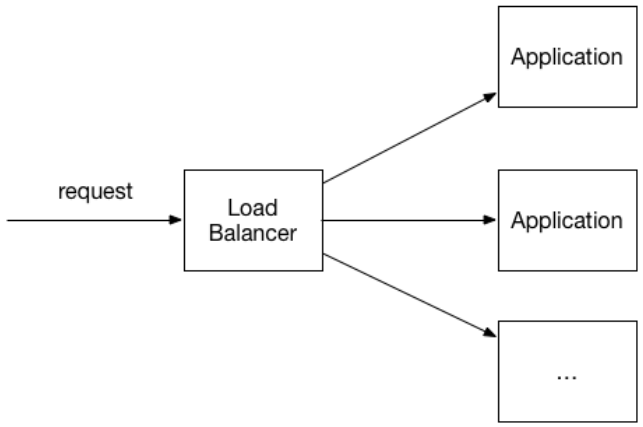


The model defines three ways to scale an application: X, Y and Z.

X-axis scaling

X-axis scaling is a commonly used way to scale an application. You simply run multiple instances of the application behind a load balancer. Figure 1.3 shows how X-axis scaling works.

Figure 1.3. X-axis scaling runs multiple, identical instances behind a load balancer

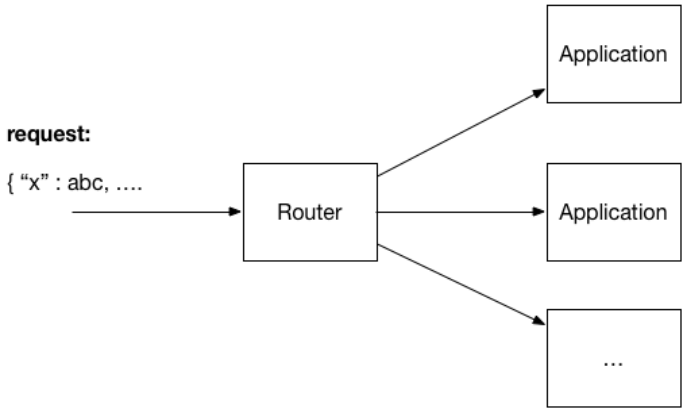


The load balancer distributes requests amongst the N identical instances of the application. This is a great way of improving the capacity and the availability of an application.

Z-axis scaling

Z-axis scaling also runs multiple instances of the application. However, unlike X-axis scaling, each server is responsible for only a subset of the data. The router in front of the instances uses an attribute of request to route it to the appropriate instance. An application might, for example, route requests using the *userId*. Figure 1.4 shows how this works.

Figure 1.4. Z-axis scaling runs multiple identical instances behind a router

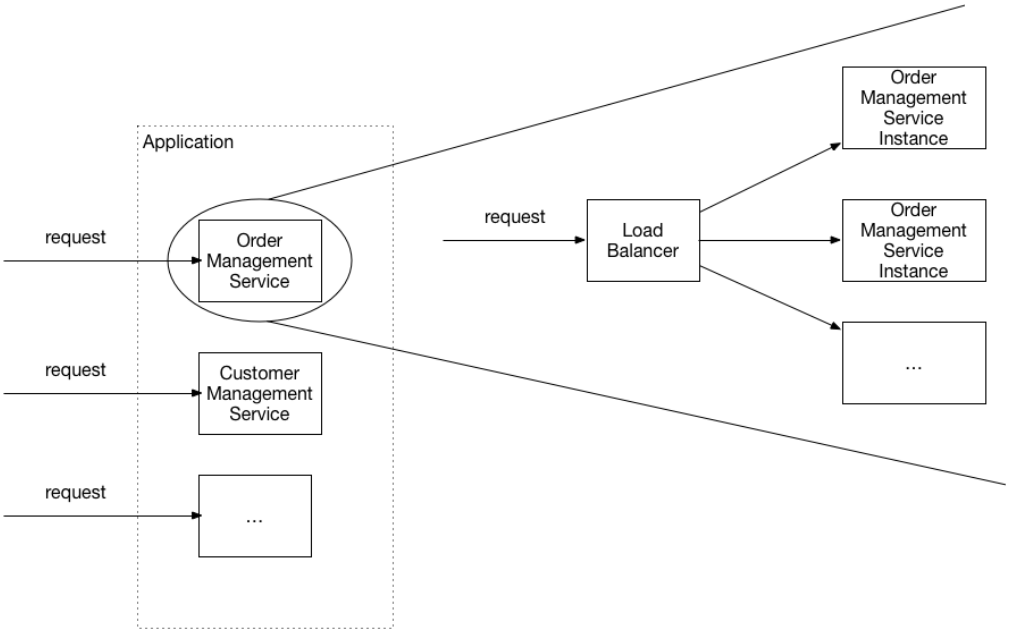


In this example, each application instance is responsible a subset of the users. The router uses the *userId* specified by the request `Authorization` header to select one of the N identical instances of the application.

Y-axis scaling

X and Z-axis scaling improve the application's capacity and availability. Neither approach, however, solves the problem of increasing development and application complexity. To solve those problems you need to apply Y-axis scaling a.k.a. functional decomposition. Y-axis scaling splits a monolithic application into a set of services. Figure 1.5 shows how this works.

Figure 1.5. Y-axis scaling splits the application into a set of services



A service is a mini-application that implements narrowed focussed functionality such as order management, customer management etc. A service is scaled using X-axis scaling. Some services might also use Z-axis scaling. For example, the `Order Service` consists of a set of load-balanced service instances.

This is my high-level definition of microservices: an architectural style that functionally decomposes an application into a set of services. Note that this definition does not say anything about size. Instead, what matters is that each service has a focussed, cohesive set of responsibilities. Later in this book I discuss what this really means. But now, let's look at how the microservice architecture is a form of modularity.

1.2.2 Microservices as a form of modularity

Modularity is essential when developing large, complex applications. A modern applications such as FTGO is too large to be developed by an individual. It is also too complex to be understood by a single person. Applications must be decomposed into

modules that are developed and understood by different people. In a monolithic application modules are defined using a combination of programming language constructs, such as Java packages, and build artifacts, such as Java JAR files. However, as the FTGO developers have discovered this approach tends not to work well in practice. Long lived, monolithic applications usually degenerate into big balls of mud.

The microservice architecture uses services as the unit of modularity. A service has an impermeable boundary that is difficult to violate. As a result, the modularity of the application is much easier to preserve over time. There are other benefits of the microservice architecture including the ability to deploy and scale services independently.

1.2.3 Each service has its own database

A key characteristic of the microservice architecture is that the services are loosely coupled. One way to achieve loose coupling is by each service having its own datastore. In the online store, for example, the `OrderService` has a database that includes the `ORDERS` table and the `CustomerService` has its database, which includes the `CUSTOMERS` table. At development time, a developer can change their service's schema without having to coordinate with developers working on other service. At runtime, the services are isolated from each other. One service will never be blocked because another service holds a database lock, for example.

Don't panic: this doesn't make Larry Ellison even richer!

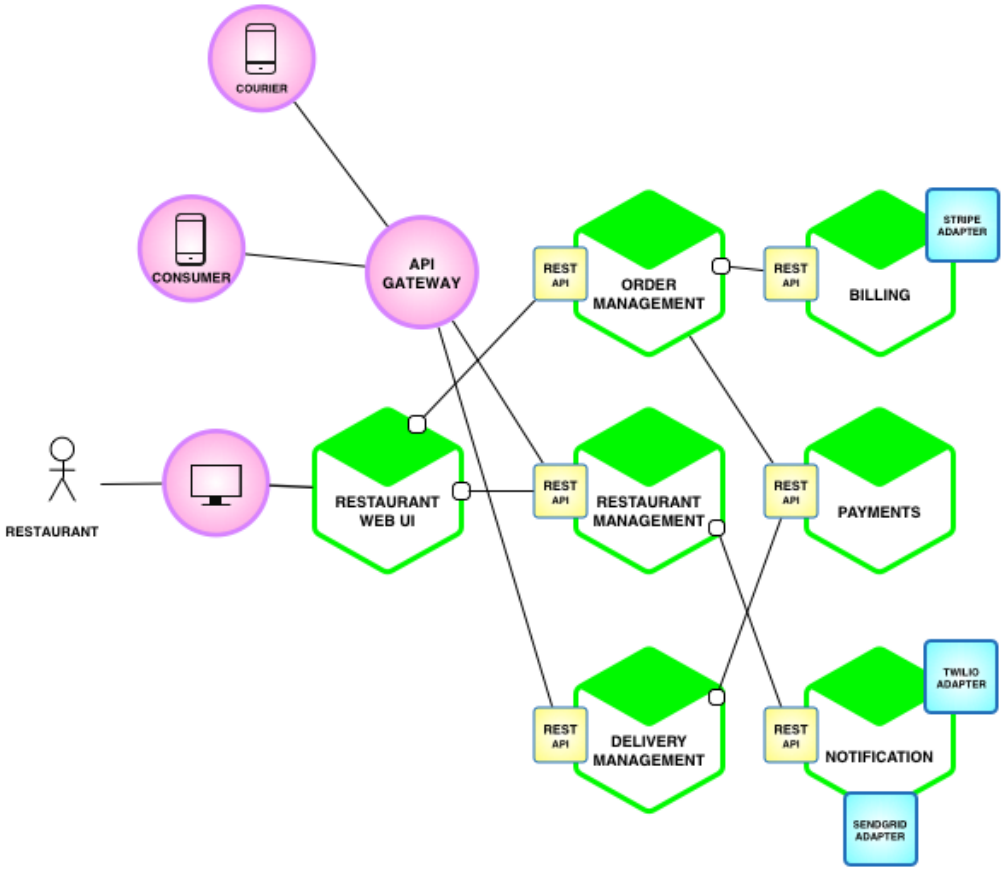
The requirement for each service to have its own database does not mean that it has its own database server. You do not, for example, have to spend 10x more on Oracle RDBMS licenses. In chapter {chapter-decomposition}, we will explore this topic in depth.

Now that we have defined the microservice architecture and described some of its essential characteristics, lets look how this applies to the FTGO application.

1.2.4 The FTGO microservice architecture

If we apply Y-axis decomposition to the FTGO application we get the architecture shown in figure 1.6. The decomposed application consists of numerous front-end and backend services.

Figure 1.6. The microservice architecture-based version of the FTGO application



The front-end services include an API gateway and the Restaurant Web UI. The API gateway, which plays the role of a facade and is described in detail in chapter {chapter-api-gateway}, provides the REST APIs that are used by the consumers' and couriers' mobile applications. The Restaurant Web UI implements the web interface that is used by the restaurants to manage menus and process orders.

The FTGO application's business logic consists of numerous backend services. Each backend service has a REST API and its own private datastore. The backend services include:

- Order service - manages orders
- Delivery service - manages delivery of orders from restaurants to consumers
- Restaurant service - maintains information about restaurants
- Restaurant order service - manages the preparation of orders
- Accounting service - handles billing and payments

Many services correspond to the modules that I described earlier this chapter. What's different is that each service and its API is very clearly defined. Each one can be independently developed, tested, deployed and scaled. Also, this architecture does a better job of preserving modularity. A developer cannot bypass a service's API and access its internal components. In chapter {chapter-refactoring} I describe how to transform an existing monolithic application into microservices.

1.2.5 Isn't the microservice architecture the same as SOA?

Some critics of the microservice architecture claim that it is nothing new and that it is just SOA. At a very high-level, there are some similarities. SOA and the microservice architecture are architectural styles that structure a system as a set of services. But once you dig deep you encounter significant differences.

SOA and the microservice architecture usually use different technology stacks. SOA applications typically use heavyweight technologies such as SOAP and other WS* standards. They often use a ESB, which is a 'smart pipe' that contain business and messaging processing logic, to integrate the services. Applications built using the microservice architecture tend to use lightweight, open-source technologies. The services communicate via 'dumb pipes', such as a message broker or lightweight protocols such as REST or gRPC.

SOA and the microservice architecture also differ in how they treat data. SOA applications typically have a global data model and share databases. In contrast, as mentioned earlier, in the microservice architecture each service has its own database. Moreover, as I describe in chapter {chapter-decomposition}, each service is usually considered to have its own domain model.

Another key difference between SOA and the microservice architecture is the size of the services. SOA is typically used to integrate large, complex monolithic applications. While services in a microservice architecture are not always tiny they are almost always much smaller. As a result, a SOA application will usually consist of a few large services where is a microservices-based application will consist of 10s or 100s of smaller services.

1.3 Benefits and drawbacks of the microservice architecture

The microservice architecture has both benefits and drawbacks. Below I describe the drawbacks but first let's consider the benefits.

1.3.1 Benefits of the microservice architecture

The microservice architecture has the following benefits:

- Each service is a small, maintainable application
- Services are independently deployable
- Services are independently scalable
- The microservice architecture enables teams to be autonomous

- Easily experiment with and adopt new technologies
- Improved fault isolation

Let's look at each benefit.

Each service is a small, maintainable application

Each service is relatively small. The code is easier for a developer to understand. The small code base doesn't slow down the IDE making developers more productive. Also, each service typically starts a lot faster than a large monolith, which again makes developers more productive, and speeds up deployments

Services are independently deployable

Each service can be deployed independently of other services. If the developers responsible for a service need to deploy a change that's local to that service they do not need to coordinate with other developers. They can simply deploy their changes. A microservice architecture makes continuous deployment feasible.

Services are independently scalable

Each service can be scaled independently of other services using X-axis cloning and Z-axis partitioning. Moreover, each service can be deployed on hardware that is best suited to its resource requirements. This is quite different than when using a monolithic architecture where components with wildly different resource requirements – e.g. CPU intensive vs. memory intensive – must be deployed together.

The microservice architecture enables teams to be autonomous

You can organize the development effort around multiple, small (e.g. two pizza) autonomous, teams. Each team is solely responsible for the development and deployment of one or more related services. Each team can develop, deploy and scale their services independently of all of the other teams.

Improved fault isolation

The microservice architecture also improves fault isolation. For example, a memory leak in one service only affects that service. Other services will continue to handle requests normally. In comparison, one misbehaving component of a monolithic architecture will bring down the entire system.

Easily experiment with and adopt new technologies

Last but not least, the microservice architecture eliminates any long-term commitment to a technology stack. In principle, when developing a new service the developers are free to pick whatever language and frameworks are best suited for that service. Of course, in many organizations it makes sense to restrict the choices but the key point is that you aren't constrained by past decisions.

Moreover, because the services are small, it becomes practical to rewrite them using better languages and technologies. It also means that if the trial of a new technology

fails you can throw away that work without risking the entire project. This is quite different than when using a monolithic architecture, where your initial technology choices severely constrain your ability to use different languages and frameworks in the future.

1.3.2 The drawbacks of the microservice architecture

Of course, no technology is a silver bullet, and the microservice architecture has a number of significant drawbacks and issues. Indeed most of this book is about how to address the drawbacks and issues of the microservice architecture. As you read about the challenges don't worry - later on, I describe ways to address them.

The drawbacks and issues of the microservice architecture are:

- Finding the right set of services is challenging
- Distributed systems are complex
- Deploying features that span multiple services requires careful coordination
- Deciding when to adopt the microservice architecture is difficult

Let's look at each one.

Finding the right set of services is challenging

One challenge with using the microservice architecture is that there isn't a concrete, well-defined algorithm for decomposing a system into services. Like much of software development, it is some what of an art. To make matter's worse, if you decompose a system incorrectly you will build a distributed monolith, a system consisting of coupled services that must be deployed together. It has the drawbacks of both the monolithic architecture and the microservice architecture.

Distributed systems are complex

Another challenge with using the microservice architecture is that developers must deal with the additional complexity of creating a distributed system. Developers must use an inter-process communication mechanism. Implementing use cases that span multiple services requires the use of unfamiliar techniques. IDEs and other development tools are focused on building monolithic applications and don't provide explicit support for developing distributed applications. Writing automated tests that involve multiple services is challenging. These are all issues that are specific to the microservice architecture. Consequently, your organization's developers must have sophisticated software development and delivery skills in order to successfully use microservices.

The microservice architecture also introduces significant operational complexity. There are many more moving parts – multiple instances of different types of service – that must be managed in production. To successfully deploy microservices you need a high-level of automation. You must use technologies such as:

- Automated deployment tooling such as Netflix Spinnaker

- An off the shelf PaaS such as Pivotal Cloud Foundry or Redhat Openshift
- A Docker orchestration platform such as Docker Swarm or Kubernetes

I describe the deployment options more detail in chapter {chapter-production}.

Deploying features that span multiple services requires careful coordination

Another challenge with using the microservice architecture is that deploying features that span multiple services requires careful coordination between the various development teams. You have to create a rollout plan that orders service deployments based on the dependencies between services. That's quite different than when using a monolithic architecture where you can easily deploy updates to multiple components atomically.

Deciding when to adopt the microservice architecture

Another challenge with using the microservice architecture is deciding at what point during the lifecycle of the application you should use this architecture. When developing the first version of an application, you often do not have the problems that this architecture solves. Moreover, using an elaborate, distributed architecture will slow down development.

This can be a major dilemma for startups whose biggest challenge is usually how to rapidly evolve the business model and accompanying application. Using the microservice architecture makes it much more difficult to iterate rapidly. A startup should almost certainly begin with a monolithic application.

Later on, however, when the challenge is how to handle complexity then it makes sense to functionally decompose the application a set of microservices. However, you might find refactoring difficult because of tangled dependencies. Later in chapter {chapter-refactoring}, I describe strategies for refactoring a monolithic application into microservices.

As you can see, the microservice architecture has many benefits but it is also has some significant drawbacks. Because of these issues, adopting a microservice architecture should not be undertaken lightly. However, for complex applications, such as a consumer-facing web application or SaaS application, it is usually the right choice. Well known sites such as eBay [PDF], Amazon.com, Groupon, and Gilt have all evolved from a monolithic architecture to a microservice architecture.

You must address numerous design and architectural issues when using the microservice architecture. What's more, many of these issues have multiple solutions, each with a different set of trade-offs. There is no one single perfect solution. In order to guide your decision making I've created the Microservice Architecture pattern language. I reference this pattern language throughout the rest of the book as I teach you about the microservice architecture. Let's look at what is a pattern language and why it is helpful.

1.4 The microservice architecture pattern language

Architecture and design is all about making decisions. You need to decide whether the monolithic or microservice architecture is the best fit for your application. And then if you pick the microservice architecture, there are lots of issues that you need to address. When making these decisions there are lots of tradeoffs to consider.

A good way to describe the various architectural and design options and improve decision making is to use a pattern language. Let's first look at why we need patterns and a pattern language. After that we will take a tour of the microservice architecture pattern language.

1.4.1 Microservices are not a silver bullet

Back in 1986, Fred Brooks, author of The Mythical Man-Month, said that in software engineering, there are no silver bullets. In other words, there are no techniques or technologies that if you adopted would give you a 10X boost in productivity. Yet 30 years later, developers are still arguing passionately about their favorite silver bullets, absolutely convinced that their favorite technology will give them a massive boost in productivity.

A lot of arguments follow the Suck/Rock dichotomy[1], which is a term coined by Neal Ford. They often have this structure: If you do X then a puppy will die so, therefore, you must do Y. For example, synchronous vs. reactive programming, object-oriented vs. functional, Java vs. JavaScript, REST vs messaging. Of course, reality is much more nuanced. No technology is a silver bullet. Every technology has drawbacks and limitations, which are often overlooked by its advocates. As a result, the adoption of a technology usually follows the Gartner hype cycle.

Microservices are not immune to the silver bullet phenomenon. Whether this architecture is appropriate for your application depends on a lot of factors. Consequently, it is bad advice to say to always use the microservice architecture. But conversely, it is equally bad advice to say never use them. Like many things, it depends!

The underlying reason for these polarized and hyped arguments about technology is that humans are primarily driven by their emotions. Jonathan Haidt in his excellent book The Righteous Mind: Why Good People Are Divided by Politics and Religion uses the metaphor of an elephant and its rider to describe how the human mind works. The elephant represents the emotion part of the human brain. It makes most of the decisions. The rider represents the rational part of the brain. It can sometimes influence the elephant but it mostly provides justifications for the elephant's decisions.

We—as in the software development community—need to overcome our emotional nature and find a better way of discussing and applying technology. A great way to discuss and describe technology is to use the pattern format.

[1] nealford.com/memeagora/2009/08/05/suck-rock-dichotomy.html

1.4.2 What is a pattern?

A pattern is a "reusable solution to a problem that occurs in particular context". This is already a major improvement because it introduces the idea of a context. Thinking about the context of a problem is important because, for example, a solution that solves the problem at the scale of Netflix might not be the best approach for an application with fewer users.

The value of patterns goes far beyond requiring you to consider the context of a problem. They force you to consider other critical yet frequently overlooked aspects of a solution. A commonly used pattern structure includes three especially valuable sections: forces, resulting context, and related patterns.

Forces

The forces section describes the issues that you must address when solving a problem in a given context. Forces can conflict so it might not be possible to solve all them. Which forces are more important depends on the context. You have to prioritize solving some forces over others. For example, code must be easy to understand and have good performance. Code written in a reactive style has better performance than synchronous code, yet is often more difficult to understand. Explicitly listing the forces is useful because it makes it clear what issues need to be solved.

Resulting context

The resulting context section describes the consequences of applying the pattern. It consists of three parts:

- benefits - the benefits of the pattern including the forces that have been resolved
- drawbacks - the drawbacks of the pattern including the unresolved forces
- issues - the new problems that have been introduced by applying the pattern.

The resulting context provides a more complete and less biased view of the solution, which enables better design decisions.

Related patterns

The related patterns section describes the relationship between this pattern and other patterns. There are five types of relationships between patterns

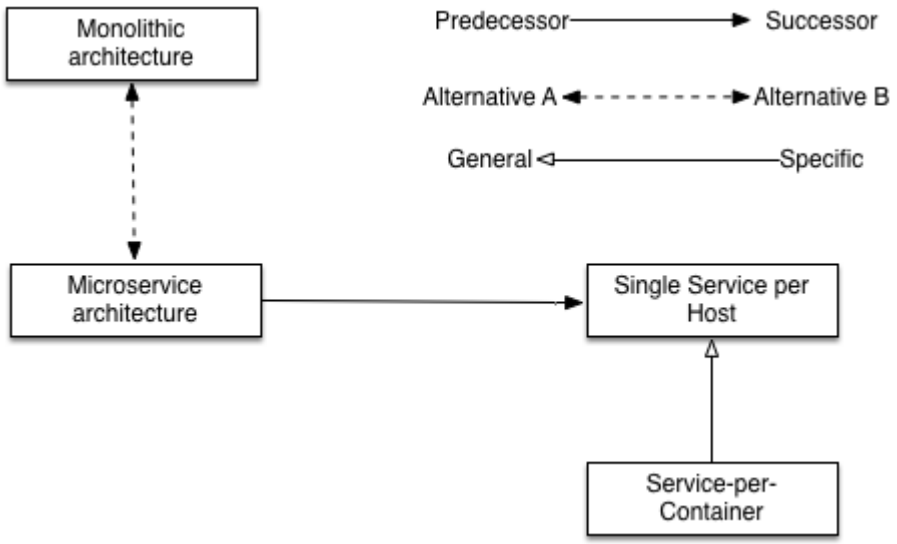
- Predecessor - a predecessor pattern is a pattern that motivates the need for this pattern. For example, the Microservice Architecture pattern is the predecessor to the rest of the patterns in the pattern language except the monolithic architecture pattern.
- Successor - a pattern that solves an issue that is introduced by this pattern. For example, if you apply the Microservice Architecture pattern you must then apply numerous successor patterns including service discovery patterns and the Circuit Breaker pattern.
- Alternative - a pattern that provides an alternative solution to this pattern. For

example, the Monolithic Architecture pattern and the Microservice Architecture pattern are alternative ways of architecting an application. You pick one or the other.

- Generalization - a pattern that is general solution to a problem. For example, in chapter {chapter-production} you will learn about the One service per host pattern, which has a few different implementation.
- Specialization - a specialized form of a particular pattern For example, in chapter {chapter-production} you will learn that the Container per service pattern is a specialization of the One service per host

The explicit description of related patterns provides valuable guidance on how to effectively solve a particular problem. Figure 1.7 shows how the relationships between patterns is visually represented.

Figure 1.7. The relationships between patterns



The different kinds of relationships between patterns are represented as follows:

- Predecessor ——▶ Successor - represents the predecessor-successor relationship.
- Alternative A ◀- - - - - - - - ▶ Alternative B - connects patterns that are alternative solutions to the same problem.
- General ◁—— Specific - indicates that one pattern is a specialization of another pattern.

Later in this chapter you will see many more examples of patterns and relationships between them.

Patterns form a pattern language

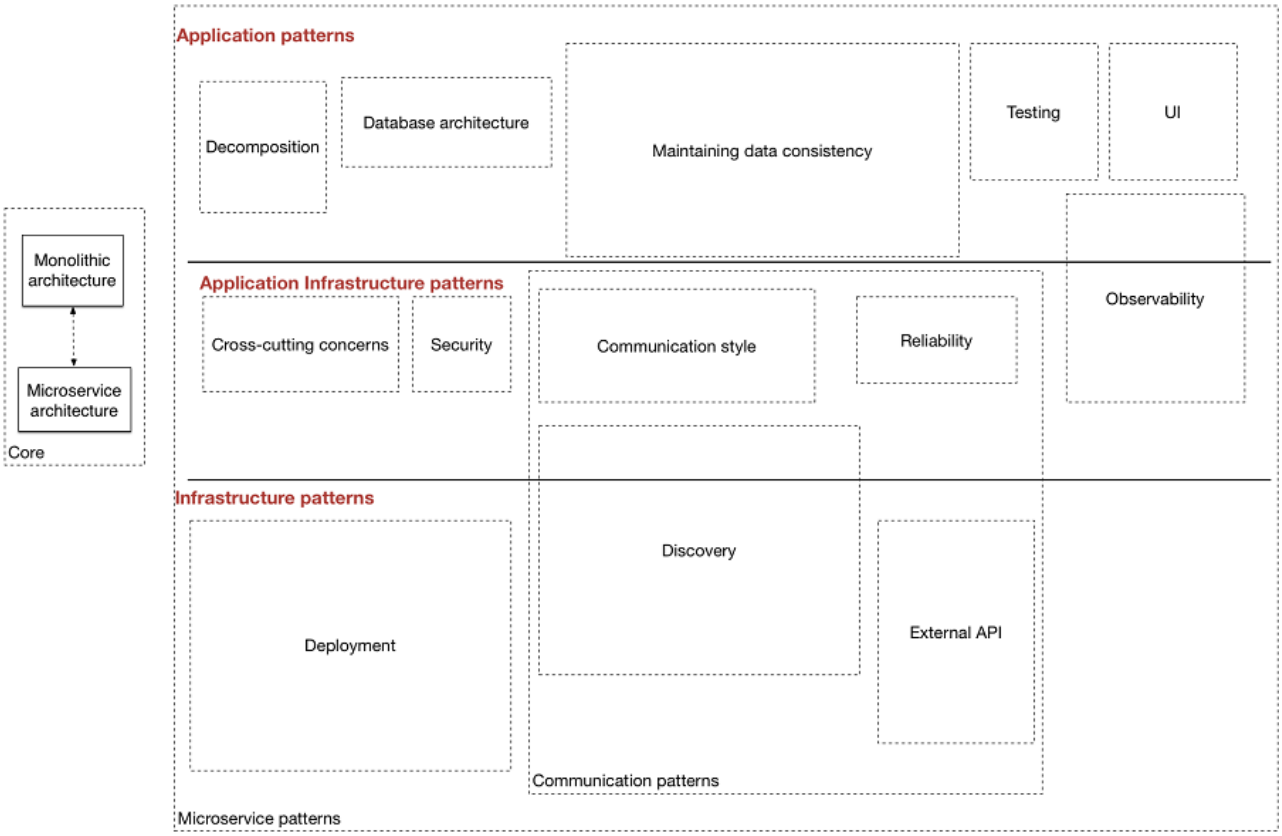
A set of related patterns that solve problems within a domain often forms what is termed a pattern language. The patterns, which solve problems at different levels of scale, work together to define an architecture for a system. The term pattern language along with the concept of a pattern was created by Christopher Alexander, a real-world architect. His writings inspired the software community to adopt the concept of patterns and pattern languages. In the middle 1990s, Christopher Alexander keynoted[2] at least one software conference. Now that we have looked at the motivations for using a pattern language, let's now look at the Microservice Architecture pattern language.

1.4.3 Overview of the Microservice architecture pattern language

The microservice architecture pattern language is a collection of patterns that help you architecture and design an application using the microservice architecture. The pattern language first helps you to first decide whether to use the microservice architecture. It describes the monolithic architecture and the microservice architecture and their benefits and drawbacks. Then, if the microservice architecture is a good fit for your application, the pattern language helps you use it effectively by solving various architecture and design issues. Figure 1.8 shows the high-level structure of the pattern language.

[2] Christopher Alexander speaking at OOPSLA 1996 - www.youtube.com/watch?v=98LdFA-_zfA

Figure 1.8. A high-level view of the Microservice architecture pattern language showing the different problem areas that the patterns solve along with the legend that explains how the different relationships between patterns are shown visually.



The pattern language consists of several groups of patterns. On the left is the core patterns group, the Monolithic Architecture pattern and the Microservice Architecture pattern. Those are the patterns that we have discussed in this chapter. The rest of the pattern language consists of groups of patterns that are solutions to issues that are introduced by using the microservice architecture pattern.

The patterns are also divided into three imperfect layers:

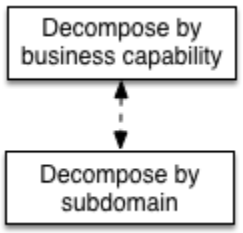
- infrastructure patterns - these are patterns that solve problems that are mostly infrastructure issues that are outside of development
- application infrastructure - these are patterns for infrastructure issues that also impact development
- application patterns - these are patterns that solve problems faced by developers

These patterns are grouped together based on the kind of problem they solve. Let's look at the main groups of patterns.

Decomposition patterns

Deciding how to decompose a system into a set of services is very much an art but there are number of strategies that can help. The decomposition patterns are different strategies that you can use to define your application's architecture. Figure 1.9 shows the patterns:

Figure 1.9. The decomposition patterns

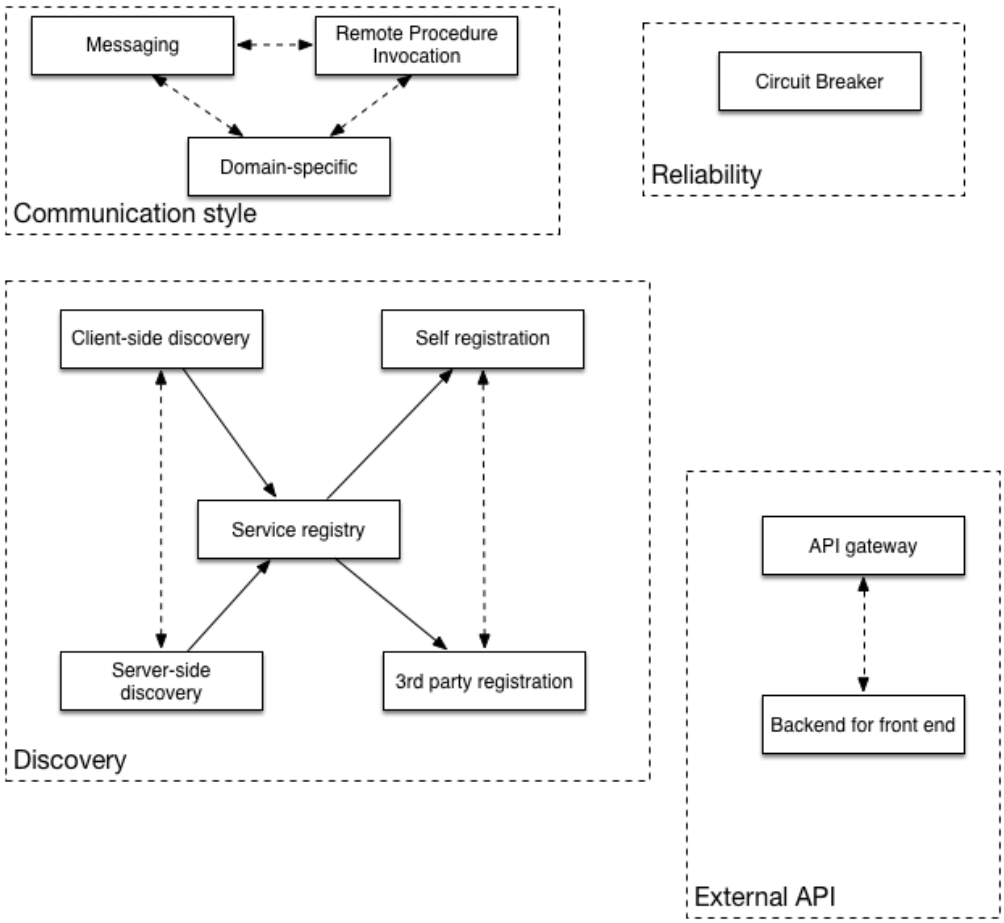


In chapter {chapter-decomposition} we will look at these patterns in detail.

Communication patterns

An application that is built using the microservice architecture is a distributed system. Consequently, inter-process communication is an important part of the microservice architecture. You must make a variety of architectural and design decisions about how your services communicate with one another and the with the outside world. Figure 1.10 shows the communication patterns.

Figure 1.10. The communication patterns



One decision you must make is which kind of IPC mechanism to use. You have a choice between asynchronous messaging or synchronous REST or RPC. In chapter {chapter-ipc} we look at the different styles of inter-process communication.

Another decision that you must make is how to do service registration and discovery. In a modern application, IP addresses of service instances are assigned dynamically. A service client that wants to make an HTTP request, for example, cannot use a configuration containing hardwired IP addresses. An application must use a service registration and discovery mechanism. In chapter {chapter-production} we look at the options.

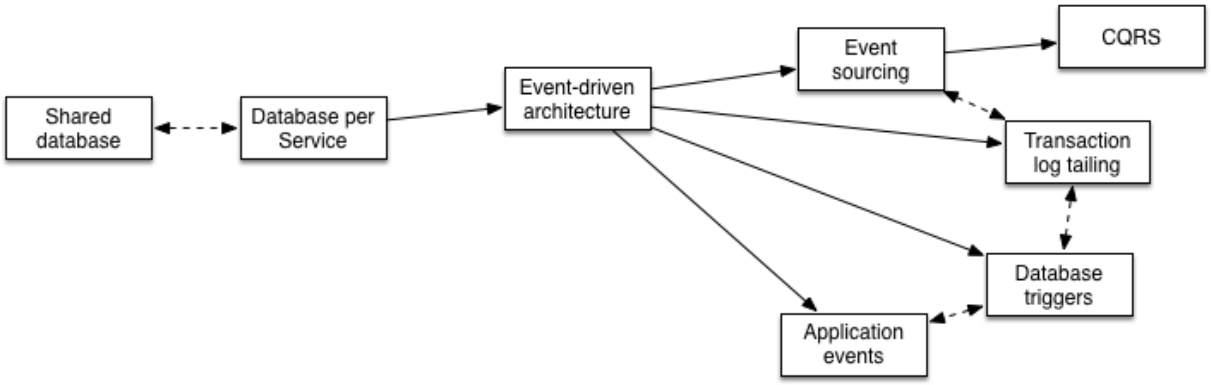
Not only, must you decide how your services communicate amongst themselves but you also have to figure out how clients of your application communicate with the services. One option, is for clients to communicate directly with the individual services. Another option is for clients to make requests via an API Gateway pattern,

which routes requests and aggregates data from multiple services. In chapter {chapter-api-gateway} we look at these patterns in more detail.

Data consistency patterns

As mentioned earlier, in order to ensure loose coupling each service has its own database. Unfortunately, having a database per service introduces some significant issues. For reasons, I describe later the traditional approach of using distributed transactions (aka. 2PC) is not a viable option for modern application. Instead, an application needs to maintain data consistency by using the Saga pattern. Figure 1.11 shows the Saga pattern and the different ways of implementing it.

Figure 1.11. The data consistency patterns

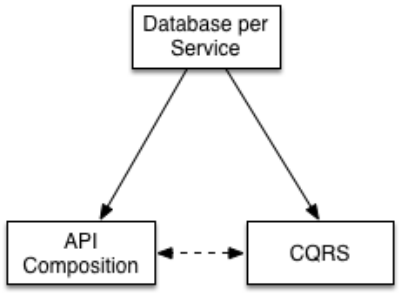


In chapters {chapter-sagas}, {chapter-ddd-aggregates} and {chapter-event-sourcing} we look at these patterns in more detail.

Querying patterns

The other challenge with using a database per service is that some queries need to join data that is owned by multiple services. A service's data is only accessible via its API and so you cannot use distributed queries against its database. As figure 1.12 shows, there are a couple of patterns that you can use to implement queries.

Figure 1.12. The querying patterns

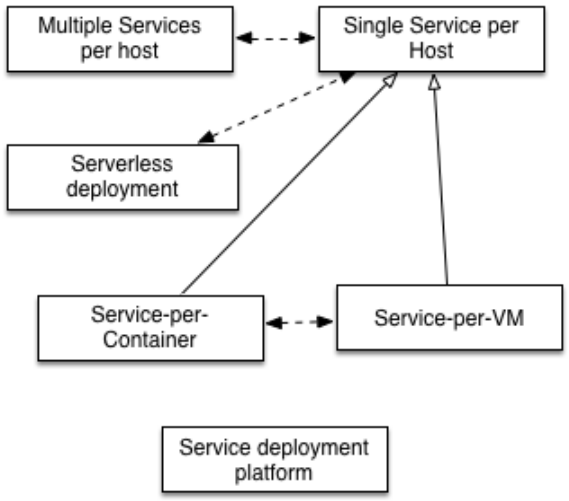


Sometimes you can use the API Composition pattern, which invokes the APIs of one or more services and aggregates results. Other times, you must use the Command Query Responsibility Segregation (CQRS) pattern, which maintains one or more easily queried replicas of the data. In chapter {chapter-queries} we look at the different ways of implementing queries.

Deployment patterns

Deploying a monolithic application is not always easy. But it is straightforward in the sense that there is a single application to deploy. You simply have to run multiple instances of the application behind a load balancer. In comparison, deploying a microservices-based application is much more complex. There are 10s or 100s of services that are written in a variety of languages and frameworks. There are many more moving parts that need to be managed. Figure 1.13 shows the deployment patterns.

Figure 1.13. The deployment patterns



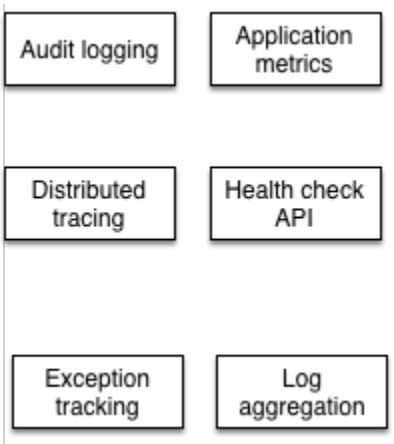
The traditional (manual) way of deploying applications (of copying an application to individual servers) doesn't scale to support a microservice architecture. You need a highly automated deployment infrastructure. Ideally, you should use a deployment platform that provides the developer with a simple UI (command line or GUI) for deploying and managing their services. The deployment platform will typically be based on VMs, Containers or Serverless technology. In chapter {chapter-production} we look at the different deployment options.

Observability patterns

A key part of operating an application is understanding its runtime behavior and troubleshooting problems such as failed requests and high latency. While understanding and troubleshooting a monolithic application is not always easy, it helps that requests are handled in a simple, straightforward way. Each incoming request is load balanced to a particular application instance, which makes a few calls to the database, and returns a response. If, for example, you need to understand how a particular request was handled you simply look at the log file of the application instance that handled the request.

In contrast, understanding and diagnosing problems in a Microservice architecture is much more complicated. A request can bounce around between multiple services before a response is finally returned into a client Consequently, there isn't just one log file to examine. Similarly, problems with latency are more difficult to diagnose since there are multiple suspects. Figure 1.14 shows the observability patterns.

Figure 1.14. The observability patterns



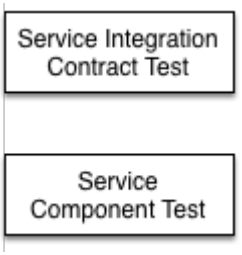
In chapter {chapter-production} I describe the Observability patterns that make it easier to understand and troubleshoot a microservices-based application.

Testing patterns

The microservice architecture makes individual services easier to test since they are

much smaller than the monolithic application. At the same time, however, it is important to test that the different services work together. Figure 1.15 shows the testing patterns.

Figure 1.15. The testing patterns



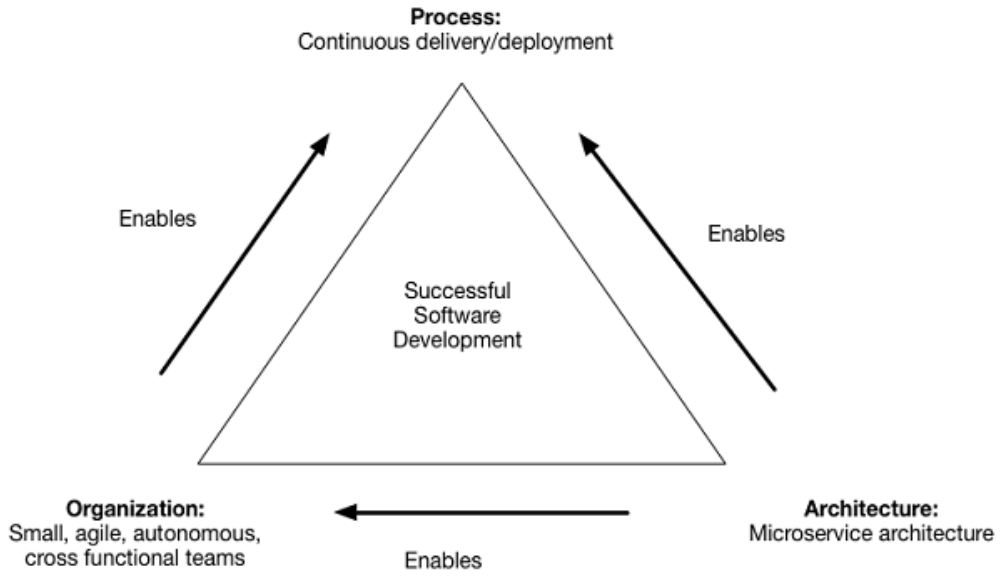
In chapter {chapter-testing}, I describe these testing patterns that enable a microservices application to be tested more easily.

Not surprisingly, these patterns are focussed on solving architect and design problems. To successfully develop software, you certainly need the right architecture. It is not the only concern, however. You must also consider process and organization.

1.5 Beyond microservices: process and organization

For large, complex application, the microservice architecture is usually the best choice. However, as well as having the right architecture, successful software development requires you to also have the organization and development and delivery process. Figure 1.16 shows the relationship between process, organization and architecture.

Figure 1.16. Successful development of large, complex applications requires a combination DevOps, which includes continuous delivery/deployment; small, autonomous teams and the microservice architecture



We have already described the microservice architecture. Let's look at organization and process.

1.5.1 Software development and delivery organization

Success inevitably means that the engineering team will grow. On the one hand, that is a good thing since more developers can get more done. The trouble with large teams is, as Fred Brooks described in **The mythical man month**, the communication overhead of a team of size N is O(N^2). If the team gets too large, then it will become inefficient due to the communication overhead. Imagine, for example, trying to do a daily standup with 20 people.

The solution is to refactor a large single team into a team of teams. Each team is small, consisting of no more than 8-12 people. It has a clearly defined business-oriented mission: developing and possibly operating one or more services, which implement a feature or a business capability. The team is cross functional and can develop, test and deploy its services without having to frequently communicate or coordinate with other teams.

The velocity of the team of teams is significant higher than that of a single large team. The microservice architecture plays a key role in enabling the teams to be autonomous. Each team can develop, deploy and scale their services without coordinating with other teams. Moreover, it is very clear who to contact when a service is not meeting its SLA.

1.5.2 Software development and delivery process

Using the microservice architecture with a waterfall development process is like driving a horse-drawn Ferrari. You would squander most of the benefit of using microservices. If you want to develop an application with the microservice architecture, it is essential that you adopt agile development and deployment practices such as Scrum or Kanban. Better yet, you should practice DevOps, which is a super-set of continuous delivery/deployment.

1.5.3 The human side of adopting microservices

Adopting the microservice architecture changes your architecture, your organization and your development processes. Ultimately, however, it changes the working environment of people, who are, as mentioned earlier, emotional creatures. Their emotions, if ignored, can make the adoption of microservices a bumpy ride. Mary, the CTO of FTGO and other leaders, will struggle to change how FTGO develops software.

The best selling book Managing Transitions by William and Susan Bridges introduces the concept of a transition, which is the process of how people respond emotionally to a change. It describes a three stage Transition Model:

1. Ending, Losing, and Letting Go - the period of emotional upheaval and resistance when people are presented with a change that forces them out of their comfort zone. They often mourn the loss of the old way of doing things. For example, when people reorganize into cross-functional teams they miss their former team-mates. Similarly, a data modeling group, which owns the global data model, will be threatened by the idea of each service having its own data model.
2. The Neutral Zone - the intermediate stage between the old and new ways of doing things where people are often confused. They are, often, struggling to learn the new way of doing things.
3. The New Beginning - the final stage where people have enthusiastically embraced the new way of doing things and are starting to experience the benefits.

The book describes how best to manage each stage of the transition and increase the likelihood of successfully implementing the change. FTGO is certainly suffering from monolithic hell and needs to migrate to a microservice architecture. There will also have to change their organization and their development processes. In order for FTGO to successfully accomplish this, however, it is essential that they take into account the transition model and consider people's emotions. In the next chapter, you will learn about the goal of software architecture and how to decompose an application into services.

1.6 Summary

- The Monolithic architecture is a good choice for simple applications but the Microservice architecture is usually a better choice for large, complex applications
- The Microservice architecture decomposes a system into a set of services each

with their own database

- The Microservice architecture accelerates the velocity of software development by enabling small, autonomous teams to work in parallel.
- The microservice architecture is not a silver bullet since there are significant drawbacks including complexity.
- The microservices pattern language guides your decision making. It helps you decide whether to use the microservice architecture. And, if you pick the microservice architecture, the pattern language helps you apply it effectively.
- The microservice architecture is insufficient. Successful software development also requires DevOps and small, autonomous teams
- Don't forget about the human side of adopting microservices

2 Decomposition strategies

This chapter covers:

- What is software architecture and why it is important
- How to decompose an application into services
- How to use the bounded context concept from Domain-Driven Design to untangle data and make decomposition easier

Let's imagine that you have been give the job of defining the architecture for what is anticipated to be a large, complex application. Your boss, the CIO, has heard that the microservices are a necessary part of accelerating software development. He has strongly suggested that you consider the microservice architecture.

When defining a microservice architecture it is essential that you use the microservices pattern language described in chapter 1. The pattern language consists of patterns that solve numerous problems including deployment, inter-process communication and strategies for maintaining data consistency. The essence, however, of the microservice architecture is functional decomposition. The first and most important aspect of the architecture is, therefore, the definition of the services. As you stand in front of that blank whiteboard in the project's first architecture meeting, you will most likely wonder how to do that!

Don't panic, in this chapter, you will learn how to define a microservice architecture for an application. I discuss how architecture determines your application's -*ilities* including maintainability, testability, and deployability, which directly impact development velocity. You will learn that the microservice architecture is an architecture style that gives an application high maintainability, testability, and

deployability. I describe strategies for decomposing an application into services. You will learn that services are organized around business concerns rather than technical concerns. I also show how to use idea from domain driven design to eliminate god classes, which are classes that are used throughout an application and cause tangled dependencies that prevent decomposition. But first, lets take a look at the purpose of architecture.

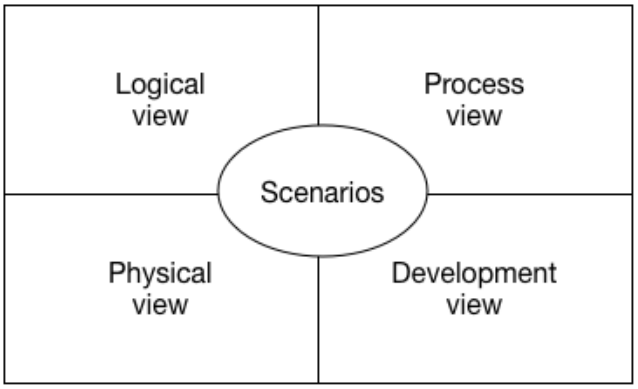
2.1 The purpose of architecture

The microservice architecture is obviously a kind of architecture. But what is it exactly and why does it matter? To answer that these question lets first look at what is meant by architecture. Len Bass and colleagues define architecture as follows:

> *The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.*
>
> -- *Documenting Software Architectures* Bass et al

This is obviously a quite abstract definition. More concretely, an application's architecture can be viewed from multiple perspectives, in the same way that a building's architecture can be viewed from the structural, plumbing, electrical, etc, perspectives. Phillip Krutchen (in his classic paper Architectural Blueprints—The "4+1" View Model of Software Architecture) designed the 4+1 view model of software architecture. The 4+1 model, which is shown in Figure 2.1 , defines four different views of a software architecture, each of which describes a particular aspect of the architecture. Each view consists of a particular set of (software) elements and relationships between them.

Figure 2.1. 4+1 view model



The four views are:

- Logical view - classes, state diagrams, etc.

- Process view - the processes and how they communicate
- Physical view - the physical (virtual machines) and network connections
- Development view - components

In addition to these four views, there are the scenarios - the +1 in the 4+1 model - that animate views. Each scenario describes how the various architectural components within a particular view collaborate in order to handle a request. A scenario in the logical view, for examples, shows how the classes collaborate. Similarly, a scenario in the process view, shows how the processes collaborate.

The the 4+1 view model is an excellent way to describe an applications's architecture. Each view describes an important aspect of the architecture. The scenarios illustrate how the elements of a view collaborate. Lets now look at why architecture is important.

2.1.1 Why architecture matters

An application has two categories of requirements. The first category are the functional requirements, which define what the application must do. They are usually in the form of use cases or user stories. Architecture has very little to do with the functional requirements. You can implement functional requirements with almost any architecture, even a big ball of mud.

Architecture is important because it enables an application to satisfy the second category of requirements, its non-functional requirements. These are also known as quality attributes and are the so called *-ilities*. The non-functional requirements define the runtime qualities such as scalability, and reliability. They also define development time qualities including maintainability, testability, and deployability. The architecture that you choose for your application determines how well it meets these quality requirements.

2.1.2 The microservice architecture is an architectural style

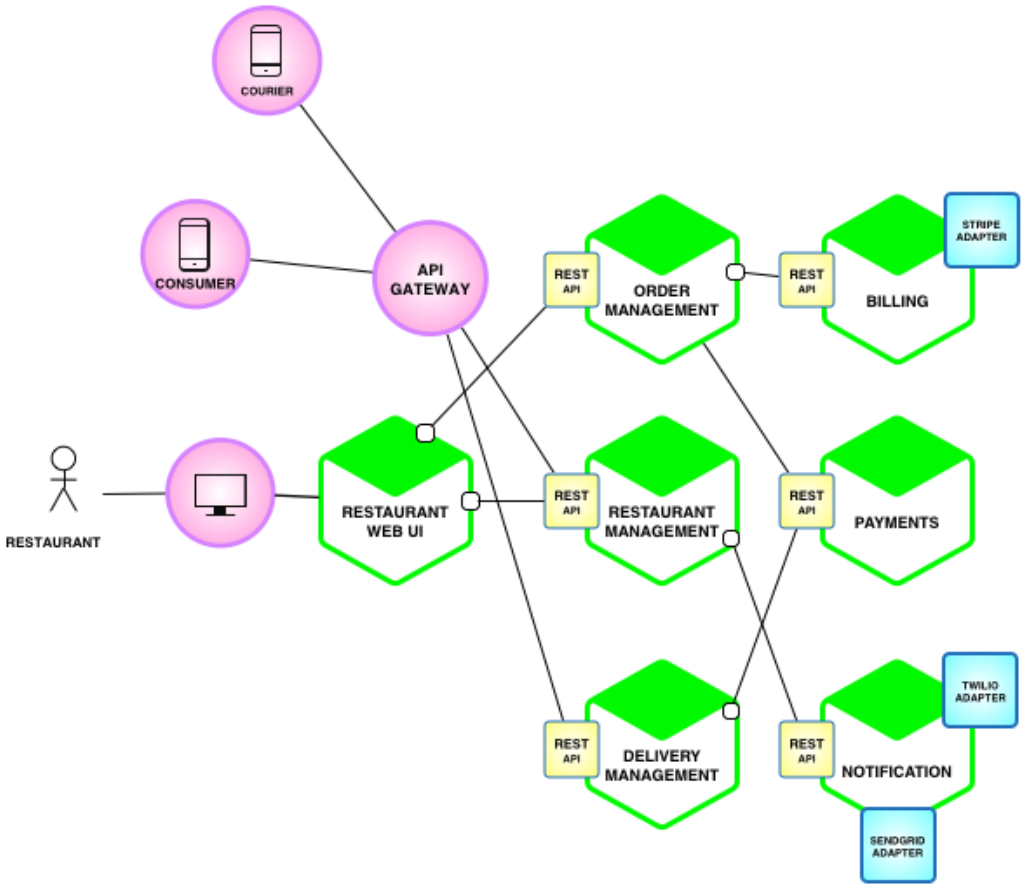
The microservice architecture is what is known as an architectural style. David Garlan and Mary Shaw define an architectural style as follows:

> *An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.*
>
> -- *David Garlan and Mary Shaw,* An Introduction to Software Architecture

A particular architectural style provides a limited palette of elements (a.k.a components) and relationships (a.k.a. connectors) from which you can define your application's architecture. The microservice architecture structures an application as a set of loosely coupled services. In other words, the components are services and the connectors are the communication protocols that enable those services to collaborate. Figure 2.2 shows a possible microservice architecture for the FTGO application. Each

service in this architecture corresponds to a different business function such as order management, and restaurant management.

Figure 2.2. A possible microservice architecture for the FTGO application. It consists of numerous services.



Later in this chapter, I describe what is meant by business function. The connectors between services are implemented using REST APIs. In chapter {chapter-ipc}, I describe inter-process communication in more detail.

A key constraint imposed by the microservice architecture is that the services are loosely coupled. Consequently, there are restrictions how the services collaborate. In order to understand those restrictions, lets attempt to define the term *service*, describe what it means to be loosely coupled and explain why this matters.

What is a service?

Intuitively, we think of a service as a standalone, independently deployable software

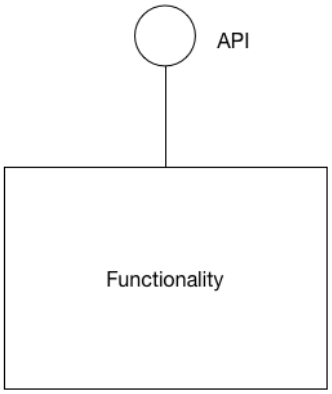
component, which implements some useful functionality. But in reality, the term service is one of those words that we use daily without having a precise definition of its meaning. For example, OASIS has a remarkably vague definition for a service:

> *a mechanism to access an underlying capability*
>
> *-- OASIS*

https://en.wikipedia.org/wiki/Service_(systems_architecture)

A service has an API, which provides its clients with access to its functionality. Figure 2.3 shows an abstract view of a service.

Figure 2.3. Abstract view of a service



A service's clients invoke the API. The service's implementation is hidden, unlike in a monolithic architecture where a developer can easy write code that bypasses a package's API and accesses its implementation. As a result, the microservice architecture enforces the application's modularity.

Typically, the API uses an inter-process communication mechanism such as a messaging or RPC but that is not always the case. Later in chapter {chapter-production}, when I discuss deployment you will see that a service can take many forms. It might be a standalone process, a web application or OSGI bundle running in a container, or a serverless cloud function. An essential requirement is, however, is that a service is independently deployable.

What is loose coupling?

An important characteristic of the microservice architecture is that the services are loosely coupled. All interaction with a service is via its API, which encapsulates its implementation details. This enables the implementation of the service to change without impacting its clients. Loosely coupled services are key to improving an application's development time attributes including its maintainability and testability. Small, loosely coupled services are much easier to understand, change and test.

The requirement for services to be loosely coupled and to collaborate only via APIs prohibits services from communicating via a database. A service's persistent data is equivalent to the fields of a class and must kept private. Keeping data private helps ensure that the services are loosely coupled. It enables a developer to change their service's schema without having to coordinate with developers working on other services. It also improves runtime isolation. For example, it ensures that one service cannot hold database locks that block another service. Later on, however, you will learn that a downside of not sharing databases is that maintaining data consistency and querying across services is more complex.

The microservice architecture structures an application as a set of small, loosely coupled services. As a result, it improves the development time attributes - maintainability, testability, deployability, etc - and enables an organization to develop better software faster. It also improves an application's scalability although it is not the main goal. To develop a microservice architecture for your application you need to identify the services and determine how they communicate. Lets now look at how to define an application's microservice architecture.

2.1.3 Defining an application's microservice architecture

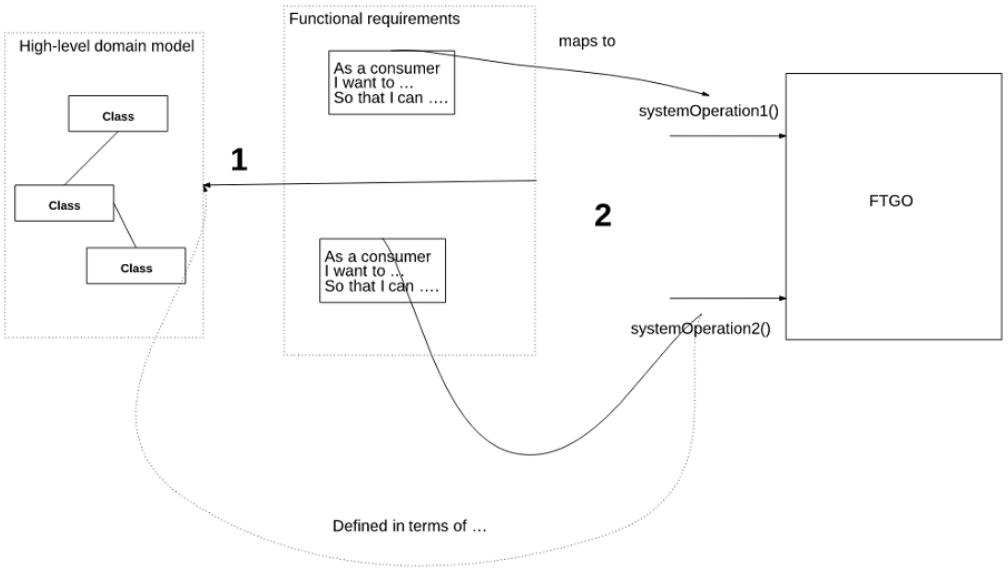
So how to create an architecture? As with any software development effort the starting point are the written requirements, hopefully domain experts and perhaps an existing application. Like much of software development, defining an architecture is more art than science. There isn't a mechanical process to follow that will produce an architecture. In this chapter, we describe a simple process for defining an application's architecture. It captures what needs to be done but it reality the process is likely to be iterative and involve a lot of creativity.

An application's raison d'etre is to handle requests, and so we will first identify the key requests. However, instead of describing the requests in terms of specific IPC technologies such as REST or messaging, I use the more abstract notion of system operation. Once we have identified the system operations, we will then identify the services that comprise the application's microservice architecture. The system operations become the architectural scenarios that illustrate how the services collaborate. Lets now look at how to identify the system operations.

2.2 Identifying the system operations

The first step of defining an application's architecture is to define the system operations. The starting point are the application's requirements, which include user stories and their associated user scenarios (note - different from the architectural scenarios). The system operations are identified and defined using a two step process, which is shown in Figure 2.4. This process is inspired by object-oriented design process described in Craig Larman's book Applying UML and Patterns. The first step creates the high-level domain model consisting of the key classes, which provides a vocabulary with which to describe the system operations. The second step identifies the system operations and describe each one's behavior in terms of the domain model.

Figure 2.4. System operations are derived from the application's requirements using a two step process



The domain model is derived primarily from the nouns of the user stories and the system operations are derived mostly from the verbs. The behavior of each system operation is described in terms of its effect on one or more domain objects and the relationships between them. A system operation can create, update or delete domain objects, as well as create or destroy relationships between them. Lets now look at how to define a high-level domain model. After that we will define the system operations in terms of the domain model.

2.2.1 Creating a high-level domain model

The first step in the process of defining the system operations is to sketch a high-level domain model for the application. Note that this domain model is much simpler that what will ultimately be implemented. The application won't even have a single domain model because, as you will learn below, each service has its own domain model. Despite being a drastic simplification, a high-level domain model is useful at this stage because it defines the vocabulary for describing the behavior of the system operations.

A domain model is created using standard techniques, such as analyzing the nouns in the stories and scenarios, and by talking to the domain experts. Consider, for example, the `Place Order` story. We can expand that story into numerous user scenarios including this one:

```
Given a consumer
  And a restaurant
When the consumer places an order for that restaurant
   with a delivery address/time that can be served by that restaurant
Then consumer's credit card is authorized
```

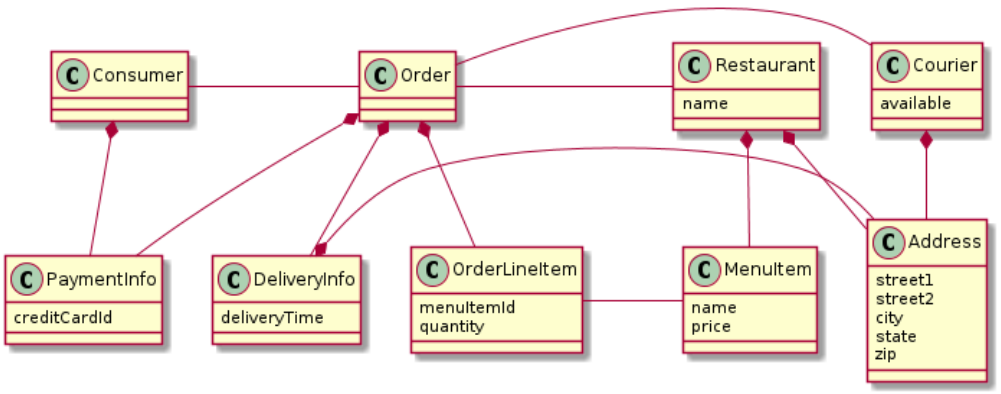
```
  And an order is created in the PENDING_ACCEPTANCE state
  And the order is associated with the consumer
  And the order is associated with the restaurant
```

The nouns in this user scenario hints at the existence of various classes including Consumer, Order, Restaurant and CreditCard. Similarly, the `Accept Order` story can be expanded into a scenarios such this one:

```
Given an order or a restaurant that is waiting to be accepted
  and a courier that is available to deliver the order
When a restaurant accepts an order with a promise to prepare by a particular time
Then the state of the order is changed to ACCEPTED
  And the order's promiseByTime is updated to the promised time
  And the courier is assigned to deliver the order
```

This scenario suggests the existence of a Courier class. The end result after a few iterations of analysis, will be a domain model that consists, unsurprisingly, of those classes and others such as MenuItem and Address. Figure 2.5 is a class diagram that shows the key classes.

Figure 2.5. The key classes in FTGO domain model



The responsibilities of each class are as follows:

- Consumer - a consumer who places orders.
- Order - an order that is placed by a consumer. It describes the order and tracks its status.
- Restaurant - a restaurant that prepares orders for delivery to consumers
- Courier - a courier who deliver orders to consumers. It tracks the availability of the courier and their current location.

A class diagram such as the one above describes one aspect of an application's architecture. But it isn't much more than a pretty picture without the scenarios to animate it. The next step is to define the system operations, which correspond to architectural scenarios.

2.2.2 Defining system operations

Once you have defined a high-level domain model, the next step is to identify the requests that the application must handle. The details of the UI are out of scope but you can imagine that in each user scenario, the UI will make requests to the backend business logic to retrieve and update data. FTGO is primarily a web application, which means that most requests are HTTP-based. However, its possible that some clients might use messaging. Instead of committing to specific protocol, it makes sense, therefore, to use the more abstract notion of a system operation to represent requests.

There are two types of system operations:

- commands - system operations that create, update and delete data.
- queries - system operations that read, i.e. query, data.

Ultimately, these system operations will correspond to REST, RPC or messaging endpoints. But for now it is useful to think of them abstractly. Lets first identify some commands.

Identifying system commands

A good starting point for identifying system commands is to analyze the verbs in the user stories and scenarios. Consider, for example, the `Create Order` story. It very clearly suggests that the system must provide a `Create Order` operation. Many other stories individually map directly to system commands. Table 2.1 lists some of the key system commands.

Table 2.1. Key system commands for the FTGO application

Actor	Story	Command	Description
Consumer	Create Order	createOrder()	creates an order
Restaurant	Accept Order	acceptOrder()	indicates that the restaurant has accepted the order and is committed to preparing it by the indicated time
Restaurant	Order Ready for Pickup	noteOrderReadyForPickup()	indicates that the order is ready for pickup
Courier	Update Location	noteUpdatedLocation()	updates the current location of the courier
Courier	Order picked up	noteOrderPickedUp()	indicates that the courier has picked up the order
Courier	Order delivered	noteOrderDelivered()	indicates that the courier has delivered the order

A command has a specification which defines its parameters, return value and its behavior in terms of the domain model classes. The behavior specification consists of pre-conditions, which must be true when the operation is invoked, and post-conditions, which are true are after the operation is invoked. Here, for example, is the specification of the `createOrder()` system operation:

Operation	createOrder(consumer id, payment method, delivery address, delivery time, restaurant id, order line items)
Returns	orderId, …
Preconditions	• the consumer exists and can place orders • the line items correspond to the restaurant's menu items • the delivery address and time can be serviced by the restaurant
Post-conditions	• the consumer's credit card was authorized for the order total • an order was created in the PENDING_ACCEPTANCE state

The pre-conditions mirror the 'givens' in the `Place Order` user scenario described earlier. The postconditions mirror the 'thens' from the scenario. When a system operation is invoked it will verify that preconditions and perform the actions required to make the postconditions true.

Here is the specification of the `acceptOrder()` system operation:

Operation	acceptOrder(restaurantId, orderId, readyByTime)
Returns	-
Preconditions	• the order.status is PENDING_ACCEPTANCE • a courier is available to deliver the order
Post-conditions	• the order.status was changed to ACCEPTED • the order.readyByTime was changed to the `readyByTime` • the courier was assigned to deliver the order

It is pre- and post-conditions mirror the user scenario from earlier.

Most of the architecturally relevant system operations are commands. Sometimes, however, queries, which retrieve data, are also important.

Identifying system queries

As well as implementing commands, an application must also implement queries. The queries provide the UI with the information a user needs to make decisions. At this stage, we don't have a particular UI design for FTGO application in mind, but consider, for example, the flow when a consumer places an order:

1. User enters delivery address and time
2. System displays available restaurants
3. User selects restaurant
4. System displays menu
5. User selects item and checks out
6. System creates order

This user scenario suggests the following queries:

- findAvailableRestaurants(deliveryAddress, deliveryTime) - retrieves the restaurants that can deliver to the specified delivery address at the specified time
- findRestaurantMenu(id) - retrieves information about a restaurant including the menus items

Of the two queries, `findAvailableRestaurants()` is probably the most architecturally significant. It is a complex query involving geosearch. The geosearch component of the query consists of finding all points - restaurants - that are near a location - the delivery address. It also filters out those restaurants are not open when the order needs to be prepared and picked up. Moreover, performance is critical since this query is executed whenever a consumer wants to place an order.

The high-level domain model and the system operations capture what the application does. They help drive the definition of the application's architecture. The behavior of each system operation is described in terms of the domain model. Each important system operation represents an architecturally significant scenario that is part of the description of the architecture. Lets now look at how to define an application's microservice architecture.

2.3 Strategies for decomposing an application into services

Once the system operations have been defined, the next step is to identify the application's services. As mentioned earlier, there isn't a mechanical process to follow. There are, however, various decomposition strategies that you can use. Each one attacks the problem from a different perspective and uses its own terminology. But with all strategies, the end result is the same: an architecture consisting of services that are primarily organized around business rather than technical concepts. Lets look at the first strategy, which defines services corresponding to business capabilities.

2.3.1 Decompose by business capability

One strategy for creating a microservice architecture is to decompose by business capability. A business capability is a concept from business architecture modeling. A business capability is something that a business does in order to generate value. The set of capabilities for a given business depend on the kind of business. For example, the capabilities of an insurance company typically include Underwriting, Claims management, Billing, Compliance, and so on/ The capabilities of an online store include Order management, Inventory management, Shipping and so on.

Business capabilities define what an organization does

An organization's business capabilities capture *what* an organization's business is. They are generally stable. In contrast, *how* a organization conducts its business changes over time, sometimes dramatically. That is especially true today, with the rapidly growing use of technology to automate many business processes. For example, it wasn't that long ago when you deposited checks at your bank by handing them to a teller. It then became possible to deposit checks using an ATM. And, now today, you can conveniently deposit most checks using your smartphone. As you can see, the `Deposit check` business capability has remained stable but the manner in which it is done has drastically changed.

Identifying business capabilities

An organization's business capabilities are identified by analyzing the organization's

purpose, structure, and business processes. Each business capability can be thought of as a service, except it is business-oriented rather than technical. Its specification consists of various components including inputs and outputs, and service-level agreements. For example, the input to an *Insurance Underwriting*capability is the consumer's application, and the outputs include approval and price.

A business capability is often focussed on a particular business object. For example, the Claim business object is the focus of the Claim management capability. A capability can often be decomposed into sub-capabilities. For example, the Claim management capability has several sub-capabilities including Claim information management, Claim review, and Claim payment management.

It is not difficult to imagine that the business capabilities for FTGO include:

- Supplier management
  - Courier management - managing courier information
  - Restaurant information management - managing restaurant menus and other information including location and opening hours
- Consumer management - managing information about consumers
- Order taking and fulfillment
  - Order management - enabling consumers to create and manage orders
  - Restaurant order management - managing the preparation of orders at a restaurant
  - Logistics
  - Courier availability management - managing the real-time availability of couriers to delivery orders
  - Delivery management - delivering orders to consumers
- Accounting
  - Consumer accounting - managing billing of consumers
  - Restaurant accounting - managing payments to restaurants
  - Courier accounting - managing payments to couriers
- …

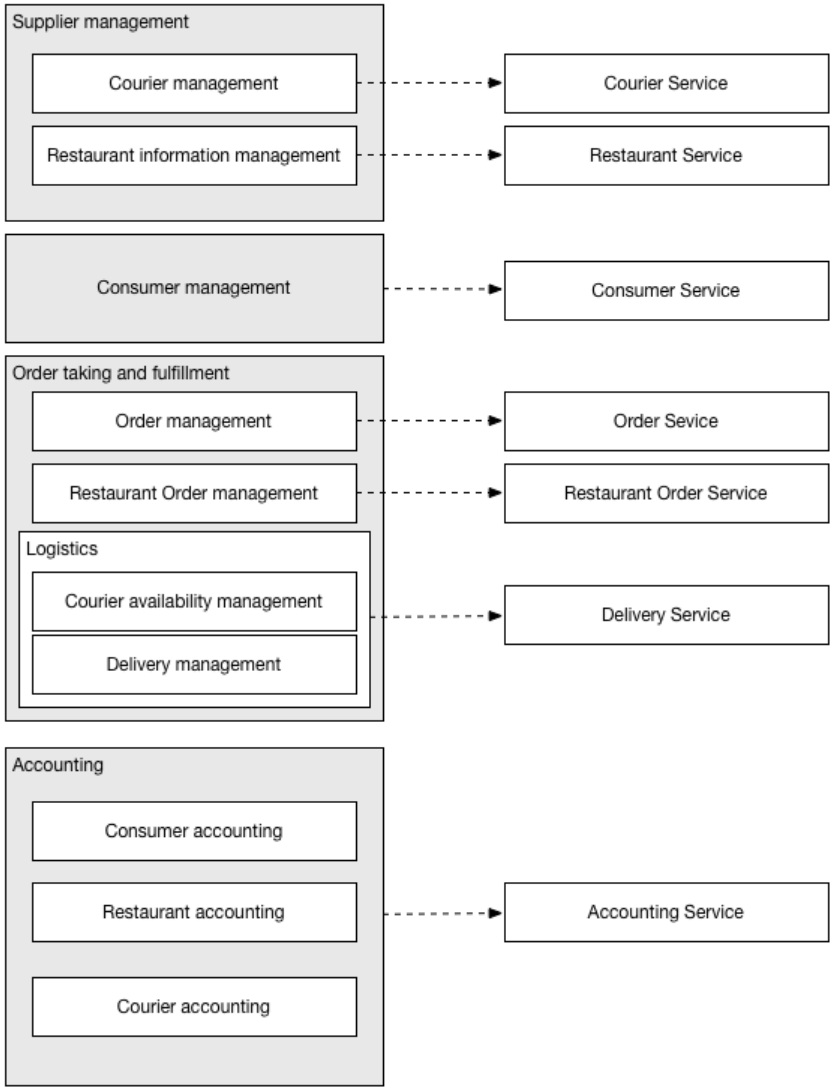
The top-level capabilities include Supplier management, Consumer management, Order taking and fulfillment, and accounting. There will likely be many other top-level capabilities including marketing related capabilities. Most top-level capabilities are decomposed into sub-capabilities. For example, Order taking and fulfillment is decomposed of into three sub-capabilities.

On interesting aspect of this capability hierarchy is that there are two restaurant related capabilities: Restaurant information management, and Restaurant order management. That is because they represent two very different aspects of restaurant operations. Lets now look at how to use business capabilities to define services.

From service to business capabilities

Once you have identified the business capabilities, you then define a service for each capability or group of related capabilities. Figure 2.6 shows the mapping from capabilities to services for the FTGO application. The top-level accounting capabilities is mapped to the `Accounting Service`, where as each of the sub-capabilities of Supplier management are mapped to a service.

Figure 2.6. Mapping FTGO business capabilities to services



A key benefit of organizing services around capabilities is that because they are stable

the resulting architecture will also be a relatively stable. The individual components of the architecture may evolve as the *how* aspect of the business changes but the architecture remains unchanged.

Having said that, it is important to remember that the services shown in figure 2.6 are merely the first attempt at defining the architecture. They may evolve over time as we learn more about the application domain. In particular, an important step in the architecture definition process is investigating how the services collaborate in each of the key architectural service. You might, for example, discover that a particular decomposition is inefficient due to excessive inter-process communication and that you must combine services. Lets take a look at the scenarios.

2.3.2 Using scenarios to determine how the service collaborate

An application's architecture consists of both software elements - the services - and the relationships between them - communication mechanisms. Consequently, after having identified the services, we must then decide how the services communicate. To do that we must is to think about how services collaborate during each scenario. The system operations define the scenarios and so drive the definition of the architecture. Since a system operation is a request from the external world, the first decision to make is which service will initially handle the request. After that we must decide what other services are involved in handling the request and how they communicate.

Assigning system operations to services

The first step is to decide which service is the initial entry point for a request. Many system operation neatly map to a service but sometimes the mapping is less obvious. Consider, for example, the `noteUpdatedLocation()` operation, which updates the courier location. On the one hand, since it is related to couriers, this operation should be assigned to the `Courier` service. But on the other hand, it is the `Delivery Service` that needs the courier location. In this case, assigning an operation to a service that needs the information provided by the operation is a better choice. In other situations, it might make sense to assign an operation to the service that has the information necessary to handle it.

Table 2.2 shows which services in the FTGO application are responsible for which operations.

Table 2.2. Mapping system operations to services in the FTGO application

Service	Operation
Order Service	• createOrder() • findAvailableRestaurants()
Restaurant Order Service	• acceptOrder() • noteOrderReadyForPickup()
Delivery Service	• noteUpdatedLocation() • noteOrderPickedUp() • noteOrderDelivered()

After having assigned operations to services, the next step is to decide how the services collaborate in order to handle each system operation.

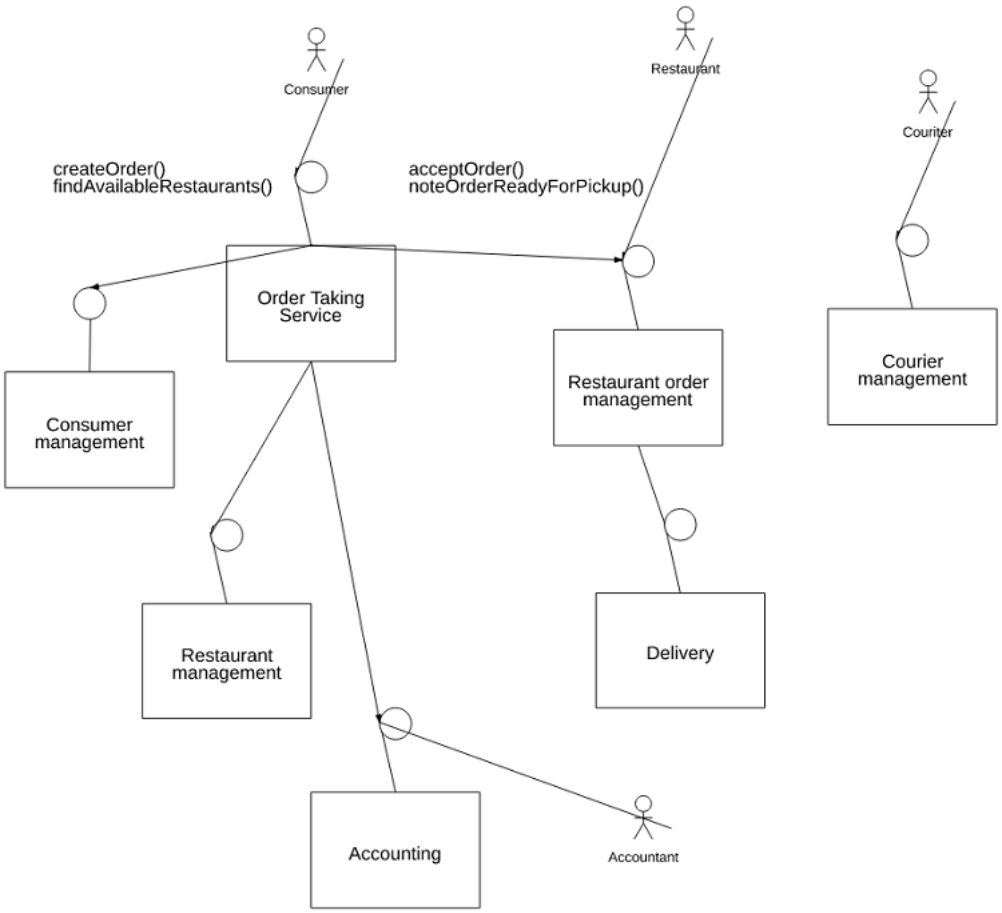
Determining how services collaborate

Some system operations are handled entirely by a single service. Other system operations, however, span multiple services. The knowledge needed to handle one of these requests might, for instance, be scattered around multiple services. For example, in the FTGO application the `Consumer Service` handles the `createConsumer()` operation entirely by itself. However, when handling the `createOrder()`operation, the `Order Service` must invoke other services including:

- `Consumer Service` - verify that the consumer can place an order and obtain their payment information
- `Restaurant Order Service` - verify the order line items and that the delivery address/time is within the restaurant's service area
- `Accounting Service` - authorize the consumer's credit card

Similarly, when the restaurant accepts an order, the `Restaurant Order Service` must invoke the `Delivery Service` to schedule a courier to deliver the order. Figure 2.7 shows the services and the dependencies between them. Each dependency represents some kind of inter-service communication.

Figure 2.7. The FTGO services and their dependencies



Even though figure 2.7 suggests that the services communicate using REST, it is important to remember that in practice they might very well use other communication mechanisms such as asynchronous messaging. In fact, in chapter {chapter-ipc}, which covers inter-process communication, and chapter {chapter-sagas}, which discusses transaction management, I describe how asynchronous messaging plays a major role in a microservice architecture.

2.3.3 *If only it were this easy...*

On the surface, the strategy of creating a microservice architecture by defining services corresponding to business capabilities looks quite reasonable. Unfortunately, however, there are some significant problems that need to be addressed.

Synchronous inter-process communication reduces availability

The first problem, is how to use inter-service communication in a way that doesn't

reduce availability. For example, the most straightforward way to implement the `createOrder()` operation is for the `Order Service` to synchronously invoke the other services using REST. The drawback of doing using a protocol such as REST is that it reduces the availability of the `Order Service`. It will not be able to create an order if any of those services is unavailable. Sometimes this is a worthwhile trade-off, but in chapter 3 you will learn that using asynchronous messaging, which eliminates tight coupling and improves availability is often a better choice.

The need for distributed transactions

The second problem is that a system operation that spans multiple services must somehow maintain data consistency across those services. For example, when a restaurant accepts an order the `Restaurant Order Service` invokes the `Delivery Service` to schedule delivery of the order. The `acceptOrder()` operation must reliably update data in the `Restaurant Order Service` and `Delivery Service`. The traditional solution is to use a two-phase commit-based, distributed transaction management mechanism. Unfortunately, as you will learn in chapter 4, this is not a good choice for modern applications, and you must use very different approach to transaction management.

God classes prevent decomposition

The third and final problem, is the existence of so-called God classes, which are classes that are used throughout the application, and make it difficult to decompose the business logic. An example of a god class in the FTGO application is the `Order` class. It has state and behavior for many different aspects of the FTGO application's business logic including order taking, restaurant order management, and delivery. Consequently, this class makes it extremely difficult to decompose any of the business logic that involves orders into the services described earlier.

Fortunately, there is a way to eliminate god classes. The technique comes from Domain-Driven Design (DDD), which provides an alternative way to decompose to an application. As with business capability based decomposition, this strategy takes a domain-oriented approach. The resulting architecture will likely be the same, despite DDD using very different terminology and having different motivations. One particularly valuable contribution of DDD is that it provides a way to eliminate the god classes.

2.3.4 Decompose by sub-domain/bounded context

DDD is an approach for building complex software applications that is centered on the development of an object-oriented, domain model. A domain model captures knowledge about a domain in a form that can be used to solve problems within that domain. It defines the vocabulary used by the team - what DDD calls the Ubiquitous language. The domain model is closely mirrored in the design and implementation of the application. DDD has two concepts that are incredibly useful when applying the microservice architecture: subdomains and bounded contexts.

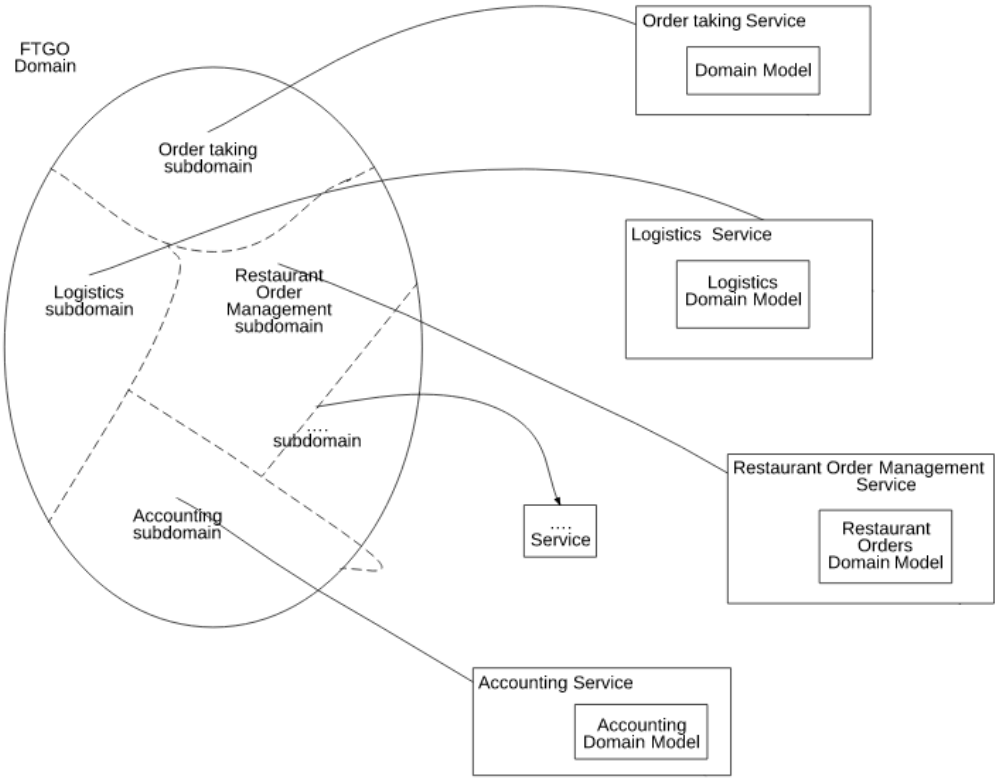
From subdomains to services

DDD is quite different than the traditional approach to enterprise modeling which creates a single model for the entire enterprise. In such a model, there would be, for example, a single definition of each business entity such as customer, order etc. The problem with kind of modeling is that getting different parts of an organization to agree on a single model is a monumental task. Also, it means that from the perspective of a given part of the organization, the model is overly complex for their needs. Moreover, the domain model can be confusing since different parts of the organization might use either the same term for different concepts or different terms for the same concept. DDD avoids these problems by defining multiple domain models, each one with an explicit scope.

DDD defines a separate domain model for each subdomain. A subdomain is a part of the domain, which is DDD's term for the application's problem space. Subdomains are identified using the same approach as identifying business capabilities: analyze the business and identify the different areas of expertise. The end result is very likely to be subdomains that are similar to the business capabilities. The examples of subdomains in FTGO include order taking, order management, restaurant order management, delivery, and financials. As you can see, these subdomains are very similar to the business capabilities described earlier.

DDD calls the scope of a domain model a bounded context. A bounded context includes the code artifacts etc that implement the model. When using the microservice architecture, each bounded context is a service or possibly a set of services. In other words, we can create a microservice architecture by applying DDD and defining a service for each subdomain. Figure 2.8 shows how the subdomains map to services, each with their own domain model.

Figure 2.8. From subdomains to services



DDD and the microservice architecture are in almost perfect alignment. The DDD concept of subdomains and bounded contexts map nicely to services within a microservice architecture. Also, the microservice architecture's concept of autonomous teams owning services is completely aligned with the DDD's concept of each domain model being owned and developed by a single team. What is even better is that the concept of a subdomain with its own domain model is a great way to eliminate God classes and thereby make decomposition easier.

Eliminating God classes

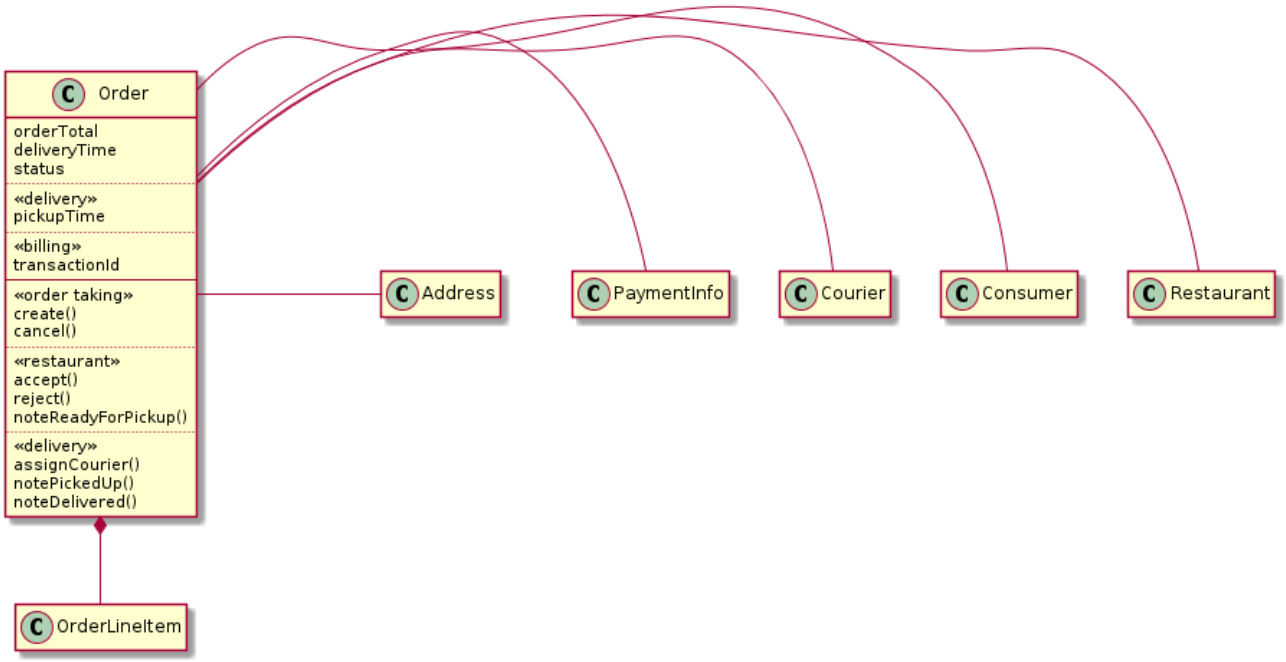
God classes[3] are the bloated classes that are used throughout an application. A god class typically implements business logic for many different aspects of the application. It typically has a large number of fields is mapped to a database table with many columns. Most applications have at least one these classes, each one representing a concept that is central to the domain: accounts in banking, orders in e-commerce, policies in insurance etc. Because a god class bundles together state and behavior for many different aspects of an application it is an insurmountable obstacle to splitting

[3] wiki.c2.com/?GodClass

any business logic that uses it into services.

The `Order` class is a great example of a god class in the FTGO application. That is not surprising since the purpose of FTGO is to deliver food orders to customers. Most parts of the system involve orders. If the FTGO application had a single domain model then the `Order` class would be a very large class. It would have state and behavior corresponding to many different parts of the application. Figure 2.9shows the structure of this class that would be created using traditional modeling techniques.

Figure 2.9. The `Order` god class



As you can see, the Order class has fields and methods corresponding order processing, restaurant order management, delivery and payments. This class also has a complex state model since that one model has to describe state transitions from disparate parts of the application. This class in its current form makes it extremely difficult to split code into services.

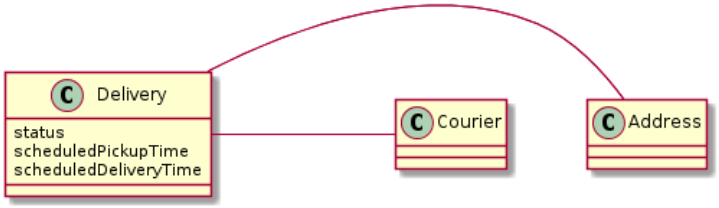
One solution is to package the `Order` class into a library and create a central `Order` database. All services that processes orders use this library and access the access database. The trouble with this approach is that it violates one of the key principles of the microservice architecture and results in undesirable, tight coupling. For example, any change to the `Order` schema requires the teams to update their code in lock step.

Another solution is to encapsulate the `Order` database in an `Order Service`, which is

invoked by the other services to retrieve and update orders. The problem this design is that the `Order Service` would be a data service with an anaemic domain model containing little or no business logic. Neither of these options is appealing but fortunately, DDD provides a solution.

DDD eliminates these god classes by treating each part of an application as separate subdomain with its own domain model. This means that service in the FTGO application that has anything to do with orders, has a domain model with its own version of the Order class. A great example of the benefit of multiple domain models is the `Delivery Service`. Its view of an Order, which is shown in figure 2.10, is extremely simple: pickup address, pickup time, delivery address, and delivery time. Moreover, rather than call it an `Order` the `Delivery Service` uses the more appropriate name of `Delivery`.

Figure 2.10. The delivery subdomain model



The `Delivery Service` is not interested in any of the other attributes of an order..

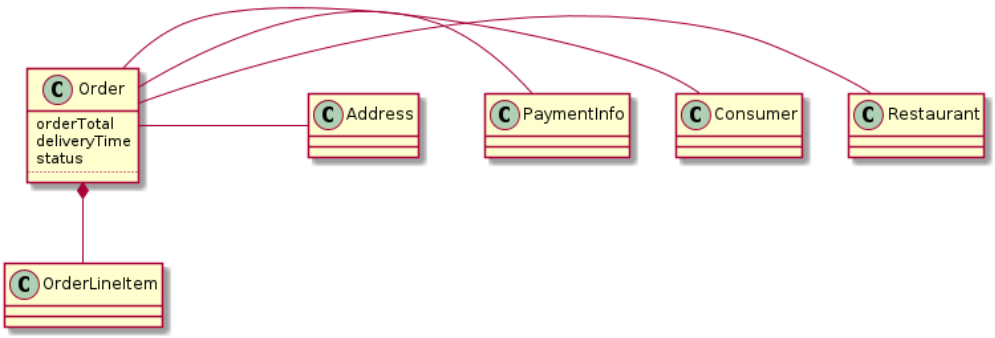
The `Restaurant Order` service also has a much simpler view of an order. Its version of an Order, which is shown figure 2.11, simply consist of a status, a `prepareByTime` and a list of line item, which tell the restaurant what to prepare.

Figure 2.11. The restaurant order subdomain model

It is unconcerned with the consumer, payment, delivery etc.

The `Order` service has the most complex view of an order, which is shown in figure 2.12. Even though it has quite a few fields and methods it is still much simpler than the original version.

Figure 2.12. Order subdomain model



The Order class in each domain model represent different aspects of the same Order business entity. The FTGO application must maintain consistency between these different objects in different services. For example, once the `Order Service` has authorized the consumer's create card it must trigger the creation in the `Restaurant Order Service`. Similarly, if the restaurant rejects the order via the `Restaurant Order Service` it must be cancelled in the `Order Management` service and the customer's credited in the billing service. In chapter 4, you will learn how to maintain consistency between services using event-driven mechanism called sagas.

As you can see, DDD has some concepts that are extremely useful when defining a microservice architecture. It works well with business capability-based decomposition. Lets look at some other guidelines that we should keep in mind developing a microservice architecture.

2.3.5 Decomposition guidelines

There are a couple of principles from object-oriented design that can be adapted and used when applying the microservice architecture. They were created by Robert C. Martin and described in his classic book Designing Object Oriented C++ Applications Using The Booch Method. The first principle is the Single Responsibility Principle (SRP) for defining the responsibilities of a class. The second principle is the Common Closure Principle (CCP) for organizing classes into packages. Lets take a look at these principles and see how they can be applied to the microservice architecture.

Single responsibility principle

The Single Responsibility Principle states that

> *A class should have only one reason to change.*
>
> *-- Robert C. Martin*

Each responsibility that a class has is a potential reason for that class to change. If a

class has multiple responsibilities that change independently then the class will not be stable. By following the SRP, you define classes that each have a single responsibility and hence a single reason for change.

We can apply SRP when defining a microservice architecture and create small, cohesive services that each have single responsibility. This will reduce the size of the services and increase their stability. The new FTGO architecture is an example of SRP in action. Each aspect of getting food to a consumer - order taking, order preparation, and delivery - is the responsibility of a separate service.

Common Closure Principle

The other useful principle is the Common Closure Principle. It states that

> *The classes in a package should be closed together against the same kinds of changes. a change that affects a package affects all the classes in that package.*
>
> *-- Robert C. Martin*

The idea is that if two classes change in lock step because of the same underlying reason then they belong in the same package. Perhaps, for example, those classes implement a different aspect of the a particular business rule. The goal is that when that business rule changes developers, only need to change code in a small number - ideally only one - of packages. Adhering to the CCP significantly improves the maintainability of an application.

We can apply CCP when creating a microservice architecture and package components that change for the same reason into the same service. Doing this will minimize the number of services that need to be changed and deployed when some requirement changes. Ideally, a change will only affect a single team and a single service. CCP is the antidote to the distributed monolith anti-pattern.

Decomposition by business capability and by subdomain along with SRP and CCP are good techniques for decomposing an application into services. In order to apply them and successfully develop a microservice architecture, you must solve some transaction management and inter-process communication issues.

2.4 Summary

- Architecture determines your application's *-ilities* including maintainability, testability, and deployability, which directly impact development velocity
- The microservice architecture is an architecture style that gives an application high maintainability, testability, and deployability
- Services in a microservice architecture are organized around business concerns - business capabilities or subdomains - rather than technical concerns
- You can eliminate God classes, which cause tangled dependencies that prevent decomposition,by applying DDD and defining a separate domain model for each service

3 Inter-process communication in a microservice architecture

This chapter covers:

- The importance of inter-process communication in a microservice architecture
- The various inter-process communication options and their trade-offs
- How to reliably send messages as part of a database transaction
- The benefits of services that communicate using asynchronous messaging

A monolithic application often uses an inter-process communication (IPC) mechanism such as HTTP/REST or messaging, to interact with other applications. But internally, components invoke one another via language-level method or function calls. In contrast, as you saw in the previous chapter, the microservice architecture structures an application as a set of services. Those services must often collaborate in order to handle a request. Since service instances are typically processes running on multiple machines they must interact using IPC. Consequently, IPC plays a much more important role in a microservice architecture than it does in a monolithic application.

The choice of IPC mechanism is an important architectural decision. It can impact application availability. What's more, as I describe in this chapter and the next, IPC even intersects with transaction management. We favor an architecture consisting of loosely coupled services that communicate internally using asynchronous messaging. Synchronous protocols such as REST are used mostly to communicate with other applications. However, it is important to remember that there are no silver bullets. In this chapter, we explore various IPC options and describe the trade-offs. I'll compare and contrast REST and messaging. I'll also describe how to send and receive messages as part of a database transaction. Lets first examine some API design issues.

3.1 API design in a microservice architecture

APIs or interfaces are central to software development. An application is comprised of modules. Each module has an interface, which defines the set of operations that module's clients can invoke. A well-designed interface exposes useful functionality while hiding the implementation. It enables the implementation to change without impacting clients.

3.1.1 Defining APIs in a microservice architecture

In a monolithic application, an interface is typically specified using a programming language construct such as a Java interface. A Java interface specifies a set of a methods that a client can invoke. The implementation class is hidden from the client. Moreover, since Java is a statically typed language, if the interface changes to be incompatible with the client the application will not compile.

APIs/interfaces are equally important in a microservice architecture. A service's API is a contract between the service and its clients. The challenge is that a service interface is not defined using a simple programming language construct. By definition, a service and its clients are not compiled together. If a new version of a service is deployed with an incompatible API there is not compilation error. Instead, there will be runtime failures.

Regardless of your choice IPC mechanism, its important to precisely define a service's API using some kind of interface definition language (IDL). Moreover, there are good arguments for using an API-first approach[4] to defining services. First, you write the interface definition. Next, you review the interface definition with the client developers. It is only after iterating on the API definition do you then implement the service. Doing this upfront design increases your chances of building a service that meets the needs of its clients.

API first design is essential

Even on really small projects I've seen problems because components don't agree on an API. For example, on one project the backend Java developer and the AngularJS front-end development both said they had completed development. The application, however, did not work. The REST and WebSocket API used by the front-end application to communicate with the backend was poorly defined. As a result, the two applications could not communicate!

The nature of the API definition depends on which IPC mechanism you are using. For example, if are using messaging then the API consists of the message channels, the message types and the message formats. If you are using HTTP then the API consists of the URLs, the HTTP verbs and the request and response formats. Later on I describe how to define interfaces in more detail.

[4] www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10

3.1.2 Evolving APIs

APIs invariably change over time as new features are added and perhaps old features are removed. In a monolithic application it is relatively straightforward to change an API and update all the callers. If you are using a statically typed language, the compiler helps by giving a list of compilation errors. The only challenge, perhaps, is the scope of the change. It might take a long time to change a widely used API.

In a microservices-based application changing a service's API is a lot more difficult. A service's clients are other services, which might be developed by other teams. The clients might even be other applications outside of the organization. You usually cannot force all clients to upgrade in lock-step with the service. Also, since modern applications are usually never down for maintenance you will typically perform a rolling upgrade of your service so both old and new versions of a service will be running simultaneously. It is important to have strategy for dealing these challenges.

How you handle a change to an API depends on the nature of the change.

Making backwards compatible changes

Some changes are minor and backwards compatible with the previous version. Examples of backwards compatible changes include adding an optional attribute to a request and adding an attribute to a response. It makes sense to design clients and services so that they observe Robustness Principle. A client that was written to use an older API should continue to work with the new version of the service. The service provides default values for the missing request attributes. Similarly, the client should ignore any extra response attributes. It's important to use an IPC mechanism and a messaging format that supports the Robustness principle.

Making breaking changes

Sometimes, however, you must make major, incompatible changes to an API. Since you can't force clients to upgrade immediately, a service must support older versions of an API for some period of time. If you are using an HTTP-based mechanism such as REST, then one approach is to embed the version number in the URL. Each service instance might handle multiple versions simultaneously. Alternatively, you could deploy different instances for each version.

Use semantic versioning

The Semantic Versioning specification[5] is a useful guide to versioning APIs. It is a set of rules that specify how version numbers are used and incremented. Semantic versioning was originally intended to be used for versioning of software packages. However, you can use it for versioning APIs in a distributed system.

The Semantic Versioning Specification (Semvers) requires a version number to consist of three parts, `MAJOR.MINOR.PATCH`. You must increment each part of a version number as follows:

[5] http://semver.org/

- `MAJOR` - when you make an incompatible change to the API
- `MINOR` - when you make backwards compatible enhancements to the API
- `PATCH` - when you make a backwards compatible bug fix

There are a couple of places you can use the version number in an API. If you are implementing a REST API, you can, as I mentioned earlier, use the major version of the as the first element of the URL path. Alternatively, if you are implementing a service that uses messaging, then you can include the version number in the messages that it publishes. The goal is properly version APIs and to evolve them in a controller fashion. Let's now look at the different IPC mechanism.

3.2 About inter-process communication mechanisms

There are lots of different IPC technologies to choose from. Services can use synchronous request/response-based communication mechanisms such as HTTP-based REST or gRPC. Alternatively, they can use asynchronous, message-based communication mechanisms such as AMQP or STOMP. There are also a variety of different messages formats. Services can use a human readable, text-based formats such as JSON, or XML. Alternatively, they could use a more efficient, binary format such as Avro, or Protocol Buffers. But before getting into the details let's first look at how services interact.

3.2.1 Interaction styles

When you are selecting an IPC mechanism for a service, it is useful to first think about how clients and services. There are a variety of client-service interaction styles. They can be categorized in two dimensions. The first dimension whether the interaction is one-to-one or one-to-many:

- one-to-one - each client request is processed by exactly one service
- one-to-many - each request is processed by multiple services

The second dimension is whether the interaction is synchronous or asynchronous:

- Synchronous - the client expects a timely response from the service and might even block while it waits
- Asynchronous - the client doesn't block and the response, if any, isn't necessarily sent immediately

The following table shows the various interaction styles

	one-to-one	**one-to-many**
synchronous	Request/Response	-
asynchronous	Request/Async response One way (a.k.a. notifications)	Publish/Subscribe Publish/Async responses

There are the following kinds of one-to-one interactions:

- Request/Response - a service client makes a request to a service and waits for a

response. The client expects the response to arrive in a timely fashion. It might event block while waiting. This is an interaction style that generally results in services being tightly coupled.

- Request/Async response - a service client sends a request to a service, which replies asynchronously. The client does not block while waiting since the service might not send the response for a long time.
- One way requests (a.k.a. notifications) - a service client sends a request to a service but no reply is expected or sent.

It is important to remember that the synchronous Request/Response interaction style mostly orthogonal to IPC technologies. A service can, for example, interact with another service using Request/Response style interaction with either REST or messaging. In other words, even if two services are communicating using a message broker the client service might be blocked waiting for a response. It doesn't necessarily mean they are loosely coupled. That is something I revisit later in this chapter when discussing the impact of inter-service communication of availability.

There are following kinds of one-to-many interactions:

- Publish/subscribe - a client publishes a notification message, which is consumed by zero or more interested services
- Publish/async responses - a client publishes a request message, and then waits for a certain amount of time for responses from interested services

Each service will typically use a combination of these interaction styles. For some services, a single IPC mechanism will be sufficient. Other services might need to use a combination of IPC mechanisms.

3.2.2 Message formats

The essence of IPC is the exchange of messages. Messages usually contain data, and so an important design decision is the format of the data. The choice of message format can impact the efficiency of IPC, the usability of the API, and its evolvability. If you are using a messaging system or protocols such as HTTP then you get to pick your message format. Some IPC mechanisms, such as gRPC, which you will learn about below, might dictate the message format. In either case, it is essential to use a cross language message format. Even if you are writing your microservices in a single language today, it's likely that you will use other languages in the future. You should not, for example, use Java serialization.

There are two main categories of message formats: text and binary. Lets look at each one.

Text-based message formats

The first category are text-based formats such as JSON and XML. An advantage of these formats is that not only are they human readable but they are self describing. A JSON message is a collection of name-value pairs. Similarly, an XML message is effectively a collection of named elements and values. This format enables a consumer

of a message to pick out the values that it is interested in and ignore the rest. Consequently, many changes to the message schema can easily be backwards compatible.

The structure of XML documents is specified by an XML schema[6]. Over time the developer community has come to realize that JSON also needs a similar mechanism. One option is to use JSON Schema[7].

A downside of using text-based messages format is that the messages tend to be verbose, especially XML. Every message has the overhead of containing the names of the attributes in addition to their values. Another drawback, is the overhead of parsing text. Consequently, if efficiency and performance is important you might want to consider using a binary format.

Binary message formats

There are several different binary formats to choose from. Popular formats include Protocol Buffers and Avro. Both of these formats provide a typed IDL for defining the structure of your messages. A compiler then generates the code that serializes and deserializes the messages. You are forced to take an API first approach to service design! Moreover, if you write your client in a statically typed language then the compiler checks that it uses the API correctly.

One difference, however, between these two binary formats is that Protocol Buffers uses tagged fields, where as an Avro consumer needs to know the schema in order to interpret message. As a result, handling API evolution is easier with Protocol Buffers than with Avro. This blog post[8] is an excellent comparison of Thrift, Protocol Buffers, and Avro.

Now that we have looked at message formats let's look at specific IPC mechanisms starting with Remote Procedure Invocation (RPI), which is a synchronous protocol.

3.3 Remote Procedure Invocation (RPI)

When using a synchronous, request/reply based IPC mechanism, a client sends a request to a service. The service processes the request and sends back a response. Some clients might block waiting for a response and others might have a reactive, non-blocking architecture. However, unlike when using messaging, the client assumes that the response will arrive in a timely fashion.

There are numerous protocols to choose from In this section, I describe REST and gRPC. I describe how to improve the availability of your services by properly handling partial failure. I explain why a microservice-based application must use a service discovery mechanism. Let's first take a look at REST.

[6] http://www.w3.org/XML/Schema
[7] json-schema.org/
[8] martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

3.3.1 Using REST

Today, it is fashionable to develop APIs in the RESTful style. REST is an IPC mechanism that (almost always) uses HTTP. Roy Fielding, the creator of REST, defines REST as follows:

> *REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.*
>
> *-- https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*

A key concept in REST is a resource, which typically represents a single business object such as a Customer or Product or a collection of business objects. REST uses the HTTP verbs for manipulating resources, which are referenced using a URL For example, a GET request returns the representation of a resource, which is often in the form of an XML document or JSON object although other formats such as binary can be used. A POST request creates a new resource and a PUT request updates a resource.

Many developers claim their HTTP-based APIs are RESTful. However, as Roy Fielding describes in a blog post[9] , not all of them actually are. To understand why, let's take a look the REST maturity model.

The REST maturity model

Leonard Richardson (no relation) defines a very useful maturity model for REST[10] that consists of the following levels.

- Level 0 - clients of a level 0 service, invoke the service by making HTTP POST requests to its sole URL endpoint. Each request specifies the action to perform, the target of the action (e.g. the business object) and any parameters.
- Level 1 - a level 1 service supports the idea of resources. To perform an action on a resource, a client makes a POST request that specifies the action to perform and any parameters.
- Level 2 - a level 2 service uses HTTP verbs to perform actions: GET to retrieve, POST to create, and PUT to update. The request query parameters and body, if any, specifies the actions parameters. This enables services to leverage web infrastructure such as caching for GET requests.
- Level 3 - the design of a level 3 service is based on the terribly named HATEOAS (Hypertext As The Engine Of Application State) principle. The basic idea is that the representation of a resource returned by a GET request contains links for performing actions on that resource. For example, a client can cancel an Order using a link in the representation returned by the GET request that retrieved the order. The benefits of HATEOAS include no longer having to hardwire URLs into

[9] roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
[10] martinfowler.com/articles/richardsonMaturityModel.html

client code[11].

I'd encourage you to review the REST APIs at your organization to see which level they correspond to.

Specifying REST APIs

As I mentioned earlier in section 3.1, you must define your APIs using an IDL. Unlike, older communication protocols such as CORBA and SOAP, REST did not original have an IDL. Fortunately, the developer community has eventually rediscovered the value of an interface definition language for RESTful APIs. The main REST IDL is the Open API Specification[12], which evolved from the Swagger open source project. The Swagger project is a set of tools for developing and documenting REST APIs. It includes tools that generate client stubs and server skeletons from an interface definition.

Benefits and drawbacks of REST/HTTP

There are numerous benefits to using a protocol that is based on HTTP:

- It is simple and familiar
- You can test an HTTP API from within a browser using, for example, the Postman plugin or from the command line using curl (assuming JSON or some other text format is used)
- It directly supports Request/reply style communication.
- HTTP is, of course, firewall friendly
- It doesn't require an intermediate broker, which simplifies the system's architecture.

There are some drawbacks to using HTTP:

- It only supports the request/reply style of communication.
- It is often challenging to map the operations that you want to perform on a business object to an HTTP verb. For example, an API should use PUT for updates but there might be multiple ways to update an order, including canceling it, changing the delivery time and so on. One solution is to create a sub-resource for updating a particular aspect of a resource. Another solution is to specify a verb as a URL query parameter.
- Reduced availability - because the client and service communicate directly without an intermediary to buffer messages, they must both be running for the duration of the exchange.
- Clients must know the locations (i.e. URL) of the service instances(s). As will describe in a later chapter this is a non-trivial problem in a modern application. Clients must use what is known as a service discovery mechanism to locate service instances.

[11] www.infoq.com/news/2009/04/hateoas-restful-api-advantages
[12] www.openapis.org

REST seems to be de facto API, but gRPC is an interesting alternative to REST. Let's take a look at how it works.

3.3.2 Using gRPC

gRPC[13] is a framework for writing cross-language RPC clients and servers. It is a binary format and, as I mentioned earlier when discussing binary message formats, you are forced to take an API first approach to service design. It provides a Protocol Buffers-based IDL for defining your APIs. Protocol Buffers is Google's language-neutral mechanism for serializing structured data. You use the Protocol Buffer compiler to generate client-side stubs and server-side skeletons. The compiler can generate code for a variety of languages including Java, C#, NodeJS and GoLang. Clients and servers exchange binary messages in the Protocol buffers format using HTTP/2.

A gRPC interface consists of one or more services and request/reply message definitions. A service definition is an analogous to Java interface. It is a collection of strongly typed methods. As well as supporting simple request/reply RPC, gRPC support streaming RPC. A server can reply with a stream of messages to the client. Alternatively, a client can send a stream of messages to the server.

gRPC uses Protocol Buffers as the message format. Protocol Buffers is, as mentioned earlier, an efficient, compact, binary format. Protocol tags uses a tagged format. Each field is numbered and has a type code. A message recipient can extract the fields that it needs and skip over the fields that it doesn't recognize. As a result, gRPC enables APIs to evolve while remaining backwards compatible.

When making a synchronous REST or gRPC request to a service, there is always the possibility that the service is unavailable or is exhibiting such high latency it is essentially unusable. This situation where some but not all parts of a distributed application are unavailable is known as partial failure. It is essential that applications handle this scenario. Otherwise, the failure of one service will cascade to other services and the availability of the application will be drastically reduced.

3.3.3 Handling partial failure

In a distributed system there is the ever present risk of partial failure. Since client and services are separate processes, a service might not be able to respond in a timely way to a client's request. The service might be down because of a failure or for maintenance. Or, the service might be overloaded and responding extremely slowly to requests.

Let's imagine, for example, that service A makes an HTTP or gRPC request to service B, which is unresponsive. A naive implementation of a client would block indefinitely waiting for a response. Not only would result in a poor user experience but in many applications, it would consume a precious resource such as a thread. Eventually the calling service would run out resources and become unable to handle additional

[13] www.grpc.io/

requests. It's essential that you design your services to handle partial failures and prevent them from cascading throughout the application.

A good to approach to follow is the one described by Netflix[14]. The strategies for dealing with partial failures include:

- Network timeouts - never block indefinitely and always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.
- Limiting the number of outstanding requests from a client to a service - impose an upper bound on the number of outstanding requests that a client can make to a particular service. If the limit has been reached, then it is probably pointless to make additional requests, and those attempts should fail immediately.
- Circuit breaker pattern - track the number of successful and failed requests and if the error rate exceeds some threshold then trip the circuit breaker so that further attempts fail immediately. If a large number of requests are failing then it suggests that the service is unavailable and that sending more requests is pointless. After a timeout period, the client should try again and, if successful, close the circuit breaker.

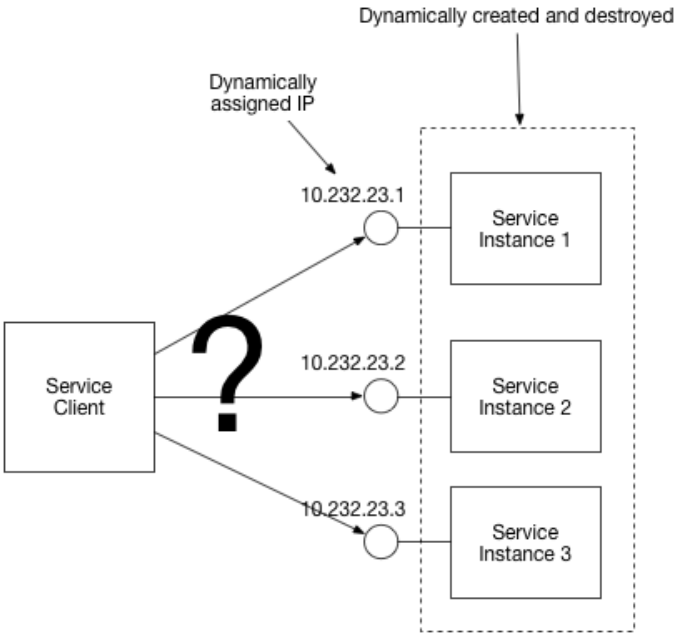
Netflix Hystrix is an open-source library that implements these and other patterns. If you are using the JVM you should definitely consider using Hystrix. And, if you are running in a non-JVM environment you should use an equivalent library. For example, the Polly library[15] is popular in the .NET community.

3.3.4 Using service discovery

Let's imagine that you are writing some code that invokes a service that has a REST API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network locations of service instances are usually static. For example, your code could read the network locations from a configuration file that is occasionally updated. However, In a modern, cloud-based microservices application, it is usually not this simple. As is shown in figure 3.1, a modern application is much more dynamic.

[14] techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html
[15] github.com/App-vNext/Polly

Figure 3.1. Service instances have dynamically assigned IP addresses.



Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures and upgrades. Consequently, your client code must use what is known as service discovery.

Overview of service discovery

As you have just seen, you cannot statically configure a client with the IP addresses of the services. Instead, an application must use a dynamic, service discovery mechanism. Service discovery is conceptually quite simple. The key component of service discovery is a service registry, which is a database of the network locations of an application's service instances.

The service discovery mechanism updates the service registry when service instances start and stop. When a client invokes a service, the service discovery mechanism queries the service registry to obtain a list of available service instances and routes the request to one of them.

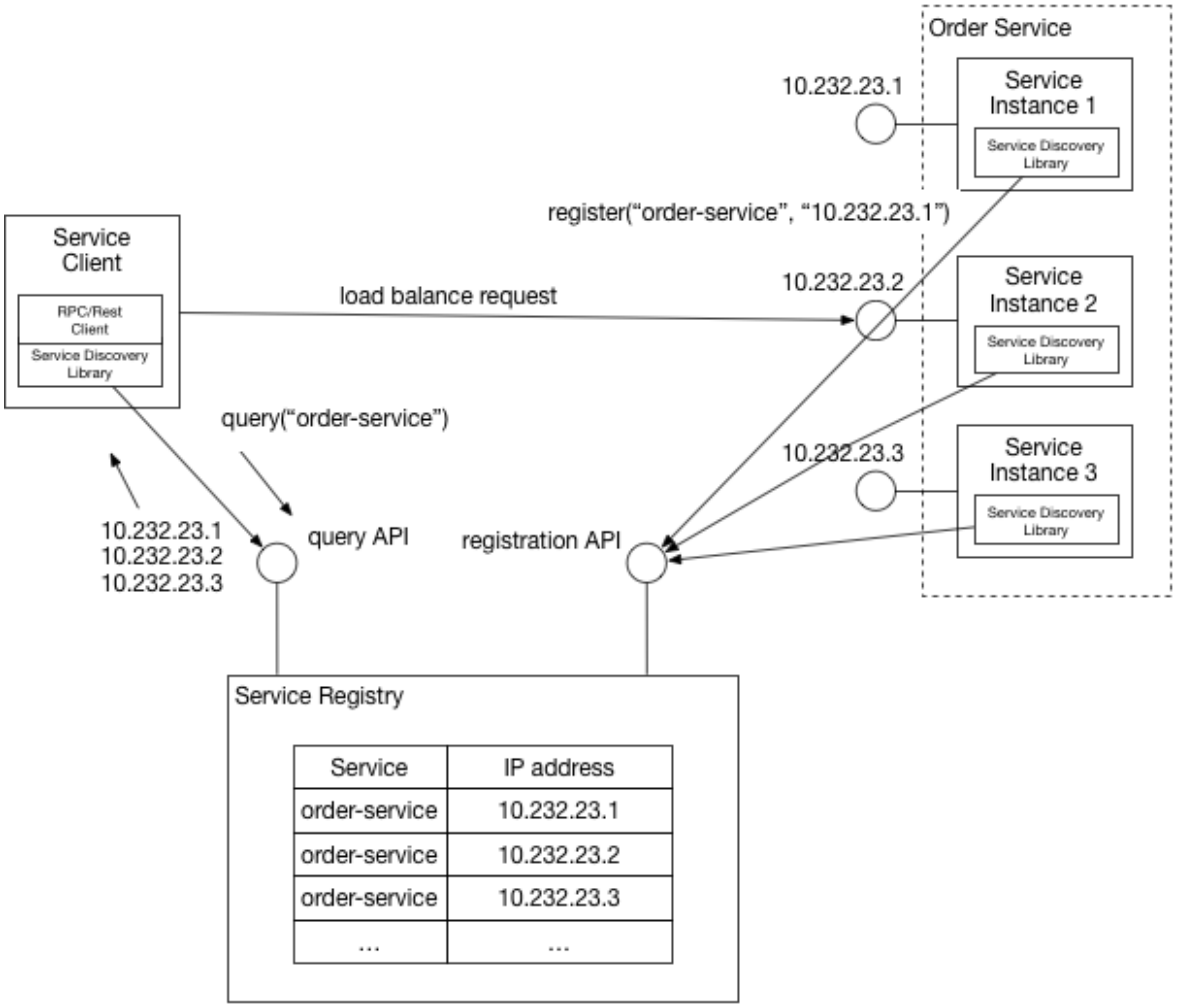
There are two main ways to implement service discovery. The first option, is for the services and their clients to interact with directly service registry. The second option is for the deployment infrastructure, which I'll describe in more detail in chapter {chapter-production}, to handle service discovery. Let's look at each option

Application-level service discovery

One way to implement service discovery is for the application's services and their clients to interact with the service registry. A service instance registers its network

location with service registry. A service client invokes a service by first querying the service registry to obtain a list of service instances. It then sends a request to one of those instances. Figure 3.2 shows how the services and the clients interact with the registry.

Figure 3.2. The service registry keeps track of the service instances. Clients query the service registry to find network locations of available service instances.



A service instance invokes the service registry's registration API to register its network location. It might also supply a health check URL, which I'll describe in more detail in chapter {chapter-production}. The health check URL is an API endpoint that the service registry invokes periodically to verify that the service instance is healthy and available to handle requests. A service registry may require a service instance to

periodically invoke a "heart beat" API in order to prevent its registration from expiring.

When a service client wants to invoke a service, it queries the service registry to obtain the a list of the service's instances. In order to improve performance, a client might cache the service instances. The service client then uses a load balancing algorithm, such a round-robin or random, to select a service instance. It then makes a request to a select service instance.

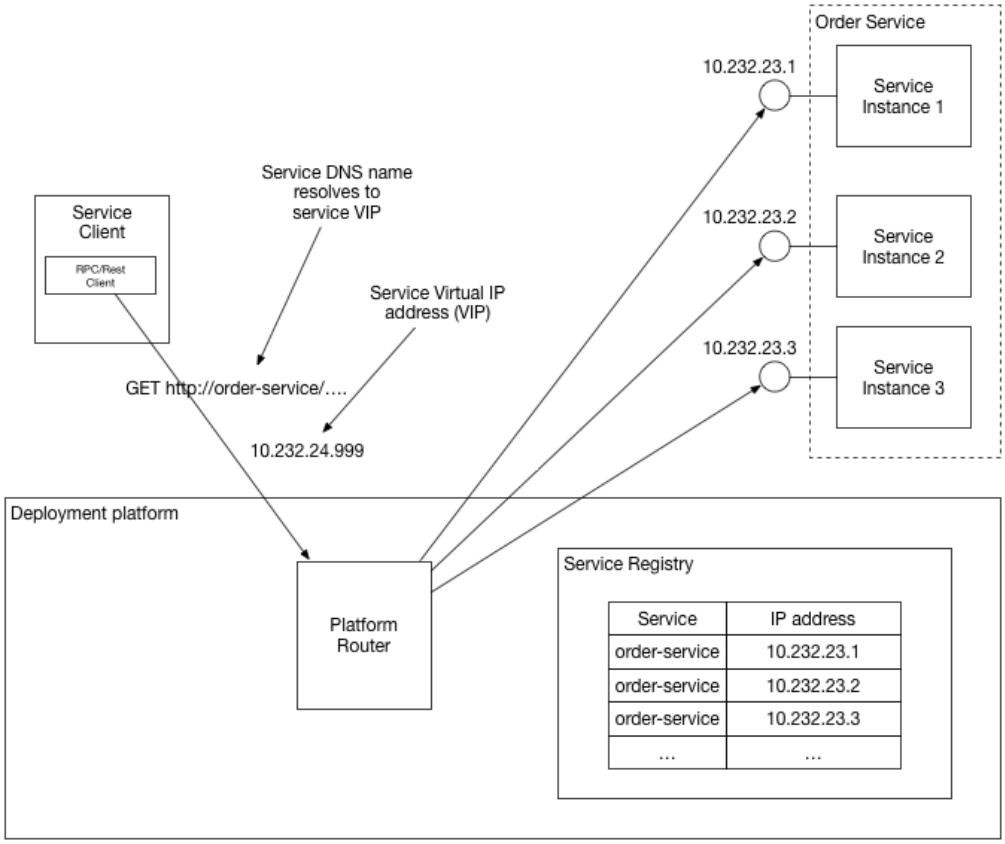
Application-level service discovery has been popularized by Netflix and Pivotal. Netflix developed and open-sourced several components: Eureka, which is a highly available service registry; the Eureka Java client; and Ribbon, which is a sophisticated HTTP client that supports the Eureka client. Pivotal have developed Spring Cloud, which is a Spring-based framework that makes it remarkably easy to use the Netflix components. Spring Cloud-based services automatically register with Eureka and Spring Cloud-based clients automatically use Eureka for service discovery.

One drawback of application-level service discovery is that you need a service discovery library for every language - and possibly framework - that you use. Spring Cloud only helps Spring developers. If you are using some other Java framework or a non-JVM language, such as NodeJS or GoLang, then you must find some other service discovery framework. Another drawback of application-level service discovery is that you are responsible for setting up and managing the service registry, which is a distraction. As a result, it is usually better to use a service discovery mechanism that is provided by the deployment infrastructure.

Platform-provided service discovery

Later in chapter {chapter-production} you will learn that many modern deployment platforms, such as Docker and Kubernetes, have a built-in service registry and service discovery mechanism. The deployment platform gives each service a DNS name and a virtual IP (VIP) address and a DNS name, which resolves to the IP address. A service client simply makes a request to the DNS name/VIP and the deployment platform automatically routes the request to one of the available service instances. As a result, service registration, service discovery and request routing are entirely handled by the deployment platform. Figure 3.3 shows how this works.

Figure 3.3. The platform is responsible for service registration, discovery and request routing



The deployment platform includes a service registry, which track the IP addresses of the deployed services. In this example, a client accesses the `Order Service` using the DNS name `order-service`, which resolves to the virtual IP address `10.1.3.4`. The deployment platform automatically load balances requests across the three instances of the `Order Service`.

The key benefit of using this approach is that all aspects of service discovery are entirely handled by the deployment platform. Neither the services nor the clients contain any service discovery code. Consequently, the service discovery mechanism is readily available to all services and clients regardless of the language or framework that they are written in.

One drawback of platform-provided service discovery is that it only supports the discovery of services that have been deployed using the platform. Let's imagine, for example, you have deployed only part of your application on Kubernetes and the rest is running in an older environment. Application-level service discovery using Eureka, for example, works across both environments, whereas Kubernetes-based service discovery only works within Kubernetes. Despite this limitation, however, I'd

recommend using platform-provided service discovery whenever possible.

Now that we have looked at synchronous IPC using REST or gRPC, let's take a look at the alternative: asynchronous, message-based communication

3.4 Asynchronous, message-based communication

When using messaging, services communicate by asynchronously exchanging messages. A service client makes a request to service by sending it a message. If the service instance is expected to reply it will do so by sending a separate message back to the client. Since the communication is asynchronous, the client does not block waiting for a reply. Instead, the client is written assuming that the reply will not be received immediately.

In this section, I describe how messaging works. I briefly compare and contrast different message brokers. I then spend some time describing several important messaging-related topics:

- Implementing request/reply using messaging
- Scaling your applications by properly implementing competing consumers
- Detecting and discarding duplicate messages
- Sending and receiving messages as part of a database transaction

Let's begin by looking at how messaging works.

3.4.1 Overview of messaging

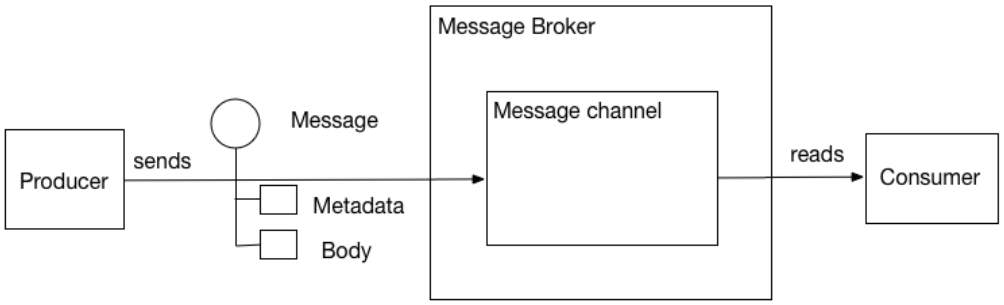
An extremely useful model of messaging is defined in the book Enterprise Integration Patterns by Gregor Hohpe. A message[16] consists of a header (a.k.a. metadata) and a message body. The header contains information such as:

- a unique message id - generated by either the sender or the messaging infrastructure
- an optional return address - the name of the message channel that a reply should be written to

The body is the data being sent and has either a text or binary format. As figure 3.4 shows, messages are exchanged over channels[17].

[16] www.enterpriseintegrationpatterns.com/Message.html
[17] www.enterpriseintegrationpatterns.com/MessageChannel.html

Figure 3.4. A producer sends a message to a consumer via message channel, which is an abstraction of the messaging mechanism provided by the message broker



Any number of producers can send messages to a channel. Similarly, any number of consumers can receive messages from a channel.

There are two kinds of channels, point-to-point[18] and publish-subscribe[19] . A point-to-point channel delivers a message to exactly one of the consumers that is reading from the channel. Service use point-to-point channels for the one-to-one interaction styles described earlier. A publish-subscribe channel delivers each message to all of the attached consumers. Service use publish-subscribe channels for the one-to-many interaction styles described above.

There are several different kinds of messages:

- Command - a message containing an command, i.e. the sender explicitly tells the receiver to do something. A command is often sent over a point-to-point channel.
- Event - a message indicating that something notable has occurred in the sender. An event is often a domain event, which represents a state change of a domain object such as an Order, or a Customer. Events are usually sent over a publish-subscribe channel.
- Document - a message that just contains data. The receiver decides how to interpret it. The reply to a command is an example of a document message. Documents are often sent over a point-to-point channel.

The approach to the microservice architecture that I describe in this book uses commands and events extensively.

3.4.2 Message brokers

Services can exchange messages directly using so-called brokerless protocols such as ZeroMQ. However, most applications use a message broker, which is an intermediary through which all communication passes. A producer gives the message to the message broker and the message broker delivers it to the consumer. An important benefit of using a message broker is that the sender does not need to know the network location

[18] www.enterpriseintegrationpatterns.com/PointToPointChannel.html
[19] www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html

of the consumer. Another benefit is that a message broker buffers messages until the consumer is able to process them.

There are many message brokers to chose from. When selecting a message broker you probably should pick one that supports a variety of programming languages. Some message brokers support standard protocols such as AMQP and STOMP, while others systems have proprietary but documented protocols. Examples of popular message brokers include ActiveMQ, RabbitMQ, Apache Kafka[20], AWS Kinesis and AWS SQS.

Each message broker uses different terminology. For example, JMS message brokers, such as ActiveMQ, are based on queues and topics; AMQP-based message brokers, such as RabbitMQ, use exchanges and queues; Apache Kafka has topics; AWS Kinesis has streams; and AWS SQS has queues. At a very high-level, they are different implementations of the message channel abstraction. The details of how each one works, however, is quite different.

Also, when selecting a message broker, there various factors to consider including:

- messaging ordering - does the message broker preserve ordering of messages?
- delivery guarantees - what kind of delivery guarantees the broker make?
- persistence - are messages persisted to disk and able to survive broker crashes?
- durability - if a consumer reconnects to the message broker will it receive the messages that were sent while it was disconnected?
- scalability - how scalable is the message broker?
- latency - what is the end to end latency?
- competing consumers - does the message broker support competing consumers?

Each broker makes different trade-offs. For example, a very low latency broker might not preserve ordering, make no guarantees to deliver messages and only store messages in memory. A messaging broker that guarantees delivery, and reliably stores messages on disk will probably have higher latency. Which kind of message broker is the best fit depends very much on your application's requirements. It is likely, however, that messaging ordering and scalability are essential.

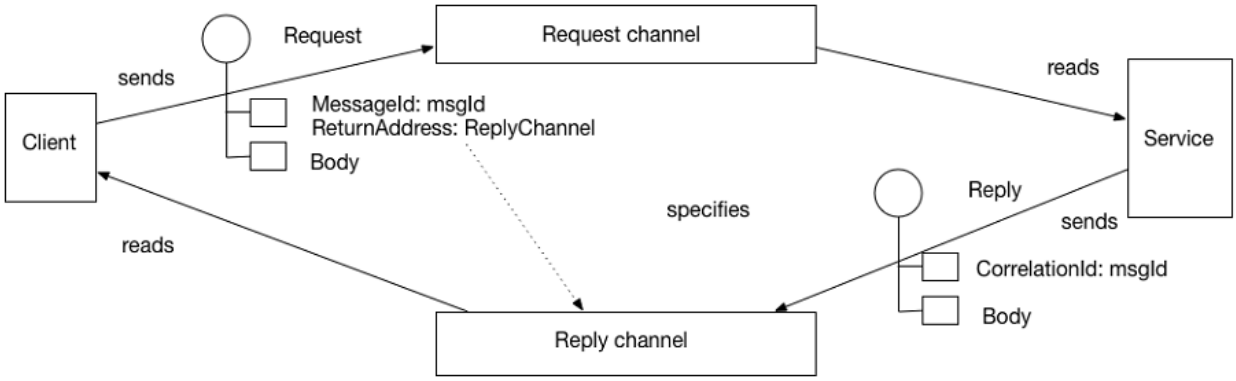
Lets now look at a few commonly encountered design issues when using messaging.

3.4.3 Implementing request/reply

Asynchronous messaging doesn't directly support request/reply style interaction. Instead, it must be implemented by the client sending a message to the service and the service sending a message to the client. Figure 3.5 shows how the client and the service interact.

[20] kafka.apache.org/

Figure 3.5. Implementing request/reply by including the reply channel and request identifier in the request message



The client must tell the service where to send a reply. It must also match reply messages to requests. Fortunately, solving these two problems is not that difficult. Each request message must contain a *reply channel* and a *request identifier*. The server writes the response message, which contains a *correlation id* that has the same value as *request identifier*, to the reply channel. The client uses the *correlation id* to match the response with the request.

3.4.4 Competing consumers

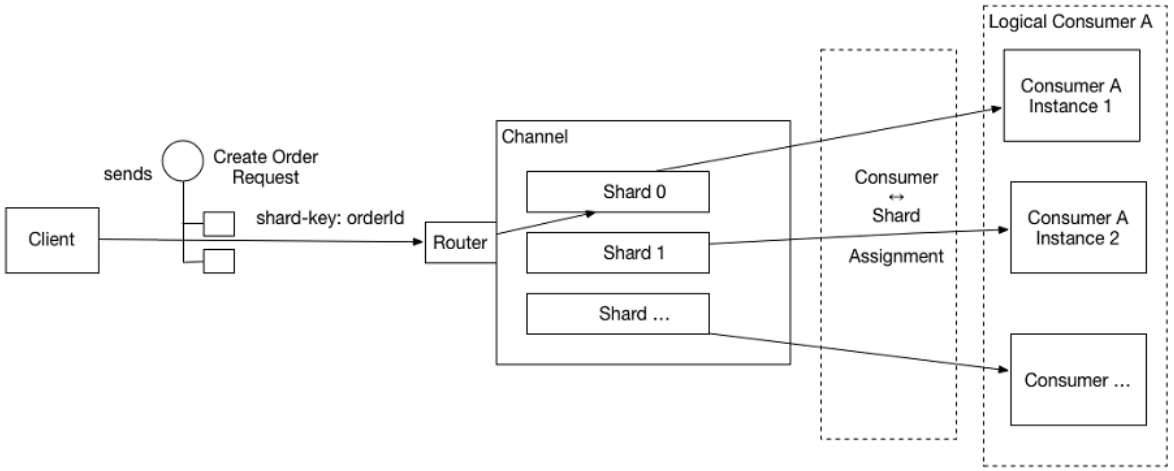
Another messaging design issue is scaling out a consumer. It is a common requirement to have multiple instances of a service into order to process messages concurrently. Moreover, even a single service instance will probably use threads to concurrently process multiple messages. Using multiple threads and service instances to concurrently process messages increases the throughput of the application. The challenge, however, with processing messages concurrently is ensuring that that each message is processed once and in order.

For example, let's imagine that there are three instances of a service reading from the same point-to-point channel and that a producer publishes `Order Created`, `Order Updated`, and `Order Cancelled` event messages sequentially. A simplistic messaging implementation could concurrently deliver each message to a different consumer. Because of delays due to network issues or garbage collections, messages might be processed out of order, which would result in strange behavior. In theory, a service instance might process the `Order Cancelled` message before another service processes the `Order Created`message!

A common solution used by modern message brokers such as Apache Kafka and AWS Kinesis is to use sharded (a.k.a. partitioned) channels. A channel consists of two or more shards, each one of which behaves as a channel. A message's metadata includes a shard key, which the message broker uses to assign the message to a particular shard/partition. Multiple instances of a service are grouped together and treated as the

same logical consumer. The message broker assigns each shard to a single consumer instance. Figure 3.6 shows how this works.

Figure 3.6. Scaling consumers by using a sharded/partitioned message channel



In the above example, each `Order` event message has the `orderId` as its shard key. Each event for a particular order is published to the same shard, which is read by a single consumer instance. As a result, those messages are guaranteed to be processed in order.

3.4.5 Handling duplicate messages

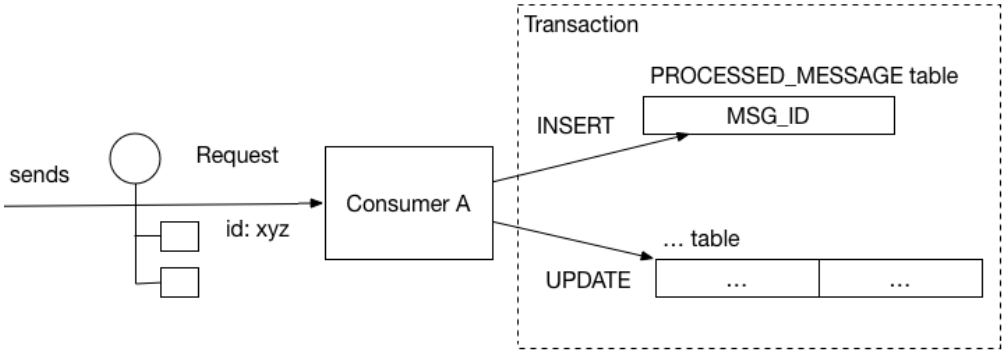
Another challenge with using a messaging, is dealing with duplicate messages. Ideally, a message broker should deliver a message only once but guaranteeing exactly once messaging is usually too costly. Instead, most message brokers promise to deliver at a message **at least** once. When the system is working normally messages are delivered only once but a failure of a client, network or message broker can result in a message being delivered multiple times. For example, a failure can cause a message acknowledgement to be lost after a consumer has processed a message and updated its database. The unacknowledged message will then be delivered again.

If the application logic that processes a message is idempotent then duplicate messages are harmless. Application logic is idempotent if it has no additional effect if called multiple times with the same input values. For instance, canceling an already cancelled order is an idempotent operation. So is creating an order with a client supplied id. Idempotent logic can be safely executed multiple times.

Unfortunately, application logic is typically not idempotent. For example, a consumer's credit card must be authorized once per order. This kind of application logic has a different effect each time it is invoked and multiple executions results in errors. The application logic must detect and discard duplicate messages. A simple solution is for a message consumer to track the messages that it has processed using the *message id* and discard any duplicates. It could, for example, store the *message*

id of each message that it consumed in a database table. Figure 3.7 shows this approach.

Figure 3.7. A consumer detects and discards duplicate messages by recording the ids of processed messages



When a consumer handles a message, it updates the database within transaction. In addition to creating and updating business entities in the database, the consumer inserts a row into a `PROCESSED_MESSAGES` table, which tracks the messages that have been already processed. If a message is a duplicate, the `INSERT` will fail and the consumer can discard the message.

3.4.6 Transactional messaging

A service often needs to publish messages as a part of transaction that includes database updates. Both the database update and the sending the message must happen within a transaction. If these operations were not done atomically then a failure could leave the system in an inconsistent state. A service might update the database, for example, but then fail to send the message.

The traditional solution is to use a distributed transaction that spans the database and the message broker. However, as you will learn in the next chapter, distributed transactions are not a good choice for modern application. Moreover, many modern brokers such as Apache Kafka do not support distributed transactions. There are a variety of ways to solve this problem and later in this chapter I describe one possible solution.

3.4.7 Defining messaging APIs

There are two aspects to designing messaging APIs. The first is defining the channels over which the communication will occur. You must define the type - point-to-point or publish-subscribe - for each channel. Sometimes a channel can be considered to belong to a service. For example, a service's request channel or reply channel is logically part of the service. Other channels are simply part of the infrastructure and are shared between services.

The second aspect of defining a messaging API is the specification of the message formats. It is essential the message formats are well defined, otherwise services will not be able to communicate. As mentioned, earlier there are various options including text-based formats such as a JSON and XML, and binary formats such as Protocol Buffers and Avro. JSON is quite popular but Avro is popular within the Apache Kafka community.

3.4.8 Benefits and drawbacks of messaging

There are many advantages to using messaging:

- loose coupling - a client makes a request simply sending a message to the appropriate channel. The client is completely unaware of the service instances. It does not need to use a discovery mechanism to determine the location of a service instance.
- message buffering - the message broker buffers messages until they can be processed. With a synchronous request/reply protocol, such as a HTTP, both the client and service must be available for the duration of the exchange. With messaging, however, messages will queue up until they can be processed by the consumer. This means, for example, that an online store can accept orders from customers, even when the order fulfillment system is slow or unavailable. The Order messages will simply queue up.
- flexible communication - messaging supports all of the interaction styles described earlier.
- explicit inter-process communication - RPC-based mechanism attempt to make invoking a remote service look the same as calling a local service. However, because of the laws of physics and the possibility of partial failure they are in fact quite different. Messaging makes these differences very explicit so developers are not lulled into a false sense of security.

There are some, however, downsides to using messaging:

- additional operational complexity - the messaging system is yet another system component that must be installed, configured and operated. Its essential that the message broker is highly available otherwise system reliability will be impacted.
- complexity of implementing request/response-based interaction - as described above request/response style interaction requires some work to implement.

Now that we have looked at using messaging based IPC, let's look at how to implement transactional messaging

3.5 Transactional messaging with the Tram framework

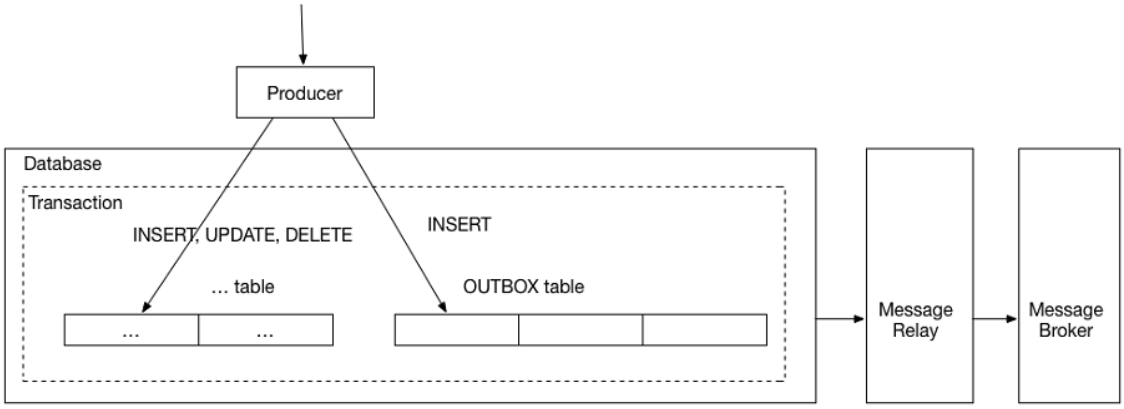
As we mentioned earlier, one challenge with using messaging, is how to send messages as part of a database transaction. For the reasons that I describe in chapter {chapter-sagas}, modern applications cannot use distributed transactions to atomically update the database and publish messages. In this section, we describe an alternative approach that reliably publishes message without using distributed transactions. You will also

learn about the Tram framework, which is a framework that I've developed to provide these capabilities in a Java application.

3.5.1 Use a database table as a message queue

A straightforward way to reliably publish messages is to use a database table as a temporary message queue. Each service that sends messages has a `OUTBOX` database table. As part of the database transaction that creates, updates and deletes business objects, the service sends messages by inserting them into the `OUTBOX` table. Atomicity is guaranteed since this is a local ACID transaction. Figure 3.8 shows how this works.

Figure 3.8. A producer reliably publishes a message by inserting it into an `OUTBOX` table as part of the transaction that updates the database



The `OUTBOX` table acts a temporary message queue. The `MessageRelay` is a component publishes the messages inserted into the `OUTBOX` to a message broker. There are a couple of different ways to move messages from the `OUTBOX` table to the message broker. Lets look at each one.

Polling the table

A very simple way to publish the messages inserted into the `OUTBOX` table is for the `MessageRelay` to poll the table for unpublished messages. It periodically queries the table:

```
SELECT * FROM OUTBOX ORDERED BY ... ASC
```

Next, the `MessageRelay` publishes those messages to the message broker, sending one to its destination message channel. Finally, it deletes those messages from the `OUTBOX` table:

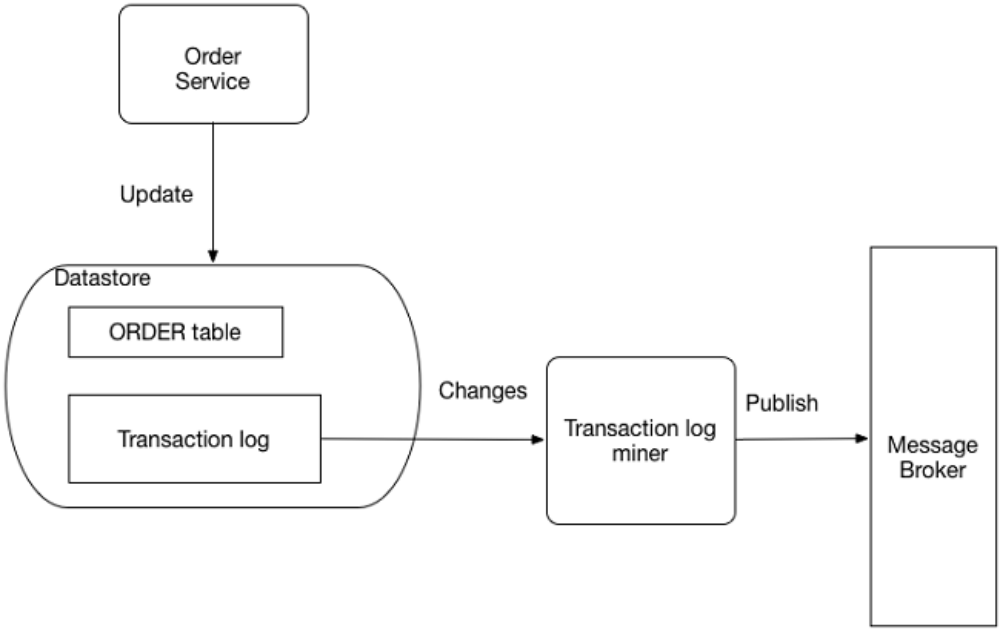
```
BEGIN
 DELETE FROM OUTBOX WHERE ID in (....)
COMMIT
```

This is a simple approach that works reasonably well at low scale. The downside is that frequently polling the database can be expensive. A more sophisticated and performant approach is to tail the database transaction log.

Transaction log tailing

A more sophisticated solution is for `MessageRelay` to tail the database transaction log (a.k.a. commit log). Every committed update by an application is represented as an entry in the database's transaction log. An application can read the transaction log and publish each change as a message to the message broker. Figure 3.9 shows how this approach works.

Figure 3.9. A service publishes messages inserted into the `OUTBOX` table by mining the database's transaction log



The `MessageRelay` reads the transaction log entries. It converts each log entry to a message and publishes that message to the message broker.

There are a few examples of this approach in use:

- LinkedIn Databus - an open-source project that mines the Oracle transaction log and publishes the changes as events. Linkedin uses Databus to synchronize various derived data stores with the system of record.
- DynamoDB streams - DynamoDB stream contains the time-ordered sequence of changes (creates, updates and deletes) made to the items in DynamoDB table in the last 24 hours. An application can read those changes from the stream and, for example, publish them as events.

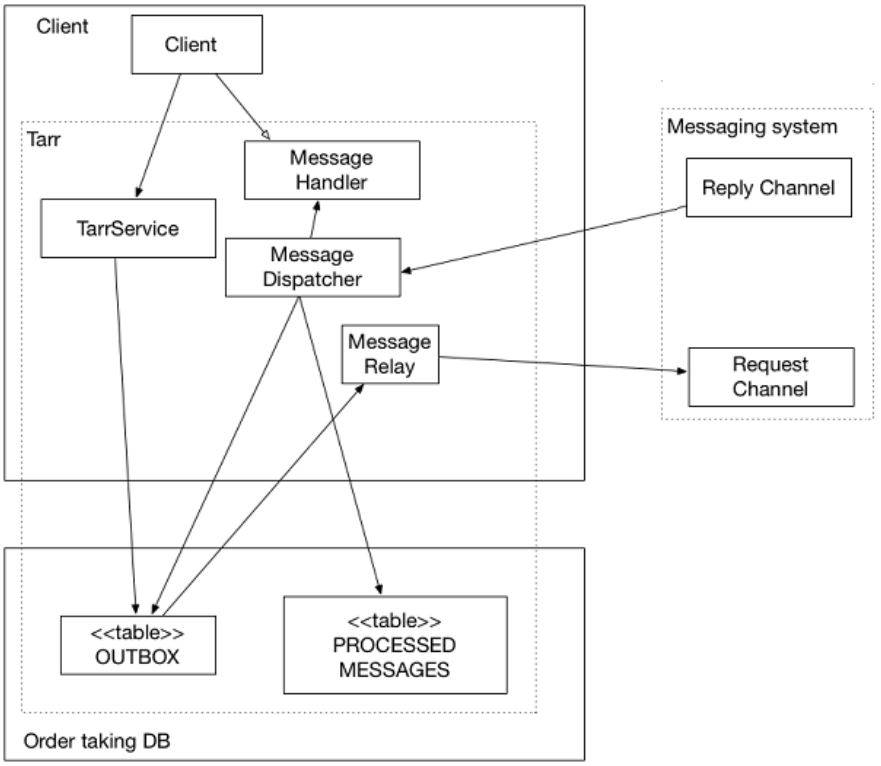
- Eventuate Local platform - an open-source project that uses the MySQL master/slave replication protocol to read changes made to an `OUTBOX` table and publish them to Apache Kafka

Although this approach is obscure, it works remarkably well. Now that we have looked at a way to reliably publish messages lets now look at a framework that does this.

3.5.2 Transactional messaging using the Tram framework

Tram is a framework that implements transactional messaging. Figure 3.10 shows the design of the framework's core. It provides an API for sending and receiving messages as part of a JDBC transaction.

Figure 3.10. The Tram framework enables a service to publish and consume messages as part of a database transaction



The Tram framework consists of Java classes and interfaces and supporting database tables. The main Java interfaces are the `MessageProducer` and `MessageConsumer`. A producer service uses the `MessageProducer` interface publish messages to message channels. A consumer service uses the `MessageConsumer` interface to subscribe to messages.

The Tram framework's two main database tables are the `OUTBOX` and

PROCESSED_MESSAGE tables. It uses the OUTBOX table to reliably publish message as part of a database transaction using the mechanism described earlier. It uses the PROCESSED_MESSAGE table to detect and discard duplicate messages. Let's look at the MessageProducer and MessageConsumer interfaces in more detail.

Sending a message using the MessageProducer interface

A service publishes a message using the MessageProducer interface.

```
java
public interface MessageProducer {

  void send(String destination, Message message);

}
```

This interface defines a send() method which has two parameters. The destination parameter specifies the destination channel. The message parameter is the message to send.

The MessageProducer publishes the message as part of a database transaction using the two step process described earlier. First, it inserts the message into an OUTBOX table. Second, it transfers the message from the OUTBOX table to the specified message channel using one of two mechanisms. It either tails the database transaction or it polls the OUTBOX table.

Subscribing to messages

A service subscribes to messages using the MessageConsumer interface.

```
public interface MessageConsumer {

  void subscribe(String subscriberId, Set<String> channels, MessageHandler
handler);
}
```

This interface defines a subscribe() method, which has the following parameters:

- subscriberId - identifies the subscriber
- channels - the channels to subscribe too
- handler - the message handler to invoke for each message

The MessageHandler interface is a functional interface that extends java.util.function.Consumer:

```
public interface MessageHandler extends Consumer<Message> {
}
```

It defines an accept() method that takes a Message parameter. A message handler performs arbitrary actions including updating a database and sending messages using the MessageProducer interface.

The Tram framework implements the following message processing logic:

```
Read message from broker
BEGIN TXN
  .. INSERT MESSAGE_ID INTO PROCESSED_MESSAGES // duplicate detection
  .. Invoke handler
  ..... Update database
  ..... Insert replies into OUTBOX table
COMMIT TXN
Acknowledge message
```

The Tram framework subscribes to specified message channels and uses the mechanism described earlier to guarantee that message handling is idempotent. The duplicate detection logic inserts the *message id* into the `PROCESSED_MESSAGES` table. If the message is a duplicate, the `INSERT` will fail with a unique key violation. If the message is not a duplicate, the Tram framework invokes the `MessageHandler`. If a failure occurs at any stage, the message broker redelivers the message to Tram framework.

Publishing and subscribing to domain events

The Tram framework also implements some higher-level concepts, which build on the core `MessageProducer` and `MessageConsumer` APIs, One high-level package is the `events` package, which enables an application to publish and consume domain events. Domain events are events that are published by an aggregate, which is a business object. I'll explain the concept of aggregate and domain event in more detail in chapter {chapter-ddd-aggregates}. If you aren't familiar with these concepts, just think of domain events as events that are published by a business object.

A service publishes a domain event using the `DomainEventPublisher` interface, which is shown in listing 3.1. This interface defines various `publish()` methods.

Listing 3.1. The Tram framework's `DomainEventPublisher` interface

```
public interface DomainEventPublisher {
  void publish(String aggregateType, Object aggregateId, List<DomainEvent>
domainEvents);
  void publish(String aggregateType, Object aggregateId,
     Map<String, String> headers, List<DomainEvent> domainEvents);
}
```

These methods take the `aggregateType`, `aggregateId` and a list of domain events as parameters. The `aggregateType` and `aggregateId` are the type and id of the aggregate that published the event. The `aggregateType` parameter identifies the message channel that the messages are written to. The `aggregateId` parameter is used as the partition key, if the message channel is partitioned/sharded. This ensures that events published by a given aggregate are consumed sequentially. When using Apache Kafka, for example, the `aggregateType` is the name of the Kafka topic and the `aggregateId` is the message key. One method has a `headers` parameter that allows the caller to publish an event message with additional headers. The `DomainEventPublisher` serializes the domain events and publishes them as messages using the `MesssageProducer` interface.

A service consumes events using the `DomainEventDispatcher` class.

`DomainEventDispatcher` uses the `MessageConsumer` interface to subscribe specified events. It dispatches each event to the appropriate handler method. Listing 3.2 shows `DomainEventDispatcher's` constructor.

Listing 3.2. The Tram framework's `DomainEventDispatcher` class which dispatches event messages to event handler methods

```
public class DomainEventDispatcher {
    public DomainEventDispatcher(String eventDispatcherId,
                DomainEventHandlers eventHandlers,
                ...) {
...
}
```

This class has a constructor that has the following parameters:

- `eventDispatcherId` - the id of the durable subscription
- `eventHandlers` - the application's event handlers

An `DomainEventHandlers` is a collection of event handler methods. An event handler has a single parameter of type `DomainEventEnvelope`, which contains the domain event and event metadata.

Here is an example of an event consumer, which I describe in more detail in chapter {chapter-ddd-aggregates}. It defines a `domainEventHandlers()` method, which returns `DomainEventHandlers` that maps event types to the corresponding handler method.

Listing 3.3. An example event consumer for Tram domain events

```
public class RestaurantOrderEventConsumer {

  public DomainEventHandlers domainEventHandlers() {                    (1)
    return DomainEventHandlersBuilder

.forAggregateType("net.chrisrichardson.ftgo.restaurantservice.Restaurant")
            .onEvent(RestaurantMenuRevised.class, this::reviseMenu)
            .build();
  }

  public void reviseMenu(DomainEventEnvelope<RestaurantMenuRevised> de) {   (2)
```

(1) Map events to event handlers
(2) An event handler for the `RestaurantMenuRevised` event

In this example, the `reviseMenu()` method handles the `RestaurantMenuRevised` event.

Sending and processing commands

As well supporting domain events, the Tram framework also implements request/reply, which I described earlier in section XYZ. A client can send a command message to a

service using the `CommandProducer` interface, which is shown in listing 3.4. This interface defines a `sendCommand()` method, which sends a command message using the `MessageProducer` interface.

Listing 3.4. A `CommandProducer` send a command message using the `MessageProducer`

```
public interface CommandProducer {
  String send(String channel, Command command, String replyTo, Map<String, String>
headers);
}
```

A service consumes command messages using the `CommandDispatcher` class. `CommandDispatcher` uses the `MessageConsumer` interface to subscribe specified events. It dispatches each command message to the appropriate handler method. Listing 3.5 shows `DomainEventDispatcher's` constructor.

Listing 3.5. The Tram framework's `CommandDispatcher` class which dispatches command messages to handlers

```
public class DomainEventDispatcher {

  public CommandDispatcher(String commandDispatcherId,
          CommandHandlers commandHandlers) {
  ...
}
```

This class has a constructor that has the following parameters:

- `commandDispatcherId` - the id of the durable subscription
- `commandHandlers` - the application's event handlers

An `CommandHandlers` is a collection of command handler methods. A command handler has a single parameter, a `CommandMessage`, which contains the command message and its metadata.

Here is an example of an command consumer, which I describe in more detail in chapter {chapter-sagas}. It defines a `commandHandlers()` method, which returns a `CommandHandlers` that maps command types to the corresponding handler method.

```
public class OrderCommandHandlers {


  public CommandHandlers commandHandlers() {                                    (1)
    return CommandHandlersBuilder
          .fromChannel("orderService")
          .onMessage(ApproveOrderCommand.class, this::approveOrder)
          ...
          .build();
  }

  public Message approveOrder(CommandMessage<ApproveOrderCommand> cm) {
```

```
    ApproveOrderCommand command = cm.getCommand();
    ...
  }
```

❶ Route a command message to the appropriate handler method

In this example, the `approveOrder()` method handles the `ApproveOrderCommand` command. It invokes the business logic to update the specified order and returns the reply message, which is sent back to the client that sent the command.

As you have seen, the Tram framework enables an application to use transactional message. It provides a low-level API for sending and receiving message transactionally. It also provides two higher-level APIs. The first is an API for publishing and consuming domain events. The second is an API for sending and processing commands. Let's look at an service design approach that uses asynchronous messaging to improve availability.

3.6 Using asynchronous messaging to improve availability

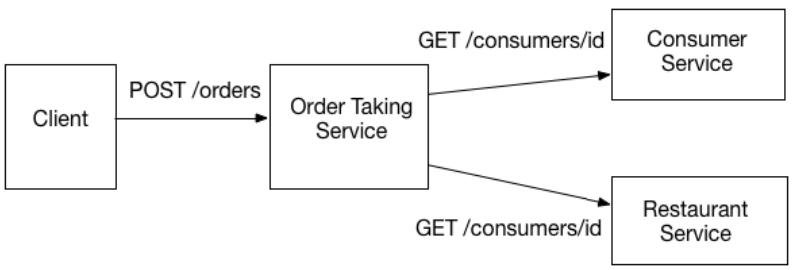
As you have just seen, there are a variety of IPC mechanisms with different trade-offs. One particular trade-off is how your choice of IPC mechanism impacts availability. In this section, you will learn that synchronous communication with other services as part of request handling reduces application availability. As a result, you should design your services to use asynchronous messaging whenever possible. Lets first look at the problem with synchronous communication and how it impacts availability.

3.6.1 Synchronous communication reduces availability

REST is an extremely popular IPC mechanism. You might be tempted to use it for inter-service communication. The problem with REST, however, is that it is synchronous protocol. An HTTP client must wait for the service to send a response. Whenever services communicate using a synchronous protocol, the availability of the application is reduced.

To understand why, let's imagine that the services involved in creating an REST APIs as shown in figure 3.11.

Figure 3.11. The `Order Service` invokes other services using REST.



The sequence of steps for creating an order is as follows:

1. Client makes a HTTP POST /orders request to the `Order Taking` service
2. `Order Taking` service retrieves consumer information by making an HTTP GET /consumers/id request to the `Consumer` service
3. `Order Taking` service retrieves restaurant information by making an HTTP GET /restaurant/id request to the `Restaurant` service
4. `Order Taking` validates the request using the consumer and restaurant information.
5. `Order Taking` creates an Order
6. `Order Taking` sends an HTTP response to the client

Since these services use HTTP, they must all be simultaneously available in order for the FTGO application to process `CreateOrder` request. The FTGO application could not create orders if any one of these three services is down. Mathematically speaking the availability of a system operation is the product of the availability of the services that are invoked by that operation. If the `Order Taking` service and the two services that it invokes are 99.5% available, then the overall availability is $99.5\%^3 = 98.5\%$, which is significantly less. Each additional service that participates in handling a request further reduces availability.

This problem isn't specific to REST-based communication. Availability is reduced whenever a service can only respond to its client after receiving a response from another. This problem exists even if services communicate using request/reply style interaction over asynchronous messaging. For example, the availability of the `Order taking` service would be reduced if it sent a message to the `Consumer`service via a message broker and then waited for a response. If you want to maximize availability then you must minimize the amount of synchronous communication. Lets look at how to do that.

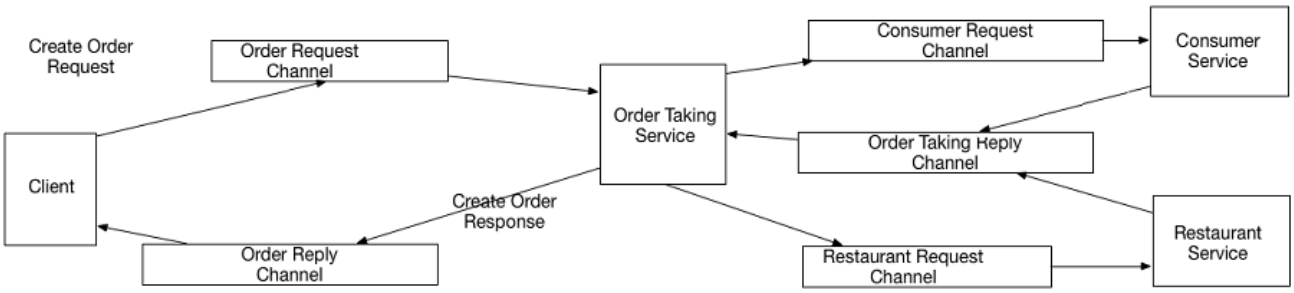
3.6.2 Eliminating synchronous interaction

There are a few different ways to reduce the amount of synchronous communication with other services while handling synchronous requests. One solution is to avoid the problem entirely by defining services that only have asynchronous APIs. That, however, is not always possible. Public APIs are, for example, commonly RESTful. Services are sometimes, therefore, required to have a synchronous APIs. Fortunately, there are ways to handle synchronous requests without making making synchronous requests. Lets look at the options.

Use asynchronous interaction styles

Ideally, all interactions should be done using the asynchronous interaction styles described earlier in this chapter. For example, lets imagine that clients' of the FTGO application used an asynchronous *request/asynchronous response* style of interaction to create orders. A client creates an order by sending a request message to the `Order Taking` service. This service then asynchronously exchanges messages with other services and eventually sends a reply message to the client. Figure 3.12 shows the design.

Figure 3.12. The FTGO appplication has higher availability if its services communicate using asynchronous messaging instead of synchronous calls



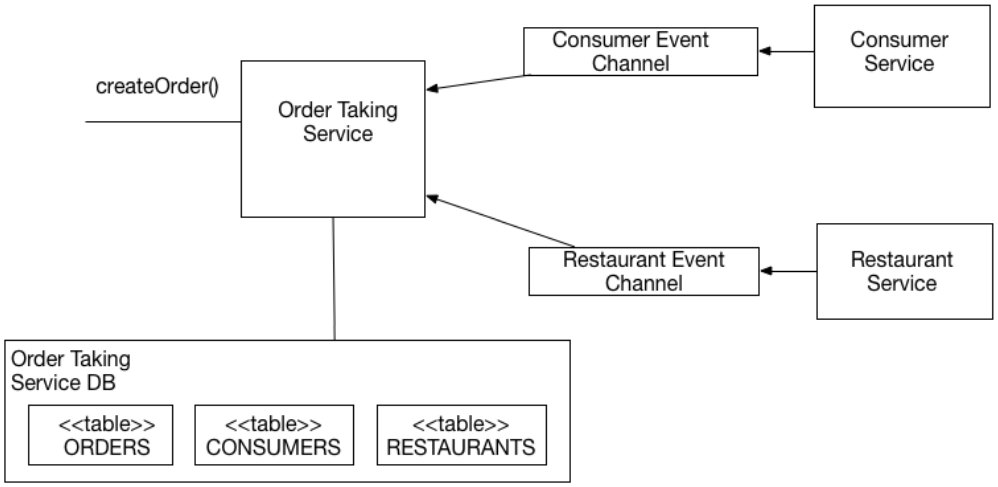
The client and the services communicate asynchronously by sending messages via messaging channels. No participant in this interaction is ever blocked waiting for a response.

Such an architecture would be extremely resilient, since the message broker buffers messages until they can be consumed. The problem, however, is that services often must an external API that uses a synchronous protocol such as REST and must respond to requests immediately. If a service has this requirement then one option is to replicate data.

Replicate data

One way to minimize synchronous requests during request processing is to replicate data. A service maintains a copy (a.k.a. replica) of the data that it needs when processing requests. It keeps the replica up to date by subscribing to events published the services that own the data it is updated. For example, the `Order Taking` service could maintain a replica of data from the `Consumer` and `Restaurant`services. This enables the `Order Taking` service to validate a request and create an order without having to interact with other services. Figure 3.13 shows the design.

Figure 3.13. The `Order Service` is self-contained because it has replicas of the consumer and restaurant data



The `Consumer` and `Restaurant` services publish events whenever their data changes. The `Order Taking` service subscribes to those events and updates its replica.

In some situations, replicating data is a useful approach and we discuss this option in more detail in chapter XYZ. However, one drawback of replication is that it can sometimes requires the replication of large amounts of data, which is inefficient. For example, it is not practical for the `Order Taking` service to maintain a replica of the data owned `Consumer` and `Restaurant` services. Another drawback of replication is that it doesn't solve the problem of how a service updates data owned by other services. When a service can't use replication then another solution is to defer interacting with other services until after it responds to its client. Lets look at how that works.

Finish processing after returning a response

Another way to eliminate synchronous communication while during request processing is for a service to handle a request as follows:

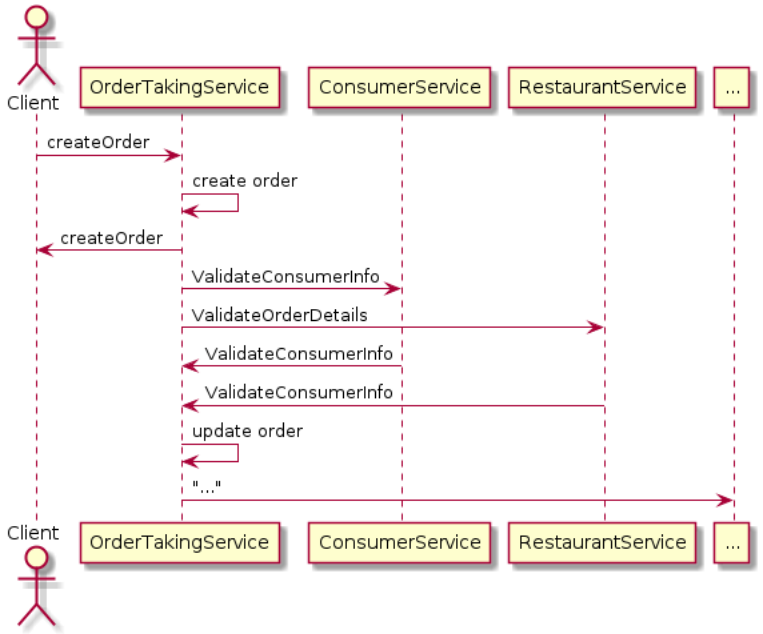
1. Validate the request using only the data available locally
2. Update its database
3. Returning a response to its client.

While handling a request, it does not synchronously interact with any other services. Any further request processing is done afterwards by asynchronously exchanging messages with other services. This approach ensures that the services are loosely coupled. Also, as you will learn in the next chapter, it enables a service to update data owned by other services.

If, for example, the `Order Taking` service used approach then it creates an order in an "unvalidated" state and then validates the order asynchronously by exchanging

messages with other services. Figure 3.14 shows what happens when the `createOrder()` operation is invoked.

Figure 3.14. The `Order Service` accepts an order without invoking any other service. It then asynchronously validates the order by exchanging messages with other services.



The sequence of events is as follows:

1. The `Order Taking` service creates an Order in a NEEDS_VALIDATION state
2. The `Order Taking` service returns a response containing the order id
3. The `Order Taking` service sends a `ValidateConsumerInfo` message to the `Consumer Service`
4. The `Order Taking` service sends a `ValidateOrderDetails` message to the `Restaurant Service`
5. The `Consumer Service` receives an `ValidateConsumerInfo` message, verifies the consumer can place an order and sends an `ConsumerValidated` message to the `Order Taking` service
6. The `Restaurant Service` receives an `ValidateOrderDetails` message, verifies the menu item are valid and that the restaurant can deliver to order's delivery address and sends an `OrderDetailsValidated`message to the `Order Taking` service
7. The `Order Taking` service receives the `ConsumerValidated` and `OrderDetailsValidated` and changes the state of the order to unauthorized
8. …

After the Order has been validated, the `Order Taking` service completes the rest of the `Order Creation` process, which I describe in the next chapter. What is nice about this approach is that even if the `Consumer` service, for example, is down, the `Order taking` service still creates orders and responds to its clients. Eventually, the `Consumer` service will come back up and process any queued messages and orders will be validated.

The drawback of a service responding before fully processing a request is that it makes the client more complex. For example, the `Order Taking` service makes minimal guarantees about the state of newly created order when returns a response. It simply creates the order and returns immediately. It does this before validating the order and authorizing the consumer's credit card. The `createOrder()` system operation has a much weaker set of post-conditions. Consequently, in order for the client to know whether the order was successfully created either it must periodically poll or the `Order Taking` service must send it a notification message. As complex as it sounds, in many situations this is the preferred approach especially since it also addresses the distributed transaction management issues we discuss in the next chapter.

3.7 Summary

- IPC has an essential role in a microservice architecture
- There are numerous IPC technologies, each with different trade-offs
- Sending messages as part of a database transaction must use the database as a temporary message queue
- Services should ideally communicate asynchronously in order to increase availability

4 Managing transactions with sagas

This chapter covers:

- Why distributed transactions are not a good fit for modern applications
- How to use sagas to maintain data consistency in a microservice architecture
- How to design saga-based business logic

Transactions are an essential ingredient of every enterprise application. Without transactions it would be impossible to maintain data consistency. ACID transactions greatly simplify the job of the developer by providing the illusion that each transaction has exclusive access to the data. The challenge with transactions in a microservice service architecture is that some system operations read and write update data owned by multiple services. For example, as we described in the previous chapter, the `createOrder()` operation spans numerous services including `Order Service`, `Restaurant Order Service`, and `Accounting Service`. Operations such as these need a transaction management mechanism that works across services.

I have already mentioned in chapter {chapter-decomposition} that the traditional approach to distributed transaction management is not a good choice for modern applications. In this chapter, you will learn why. I describe an alternative way to maintain data consistency known as a saga, which is a message-driven sequence of local transactions. You will learn how to develop saga-based business logic for a microservices-based application. Lets begin by looking at the challenges of transaction management in the microservice architecture.

4.1 Transaction management in a microservice architecture

Almost every request handled by an enterprise application is executed within a database transaction. Enterprise application developers use frameworks and libraries that simplify transaction management. Some frameworks and libraries provide a programatic API for explicitly beginning, committing and rolling back transactions. Other frameworks, such as the Spring framework, provide a declarative mechanism. Spring provides an `@Transactional` annotation that arranges for method invocations to be automatically executed within a transaction. As a result, it is straightforward to write transactional business logic.

Or, to be more precise, transaction management is straightforward in a monolithic application that accesses a single database. A more complex monolithic application might use multiple databases and message brokers. What's more, in a microservice architecture transactions span multiple services, each of which has its own database. In this situation, the application must use a more elaborate mechanism to manage transactions. And, as you will learn, the traditional approach of using distributed transactions is not viable option for modern applications. Instead, a microservice-services application must use what are known as sagas. But, before I explain the concept of a saga, let's first look at why transaction management is challenging in a microservice architecture.

4.1.1 The need for 'distributed transactions' in a microservice architecture

Let's imagine that you the FTGO developer responsible for implementing the `createOrder()` system operation. As I described in chapter {chapter-decomposition}, this operation must verify that the consumer can place an order, verify the order details, authorize the consumer's credit card and create an `Order` in the database. It is relatively straightforward to implement this operation in the monolithic FTGO application. All of the data required to validate the order is readily accessible. What's more, you can use an ACID transaction to ensure data consistency. You might simply use Spring's `@Transactional` annotation on the `createOrder()` service method.

In contrast, implementing the same operation in a microservice architecture is much more complicated. The needed data is scattered around multiple services. The `createOrder()` operation must access numerous services including `Consumer`, `Restaurant Order`, and `Accounting Service`. Since each service has its own database, you will need to use a mechanism to maintain data consistency across those databases.

4.1.2 The trouble with distributed transactions

The traditional approach to maintaining data consistency across multiple services, databases or message brokers is to use distributed transactions. The de facto standard for distributed transaction management is X/Open Distributed Transaction Processing (DTP) Model (X/Open XA)[21]. XA uses two-phase commit (2PC) to ensure that all

[21] en.wikipedia.org/wiki/X/Open_XA

participants in a transaction either commit or rollback. An XA compliant technology stack consists of XA compliant databases and message brokers, database drivers, and messaging APIs and an inter-process communication mechanism that propagates the XA global transaction id. Most SQL databases are XA compliant, as are some message brokers. Java EE applications can, for example, use JTA to perform distributed transactions.

As simple as this sounds, there are a variety of problems with distributed transactions. One problem is that many modern technologies including NoSQL databases such as MongoDB and Cassandra do not support them. Also, distributed transactions are not supported by modern message brokers such as RabbitMQ or Apache Kafka. As a result, if you insist on using distributed transactions then you are unable to use many modern technologies.

Another problem with distributed transactions that since they are a form of synchronous IPC they reduce availability. In order for a distributed transaction to commit, all of the participating services must be available. As described earlier, the availability is the product of the availability of all of the participants in the transaction. If a distributed transaction involves two services that are 99.5% available, then the overall availability is 99%, which is significantly less. Each additional service that is involved in a distributed transaction further reduces availability. There is even Eric Brewer's CAP theorem[22], which states that a system can only have two of the following three properties: consistency, availability and partition tolerance. Today, architects prefer to have a system that is available instead one that is consistency.

On the surface, distributed transactions are very appealing. From a developer's perspective, they have the same programming model as local transactions. However, because of the above problems distributed transactions are not a viable technology for modern applications. When discussing message systems in chapter 3 we showed how to deal with these issues and send messages as part of a database transaction without using a distributed transactions. To solve the more complex problem of maintaining data consistency in a microservice architecture an application must use a different mechanism, which builds on the concept of loosely coupled, asynchronous services: sagas.

4.1.3 Using sagas to maintain data consistency

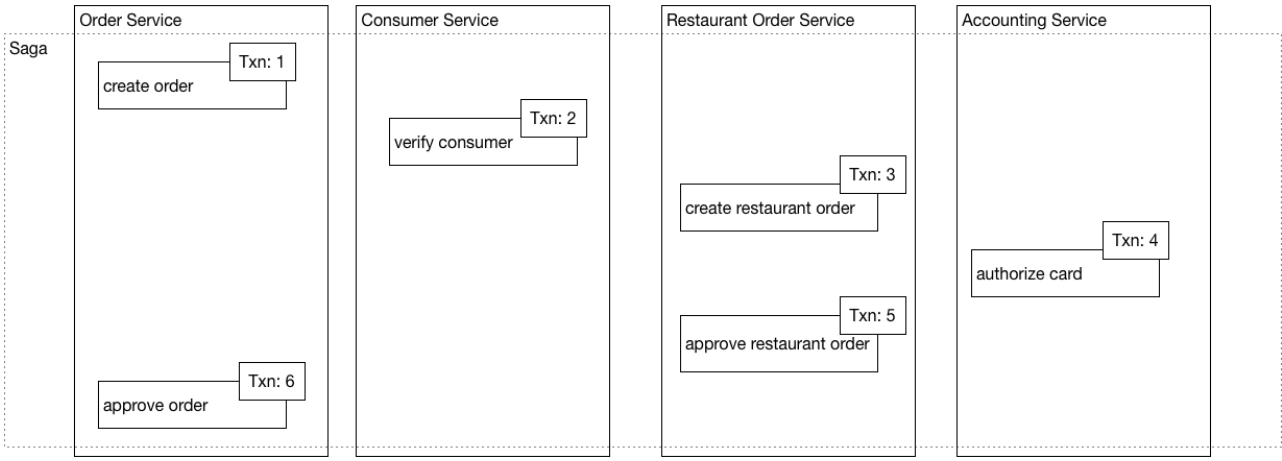
Sagas are a mechanism for maintaining data consistency that avoids the problems of distributed transactions. A saga is a sequence of local transactions. Each local transaction updates data within a single service. It uses the familiar ACID transaction frameworks and libraries mentioned earlier. The local transaction is initiated by the external request corresponding to the system operation. Each subsequent step is triggered by the completion of the previous step.

For example, the `Order Service` implements the `createOrder()` operation using the saga shown in figure 4.1. The first local transaction is initiated by the external request

[22] The CAP theorem, en.wikipedia.org/wiki/CAP_theorem

to create an order. The other five local transactions are each triggered by completion of the previous one.

Figure 4.1. Creating an `Order` using a saga



This saga consists of the following local transactions:

1. Order Service: create an `Order` in a `CREATE_PENDING` state
2. Consumer Service: verify that the consumer can place an order
3. Restaurant Order Service: validate order details and create a `Restaurant Order`
4. Accounting Service: authorize consumer's credit card
5. Restaurant Order Service: change the state of the `Restaurant Order` to `APPROVED`
6. Order Service: change state of the Order to `APPROVED`

As you will learn below, the services that participate in a saga communicate using messages. A service publishes a message when a local transaction completes in order to trigger the next step in the saga. Not only does using messaging ensure the saga participants are loosely coupled but it also guarantees that a saga completes. That's because if the recipient of a message is temporarily unavailable, the message broker buffers the message until it can be delivered.

Sagas have very different characteristics than ACID transactions, which make development more difficult. Unlike an ACID transaction, a saga is not atomic. The effects of each of a saga's local transactions are visible as soon as that transaction commits. As a result, developers must write business logic that can cope with eventually consistent data. For example, the `Order Service` must handle the scenario where a user attempts to cancel an order that is still being validated. As a result, the business logic is more complex.

Another challenge with using sagas, is that unlike with ACID transactions, rollback doesn't happen for free. Instead, the developer must write code to explicitly rollback a saga by undoing the changes made previously using what are called compensating

transactions. For example, the `CreateOrder` saga must implement a compensating transaction that cancels the order if, for example, it was found to be invalid. Although the business logic of compensating transactions is usually straightforward, they add to overall complexity of the business logic.

As you can see, using sagas involves solving some tricky business logic design issues. But before investigating how to tackle those problems let's look at the mechanics of reliably invoking and sequencing saga transactions.

4.2 Designing a saga's sequencing logic

A saga is initiated by system operation that needs to update data in multiple services. The saga's sequencing logic selects and executes the first local transaction. Once that transaction completes, the saga's sequencing logic selects and executes the next transaction. This process continues until the saga completes. If a local transaction fails because it would violate a business rule, the saga undoes what has already been done by execute the compensating transactions in reverse order.

There are a couple of different ways to structure a saga's sequencing logic:

- Orchestration - centralize a saga's decision making and sequencing business logic in a saga coordinator class. A saga orchestrator sends command messages to saga participants telling them which operations to perform.
- Choreography - distribute the decision making and sequencing among the saga participants. They primarily communicate by exchanging events.

Let's look at each option starting with choreography.

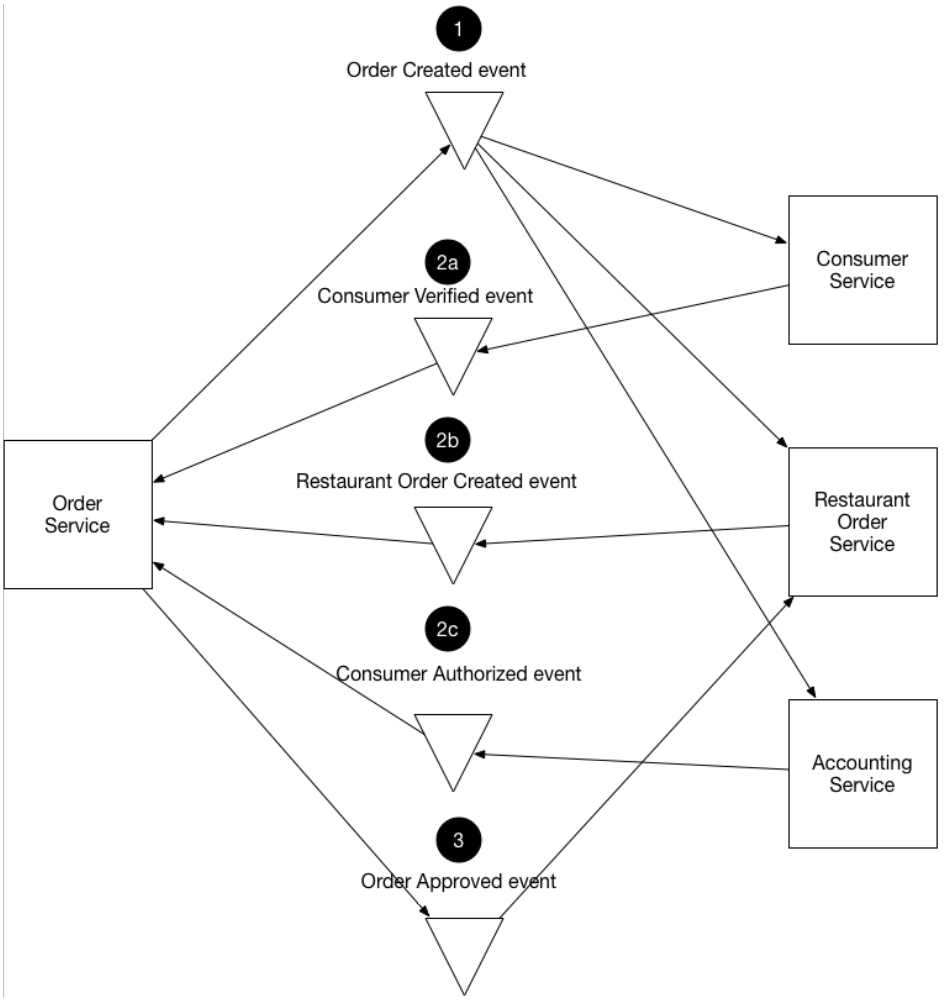
4.2.1 Choreography-based sagas

One way you can implement a saga is by using choreography. When using choreography, there is no central coordinator telling the saga participants what to do. Instead, the saga participants 'simply' know how to respond to one another's events.

Implementing the `Create Order` saga using choreography.

For example, you would implement a choreography-based `Create Order` saga by having the `ConsumerService`, `RestaurantOrderService` and the `AccountingService` subscribe to the `OrderService's` events and vice versa. Figure 4.2 shows how this works.

Figure 4.2. Implementing the `CreateOrder` saga using choreography. The saga participants communicate by exchanging events.



The happy path through this saga is as follows:

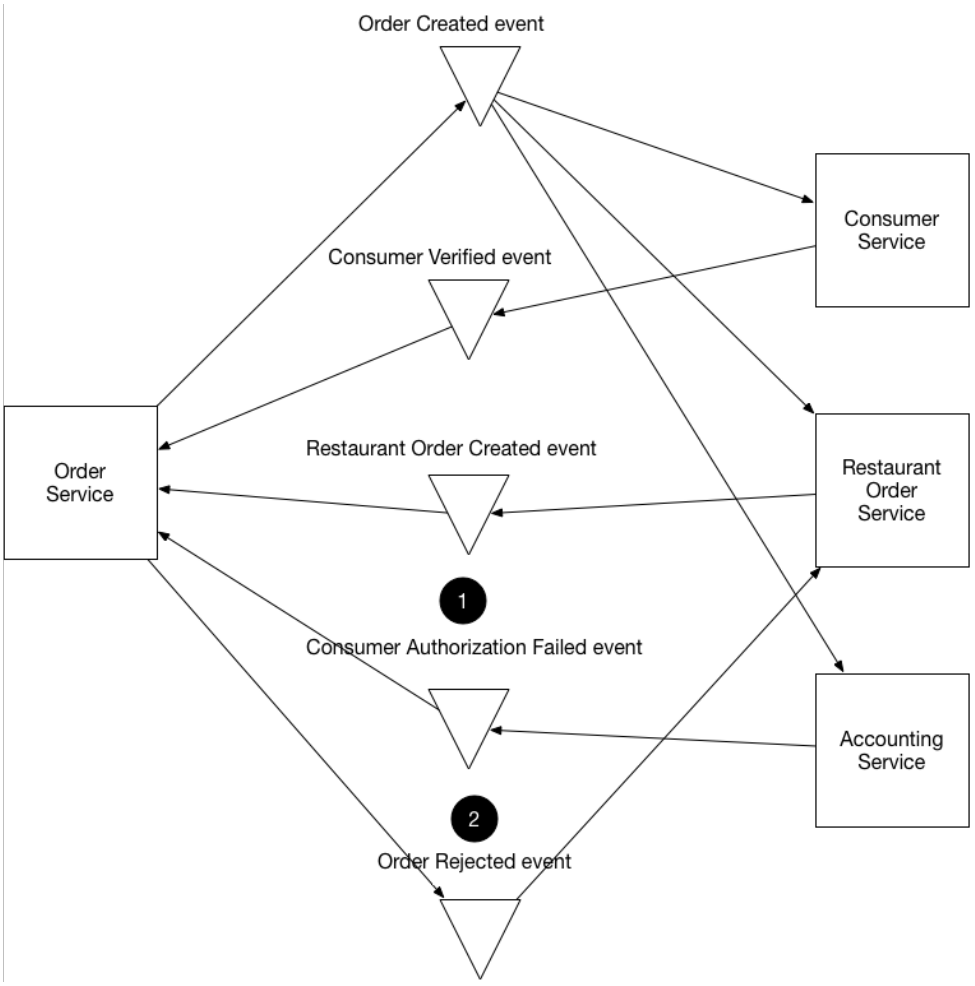
1. The `OrderService` creates an `Order` in the `CREATE_PENDING` state and publishes an `OrderCreated` event.
2. The `OrderCreated` event is received by the following services:
   a. The `ConsumerService`, which verifies that the consumer can place the order, and publishes `ConsumerVerified` event
   b. The `RestaurantOrderService`, which validate the `Order`, creates a `RestaurantOrder` in a `CREATE_PENDING` state, and publishes `RestaurantOrderCreated` event
   c. The `AccountingService`, which charges the consumer's credit card and

publishes `ConsumerAuthorized` event

3. The `OrderService` receives the `ConsumerVerified`, `RestaurantOrderCreated` and `ConsumerAuthorized` events , changes the state of the `Order` to `APPROVED` and publishes an `OrderApproved` event.
4. The `RestaurantOrderService`, receives the `OrderApproved` event and changes the state of the `RestaurantOrder` to `CREATED`

The `CreateOrder` saga must also handle the scenario where a saga participant rejects the `Order` and publishes some kind of 'failure' event. For example, the authorization of the consumer's credit card might fail. The saga must execute the compensating transactions to undo what has already been done. Figure 4.3 shows the flow of events when the `AccountingService` can't authorize the consumer's credit card.

Figure 4.3. Implementing the `CreateOrder` saga using choreography



The sequence of events is as follows:

1. The `AccountingService` publishes a `ConsumerAuthorizationFailed` event
2. `OrderService` changes the state of the `Order` to `REJECTED` and publishes an `OrderRejected` event.
3. `RestaurantOrderService` receives the `OrderRejected` event, changes the state of the `RestaurantOrder` to `REJECTED`.

Reliable event-based communication

There are a couple of issues that you must consider when implementing choreography-based sagas. The first issue is ensuring that a saga participant updates its database and publishes an event as part of a database transaction. Each step of a choreography-based saga updates the database and publishes an event. For example, in the `Create Order` saga, the `Restaurant Order Service` receives an `Order Created` event, creates a `Restaurant Order`, and publishes a `Restaurant Order Created` event. Consequently, the saga participants must use a mechanism such as the one provided by the Tram framework in order for the sagas to work reliably.

The second issue is that a saga participant must be able to map a received event to its own data. For example, when the `Order Service` receives a `Restaurant Order Created`, it must be able to lookup the corresponding `Order`. The solution is for a saga participant to publish events containing additional data that enables other participants to perform the mapping. The `Restaurant Order Service`, for example, publishes a `RestaurantOrderCreated` event containing the `orderId` from the `OrderCreated` event. When the `Order Service` receives a `Restaurant Order Created` event, it uses the `orderId` to retrieve the corresponding `Order`. This extra data plays the role of a *correlation id*.

Benefits and drawbacks of choreography-based sagas

Choreography-based sagas have several benefits. One benefit is that it simply requires services to publish events when they create, update or delete business objects. Unlike, when using orchestration, which is described below, no other libraries or frameworks are required. Another benefit of choreography is that the participants are loosely coupled and don't have direct knowledge of each other

There are, however, some drawbacks. One drawback of choreography is that it introduces cyclic dependencies between the services. That's because the services must subscribe to one another's events. For example, the `Order Service` subscribes to events published by the `Accounting Service` and vice versa. While this is not necessarily a problem, cyclic dependencies are considered a design smell.

Another drawback of choreography is that it potentially makes some domain objects more complex. For example, the `Order` knows about the order validation process. It must track the events published by the `ConsumerService`, `RestaurantOrderService` and `AccountingService` in order to decide whether it has been approved or rejected. As a result, an `Order` has additional states that increase its complexity as well as the complexity of any code that uses it.

The third drawback of choreography is that unlike with orchestration there isn't a single place in the code that defines how to validate an `Order`. Instead, choreography distributes knowledge of how to do that among the services. It is much more difficult for a developer to understand how a given saga works. Because of these drawbacks, it is sometimes better to use orchestration, which centralizes the sequencing logic in a single class.

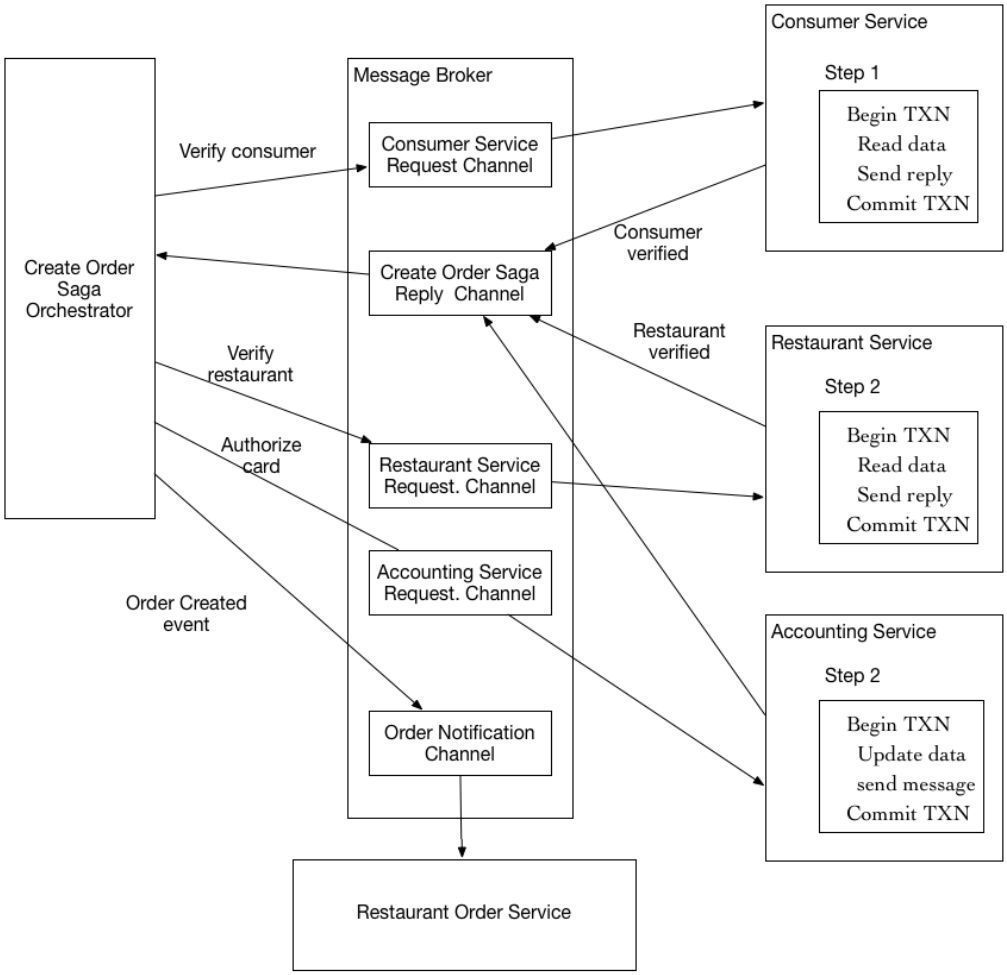
4.2.2 Orchestration-based sagas

Orchestration is another way to implement sagas. When using orchestration, you define an orchestrator class whose sole responsibility is to tell the saga participants what to do. The saga orchestrator communicates with the participants using command/reply-style interaction. To execute a saga step, it sends a command message to a participant telling it what operation to perform. Once the saga participant performs the operation, it sends a reply message to the orchestrator. The orchestrator then processes the message and determines which saga step to perform next.

Implementing the `CreateOrder` saga using orchestration

For instance, the `Create Order` saga can be orchestrated by the `CreateOrderSaga` class. This class contains the logic that implements the state machine that validates the order and authorizes the credit card. As figure 4.4 shows, `CreateOrderSaga` sends commands to several services using the `RestaurantOrderService` and the `ConsumerService`. Since the saga uses asynchronous messaging the orchestrator and the participants communicate using message channels. Each saga participant reads command messages from its own command message channel. The `Create Order` saga orchestrator sends commands to the saga participants' command channels and reads replies from its reply channel.

Figure 4.4. Implementing the `CreateOrder` saga using orchestration



The `Order Service` first creates an `Order` and the saga orchestrator. After that, the flow for the happy path is as follows:

1. The saga orchestrator sends a `Verify Consumer` command to the `Consumer Service`
2. The `Consumer Service` replies with a `Consumer Verified` message.
3. The saga orchestrator sends a `Create Restaurant Order` command to the `Restaurant Order Service`
4. The `Restaurant Order Service` replies with a `Restaurant Order Created` message
5. The saga orchestrator sends a `Authorize Card` message to the `Accounting Service`
6. The `Accounting Service`sends replies with a `Card Authorized` message

7. The saga orchestrator sends an `Approve Restaurant Order` command to the `Restaurant Order Service`
8. The saga orchestrator sends an `Approve Order` command to the `Order Service`.

Note that in final step, the saga orchestrator sends a command to the `Order Service` even though it is component of the `Order Service`. In principle, the `Create Order` saga could approve the `Order` by updating it directly. However, in order to be consistent, the saga treats the `Order Service` as just another participant.

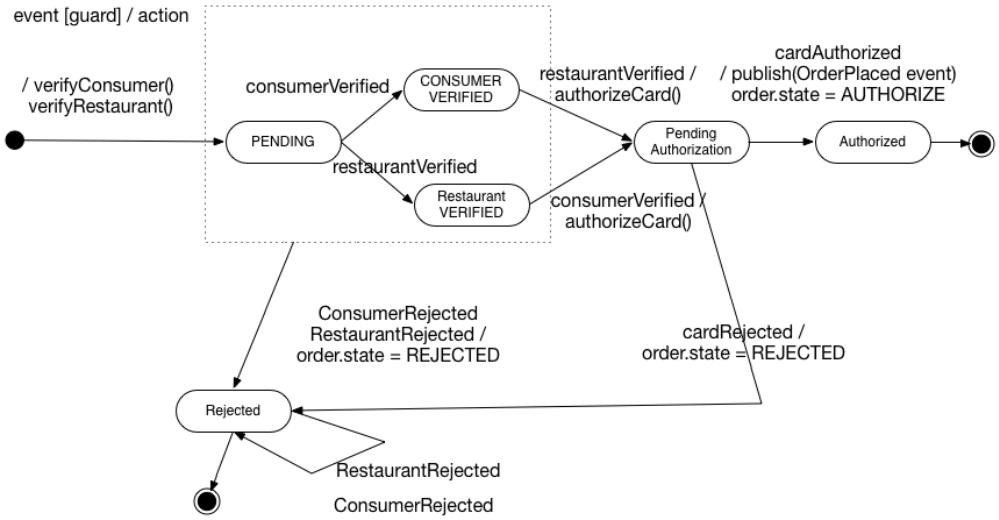
Diagrams such as figure 4.4 each depict one scenario for a saga. However, a saga is likely to have numerous scenarios. For example, the `Create Order` saga has four scenarios. In addition to the happy path, the saga can fail due to an failure in the `Consumer Service`, `Restaurant Order Service` or `Accounting Service`. It's useful, therefore, to model a saga as a state machine since it describes all possible scenarios.

Modeling saga orchestrators as state machines

A good way to model a saga orchestrator is as a state machine. A state machine consists of a set of states and a set of transitions between states that are triggered by events. Each transition can have an action, which for a saga is the invocation of a saga participant. The transitions between states are triggered the completion of a local transaction performed by a saga participant. The current state and the specific outcome of the local transaction determine the state transition and what action, if any, to perform.

State machines are a useful formalism that make it easier to design saga. Their structure and behavior is well defined. There are also effective testing strategies for state machines. As result, using a state machine model makes designing, implementing and testing sagas easier. Figure 4.5 shows the state machine model for the `Create Order` saga.

Figure 4.5. The state machine model for the `Create Order Saga`



When `Create Order` saga is created by the `Order Service`, its initial action is to verify the consumer and its initial state is `Verifying Consumer`. The response from the `Consumer Service` triggers the next state transition. If the consumer was successfully verified, the saga creates the `Restaurant Order` and transitions to the `Creating Restaurant Order` state. If, however, the consumer verification failed, the saga rejects the `Order` and transitions to the `Rejecting Order` state. The state machine undergoes numerous other state transitions driven by the responses from saga participants until it reaches a final state of either `Rejected` or `Succeeded`.

Saga orchestration and reliable messaging

In order for saga orchestrators and participants to work reliably, they must use transactions to update the database and exchange messages. But, as I described in both this chapter and chapter {chapter-ipc}, an application cannot updating a database and send a message using 2PC. Instead, sagas must use an approach such as the one provided by the Tram framework. The Tram framework publishes messages as part of the database transaction by using a database table as a temporary outbound message queue. The framework also consumes messages as part of a database transaction. Lets look at how to make saga orchestration and messaging transactional beginning with the creation of the orchestrator.

A service initiates a saga by instantiating a saga orchestrator and saving in the database. Within that same transaction, the saga orchestrator sends a message to the first saga participant using the Tram framework, which inserts the message into the `OUTBOX` table.

```
BEGIN TXN
  ... other database updates ...
  INSERT INTO SAGA ...
```

```
  INSERT INTO OUTBOX ...
COMMIT TXN
```

Later on, the Tram framework sends the message to the specified saga participant's command channel.

A saga participant processes a message sent by the saga orchestrator using code that looks like this:

```
Read message from message broker
TRY
  Begin TXN
    ... Detect duplicates ...
    ... Update persistent business objects ...
    INSERT INTO OUTBOX ...
  Commit TXN
CATCH
  Send reply containing failure
Acknowledge message
```

First, the Tram framework reads the request from the participant's command channel. Next, it begins a transaction. Within the transaction, the Tram framework determines whether the message is a duplicate and if so, discards it. The saga participant then invokes the business logic, which updates the database. It also sends a reply using the Tram framework. The reply message contains the outcome of the transaction along with any data required by later transaction steps. The saga participant then commits the transaction. Finally, it acknowledges the message.

The TRY/CATCH statement represents error handling logic for when the transaction is rolled back because of a permanent failure. Examples of permanent failures include the violations of preconditions such as a non-existent domain object or a business rule. Other kinds of failures such an optimistic locking and other concurrency related errors are transient failures since retrying the transaction will fix the problem. When a permanent error occurs, the saga participant sends a reply message containing the error. If some other kind of error occurs, the message won't be acknowledged and so it will be redelivered.

A saga orchestrator also uses the Tram framework to process reply messages from saga participants. It execute a message processing loop that read messages from the orchestrator's reply channel, discards duplicates, invokes the saga logic, and publishes further messages. The code looks something like this:

```
Read message from reply channel
BEGIN TXN
  ... Detect duplicates ...
  SELECT * SAGA
  UPDATE ... SAGA ...
  INSERT INTO OUTBOX ...
COMMIT TXN
Acknowledge message
```

First, the Tram framework invokes a message broker API to reads the message from

the saga orchestrator's channel. Next, within a transaction, the Tram framework checks whether the message is a duplicate and if so, discards it. It then invokes the saga logic, which updates the saga orchestrator, and generates a command message. The Tram framework inserts the message into the `OUTBOX` table. Finally, Tram framework invokes a message broker API to acknowledge the reply message. A failure at any step prior to acknowledging the message will cause the message to be reprocessed again. The duplicate detection logic ensures the message will be processed exactly once. Later on, I'll describe the implementation of the `Create Order` saga orchestrator in more detail. But first, let's take a look at the benefits and drawbacks of using saga orchestration.

Benefits and drawbacks of orchestration-based sagas

Orchestration-based sagas have several benefits. One benefit of orchestration is that it doesn't introduce cyclic dependencies. The saga orchestrator invokes the saga participants but the participants do not invoke the orchestrator. As a result, the orchestrator depends on the participants but not vice versa.

Another benefit of orchestration is that it improves separation of concerns and simplifies the business logic. The saga coordination logic is localized in the saga orchestrator. The domain objects are simpler and have no knowledge of the sagas that they participate in. For example, when using orchestration, the `Order` class has no knowledge of any of the sagas and so has a simpler state machine model. During the execution of the `Create Order` saga it transitions directly from the `CREATE_PENDING` state to the `AUTHORIZED` state. The `Order` class does not have any intermediate states corresponding to the steps of the saga. As a result, the business is much simpler.

Orchestration also has some drawbacks. One downside is that there is a risk of centralizing too much business logic in the orchestrator. This results in a design where the smart orchestrator tells the dumb services what operations to do. Fortunately, you can avoid this problem by designing orchestrators that are solely responsible for sequencing and do not contain any other business logic. As a result, you should consider using orchestration for all but the simplest sagas.

4.3 The impact of sagas on business logic

Now that we have looked at the mechanics of how sagas work, let's look at how sagas impact the design of business logic. Traditional business logic design relies heavily on ACID transactions, which have several important features:

- Easily rollback - business logic that executes within an ACID transaction can easily rollback the transaction if it detects the violation of a business rule
- Consistency - the updates made by an ACID transaction appear once the transaction commits and so the database is always consistent.
- Serializability - ACID transactions that execute concurrently behave as they are executed serially.

Sagas work in a completely different way. Since the changes made by each step of a

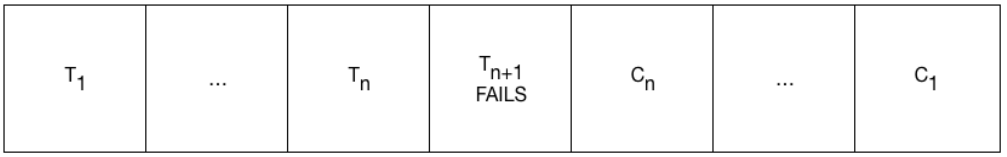
saga are committed once that step completes, you cannot automatically rollback a saga. Nor do they have the consistency and serializability of ACID transactions. The intermediate state of one saga is visible to other sagas. As a result, designing business logic in a saga-based application is more challenging. Lets look at the problems you will encounter when using sagas, starting with the challenge of rolling back a saga.

4.3.1 Sagas uses compensating transactions to rollback changes

A key challenge with using sagas is rolling back a saga when, for example, a business rule is violated. For example, the authorization of the consumer's credit card might fail due to insufficient funds. However, unlike ACID transactions, which simply rollback if there is a failure, sagas must explicitly undo what changes have been made so far. You must write what are known as compensating transactions.

Lets suppose that the $(n + 1)^{th}$ transaction of a saga fails. The effects of the previous *n* transactions must be undone. Conceptually, each of those steps T_i has a corresponding compensating transaction C_i, which undoes the effects of the T_i. To undo the effects of those first *n* steps the saga must execute each C_i in reverse order. The sequence of steps is T_1, … T_n, C_n, …, C_1 as is shown in figure 4.6. In this example, T_{n+1} fails, which requires steps T_1, … T_n to be undone.

Figure 4.6. When a step of a saga fails because of a business violation the updates made by previous steps must be explicitly undone by executing compensating transactions.



The saga executes the compensation transactions in reverse order of the of the forward transactions: C_n, …, C_1. The mechanics of sequencing the C_is is not really any different than sequencing the T_is. Consider, for example, the `Create Order` saga. This saga can fail for a variety of reasons: the consumer information is invalid or the consumer is not allowed to create orders; the restaurant information is invalid or the restaurant is unable to accept orders; or the authorization of the consumer's credit card fails. If a failure does occur, the saga participant sends a failure message. The `Create Order` saga coordinator must then execute a compensating transaction that cancels the order.

4.3.2 Sagas are interwoven

Another challenge with developing saga-based business logic is that the effects of each step of a saga are immediately visible once that step completes. A saga might, for example, make a series of modifications to a domain object. Since each step of a saga is a local transaction, each modification is visible to other sagas. The transactions of multiple sagas that read and write the same data will be interwoven. A saga will often need to update a business object that is still being updated by another saga. You must,

therefore, design your application's business logic to correctly handle domain objects that are in an intermediate state. As a result, your business logic is potentially more complex. Let's look at some examples.

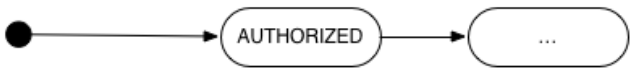
The problem of interwoven sagas in a banking application

The severity of this problem depends on the application domain. Consider, for example, a banking application that uses a saga to transfer money between accounts. In this application, the Money Transfer saga debits the *from* account and credits the *to* account. If the *to* account is closed, the saga's compensating transaction credits the *from* account. But let's imagine that another saga attempts debit the *from*account before the compensating transaction has reversed the debit. The second saga might discover that there are insufficient funds in the *from* account. This scenario would not occur in a monolithic application that transferred money using a single ACID transactions.

The problem of interwoven sagas in the FTGO application

The problem is worse in other application domains, such as the FTGO application. Figure 4.7 shows part of the state model for an `Order` in the monolithic FTGO application, which uses ACID transactions. This application does not create an `Order` until it has been validated. It never creates an invalid `Order`.

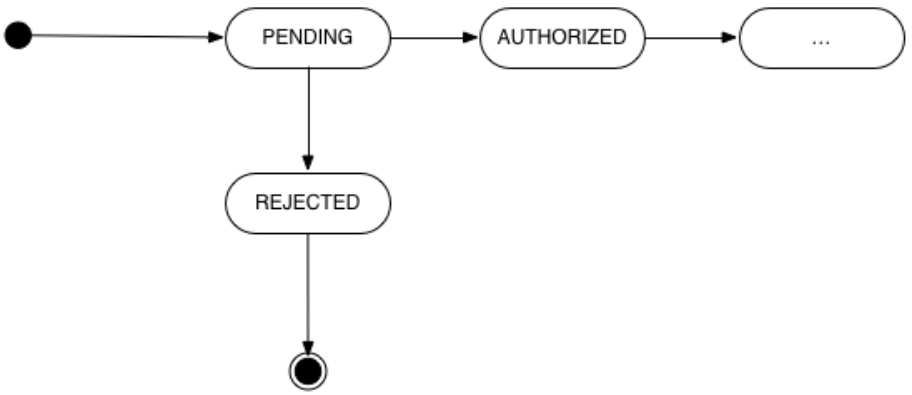
Figure 4.7. The state model for an `Order` in a monolithic application



The application creates an `Order` in the `AUTHORIZED` state.

In contrast, in the microservices-based FTGO application, which uses sagas, unvalidated `Orders` are immediately visible to other transactions. As you can see in Figure 4.8, the `Order Service` creates the `Order` in the `CREATE_PENDING` state.

Figure 4.8. The more complex state model of an `Order` in a saga-based application.

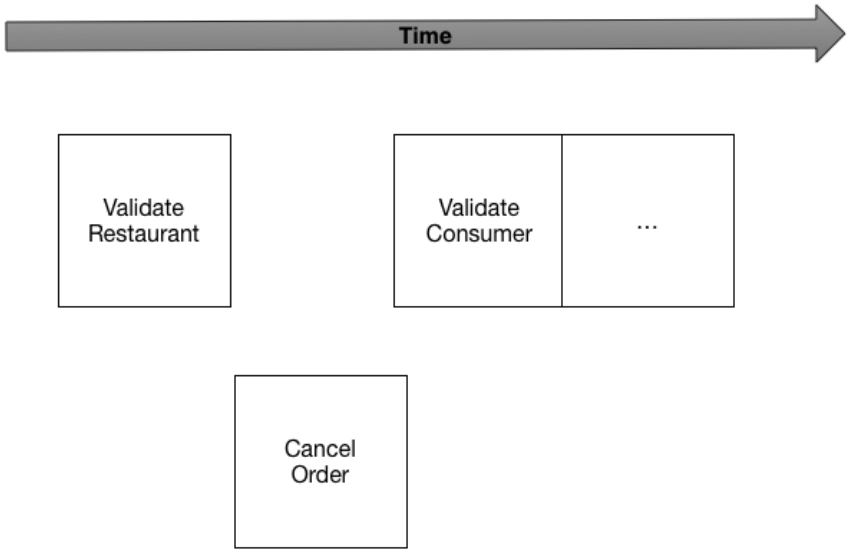


The `CREATE_PENDING` state is a state that does not exist (or is at least not visible) in a monolithic application. What's more, in an application that uses choreography-based sagas, where an `Order` must keep track of the events that it has received, it's likely to have many more state. Other system operations and compensating transaction require even more states. As a result, when using sagas domain objects, such as `Order`, have a more complex state machine, which makes the business logic more complex.

Designing sagas that can be interwoven

In order to understand the problem and how to solve it let's take a look at the the `cancelOrder()` system operation. In a traditional monolithic application that uses ACID transactions, this operation isn't executed until after an order was created completely. The *order id* isn't even visible until the `createOrder()` transaction commits. In contrast, the `cancelOrder()` system operation in the saga-based FTGO application must handle the scenario, shown in figure 4.9, where the order is still in the process of being validated by the `Create Order` saga.

Figure 4.9. The problem of interwoven sagas. The user attempts to cancel an order while it is still in the process of being validated by the `Create Order` saga.



One way to handle this scenario is for the `cancelOrder()` operation to fail and tell the user that they must try again to cancel the order. In many ways, this is the simplest approach. The `cancelOrder()`operation simply checks the state of `Order` and throws an exception if the `Create Order` saga is not yet complete. The drawback, however, is that it creates a bad user experience. A better approach is for `cancelOrder()` to some how undo the effects of the `Create Order` saga that is still in process and cancel the order.

One solution is for the `Cancel Order` saga to set a flag in the `Order` to indicate that it must be cancelled. The `Create Order` saga could then notice that the flag has been set

and cancel the `Order`. The drawback of this approach is that the sagas are no longer independent. The `Create Order` saga needs to know how to cancel an `Order`. This lack of modularity will likely result in business logic that is difficult to maintain.

Another way to solve this problem, is for the `Cancel Order` saga to 'wait' until the `Create Order` saga has completed. An application can implement accomplish by using a locking mechanism. A saga that wants to create or update a domain object must first obtain a lock. If another saga attempts to update the same domain object it is blocked until the first saga completes and releases the lock. For example, the `Create Order` saga would lock the `Order` until it completes at which point the `Cancel Order` saga would acquire the lock and begin the process of canceling the `Order`.

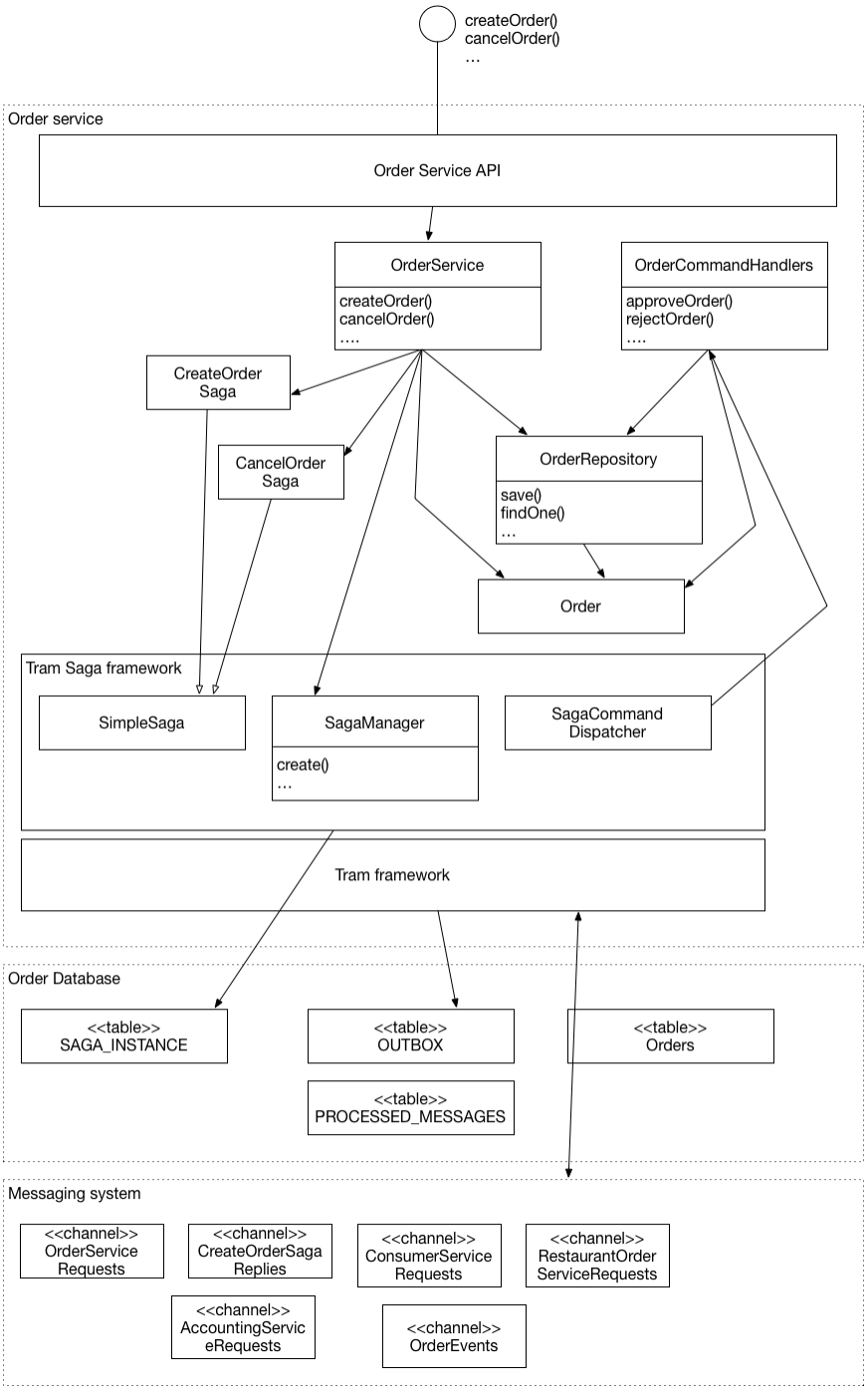
This approach has several benefits and drawbacks. One benefit is that it ensures that the sagas are decoupled from one another. The `Create Order` and `Cancel Order` sagas have no knowledge of each another. Another benefit is that effectively hides any intermediate states. The `Cancel Order` saga is blocked until the `Create Order` saga completes. As a result, it doesn't see any intermediate states so the business logic is much simpler.

The use of locks has some drawback, however. There is the possibility of deadlocks where, for example, two sagas are blocked waiting for locks the other holds. The solution is to implement a deadlock detection algorithm that performs a rollback of a saga to break a deadlock and re-executes it.

4.4 The design of the `Order Service` and its sagas

Now that we have looked at various saga design and implementation issues let's look at an example. The `Order Service` provides an API for creating and managing orders. It consists of various components including an `OrderService` class, an `Order` entity and several saga orchestrator classes including `CreateOrderSaga` and `CancelOrderSaga`. Figure 4.10 shows the design of this service.

Figure 4.10. The design of the `Order Service` and its sagas



Some parts of the `Order Service` have a familiar design. Just as in a traditional application, the core of the business logic is implemented by the `OrderService`, `Order`, and `OrderRepository` classes. In this chapter, I am only going to briefly describe these classes. I describe them in more detail in chapter {chapter-ddd-aggregates}.

What's less familiar about the `Order Service` are the saga-related classes. The `Order Service` defines several saga orchestrators including `CreateOrderSaga` and `CancelOrderSaga`. There is also the `OrderCommandHandlers` class, which handles the messages sent by sagas to the `Order Service`. Let's look in more detail at the design starting with the Tram saga framework.

4.4.1 About the Tram saga framework

The `Order Service's` sagas are implemented using the Tram saga framework. The framework has two main packages, an orchestration package for writing saga orchestrators, and a participant package for writing saga participants. Let's look at each package.

The Saga orchestration package

The orchestration package provides a domain-specific language (DSL) for defining sagas, and a `SagaManager` class, which creates and manages saga instances. The DSL provides a simple way to define a saga's state machine. You specify the sequence of steps and for each step you specify the forward transaction and the corresponding compensating transactions. Each transaction sends a message to a participant.

Listing 4.1 shows an excerpt of a saga definition that illustrates some of the key features of the DSL. A saga implements the `SimpleSaga` interface. `SimpleSaga` is a generic interface that has one type parameter, the class of the saga's data. The saga's data is the persistent state of the saga. The saga class, which is this example is `CreateOrderSaga` is a stateless singleton.

Listing 4.1. Part of a saga definition, which illustrates the key features of the saga DSL.

```
public class CreateOrderSaga implements SimpleSaga<CreateOrderSagaData> {

  private SagaDefinition<CreateOrderSagaData> sagaDefinition =
        step()
           .withCompensation(this::rejectOrder)
        .step()
           .invokeParticipant(this::verifyConsumer)
        ...
        .build();

  @Override
  public SagaDefinition<CreateOrderSagaData> getSagaDefinition() {
   return sagaDefinition;
  }
```

A saga must define a `getSagaDefinition()`, which returns the saga definition that is created using the DSL. The DSL provides methods such as `step()`,

`withCompensation()` and `invokeParticipant()`. The `step()` method defines a step, which consists of a forward transaction or a compensating transaction or both. The `invokeParticipant()` method defines a forward transaction and the `withCompensation()`method defines a compensating transaction. Both methods have a single parameter , which is a `Function` that takes the saga's data as a parameter and returns `CommandWithDestination`. The `SagaManager`invokes this function to get the command message to send to a saga participant. A `CommandWithDestination` contains a message and the destination channel to send it to.

The `SagaManager` is the other key part of the orchestration package. An application has an instance of a `SagaManager` for each type of saga. The `SagaManager` class provides an API for creating saga instances. It interprets the saga's definition, sends commands to saga participants, and handles replies. The `SagaManager` persists saga instances in the database. It exchanges messages with saga participants using the Tram framework, which ensures that the message processing happens as part of a database transaction that updates the persistent saga instance.

When an application creates a saga instance, the `SagaManager` performs the following actions:

1. Execute the first forward transaction
2. Send the command message to the specified participant
3. Saves the saga instance in the database.

When it receives a reply, the `SagaManager` performs the following actions:

1. Retrieve the saga instance from the database
2. Invokes the current step to handle the reply
3. Invoke the next step
4. Send a command message to the specified participant
5. Save the updated saga instance in the database.

If a saga participant fails, the saga manager executes the compensating transactions in reverse order.

The Saga participants package

The participant package provides an API for writing saga participants. It builds on the Tram framework's command package and defines a `SagaCommandDispatcher` class. The `SagaCommandDispatcher`dispatches command messages to command handlers. It manages locks that serializes access to to domain objects. Let's look at the saga framework in action and see an example of a saga orchestrator and particpant.

4.4.2 The `OrderService` class

The `OrderService` class is the domain service, which is called by the service's API layer. It defines methods for creating and updating orders. I'll describe this class in more detail in chapter {chapter-ddd-aggregates}. For now I'll focus on the `createOrder()` method. Listing 4.2 shows an excerpt of the `OrderService`, which includes the `createOrder()` method. This method first creates an `Order` and then

creates an `CreateOrderSaga` to validate the order.

Listing 4.2. The `OrderService` class and its `createOrder()` method

```
@Transactional                                                          (1)
public class OrderService {

  @Autowired
  private SagaManager<CreateOrderSagaData> createOrderSagaManager;

  @Autowired
  private OrderRepository orderRepository;

  @Autowired
  private DomainEventPublisher eventPublisher;

  public Order createOrder(OrderDetails orderDetails) {
    ResultWithEvents<Order> orderAndEvents = Order.createOrder(orderDetails); (2)
    Order order = orderAndEvents.result;
    orderRepository.save(order);                                        (3)

    eventPublisher.publish(Order.class,                                 (4)
                           Long.toString(order.getId()),
                           orderAndEvents.events);

    CreateOrderSagaData data =
        new CreateOrderSagaData(order.getId(), orderDetails);           (5)
    createOrderSagaManager.create(data, Order.class, order.getId());

    return order;
  }

  ...
}
```

(1) Ensure that service methods are transactional
(2) Create the `Order`
(3) Persist the `Order` in the database
(4) Publish domain events
(5) Create a `CreateOrderSaga`

The `createOrder()` method creates an `Order` by calling the factory method `Order.createOrder()`. It then persists the `Order` using the `OrderRepository`, which is a JPA-based repository. It creates the `CreateOrderSaga` by calling `SagaManager.create()` passing a `CreateOrderSagaData` containing the *id* of the newly saved `Order` and the `OrderDetails`.

4.4.3 The design of the *`CreateOrderSaga`* orchestrator

The `CreateOrderSaga` class orchestrates the validation of an order. It implements the state machine shown earlier in figure 4.5. The `CreateOrderSaga` class implements the `SimpleSaga` interface. The persistent state class for `CreateOrderSaga` is

`CreateOrderSagaData`. The heart of the `CreateOrderSaga` class is the saga definition, which is shown in Listing 4.3. It is defined using the DSL provided by the Tram saga framework.

Listing 4.3. The definition of the `CreateOrderSaga`

```
public class CreateOrderSaga implements SimpleSaga<CreateOrderSagaData> {

  private SagaDefinition<CreateOrderSagaData> sagaDefinition =
        step()
           .withCompensation(this::rejectOrder)
        .step()
           .invokeParticipant(this::verifyConsumer)
        .step()
           .invokeParticipant(this::createRestaurantOrder)
           .onReply(CreateRestaurantOrderReply.class,
                    this::handleCreateRestaurantOrderReply)
           .withCompensation(this::rejectRestaurantOrder)
        .step()
           .invokeParticipant(this::authorizeCard)
        .step()
           .invokeParticipant(this::approveOrder)
        .step()
           .invokeParticipant(this::approveRestaurantOrder)
        .build();

 @Override
 public SagaDefinition<CreateOrderSagaData> getSagaDefinition() {
  return sagaDefinition;
 }
```

In order to better understand how the `CreateOrderSaga` works, let's look at the definition of the third step of the saga. This step invokes the `RestaurantOrderService` to create a `RestaurantOrder`. Listing 4.4 shows the definition of this step along with the three helper methods that it uses.

Listing 4.4. The definition of the third step of the saga and some helper methods that create the messages to send to the saga recipients.

```
public class CreateOrderSaga ...

  private SagaDefinition<CreateOrderSagaData> sagaDefinition =
    ...
    .step()
     .invokeParticipant(this::createRestaurantOrder)              (1)
     .onReply(CreateRestaurantOrderReply.class,                   (2)
              this::handleCreateRestaurantOrderReply)
     .withCompensation(this::rejectRestaurantOrder)               (3)
    ...
  ;

  private CommandWithDestination createRestaurantOrder(CreateOrderSagaData data) {
     return send(new
CreateRestaurantOrder(data.getOrderDetails().getRestaurantId(),
```

```
                                          data.getOrderId(),
                                          data.getOrderDetails()))
            .to("restaurantService")
            .build();
  }

  private void handleCreateRestaurantOrderReply(CreateOrderSagaData data,
                                                CreateRestaurantOrderReply reply) {
   data.setRestaurantOrderId(reply.getRestaurantOrderId());
  }

  private CommandWithDestination rejectRestaurantOrder(CreateOrderSagaData data) {
     return send(new RejectRestaurantOrder(
              data.getOrderDetails().getRestaurantId(),
              data.getOrderId()))
            .to("restaurantService")
            .build();

  }
```

1. The forward transaction that sends a `CreateRestaurantOrder` message to the `Restaurant Order Service`
2. Save the `restaurantOrderId` returned in the reply `Restaurant Order Service`
3. The compensating transaction that undoes the create by sending a `RejectRestaurantOrder` message to the `Restaurant Order Service`

The call to `invokeParticipant()` specifies the saga must create the command message for the forward transaction by calling `createRestaurantOrder()`. This method creates a `CreateRestaurantOrder`destined for the `Restaurant Order Service`. Similarly, the call to `withCompensation()` specifies that the `rejectRestaurantOrder()` method should be called to create command message for the compensating transaction. The `rejectRestaurantOrder()` creates a `RejectRestaurantOrderCommand` to send to the `Restaurant Order Service`. The call to `onReply()`, specifies that the saga must call `handleCreateRestaurantOrderReply()` when it handle a successful reply from the `RestaurantOrderService`. This method stores the *id* of the newly created `RestaurantOrder` in the saga data so that it can be used by other saga steps.

4.4.4 The `OrderCommandHandlers` class

The `OrderCommandHandlers` class, which is shown in listing 3.6, defines methods that handle command messages sent by sagas, such as `CreateOrderSaga`, to the `Order Service`. Each of its methods updates an `Order`, publishes domain events, and sends a reply message to the saga. The handler methods are invoked by the Tram framework, which implements transactional messaging.

```
public class OrderCommandHandlers {

  @Autowired
  private OrderRepository orderRepository;

  @Autowired
  private DomainEventPublisher eventPublisher;
```

```
public CommandHandlers commandHandlers() {                                  (1)
  return CommandHandlersBuilder
        .fromChannel("orderService")
        .onMessage(ApproveOrderCommand.class, this::approveOrder)
        .onMessage(RejectOrderCommand.class, this::rejectOrder)
        ...
        .build();

}

public Message approveOrder(CommandMessage<ApproveOrderCommand> cm) {
  long orderId = cm.getCommand.getOrder();
  updateOrder(orderId, Order::noteAuthorized);                              (2)
  return withSuccess();                                                     (3)
}



public Message rejectOrder(CommandMessage<RejectOrderCommand> cm) {
  long orderId = cm.getCommand.getOrder();
  updateOrder(orderId, Order::noteRejected);                                (4)
  return withSuccess();
}

private Order updateOrder(long orderId, Function<Order,
                          List<DomainEvent>> updater) {
  Order order = orderRepository.findOne(orderId);
  eventPublisher.publish(Order.class, Long.toString(orderId),
                         updater.apply(order));
  return order;
}
```

(1) Route a command message to the appropriate handler method
(2) Change the state of the `Order` to authorized
(3) Return a `Success` message
(4) Change the state of the `Order` to rejected

The `approveOrder()` and `rejectOrder()` methods update the specified `Order` using the `updateOrder()` helper method. This method loads the `Order`, invokes the specified method, and publishes any generated domain events. The other services that participate in sagas have similar command handler classes that update their domain objects.

4.4.5 The *OrderServiceConfiguration* class

The `Order Service` uses the Spring framework. Its components are instantiated and wired together by the `OrderServiceConfiguration` class, which is a Spring `@Configuration` class. Listing 4.6 is an excerpt of this class.

Listing 4.5. Some of the Spring @Beans that configure the `Order Service`

```
@Configuration
public class OrderServiceConfiguration {
```

```
 @Bean
 public OrderService orderService() {
  return new OrderService();
 }

 @Bean
 public SagaManager<CreateOrderSagaData> createOrderSagaManager(CreateOrderSaga
saga) {
  return new SagaManagerImpl<>(saga);
 }

 @Bean
 public CreateOrderSaga createOrderSaga() {
  return new CreateOrderSaga();
 }

 @Bean
 public OrderCommandHandlers orderCommandHandlers() {
  return new OrderCommandHandlers();
 }

 @Bean
 public SagaCommandDispatcher  orderCommandHandlersDispatcher(OrderCommandHandlers
orderCommandHandlers) {
  return new SagaCommandDispatcher("orderService",
orderCommandHandlers.commandHandlers());
 }

 ...

}
```

This class defines several Spring @Beans including `orderService`, `createOrderSagaManager`, `createOrderSaga`, `orderCommandHandlers` and `orderCommandHandlersDispatcher`.

The `CreateOrderSaga` is only one of the `Order Service's` many sagas. Many of its other system operations also use sagas. For example, the `cancelOrder()` operation uses a `CancelOrderSaga` and the `reviseOrder()` operation uses a `ReviseOrderSaga`. As a result, even though many services have an external API API that uses a synchronous protocol, such as REST or gRPC, a large amount of inter-service communication will use asynchronous messaging.

As you can see, transaction management and some aspects of business logic design are quite different in a microservice architecture. Fortunately, saga orchestrators are usually quite simple state machines and you can use a saga framework simplify your code. Nevertheless, transaction management is certainly more complicated than in a monolithic architecture. That is usually, however, a small price to pay for the tremendous benefits of microservices.

4.5 Summary

- XA/2PC-based distributed transactions are not a good fit for modern applications and sagas, which are sequences of message-driven local transactions, are a better

way to maintain data consistency in a microservice architecture

- Designing saga-based business logic can be challenging because sagas can be interwoven and must use compensating transactions to rollback changes. An application must sometimes use locking in order to simplify the business logic even though that risks deadlocks.
- Simple sagas can sometimes use choreography but orchestration is usually a better approach for complex sagas
- Modeling saga orchestrators as state machines simplifies development and testing

5 Designing business logic in a microservice architecture

This chapter covers:

- Business logic organization patterns
- Designing business logic with Domain-driven design (DDD) aggregates
- Using domain events in your application

The heart of an enterprise application is the business logic that implements the business rules. In a microservice architecture the business logic is spread over multiple services. As I described in chapter {chapter-ipc}, some external invocations of the business logic are handled by a single service. Other, more complex requests are handled by multiple services and sagas are used to enforce data consistency. In this chapter, I describe how to implement a service's business logic.

The raison d'etre for a service is to handle requests from its clients. Some clients are external to the application, such as other applications or the user. Other clients are other services including saga orchestrators. Inbound requests are handled by an adapter such as a web controller or messaging gateway, which invokes the business logic. The business logic typically updates a database, possibly invokes other services, and returns response to the request.

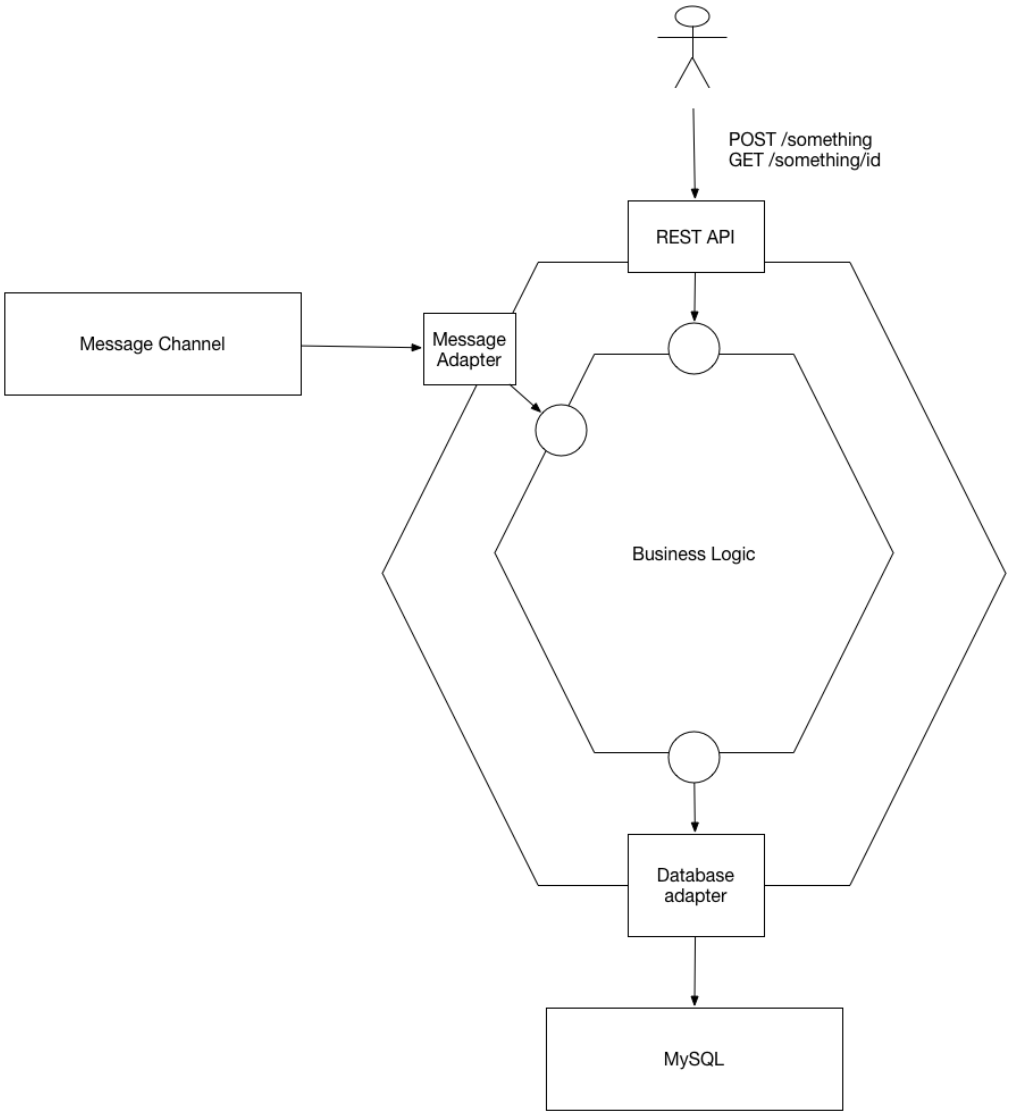
Developing complex business logic is always challenging. What's more, the microservice architecture presents some distinctive challenges. As I described in chapter {chapter-sagas}, a microservices-based application often uses sagas to maintain consistency across multiple services. Furthermore, as you will learn below, in a microservice architecture business logic often needs to generate events when data changes. In this chapter, you will learn how to structure business logic as a collection

of domain-driven design (DDD) aggregates that emit events. Let's first look at the different ways of organizing business logic.

5.1 Business logic organization patterns

The core of a service is its business logic. But as figure 5.1 shows, a service also consists of one or more adapters. It will have inbound adapters, which handle requests from clients and invoke the business logic. It will typically have outbound adapters, which enable the business logic to invoke other services and applications.

Figure 5.1. The structure of a typical service



This service consists of the business logic and the following adapters:

- REST API adapter, which, exposes an HTTP API.
- Inbound message gateway, which consumes messages from a message channel.
- Database adapter, which accesses the database
- Outbound message adapter, publishes messages to a message broker

Sitting at the core of the service is the business logic, which is typically the most

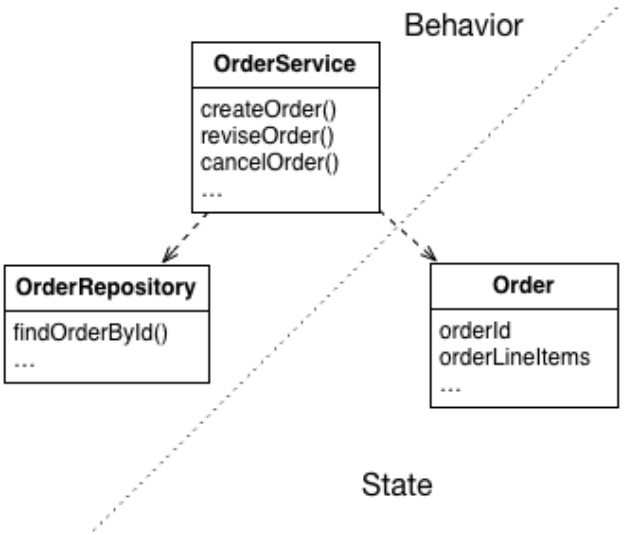
complex part of the service, is invoked by the inbound adapters. The business logic invokes the outbound adapters to access the database and publish messages.

When developing business logic, you should consciously organize your business logic in the way that's the most appropriate for your application. After all, I'm sure you've experienced the frustration of having to maintain someone else's badly structured code. Most enterprise applications are written in an object-oriented language such as Java and so consists of classes and methods. However, using an object-oriented language does not guarantee that the business logic has an object-oriented design. The key decision you must make when developing business logic is whether to use an object-oriented approach or a procedural approach. There are two main patterns for organizing business logic: the procedural Transaction Script pattern, and the object-oriented Domain Model pattern.

5.1.1 Transaction script pattern

While I am a strong advocate of the object-oriented approach, there are some situations where it is overkill, such as when you are developing simple business logic. In such a situation, a better approach is to write procedural code and use what Martin Fowler calls the Transaction Script pattern [Fowler 2002]. Rather than doing any object-oriented design, you simply write a method, which is called a transaction script, to handle each request from the presentation tier. As figure 5.2 shows, An important characteristic of this approach is that the classes that implement behavior are separate from those that store state.

Figure 5.2. Organizing business logic as transaction scripts



When using the Transaction Script pattern, the scripts are usually located in service classes, which in this example is the `OrderService` class. A service class has one

method for each request/system operation. The method implements the business logic for that request. The data objects, which in this example is the `Order` class, are pure data with little or no behavior.

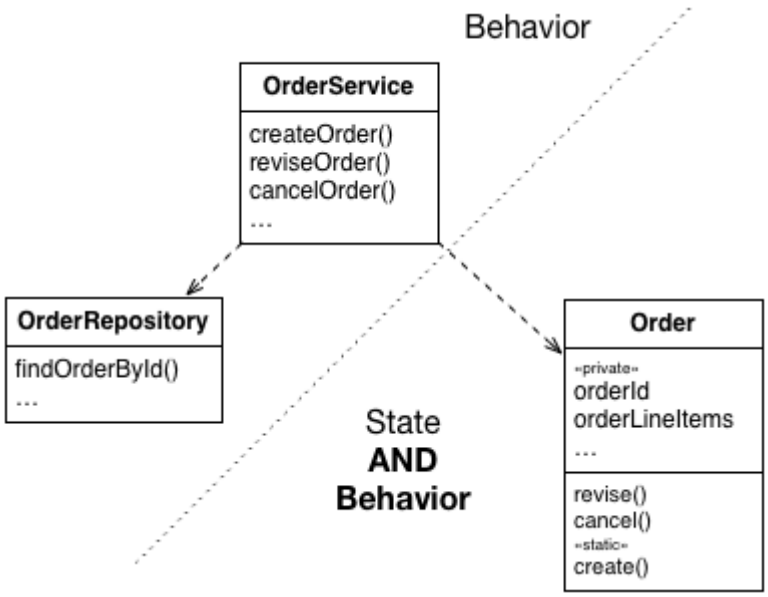
This style of design is highly procedural, and relies on few of the capabilities of object-oriented programming (OOP) languages. This what you would create if you were writing the application in C or another non-OOP language. Nevertheless, you should not be ashamed to use a procedural design when it is appropriate. This approach works well for simple business logic. The drawback is that this tends not to be good way to implement complex business logic.

5.1.2 Domain Model pattern

The simplicity of the procedural approach can be quite seductive. You can just write code without having to carefully consider how to organize the classes. The problem is that if your business logic becomes complex, then you can end up with code that's a nightmare to maintain. In fact, in the same way that a monolithic application has a habit of continually growing, transaction scripts have the same problem. Consequently, unless you are writing an extremely simple application you should resist the temptation to write procedural code and instead apply the Domain Model pattern and develop an object-oriented design.

In an object-oriented design, the business logic consists of an object model, which is a network of relatively small classes. These classes typically correspond directly to concepts from the problem domain. In such a design some classes have only either state or behavior but many contain both, which is the hallmark of a well-designed class. Figure 5.3 shows an example of the Domain Model pattern.

Figure 5.3. Organizing business logic as a domain model



As with the Transaction Script pattern, a `OrderService` class has a method for each request/system operation. However, when using the Domain Model pattern the service methods are usually very simple. That is because a service method almost always delegates to persistent domain objects, which contain the bulk of the business logic. A service method might, for example, simply load a domain object from the database and invoke one of its methods. In this example, the `Order` class has both state and behavior. Moreover, its state is private and can only be accessed indirectly via its methods.

Using an object-oriented design has a number of benefits. First, the design is easier to understand and maintain Instead of consisting of one big class that does everything, it consists of a number of small classes that each have a small number of responsibilities. In addition, classes such as Account, BankingTransaction, and OverdraftPolicy closely mirror the real world, which makes their role in the design easier to understand. Second, our object-oriented design is easier to test: each class can and should be tested independently. Finally, an object-oriented design is easier to extend because it can use well-known design patterns, such as the Strategy pattern and the Template Method pattern [Gang of Four], that define ways of extending a component without actually modifying the code

The Domain Model pattern works well. However, there are a number of problems with this approach, especially in a microservice architecture. To address those problems you need to use a refinement of OOD known as Domain Driven design.

5.1.3 About DDD

Domain-Driven design is a refinement of OOD and is an approach for developing

complex business logic. I first introduced DDD earlier in chapter {chapter-decomposition}, when describing how DDD subdomains are a useful concept when decomposing an application into services. When using DDD, each service has its own domain model, which avoids the problems of a single, application-wide domain model. Subdomains and the associated concept of Bounded Context are two of the strategic DDD patterns.

DDD also has some tactical patterns that are building blocks for domain models. Each pattern is a role that a class plays in a domain model and defines the characteristics of the class. The building blocks that have been widely adopted by developers include:

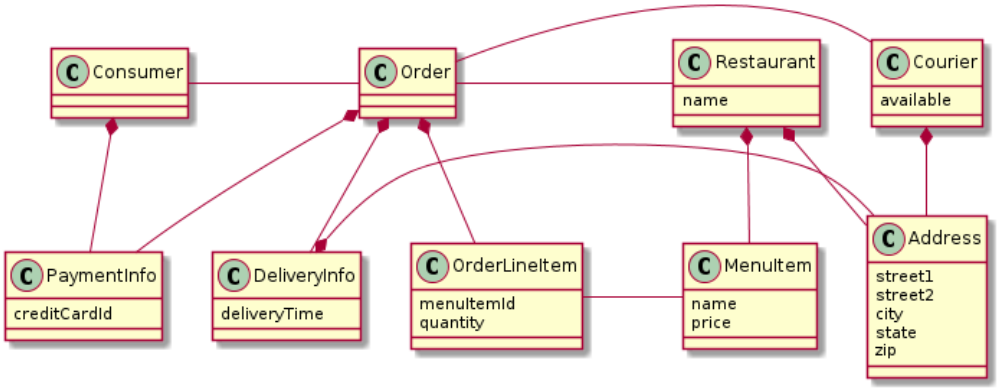
- Entity - an object that has a persistent identity. Two entities whose attributes have the same values are still different objects. In a Java EE application, classes that are persisted using JPA `@Entity` are usually DDD entities.
- Value object - an object that is a collection of values. Two value objects whose attributes have the same values can be used interchangeably. An example of a value object is a `Money` class, which consists of a currency and an amount.
- Factory - an object or method that implements object creation logic that is too complex to be done directly by a constructor. A factory might be implemented as a static method of a class.
- Repository - an object that provides access to persistent entities and encapsulates the mechanism for accessing the database.
- Service - an object that implements business logic that does not belong in an entity or a value object.

These building blocks used by many developers. Some are supported by frameworks such as JPA and the Spring framework. There is, however, one more building block, which has been generally ignored (myself included!) except by DDD purists: aggregates. It turns out, however, that aggregates are an extremely useful concept when developing microservices. Let's first look at some subtle problems with classic OOD that are solved by using aggregates.

5.2 Using DDD aggregates

In traditional object-oriented design, a domain model is a collection of classes and relationships between classes. The classes are usually organized into packages. For example, figure 2.5 shows part of a domain model for the FTGO application.

Figure 5.4. Organizing business logic as a domain model



This example has several classes corresponding to business objects: `Consumer`, `Order`, `Restaurant` and `Courier`. In many ways this works well. But interestingly in a traditional domain model is missing the explicit boundaries of each 'business object'. This lack of boundaries can sometimes cause problems, especially in microservice architecture.

5.2.1 The problem with fuzzy boundaries

Lets imagine, for example, that you want to perform an operation, such as a load or delete, on an `Order`. What exactly does that mean? What is the scope an operation? You would certainly load or delete the `Order` object. But in reality there is more to an `Order` than simply the `Order` object. There are also the order line items, the payment information, etc. The boundaries of a domain object are left up to intuition of a developer.

As well as a conceptual fuzziness, the lack of explicit boundaries, cause problems when updating a business object. A typical business object will have invariants, which are business rules that must be enforced at all times. Lets suppose, for example, that an `Order` has a minimum order amount. The application must ensure that any attempt to update an order does not violate an invariants such as the minimum order amount.

For example, FTGO provides a collaborative ordering feature that enables multiple consumers to work together to create an order. Let's imagine that two consumers - Sam and Mary - are working together on an order and simultaneously decide that the order exceeds their budget. Sam reduces the quantity of samosas, and Mary reduces the quantity of naan bread.

From the application's perspective, both consumers retrieve the order and its line items from the database. Both consumers then update a line item to reduce the cost of the order. From each consumers's perspective the order minimum is preserved. Here is the sequence of database transactions.

Consumer - Mary	Consumer - Sam
BEGIN TXN SELECT ORDER_TOTAL FROM ORDER WHERE ORDER ID = X SELECT * FROM ORDER_LINE_ITEM WHERE ORDER_ID = X … END TXN	BEGIN TXN SELECT ORDER_TOTAL FROM ORDER WHERE ORDER ID = X SELECT * FROM ORDER_LINE_ITEM WHERE ORDER_ID = X … END TXN
Verify minimum is met	
BEGIN TXN UPDATE ORDER_LINE_ITEM SET VERSION .. WHERE VERSION = <loaded version> END TXN	
	Verify minimum is met
	BEGIN TXN UPDATE ORDER_LINE_ITEM SET VERSION .. WHERE VERSION = <loaded version> END TXN

Each consumer changes a line item using a sequence of two transactions. The first transaction loads the order and its line items. The UI verifies that order minimum is satisfied before executing the second transaction. The second transaction updates the line item quantity using an optimistic offline locking check that verifies that the order line is unchanged since it was loaded by the first transaction.

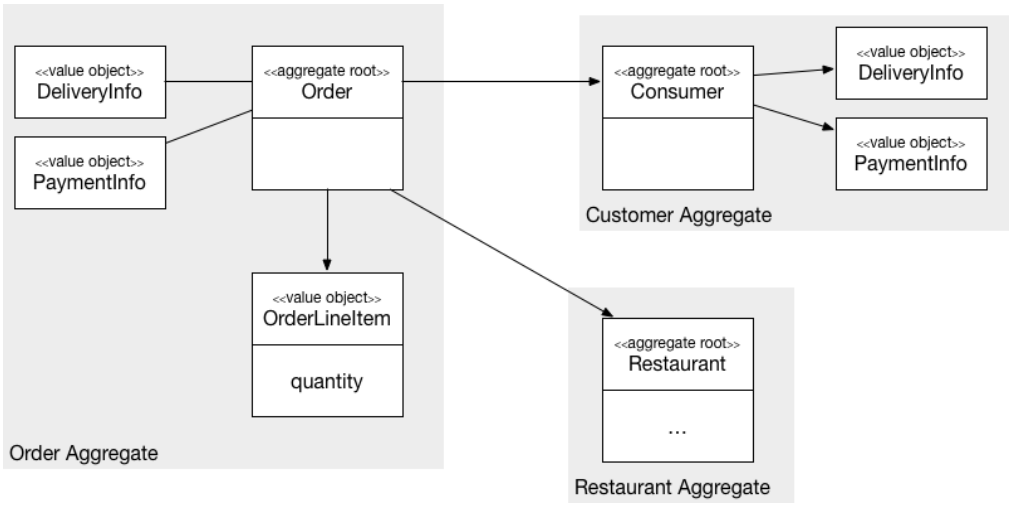
In this scenario, Sam reduces the order total by $X and Mary reduces it by $Y. As a result, the `Order` is no longer valid even though the application verified that the order was still satisfied the order minimum after each consumer's update. As you can see, directly updating part of a business object can result in the violate of the business rules. DDD aggregates are intended to solve this problem.

5.2.2 Aggregates have explicit boundaries

An aggregate is a cluster of domain objects within a boundary that can be treated as a unit. It consists of a root entity and possibly one or more other entities and value objects. Many 'business objects' are modeled as aggregates. For example, in chapter {chapter-decomposition} we created a rough domain model by analyzing the nouns used in the requirements and by domain experts. Many of those nouns, such as `Order`, `Consumer` and `Restaurant` are actually aggregates.

Figure 5.5 shows the `Order` aggregate and its boundary. An `Order` aggregate consists of an `Order` entity, one or more `OrderLineItem` value objects along with other value objects such as a delivery `Address`and `PaymentInformation`

Figure 5.5. The `Order` aggregate and its boundary.



Aggregates decompose a domain model into chunks, which are individually easier to understand. It also clarifies the scope of operations such as load, update and delete. These operations act on the entire aggregate rather than parts of it. An aggregate is often loaded it in its entirety from the database thereby avoiding any complications of lazy loading. When an aggregate is deleted, all of its objects and not others are deleted from a database.

Aggregates are consistency boundaries

Updating an entire aggregate rather than its parts solves the consistency issues, such as the example described earlier. Update operations are invoked on the aggregate root, which enforces invariants. Concurrency is handled by locking by, for example, using a version number, the aggregate root. For example, instead of updating line items quantities directly, a client must invoke a method on the root of the `Order` aggregate, which enforces invariants such as the minimum order amount. Note, however, this approach does not require the entire aggregate to be updated in the database. An application might, for example, only update the rows corresponding to the `Order` object and the updated `OrderLineItem`.

Identifying aggregates is key

In DDD, a key part of designing a domain model is identifying aggregates, their boundaries and their roots. The details of the aggregates' internal structure is secondary. The benefit of aggregates, however, goes far beyond modularizing a domain model. That is because aggregates must obey certain rules.

5.2.3 Aggregate rules

DDD requires aggregates to obey a set of rules. These rules ensure that an aggregate is a self-contained unit that can enforce its invariants. Lets look at each of the rules.

Rule #1: Reference only the aggregate root

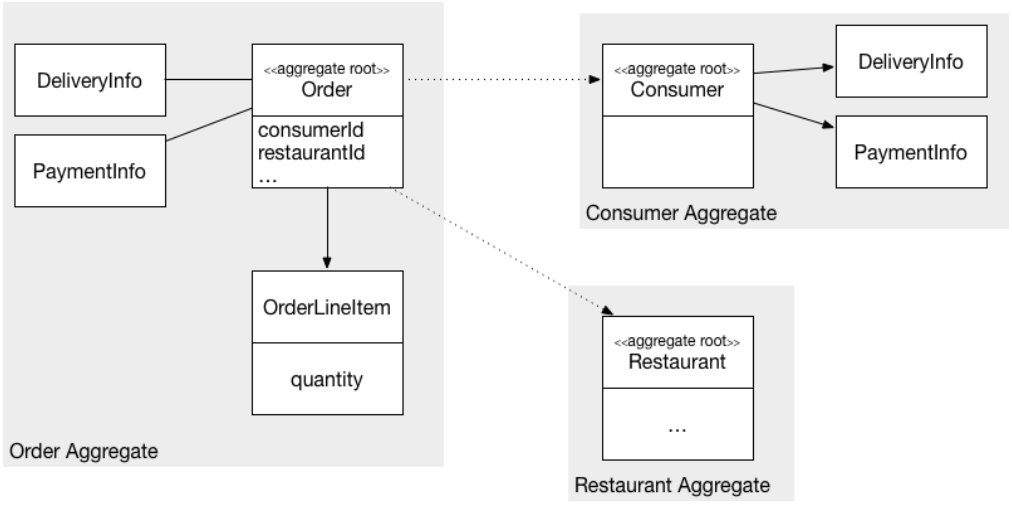
The example above illustrated the perils of directly updating `OrderLineItems` directly. The goal of the first aggregate rule is to eliminate this problem. It requires that the root entity is the only part of an aggregate that can be referenced by classes outside of the aggregate. A client can only update an aggregate by invoking a method on the aggregate root.

A service, for example, uses a repository to load an aggregate from the database and obtain a reference to the aggregate root. It updates an aggregate by invoking a method on the aggregate root. This rule ensures that the aggregate can enforce its invariant.

Rule #2: Inter-aggregate references must use primary keys

Another rule is that aggregates reference each other by identity (e.g. primary key) instead of object references. For example, as figure 5.6 shows, an `Order` references its `Consumer` using a `consumerId` rather than a reference to the `Consumer` object. Similarly, an `Order` references a `Restaurant` using a `restaurantId`.

Figure 5.6. References between aggregates are by primary key rather than by object reference



This approach is quite different than traditional object modeling, which considers foreign keys in the domain model to be a design smell. It does, however, have a number of benefits. The use of identity rather than object references means that the aggregates are loosely coupled. It ensures that the aggregate boundaries between aggregates are well defined and avoids accidentally updating a different aggregate Also, if an aggregate is part of another service there isn't a problem of object references that span services.

This approach also simplifies persistence since the aggregate is the unit of storage. It makes it easier to store aggregates in a NoSQL database such as MongoDB. It also eliminates the need for transparent lazy loading and its associated problems. Scaling

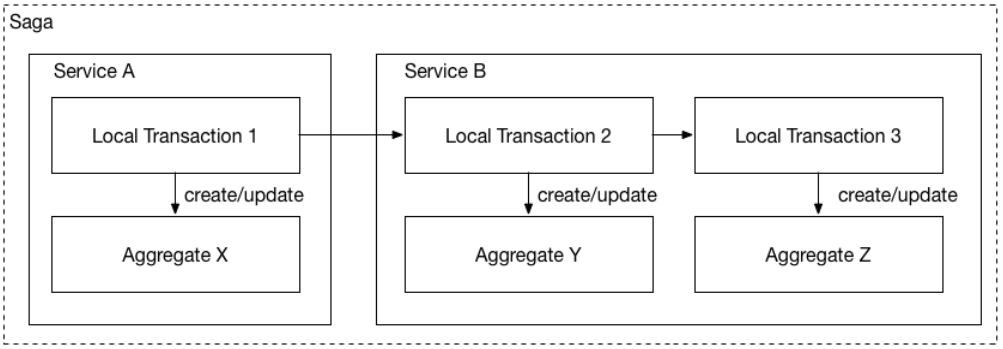
the database by sharding aggregates is relatively straightforward.

Rule #3: One transaction creates or updates one aggregate

Another rule aggregates must obey is that a transaction can only create or update a single aggregate. When I first read about it many years ago, this rule made no sense! At the time, I was developing traditional monolithic applications that used an RDBMS and so transactions could update multiple aggregates. Today, however, this constraint is perfect for the microservice architecture. It ensures that a transaction is contained within a service. This constraint also matches the limited transaction model of most NoSQL databases.

This rule makes it more complicated to implement operations that need to create or update multiple aggregates. However, this is exactly the problem that sagas, which I described in chapter {chapter-sagas}, are designed to solve. Each step of the saga creates or updates exactly one aggregate. Figure 5.7 shows how this works.

Figure 5.7. An application uses a saga to update multiple aggregates. Each step of the saga updates one aggregate.



In this example, the saga consists of three transactions. The first transaction updates aggregate 'X' in in service 'A'. The other two transactions are both in service 'B'. One transaction updates aggregate 'X' and the other updates aggregate 'Y'.

An alternative approach to maintaining consistency across aggregates is to cheat and update multiple aggregates within a transaction. For example, service 'B' could update aggregates 'Y' and 'Z' in a single transaction. Of course, this is only possible when using a database such as an RDBMS, that supports a rich transaction model. If you are using a NoSQL database that only has simple transactions there is no other option except to use sagas.

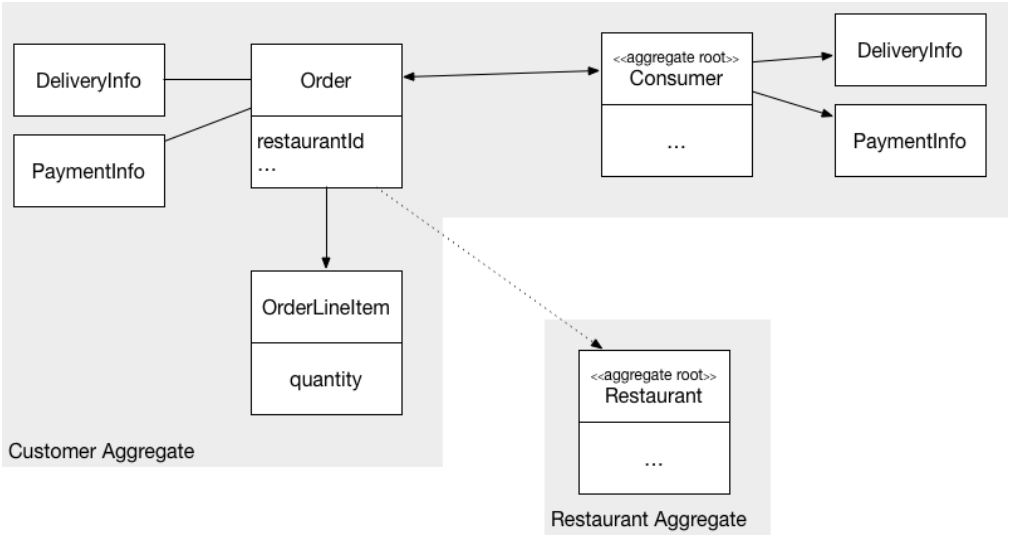
Or is there? It turns out that aggregate boundaries are not set in stone. When developing a domain model, you get to choose where the boundaries lie. But like a 20th century colonial power drawing national boundaries you need to be careful.

5.2.4 Aggregate granularity

When developing a domain model, a key decision you must make is how large to make each aggregate. On the one hand, aggregates should ideally be small. Because updates to each aggregate are serialized, more fine grained aggregates will increase the number of simultaneous requests that the application can handle and so improve scalability. It will also improve the user experience since it reduces the chance of two users attempting conflicting updates of the same aggregate. On the other hand, because an aggregate is the scope of transaction you might need to define larger aggregate in order to make a particular update atomic.

For example, earlier I described how in the FTGO application's domain model `Order` and `Consumer` are separate aggregates. An alternative design is to make `Order` part of the `Consumer` aggregate. Figure 5.8 shows this alternative design.

Figure 5.8. An alternative design defines an aggregate that contains `Customer` and `Order`



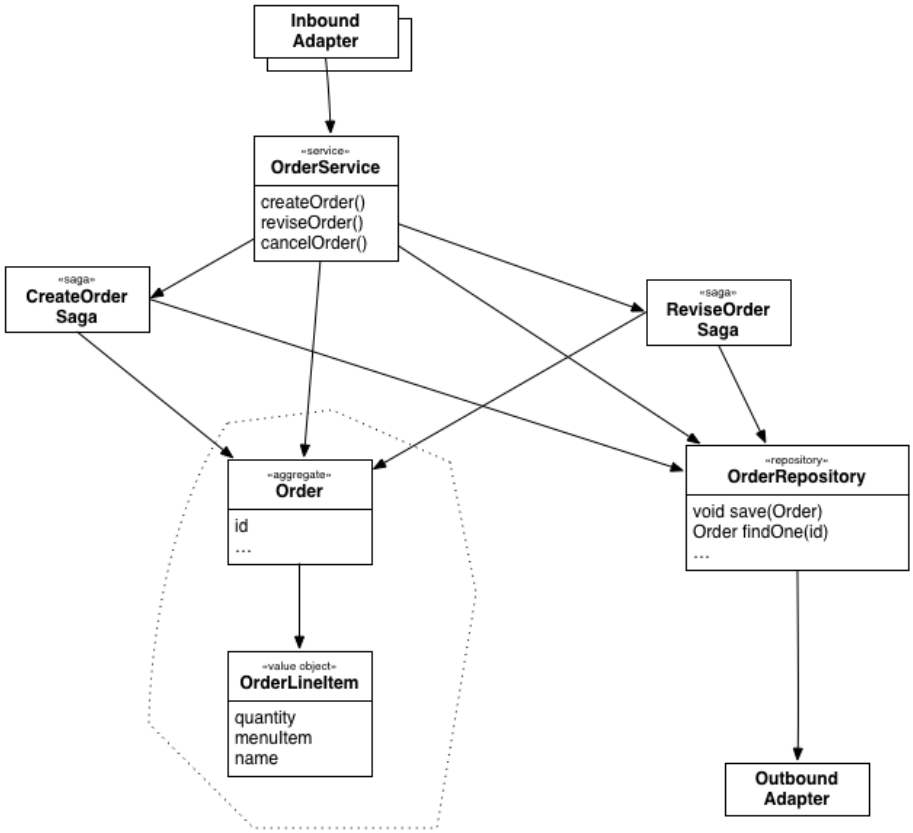
A benefit of this larger `Consumer` aggregate is that the application can atomically update a `Consumer` and one or more its `Orders`. A drawback of this approach is that it reduces scalability. Transactions that update different orders for the same customer would be serialized. Similarly, two users would conflict if they attempted to edit different orders for the same customer.

Another drawback of this approach in a microservice architecture is that it is an obstacle to decomposition. The business logic for `Orders` and `Consumers`, for example, must be collocated in the same service, which makes the service larger. Because of these issues, it is best to make aggregates as fine-grained as possible.

5.2.5 Designing business logic with aggregates

In a typical (micro)service, the bulk of the business logic consists of aggregates. The rest of the business logic resides in the domain services and the sagas. The sagas orchestrate sequences of local transactions in order to enforce data consistency. The services are usually how the business logic is invoked by inbound adapter. Services use a repository to retrieve aggregates from the database or save aggregates to the database. Each repository is implemented by an outbound adapter that accesses the database. Figure 5.9 shows the aggregate-based design of the business logic for the `Order Service`.

Figure 5.9. An aggregate-based design for the `Order Service`



The business logic consist of the `Order` aggregate, the `OrderService` service class, the `OrderRepository` and one or more sagas. The `OrderService` invokes the `OrderRepository` to save and load `Orders`. For simple requests that are local to the service, the service invokes `Order` aggregates directly. But if an update request spans multiple services then the `OrderService` will create a saga as I described in chapter {chapter-sagas}. Below we will take a look at the code but first, lets look a concept

which is closely related to aggregates: domain events.

5.3 Publishing domain events

Merriam-Webster lists several definitions of the word 'event' including:

> *a : something that happens : occurrence*
> *b : a noteworthy happening*
> *c : a social occasion or activity*
> *d : an adverse or damaging medical occurrence a heart attack or other cardiac event*
>
> *-- Merriam-Webster, https://www.merriam-webster.com/dictionary/event*

In the context of DDD, a domain event is something that has happened to an aggregate. It is represented by an class in the domain model. An event usually represents a state change. Consider, for example, an `Order` aggregate in the FTGO application. Its state changing events include `Order Created`, `Order Cancelled`, `Order Shipped` etc. An `Order` aggregate might, if there are interested consumers, publish one of the events each time it undergoes a state transition.

5.3.1 Why publish change events?

Domain events are useful because other parties - users, other applications, or other components within the same application - are often interested in knowing about an aggregate's state changes. Here are some example scenarios:

1. Notifying a service that maintains a replica that the source data has changed.
   a. In a microservice architecture, it is common for a service to maintain a replica of data that is owned by a different service.
   b. Domain events are a convenient way to keep those replicas up to date.
   c. Whenever a service updates its data, it publishes a domain event, which tells other services to update their replicas.
2. Notifying a different application via a registered webhook, or via a message broker in order to trigger the next step in a business process.
3. Notifying a different component of the same application in order, for example, to send a WebSocket message to a user's browser or update a text database such as ElasticSearch.
4. Sending notifications - text messages or emails - to users informing them that their order has shipped, their Rx prescription is ready for pick up, or their flight is delayed.
5. Monitoring domain events to verify that the application is behaving correctly.
6. Analyzing events to model user behavior.

The trigger for the notification in all of these scenarios is the state change of an aggregate in an application's database.

5.3.2 What is a domain event

A domain event is a class whose name is in past tense or to be more precise a past-

participle verb. It has properties that meaningfully convey the event. Each property is either primitive value or value object. For example, an `OrderCreated` event class has an `orderId` property.

A domain event typically also has metadata such as the event id, and a timestamp. It might also have the identity of the user that made the change, since that is useful for auditing. The metadata can be part of the event object, perhaps defined in a superclass. Alternatively, the event metadata can be in an envelope object that wraps the event object. The `id` of the aggregate that emitted the event might also be part of the envelope rather than an explicit event property.

The `OrderCreated` event is an example of a domain event. It does not have any fields because the Order's `id` is part of the event envelope. Listing 5.1 shows the `OrderCreated` event class and the `DomainEventEnvelope` class.

Listing 5.1. The `OrderCreated` event and the `DomainEventEnvelope` class, which contains an event and its metadata

```
interface DomainEvent {}

class OrderCreated implements DomainEvent {}

class DomainEventEnvelope<T extends DomainEvent> {
  private String aggregateType;
  private Object aggregateId;
  private T event;
  ...
}
```

The `DomainEvent` interface is a marker interface that identifies a class a DomainEvent. The `DomainEventEnvelope` is a class that contains event metadata and the event object. It is a generic class that is parameterized by the domain event type.

5.3.3 Event enrichment

Lets imagine, for example, that you are writing an event consumer that processes `Order` events. On the one hand, the `OrderCreated` event class shown above captures the essence of the event. But on the other hand, many event consumers need the order details when processing an `OrderCreated` event.

Sometimes the data needed by event consumers is readily available to the aggregate at the time it publishes the event. In many cases the data is often part of the aggregate. The aggregate can simply include that data in the event. This technique, which simplifies event consumers since not longer have to request that data, is is known as event enrichment. In the `OrderCreated` event, the `Order` aggregate can enrich the event by including the order details. Listing 5.2 shows the enriched event.

Listing 5.2. The enriched `OrderCreated` event containing data that its consumers typically need

```
class OrderCreated implements Event {
  private List<OrderLineItem> lineItems;
  private DeliveryInformation deliveryInformation;
  private PaymentInformation paymentInformation;
  ...
}
```

This version of the `OrderCreated` event contains the order details. As a result, an event consumer no longer needs to fetch that data.

While event enrichment simplifies consumers, the drawback is that it risks making the event classes less stable. An event class potentially needs to change whenever the requirements of its consumers change. This can reduce maintainability since this kind of change can impact multiple parts of the application. It can also be futile effort to satisfy every consumer. Fortunately, in many situations it is fairly obvious which properties to include in an event. Now that we have covered the basics of domain events, let's look at how to discover them.

5.3.4 Identifying domain events

There are a few different strategies for identifying domain events. Often the requirements will describe scenarios where notifications are required. The requirements might include language such "when X happens do Y". For example, one requirement in the FTGO application is "When an Order is placed send the consumer an email". A requirement for a notification suggests the existence of a domain event.

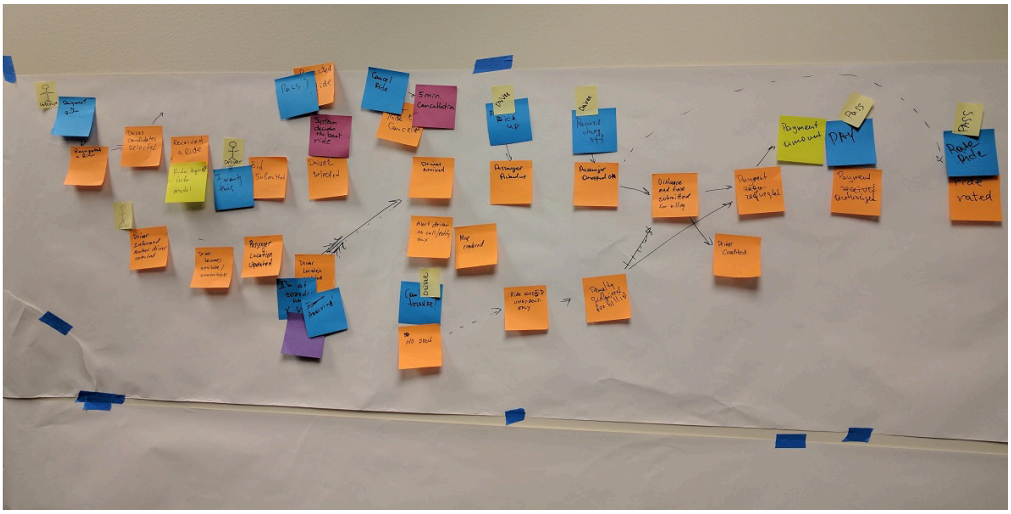
Another approach, which is increasing in popularity, is to use event storming. Event storming is an event-centric workshop format for understanding a complex domain. It involves gathering domain experts in a room, lots of post-it notes and a very large surface - whiteboard or paper roll - to stick the notes on. The result of event storming is an event-centric domain model consisting of aggregates and events.

Event storming consist of three main steps:

1. Brainstorm events - ask the domain experts to brainstorm the domain events. Domain events are represented by orange post-it notes that are laid out in a rough timeline on the modeling surface.
2. Identify event event triggers - ask the domain experts to identify the trigger of each event, which is one of the following:
    - user actions - represented as a command using a blue post-it note
    - external system - represented by a purple post-it note
    - another domain event
    - passing of time
3. Identify aggregates - ask the domain experts to identify the aggregate that consumes each command and emits the corresponding event.

Figure 5.10 shows the result of an event storming workshop.

Figure 5.10. The result of an event storming workshop. The orange post-it notes are the events laid out along a timeline.



Event storming is a useful technique for quickly creating a domain model. Now that we have covered the basics of domain events lets now look at the mechanics of generating and publishing them.

5.3.5 Generating and publishing domain events

Communicating using domain events is a form of asynchronous messaging, which I described in chapter {chapter-ipc}. The before the business logic can publish them to a message broker, it must first create them. Lets look at how to that.

Generating domain events

Conceptually, domain events are published by aggregates. An aggregate knows when its state changes and hence what event to publish. An aggregate could invoke a messaging API directly. The drawback of this approach is that since aggregates cannot use dependency injection the messaging API would need to passed around as a method argument. That would intertwine infrastructure concerns and business logic, which is extremely undesirable.

A better approach is to split responsibility between the aggregate and the service (or equivalent class) that invokes it. Services can use dependency injection to obtain a reference to the messaging API and so can easily publish events. The aggregate generates the events whenever its state changes and returns them to the service. There are a couple of different ways an aggregate can return events back to the service. One option is for the return value of an aggregate method to include a list of events. For example, listing 5.3 shows how a `RestaurantOrder` aggregate's `accept()` method can

return a `RestaurantOrderAcceptedEvent` to its caller:

Listing 5.3. The `RestaurantOrder` aggregate's command method updates the aggregate and returns a `RestaurantOrderAcceptedEvent`

```
public class RestaurantOrder {

   public List<DomainEvent> accept(LocalDateTime readyBy) {
    ...
    this.acceptTime = LocalDateTime.now();
    this.readyBy = readyBy;
    return singletonList(new RestaurantOrderAcceptedEvent(readyBy));
  }
}
```

The service, which invokes the aggregate root's method, then publishes the events. For example, listing 5.4 shows how the `RestaurantOrderService` invokes `RestaurantOrder.accept()` and publishes the events:

Listing 5.4. The `RestaurantOrderService` calls `RestaurantOrder.accept()` and publishes the domain events

```
public class RestaurantOrderService {

  @Autowired
  private RestaurantOrderRepository restaurantOrderRepository;

  @Autowired
  private DomainEventPublisher domainEventPublisher;

  public void accept(long orderId, LocalDateTime readyBy) {
    RestaurantOrder restaurantOrder = restaurantOrderRepository.findOne(orderId);
    List<DomainEvent> events = restaurantOrder.accept(readyBy);
    domainEventPublisher.publish(RestaurantOrder.class, orderId, events);
  }
```

The `RestaurantOrderService` defines an `accept()` method. This method first invokes the `RestaurantOrderRepository` to load the `RestaurantOrder` from the database. It then invokes updates the `RestaurantOrder`. The `RestaurantOrderService` then publishes any returned events by calling `DomainEventPublisher.publish()`, which is described below.

This approach is quite simple. Methods that would otherwise have a void return type now return `List<Event>`. The only potential drawback is the return type of non-void methods is now more complex. They must return an object containing the original return value and `List<Event>` You will see an example of such a method below.

Another option is for the aggregate root to accumulate events in a field. The service then retrieves the events and publishes them. For example, listing 5.5 shows a variant of the `RestaurantOrder` class that works this way.

Listing 5.5. The `RestaurantOrder` extends a superclass, which records domain events

```
public class RestaurantOrder extends AbstractAggregateRoot {

  public void accept(LocalDateTime readyBy) {
    ...
    this.acceptTime = LocalDateTime.now();
    this.readyBy = readyBy;
    registerDomainEvent(new RestaurantOrderAcceptedEvent(readyBy));
  }

}
```

`RestaurantOrder` extends `AbstractAggregateRoot`, which defines a `registerDomainEvent()` method that records the event. A service would call `AbstractAggregateRoot.getDomainEvents()` to retrieve those events.

My preference is for the first option of method returning events to the service. However, accumulating events in the aggregate root is also a viable option. In fact, the Spring Data Ingalls release train implements a mechanism that automatically publishes events to the Spring Application drawback. The main drawback is that to reduce code duplication aggregate roots should extend a superclass such as `AbstractAggregateRoot`, which might conflict with a requirement to extend some other superclass. Another issue is while its easy for the aggregate root's methods to call `registerDomainEvent()` methods in other classes in the aggregate would find it challenging. They would mostly likely need to somehow pass the events to the aggregate root.

How to reliably publish domain events?

In chapter {chapter-ipc}, I described how to reliably send messages as part of a local database transaction. Domain events are no different. A service must use transactional messaging to publish events to ensure that they are published as part of the transaction that updates the aggregate in the database. The Tram framework, which I described in chapter {chapter-ipc}, implements such a mechanism, It insert events into an `OUTBOX` table as part of the ACID transaction that updates the database, After the transaction commits, the events that were inserted into the `OUTBOX` table are then published to the message broker.

The `Tram` framework provides a `DomainEventPublisher` interface, which is shown in listing 3.1. It defines several overloaded `publish()` methods that take the aggregate type and id as parameters, along with a list of domain events.

Listing 5.6. The Tram framework's `DomainEventPublisher` interface

```
public interface DomainEventPublisher {
 void publish(String aggregateType, Object aggregateId, List<DomainEvent>
domainEvents);
```

It uses the Tram framework's `MessageProducer` interface to publish those events transactionally.

5.3.6 Consuming domain events

Domain events are ultimately published as messages to a message broker, such as Apache Kafka. A consumer could the use the broker's client API directly. However, it is more convenient to use a higher-level API such as the Tram framework's `DomainEventDispatcher`, which I described in chapter {chapter-ipc}. A `DomainEventDispatcher` dispatches domain events to the appropriate handle method. Listing 3.3 shows an example event handler class. `RestaurantOrderEventConsumer` subscribes to events published by the `Restaurant Service` whenever a restaurant's menu is updated. It is responsible for keeping the `Restaurant Order Service's` replica of the data up to date.

Listing 5.7. The Tram framework's `DomainEventDispatcher` class which dispatches event messages to event handler methods

```
public class RestaurantOrderEventConsumer {
  @Autowired
  private RestaurantRepository restaurantRepository;

  @Autowired
  private DomainEventPublisher eventPublisher;

  public DomainEventHandlers domainEventHandlers() {                    (1)
    return DomainEventHandlersBuilder

.forAggregateType("net.chrisrichardson.ftgo.restaurantservice.Restaurant")
            .onEvent(RestaurantMenuRevised.class, this::reviseMenu)
            .build();
  }

  public void reviseMenu(DomainEventEnvelope<RestaurantMenuRevised> de) {   (2)

    String restaurantIds = de.getAggregateId();
    Restaurant restaurant =
restaurantRepository.findOne(Long.parseLong(restaurantIds));

    List<DomainEvent> events =
restaurant.reviseMenu(de.getEvent().getRevisedMenu());

    eventPublisher.publish(RestaurantOrder.class, restaurantIds, events);
  }

}
```

(1) Map events to event handlers

(2) An event handler for the `RestaurantMenuRevised` event

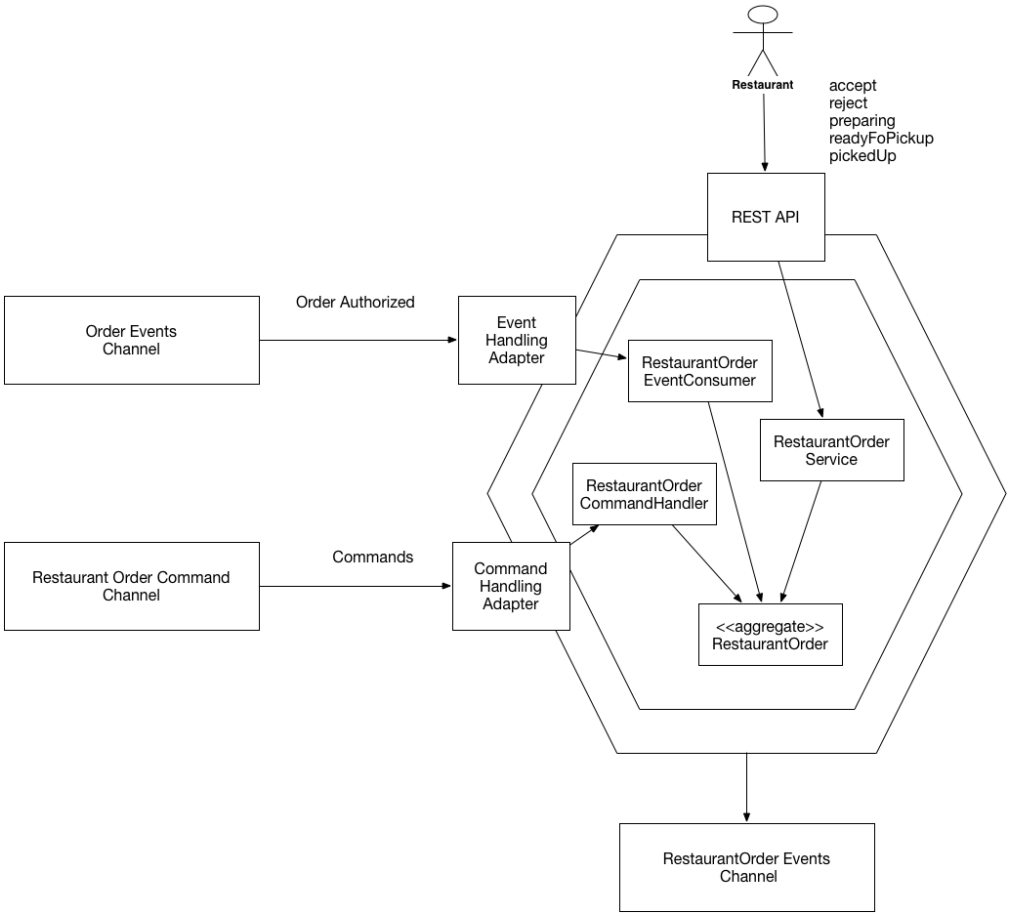
The `reviseMenu()` method handles `RestaurantMenuRevised` events. It calls `Restaurant.reviseMenu()`, which updates the restaurant's menu. That method returns

a list of domain events, which are published by the event handler. Now that we have looked at aggregates and domain events lets now look at some example business logic that is implemented using aggregates.

5.4 Restaurant order management business logic

The first example is the `RestaurantOrderService`, which enables a restaurant to manage their orders. The two main aggregates in this service are the `Restaurant` and `RestaurantOrder` aggregates. The `Restaurant` aggregate, which knows the restaurant's menu and opening hours, validates orders. A `RestaurantOrder` represents an order that restaurants must prepare for pickup by a courier. Figure 5.11shows the key components of the service's business logic as well the adapters that invoke the business logic in response to REST API calls, and command and event messages.

Figure 5.11. The design of the `Restaurant Order Service`



The `Restaurant Order Service` handles two kinds of external requests. First, it has a REST API, which is invoked by a user interface used by workers at the restaurant. The Restaurant UI uses the REST API to update the state of an order. These requests invoke the domain service called `RestaurantService`. Second, the `Restaurant Order Service` consumes command messages, which are asynchronous requests to update the state of a `RestaurantOrder`. These messages invoke the `RestaurantOrderCommandHandler` to create or update a `RestaurantOrder` aggregate. Lets take a closer look at the design of the `RestaurantOrderService` design starting with the `RestaurantOrder` aggregate.

5.4.1 The `RestaurantOrder` aggregate

The `RestaurantOrder` is one of the aggregates of the `RestaurantOrderService`. As I described in chapter {chapter-decomposition}, when describing the concept of a Bounded Context, this aggregate represents the restaurant's view of an order. It does not contain information about the consumer such as their identity, the delivery information or payment details. It is simply focussed on enabling a restaurant's kitchen to prepare the order for pickup. Moreover, the `RestaurantOrderService` does not generate a unique id for this aggregate. Instead, it uses the 'id' supplied by the `OrderService`. Lets first look at the structure of this class and then we will look at its methods.

Structure of the `RestaurantOrder` class

The `RestaurantOrder` class is similar to a traditional domain class. The main difference is that references to other aggregates are by primary key. Listing 5.3 shows an excerpt of the code for this class.

Listing 5.8. Part of the `RestaurantOrder` class, which is a JPA entity

```
@Entity(table="restaurant_orders")
public class RestaurantOrder {

  @Id
  private Long id;
  private RestaurantOrderState state;
  private Long restaurantId;

  @ElementCollection
  @CollectionTable(name="restaurant_order_line_items")
  private List<RestaurantOrderLineItem> lineItems;

  private LocalDateTime readyBy;
  private LocalDateTime acceptTime;
  private LocalDateTime preparingTime;
  private LocalDateTime pickedUpTime;
  private LocalDateTime readyForPickupTime;
  ...
```

This class is persisted with JPA and is mapped to the `RESTAURANT_ORDERS` table. The `restaurantId` field is a `Long` rather than an object reference to a `Restaurant`.

The `readyBy` field stores the estimate of when the order will be ready for pickup. The `RestaurantOrder` class has several fields that track the history of the order including `acceptTime`, `preparingTime` and `pickupTime`. Lets now look at this class's methods.

Behavior of the `RestaurantOrder` aggregate

The `RestaurantOrder` aggregate defines several methods. As you saw earlier, it has a static `create()` method, which is a factory method that creates a `RestaurantOrder`. There are also some methods that invoked when the restaurant updates the state of the order:

- `accept()` - the restaurant has accepted the order
- `preparing()` - the restaurant has started preparing the order, which means that it can no longer be changed or cancelled
- `readyForPickup()` - the order can now be picked up

Listing 5.9 shows some of its methods.

Listing 5.9. Some of the `RestaurantOrder`'s methods

```
public class RestaurantOrder {

public static ResultWithEvents<RestaurantOrder> create(Long id,
RestaurantOrderDetails details) {
  return new ResultWithEvents<>(new RestaurantOrder(id, details), new
RestaurantOrderCreatedEvent(id, details));
}

public List<RestaurantOrderPreparationStartedEvent> preparing() {
  switch (state) {
    case ACCEPTED:
      this.state = RestaurantOrderState.PREPARING;
      this.preparingTime = LocalDateTime.now();
      return singletonList(new RestaurantOrderPreparationStartedEvent());
    default:
      throw new UnsupportedStateTransitionException(state);
  }
}

public List<DomainEvent> cancel() {
    switch (state) {
      case CREATED:
      case ACCEPTED:
        this.state = RestaurantOrderState.CANCELLED;
        return singletonList(new RestaurantOrderCancelled());
      case READY_FOR_PICKUP:
        throw new RestaurantOrderCannotBeCanceledException();

      default:
        throw new UnsupportedStateTransitionException(state);

    }
  }
```

The `create()` method creates a `RestaurantOrder`. The `preparing()` method is called when the restaurant starts preparing the order. It changes the state of the order to `PREPARING`, records the time, and publishes an event. The `cancel()` method is called when a user attempts to cancel an order. If the cancellation is allowed this method changes the state of the order and returns an event. Otherwise, it throws an exception.

These methods are invoked in response to REST API requests as well as events and command messages. Lets look at the classes that invoke the aggregate's method.

The `RestaurantOrderService` domain service

The `RestaurantOrderService` is invoked by the REST API. It defines various methods for changing the state of an order including `accept()`, `reject()`, `preparing()` etc. Each method loads the specifies aggregate, calls the corresponding method on the aggregate root and publishes any domain events. Listing 5.4 shows an excerpt of this class, which shows the `accept()` method.

Listing 5.10. An excerpt of the `RestaurantOrderService`. The `accept()` method updates the specified `RestaurantOrder` and publishes domain events.

```
public class RestaurantOrderService {

  @Autowired
  private RestaurantOrderRepository restaurantOrderRepository;

  @Autowired
  private DomainEventPublisher domainEventPublisher;

  public void accept(long orderId, LocalDateTime readyBy) {
    RestaurantOrder restaurantOrder =
       restaurantOrderRepository.findOne(orderId);
    List<DomainEvent> events = restaurantOrder.accept(readyBy);
    domainEventPublisher.publish(RestaurantOrder.class, orderId, events);
  }

  // ...
}
```

The `accept()` method is invoked when the restaurant accepts a new order. It has two parameters:

- `orderId` - id of the order to accept.
- `readyBy` - the estimated time when the order will be ready for pickup

This method retrieves the `RestaurantOrder` aggregate and calls its `accept()` method. It publishes any generated events. Lets now look at the class that handles asynchronous commands.

The `RestaurantOrderCommandHandler` class

The `RestaurantOrderCommandHandler` class is responsible for handling command messages sent by the various sagas implemented by the `Order Service`. This class

defines a handler method for each command, which either creates or updates a `RestaurantOrder`. Listing 5.11 shows an excerpt of this class including some example command handler methods.

Listing 5.11. An excerpt of the `RestaurantOrderCommandHandler`, which handles command messages sent by sagas

```
public class RestaurantOrderServiceCommandHandler {

 @Autowired
 private RestaurantOrderRepository restaurantOrderRepository;

 @Autowired
 private DomainEventPublisher domainEventPublisher;

 public CommandHandlers commandHandlers() {                                    (1)
  return CommandHandlersBuilder
          .fromChannel("orderService")
          .onMessage(CreateRestaurantOrder.class, this::createRestaurantOrder)
          .onMessage(ConfirmCreateRestaurantOrder.class,
                  this::confirmCreateRestaurantOrder)
          .onMessage(CancelCreateRestaurantOrder.class,
                  this::cancelCreateRestaurantOrder)
          .build();
 }

 private Message createRestaurantOrder(CommandMessage<CreateRestaurantOrder>
                                                  cm) {
  CreateRestaurantOrder command = cm.getCommand();
  long restaurantId = command.getRestaurantId();
  Long restaurantOrderId = command.getOrderId();
  RestaurantOrderDetails restaurantOrderDetails =
      command.getRestaurantOrderDetails();

  try {
    restaurant.verifyRestaurantDetails(restaurantOrderDetails);              (2)
  } catch (RestaurantDetailsVerificationException e) {
    return withFailure();
  }

  ResultWithEvents<RestaurantOrder> rwe = RestaurantOrder                    (3)
          .create(restaurantOrderId, restaurantId,
                  command.getRestaurantOrderDetails());
  restaurantOrderRepository.save(rwe.result);                                (4)
  domainEventPublisher.publish(RestaurantOrder.class, restaurantOrderId      (5)
          .toString(), rwe.events);
  return withLock(RestaurantOrder.class, restaurantOrderId).withSuccess();   (6)
 }

 private Message confirmCreateRestaurantOrder
         (CommandMessage<ConfirmCreateRestaurantOrder> cm) {
  Long orderId = cm.getCommand.getOrderId();
  RestaurantOrder ro = restaurantOrderRepository.findOne(orderId);           (7)
  List<DomainEvent> events = ro.confirmCreate();                             (8)
  domainEventPublisher
      .publish(RestaurantOrder.class, orderId.toString(), events);
```

```
 return withSuccess();
}

  ...
```

1. Map command messages to handler messages
2. Verify order details
3. Create a `RestaurantOrder`
4. Persist it in the database
5. Publish domain events
6. Reply to command
7. Find existing `RestaurantOrder`
8. Update it

The responsibilities of each method are as follows:

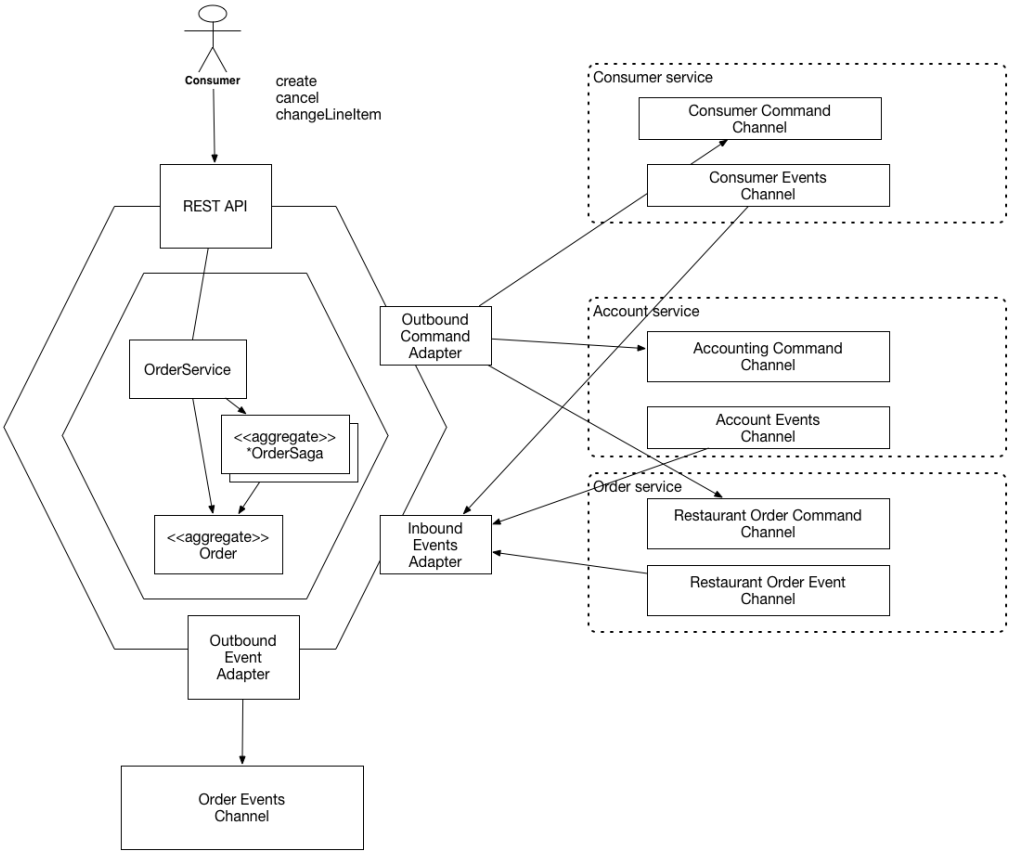
- `create()` - creates a `RestaurantOrder` in the `CREATE_PENDING` state
- `confirmCreate()` - confirms the creation of the `RestaurantOrder`
- `cancelCreate()` - cancels the creation of the `RestaurantOrder`

The `create()` method validates the request to create the `RestaurantOrder`, creates a new `RestaurantOrder` and saves it in the database. The `confirmCreateRestaurantOrder()` method updates a `RestaurantOrder`. All of the handler methods publish domain events. Now that you have seen the business logic for a relatively simple service, lets look at a more complex example: the `Order Service`.

5.5 *Order service* business logic

The `Order` aggregate is the central aggregate of the `Order Service`. As you saw in earlier chapters, the `Order Service` enables a consumer to create, update and cancel orders. Figure 5.12 shows the high-level design of the service.

Figure 5.12. The design of the `Order Service`



The service has various adapters including a REST API for creating, canceling and revising orders and adapters for exchanging messages with other services. It sends commands to services such as the `Consumer Service`, the `Accounting Service` and the `Restaurant Order Service`. This service also publishes domain events. Its domain logic consists of the `OrderService` domain service, some saga classes and the `Order` aggregate. Lets take a look at the `Order` aggregate.

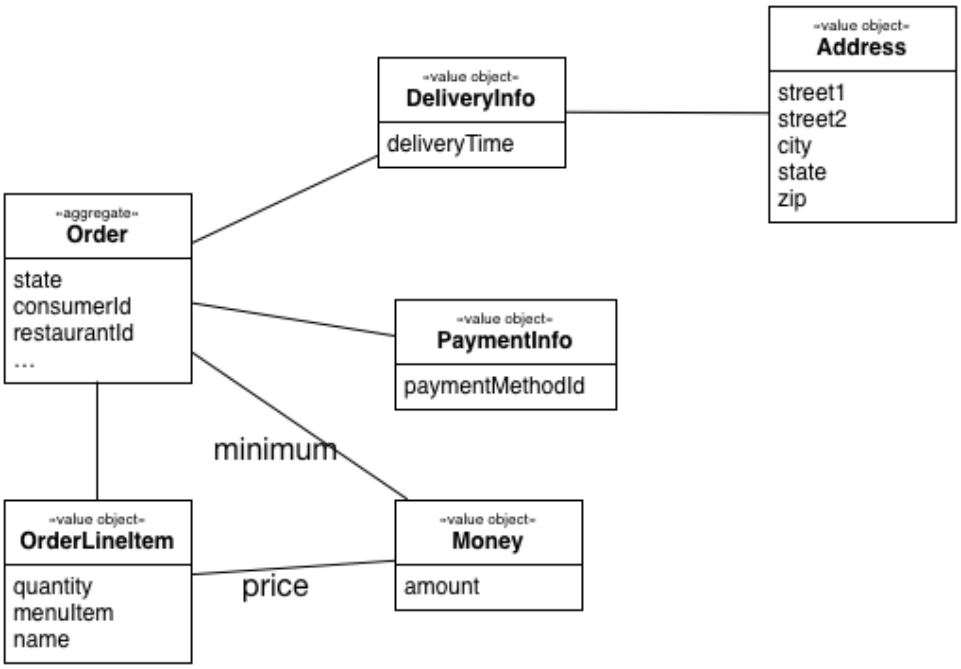
5.5.1 The `Order` Aggregate

The `Order` aggregate represents an order placed by a consumer. Lets first look at the structure of the `Order` aggregate and then look at its methods.

The structure of the `Order` aggregate

The root of the `Order` aggregate is the `Order` class. The `Order` aggregate also consists of value objects such as `OrderLineItem`,`DeliveryInfo` and `PaymentInfo`. Figure 5.13 shows the structure of the `Order`aggregate.

Figure 5.13. The design of the `Order` aggregate



The `Order` class has a collection of `OrderLineItems`. Since the `Order's Consumer` and the `Restaurant` are other aggregates it references them by primary key value. The `Order` class has a `DeliveryInfo` class, which stores the delivery address and the desired delivery time, and a `PaymentInfo`, which stores the payment info. Listing 5.12 shows the code.

Listing 5.12. The `Order` class and its fields

```
@Entity
@Table(name="orders")
@Access(AccessType.FIELD)
public class Order {

  @Id
  @GeneratedValue
  private Long id;

  @Version
  private Long version;

  private OrderState state;
  private Long consumerId;
  private Long restaurantId;

  @Embedded
  private OrderLineItems orderLineItems;
```

```
@Embedded
private DeliveryInformation deliveryInformation;

@Embedded
private PaymentInformation paymentInformation;

@Embedded
private Money orderMinimum;
```

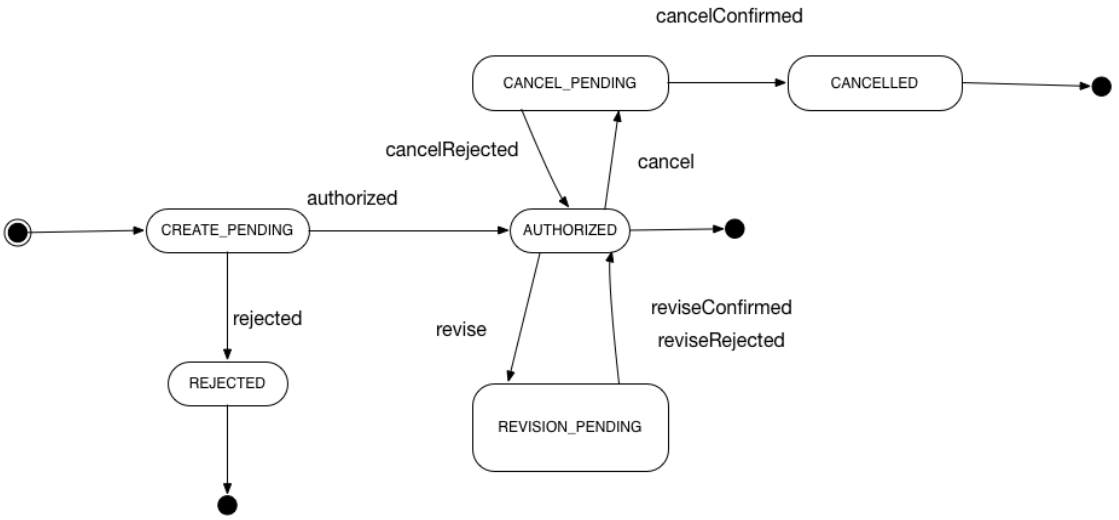
This class is persisted with JPA and is mapped to the `ORDERS` table. The `id` field is the primary key. The `version` field is used for optimistic locking. The state of an `Order` is represented by the `OrderState`enumeration. The `DeliveryInformation` and `PaymentInformation` fields are mapped using the `@Embedded` annotation and are stored as columns of the `ORDERS` table. The `orderLineItems` field is an embedded object that contains the order line items. The `Order` aggregate consists of more than just fields. It also implements business logic, which can be described by a state machine. Lets take a look at the state machine.

The `Order` aggregate state machine

The `Order` aggregate has several methods including `createOrder()`, `cancelOrder()` and `reviseOrder()`. Many of these methods create sagas. As a result, they are more complex than those for the `RestaurantOrder` aggregate. Typically, either the method or the first step of the saga verifies that the operation can be performed and changes the state of the `Order` to a 'pending' state. Eventually, once the saga has invoked the participating services, it then updates the `Order` to reflect the outcome of the saga. For example, as I described in chapter {chapter-sagas}, the `Create Order Saga` has multiple participants services including `Consumer Service`, `Accounting Service`, and `Restaurant Order Service`. It first creates an `Order` in a `CREATE_PENDING` state and then later changes its state to either `AUTHORIZED` or `REJECTED`.

The behavior of an `Order` can be modeled as a state machine. Figure 5.14 shows the state machine.

Figure 5.14. The state machine model of the `Order` aggregate



An `Order` is first created in the `CREATE_PENDING` state. Once it has been verified and the consumer's credit card successfully authorized, the `Order` transitions to the `AUTHORIZED` state. If either validation or authorization fails then it becomes `REJECTED`. Operations such as `revise()` and `cancel()` transition the `Order` to a 'pending' state. Once the operation has been verified the `Order` transitions to some other state. For example, a successful `cancel()` operation first transitions the `Order` to the `CANCEL_PENDING` state and then to the `CANCELLED` state. Lets now look at the how the `Order` aggregate implements this state machine.

The `Order` aggregate's methods

I'll first describe the business logic that creates an `Order`. After that we will look at how an `Order` is updated. Listing 5.13 shows the `Order's` methods that are involved in creating an `Order`.

Listing 5.13. The methods of the `Order` class that involved in the order creation process

```
public class Order { ...

  public static ResultWithEvents<Order> createOrder(OrderDetails orderDetails) {
    Order order = new Order(orderDetails);
    List<DomainEvent> events =
         singletonList(new OrderCreatedEvent(orderDetails));
    return new ResultWithEvents<>(order, events);
  }

  public Order(OrderDetails orderDetails) {
    this.orderLineItems = new OrderLineItems(orderDetails.getLineItems());
    this.orderMinimum = orderDetails.getOrderMinimum();
```

```
    this.state = CREATE_PENDING;
  }
  ...

  public List<DomainEvent> noteAuthorized() {
    switch (state) {
      case CREATE_PENDING:
        this.state = AUTHORIZED;
        return singletonList(new OrderAuthorized());
      ...
      default:
        throw new RuntimeException("Unknown state: " + state);
    }
  }

  public List<DomainEvent> noteRejected() {
    switch (state) {
      case CREATE_PENDING:
        this.state = REJECTED;
        return singletonList(new OrderRejected());
        ...
      default:
        throw new UnsupportedStateTransitionException(state);
    }

  }
```

The `createOrder()` method is a static method that creates an Order and publishes an `OrderCreatedEvent`. The initial state of the `Order` is `CREATE_PENDING`. When the `CreateOrderSaga` completes it will invoke either `noteAuthorized()` or `noteRejected()`. The `noteAuthorized()` method is invoked when the consumer's credit card has been successfully authorized. The `noteRejected()` method is called when one of the services rejects the order or authorization fails. As you can see, the `state` of the `Order` aggregate determines its behavior of most of its methods. Like the `RestaurantOrder` aggregate, it also emits events.

In addition to `createOrder()`, the `Order` class defines several update methods. For example, the `ReviseOrderSaga` revises an order by first invoking the `revise()` method and then, once it has verified that the revision can be made, it invokes the `confirmRevised()` method. Listing 5.14 shows these methods.

Listing 5.14. The `Order` method for revising an `Order`

```
class Order ...

  public List<DomainEvent> revise(OrderRevision orderRevision) {
    switch (state) {

      case AUTHORIZED:
        LineItemQuantityChange change =
orderLineItems.lineItemQuantityChange(orderRevision);
        if (change.newOrderTotal.isGreaterThanOrEqual(orderMinimum)) {
          throw new OrderMinimumNotMetException();
        }
```

```
        this.state = REVISION_PENDING;
        return singletonList(new OrderRevisionProposed(orderRevision,
change.currentOrderTotal, change.newOrderTotal));

      default:
        throw new UnsupportedStateTransitionException(state);
    }
  }

  public List<DomainEvent> confirmRevision(OrderRevision orderRevision) {
    switch (state) {
      case REVISION_PENDING:
        LineItemQuantityChange licd =
orderLineItems.lineItemQuantityChange(orderRevision);

        orderRevision.getDeliveryInformation().ifPresent(newDi ->
this.deliveryInformation = newDi);

        if (!orderRevision.getRevisedLineItemQuantities().isEmpty()) {
          orderLineItems.updateLineItems(orderRevision);
        }

        this.state = AUTHORIZED;
        return singletonList(new OrderRevised(orderRevision,
licd.currentOrderTotal, licd.newOrderTotal));
      default:
        throw new UnsupportedStateTransitionException(state);
    }
  }

}
```

The `revise()` method is called to initiate the revision of an order. Among other things, it verifies that the revised order will not violate the order minimum and changes to the state of the order to `REVISION_PENDING`. Once the `ReviseOrderSaga` has successfully updated the `Restaurant Order service` and the `Accounting Service` it then calls `confirmRevision()` to complete the revision.

5.5.2 The `OrderService` service class

The `Order Service` exposes an REST API for creating and updating orders. Each REST API call invokes the `OrderService`, which is a DDD service. Most of its methods create a saga to orchestrate the creation and updating of `Order` aggregates. As a result, this service is more complicated than the `RestaurantOrderService` class you saw earlier. Listing 5.15 shows an except of this class.

```
@Transactional
public class OrderService {

  @Autowired
  private OrderRepository orderRepository;


  @Autowired
  private SagaManager<CreateOrderSagaState, CreateOrderSagaData>
createOrderSagaManager;
```

```
  @Autowired
  private SagaManager<ReviseOrderSagaState, ReviseOrderSagaData>
reviseOrderSagaManagement;

  public Order createOrder(OrderDetails orderDetails) {

    CreateOrderSagaData data = new CreateOrderSagaData(orderDetails);
    createOrderSagaManager.create(data);
    return orderRepository.findOne(data.getOrderId());
  }

  public Order reviseOrder(Long orderId, Long expectedVersion, OrderRevision
orderRevision)  {

    ReviseOrderSagaData sagaData = new ReviseOrderSagaData(orderId,
expectedVersion, orderRevision);
    reviseOrderSagaManagement.create(sagaData);
    return orderRepository.findOne(orderId);
  }
```

The `OrderService` is injected with various dependencies including the `OrderRepository` and several saga managers. The `createOrder()` and `reviseOrder()` methods create sagas by calling the appropriate saga manager's `create()` method. They both return the order after it has been updated by the first step of the saga.

In many ways, the business logic for a microservice-based application is not that much different than that for a monolithic application. It is comprised of classes such as services, JPA-backed entities and repositories. There are some differences, however. A domain model is organized as a set of DDD aggregates, which impose various design constraints. Unlike in a traditional object model, references between classes in different aggregates are in terms of primary key value instead of object references. Also, a transaction can only create or update a single aggregate. It also useful to aggregates to publish domain events when their state changes.

Another major difference is that services often use sagas to maintain data consistency across multiple services. For example, the `Restaurant Order Service` merely participates in a sagas, it does not initiate them. In contrast, the `Order Service` relies heavily on sagas when creating and updating orders. That is because `Orders` must be transactionally consistent with data owned by other services. As a result, most `OrderService` methods create a saga rather than update an `Order` directly.

In this chapter, we have described how to implement business logic using a traditional approach to persistence. That has involved integrating messaging and event publishing with database transaction management. Event publishing code that is sprinkled throughout the business logic. In the next chapter, we will look at an alternative event-centric approach to writing business logic in a microservice architecture: event sourcing. You will learn that a key benefit of event sourcing is that the event generation logic is integral to the business logic rather than being bolted on.

5.6 Summary

- When implementing simple business logic you can use the procedural Transaction script pattern but when implementing complex business logic you should consider using the object-oriented Domain model pattern.
- A good way to organize a service's business logic is as a collection of DDD aggregates. DDD aggregates are useful because they modularize the domain model, eliminate the possibility of object reference between services and ensure that each ACID transaction is within a service.
- An aggregate should sometimes publish domain events when it is created or updated by an API call, an event, or a command sent by a saga. Domain events have a wide variety of uses. Subscribers can, for example, notify users and other applications, publish web socket messages to a user's browser, and update replicated data.

6 Developing business logic with event sourcing

This chapter covers:

- Using event sourcing to develop business logic
- Implementing an event store
- Integrating sagas and event sourcing-based business logic
- Implementing saga orchestrators using event sourcing

In the previous chapter I described how to structure a service's business logic as a set of DDD aggregates. Many of those aggregates publish domain events. On the one hand, the event publishing "logic" works reasonably well. But on the other hand, it is "bolted" onto the business logic. The business logic continues to work even when the developer forgets to publish an event. In this chapter you will learn about event sourcing, which is an event-centric way of writing business logic and persisting domain objects. It stores data as events in a type of database known as event store.

Event sourcing is not a new idea. I first learned about event sourcing 5+ years ago but it remained a curiosity until I started developing applications with a microservice architecture. That is because, as you will see, event sourcing is a great way to implement business logic in an event-driven microservice architecture. It naturally generates the domain events, which communicate changes to data between services.

In this chapter, I describe how event sourcing works and how to use it to write business logic. I also discuss the benefits and drawbacks of event sourcing. I describe how to implement an event store. I also show how event sourcing is a good foundation for implementing sagas, which are, as I described in chapter {chapter-sagas}, a good way to maintain data consistency in a microservice architecture. But first, lets look at how

to develop business logic with event sourcing.

6.1 Developing business logic using event sourcing

Event sourcing is a different way of structuring the business logic and persisting aggregates. It persists an aggregate as a sequence of events. Each event represents a state change of the aggregate. An application recreates the current state of an aggregate by replaying the events.

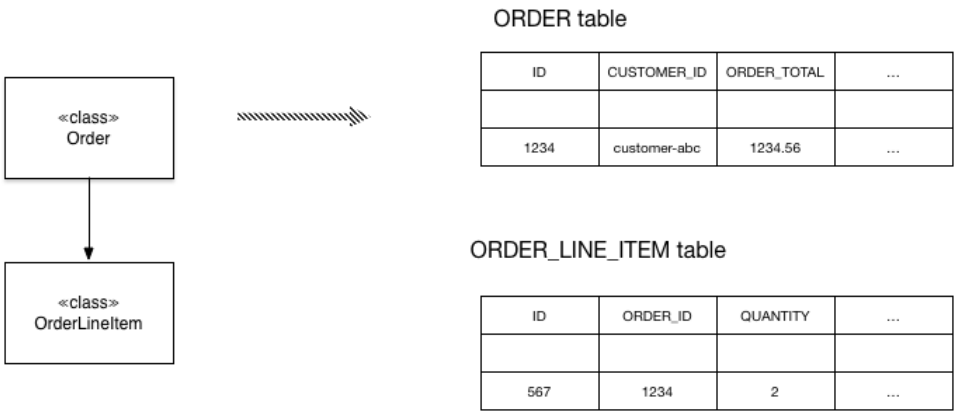
Event sourcing has several important benefits. It preserves the history of aggregates, which is valuable for auditing and regulatory purposes. Also, it reliably publishes domain events, which is particularly useful in a microservice architecture. Event sourcing also has drawbacks. There is a learning curve because its a different way to write your business logic. Querying the event store is often difficult, which you to use the Command Query Responsibility (CQRS) pattern, which I describe in chapter {chapter-queries}.

In this section, I describe the limitations of traditional persistence. I then describe event sourcing in more detail and describe how it overcomes those limitations. I also show how to implement the `Order`aggregate using event sourcing. I will also discuss the benefits and drawbacks of event sourcing. Lets first look at the limitations of the traditional approach to persistence.

6.1.1 The trouble with traditional persistence

The traditional approach to persistence maps classes to database tables, fields of those classes to table columns, and instances of those of classes to rows in those tables. For example, figure 6.1 shows how the `Order` aggregate, which I described in chapter {chapter-ddd-aggregates}, is mapped to the `ORDER` table. Its `OrderLineItems` are mapped to the `ORDER_LINE_ITEM` table.

Figure 6.1. The traditional approach to persistence maps classes to tables and objects to rows in those tables.



The application persists an order instance as rows in the `ORDER` and `ORDER_LINE_ITEM` tables. Some applications implement persistence using an ORM framework, such as JPA, while others use lower-level frameworks, such as MyBATIS.

This approach clearly works well since most enterprise applications store data this way. It has some drawbacks and limitations:

- Object-Relational impedance mismatch
- Lack of aggregate history
- Implementing audit logging is tedious and error prone
- Event publishing is bolted on to the business logic

Lets look at each of these problems starting with the Object-Relational impedance mismatch problem.

Object-Relational impedance mismatch

One age old problem is the so-called Object-Relational impedance mismatch problem. There is a fundamental conceptual mismatch between the tabular relational schema and the graph structure of a rich domain model with its complex relationships. Some aspects of this problem are reflected in polarized debates over the suitability of Object/Relational mapping (ORM) frameworks. For example, Ted Neward said that "Object-Relational mapping is the Vietnam of Computer Science"[23]. Although to be fair, I've used Hibernate successfully to develop applications where the database schema has been derived from the object model. However, the problems are deeper than the limitations of any particular ORM framework.

Lack of aggregate history

Another limitation of traditional persistence is that it only stores the current state of an aggregate. Once an aggregate has been updated, its previous state is lost. If an application must preserve the history of an aggregate, perhaps for regulatory purposes, then developers must implement this mechanism themselves. It is time consuming to implement an aggregate history mechanism and involves duplicating code that must be synchronized with the business logic.

Implementing audit logging is tedious and error prone

Another issue is audit logging. Many applications must maintain an audit log which tracks which users have changed an aggregate. Some applications require auditing for security or regulatory purposes. In other applications, the history of user actions is an important feature. For example, issue trackers and task management applications such as Asana and JIRA display the history of changes to tasks and issues. The challenge of implementing auditing is that as well as being a time consuming chore, there is a risk that the auditing logging code and the business logic will diverge resulting in bugs.

[23] blogs.tedneward.com/post/the-vietnam-of-computer-science/

Event publishing is bolted on to the business logic

Another limitation of traditional persistence is that it usually doesn't support publishing domain events. Domain events, which I described in chapter {chapter-ddd-aggregates}, are events that are published are by an aggregate when its state changes. They are a useful mechanism for synchronizing data and sending notifications in microservice architecture. Some ORM frameworks, such as Hibernate, can invoke application-provided callbacks when data objects change. However, there is no support for automatically publishing messages as part of the transaction that updates the data. Consequently, as with history and auditing, developers must bolt on event generation logic, which risks not being synchronized with the business logic. Fortunately, there is a solution to these issues: event sourcing.

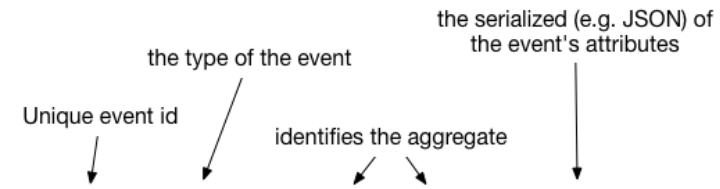
6.1.2 Overview of event sourcing

Event sourcing is an event-centric technique for implementing business logic and persisting aggregates. An aggregate is stored in the database as a series of events. Each event represents a state change of the aggregate. An aggregate's business logic is structured around the requirement to produce and consume these events. Lets see how that works.

Event sourcing persist aggregates using events

Earlier, in section XYZ, I described how traditional persistence maps aggregates to tables, their fields to columns, and their instances to rows. Event sourcing is a very different approach to persisting aggregates, which builds on the concept of domain events. Each aggregate is persisted as a sequence of events in the database, which is also known as an event store, Consider, for example, the `Order`aggregate. As figure 6.2 shows, rather than store each `Order` as a row in an `ORDER` table, event sourcing persists each `Order` aggregate as a sequence of domain events `Order Created`, `Order Approved`, `Order Shipped` and so on.

Figure 6.2. Event sourcing persists each aggregate as a sequence of events. An SQL-based application can store the events in an `EVENTS` table.



event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...	...	...	...	...

EVENTS table

When an aggregate is created or updated, events emitted by the aggregate are inserted into the `EVENTS` table. An aggregate is loaded from the event store by retrieving its events and replaying them. Specifically, loading an aggregate consists of the following three steps:

1. Load the events for the aggregate
2. Create an aggregate instance by using its default constructor
3. Iterate through the events calling apply()

For example, the Eventuate Client framework, which I describe below in section XYZ, uses code similar to the following to reconstruct an aggregate:

```
java
Class aggregateClass = ...;
Aggregate aggregate = aggregateClass.newInstance();
for (Event event : events) {
  aggregate = aggregate.applyEvent(event);
}
// use aggregate...
```

It creates an instance of the class, and iterates through the events calling the aggregate's `applyEvent()` method. If you are familiar with functional programming, you might recognize this as a fold or reduce operation.

In some ways, the way in which an application reconstructs the in-memory state of an

event sourcing-based aggregate is not all that different than how an ORM framework such as JPA or Hibernate loads an entity. An ORM framework loads an object by executing one or more SELECT statements to retrieve the current persisted state; instantiating objects using their default constructors; and uses reflection to initialize those objects. What is different about event sourcing is that the reconstruction of the in-memory state is accomplished using events. Lets now look at how the requirements that event sourcing places on domain events.

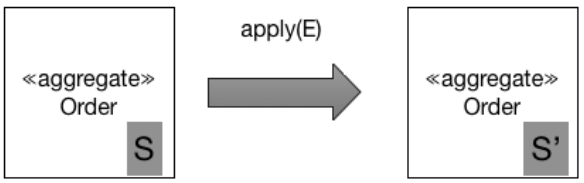
Events represent state changes

In chapter {chapter-ddd-aggregates}, I described domain events as mechanism for notifying subscribers of changes to aggregates. I discussed how events can either contain minimal data, such as just the aggregate *id*, or can be enriched to contain data that is useful to a typical consumer. For example, the `Order Service` can publish an `OrderCreated` event when an order is created. An `OrderCreated` event might simply contain the `orderId`. Alternatively, the event could contain the complete order so consumers of that event do not have fetch the data from the `Order Service`. Whether events are published and what those events contain is driven by the needs of the consumers. With event sourcing, however, it's primarily the aggregate that determines the events and their structure.

Events are not optional when using event sourcing. Every state change of an aggregate including its creation is represented by a domain event. Whenever the aggregate's state changes it must emit an event. For example, an `Order` aggregate must emit an `OrderCreated` event when it is created and an `Order*` event whenever it is updated. This is a much more stringent requirement than before, when an aggregate only emitted events that were of interest to consumers.

What's more, an event must contain the data that the aggregate needs to perform the state transition. Lets suppose, as figure 6.3 shows, that the current state of the aggregate is `S` and the new state is `S'`. An event `E` that represents the state change must contain the data such that when an `Order` is in state `S` calling `order.apply(E)` will update the `Order` to state `S'`. Below you will see that `apply()` is a method that performs the state change represented by an event.

Figure 6.3. Applying event E when the Order is in state S must change the Order state to S'. The event must contain the data necessary to perform the state change.



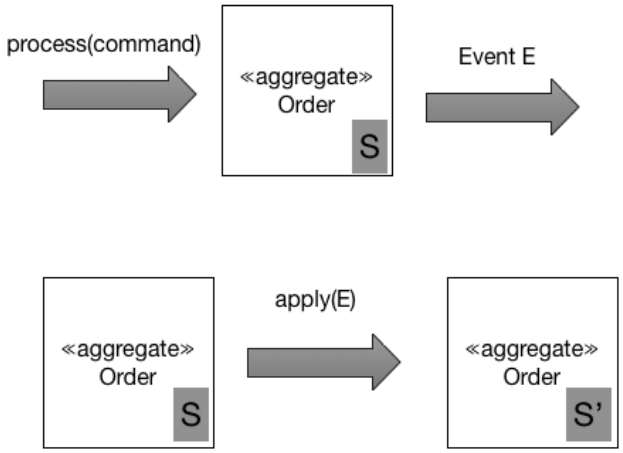
Some events, such as the `Order Shipped` event, contain little or no data and just represent the state transition. The `apply()` method handles an `Order Shipped` event by simply changing the `Order's` status field to `SHIPPED`. Other events, however, contain a

lot of data. An `OrderCreated` event, for example, must contain all of the data needed by the `apply()` method to initialize an `Order` including its line items, payment information and delivery information etc. Because events are used to persist an aggregate, you no longer have the option of using a minimal `OrderCreated` event that just contains the `orderId`.

Aggregate methods are all about events

The business logic handles a request to update an aggregate by calling a command method on the aggregate root. In a traditional application, a command method typically validates its arguments and then updates one or more of the aggregate's fields. Command methods in an event sourcing-based application work quite because they must generate events. As figure 6.4 shows, the outcome of invoking an aggregate's command method is a sequence of events that represent the state changes that must be made. These events are persisted in the database and applied to the aggregate to update its state.

Figure 6.4. Processing a command generates events, without changing the state of the aggregate. An aggregate is updated by applying an event

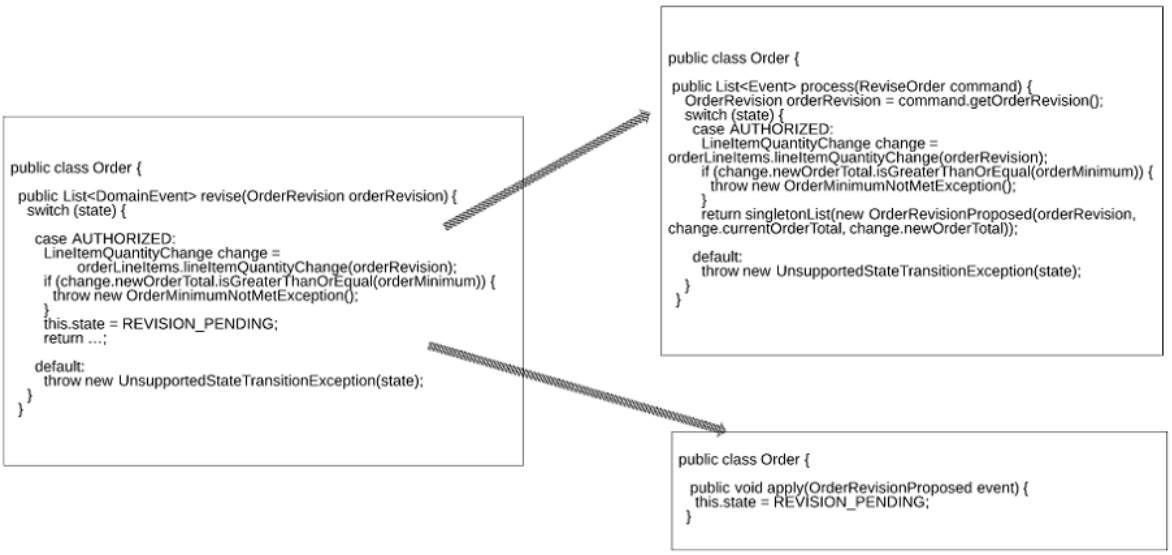


The requirement to generate events and apply them requires a restructuring - albeit mechanical - of the business logic. Event sourcing refactors a command method into two or more methods. The first method takes a command object parameter, which represents the request, and determines what state changes need to be performed. It validates its arguments and without changing the state of the aggregate returns a list of events representing the state changes. The other methods each take a particular event type as a parameter and update the aggregate.

The Eventuate Client framework, which is an event sourcing framework that is described in more detail below, names these methods `process()` and `apply()`. A `process()` method takes a command object, which contains the arguments of the update request, as a parameter and returns a list of events. An `apply()` method takes an

event as a parameter and returns void. An aggregate will define multiple overloaded versions of these methods: one `process()` method for each command class; and one `apply()` method for each event type emitted by the aggregate. Figure 6.5 shows an example.

Figure 6.5. Event sourcing splits a method that updates an aggregate into a `process()` method, which takes a command and returns events, and one or more `apply()` methods, which take an event and update the aggregate.



In this example, the `reviseOrder()` method is replaced by a `process()` method and an `apply()` method. The `process()` method takes a `ReviseOrder` command as a parameter. This command class is defined by applying the *Introduce Parameter Object* refactoring to the `reviseOrder()` method. The `process()` method either returns an `OrderRevisionProposed` event or throws an exception if the order cannot be revised or the proposed revision does not meet the order minimum. The `apply()` method for the `OrderRevisionProposed` event changes the state of the `Order` to `REVISION_PENDING`.

An aggregate is created using the following steps:

1. Instantiate aggregate root using its default constructor
2. Invoke `process()` to generate the new events
3. Update the aggregate by iterating through the new events calling its `apply()`
4. Save the new events in the event store

An aggregate is updated using the following steps:

1. Load aggregate's events from the event store
2. Instantiate the aggregate root using its default constructor
3. Iterate through the loaded events calling `apply()` on the aggregate root

4. Invoke its `process()` method to generate new events
5. Update the aggregate by iterating through the new events calling `apply()`
6. Save the new events in the event store

To see this in action, lets now look at the event sourcing version of the `Order` aggregate.

Event sourcing-based `Order` aggregate.

The event sourcing version of the `Order` aggregate has some similarities to the JPA-based version I showed earlier in chapter {chapter-ddd-aggregates}. Its fields are almost identical and it emits similar events. What's different is that its business logic is implemented in terms of processing commands that emits events and applying those events. For each method that updates the JPA-based aggregate, the event sourcing version defines numerous commands including `CreateOrder`, `NoteAuthorized`, and `CancelOrder`. The `Order` aggregate defines a `process()` method for each command. It also defines an `apply()` method for each of the emitted events. Listing 6.1 shows the `Order` aggregate's fields and the methods responsible for creating it.

Listing 6.1. The `Order` aggregate's fields and the methods that creates it

```
java
public class Order {

  private OrderState state;
  private Long consumerId;
  private Long restaurantId;
  private OrderLineItems orderLineItems;
  private DeliveryInformation deliveryInformation;
  private PaymentInformation paymentInformation;
  private Money orderMinimum;


  public Order() {
  }

  public List<Event> process(CreateOrderCommand command) {
    return events(new OrderCreatedEvent(command.getOrderDetails()));
  }

  public void apply(OrderCreatedEvent event) {
    OrderDetails orderDetails = event.getOrderDetails();
    this.orderLineItems = new OrderLineItems(orderDetails.getLineItems());
    this.orderMinimum = orderDetails.getOrderMinimum();
    this.state = CREATE_PENDING;
  }
```

This class's fields are similar to those of the JPA-based `Order`. The only difference is that the aggregate's `id` is not stored in the aggregate. In contrast, the `Order's` methods are quite different. The `createOrder()` factory method has been replaced by `process()` and `apply()` methods. The `process()` method takes

a `CreateOrder` command and emits an `OrderCreated` event. The `apply()` method takes the `OrderCreated` and initializes the fields of the `Order`.

Lets now look at the slightly more complex business logic for revising an order. Previously this business logic consisted of three methods: `reviseOrder()`, `confirmRevision()` and `rejectRevision()`. The event sourcing version replaces these three methods with three `process()` methods and some `apply()` methods. Listing 6.2 shows the event sourcing version of `reviseOrder()` and `confirmRevision()`.

Listing 6.2. The `process()` and `apply()` methods that revise an `Order` aggregate

```
public class Order {

public List<Event> process(ReviseOrder command) {
  OrderRevision orderRevision = command.getOrderRevision();
  switch (state) {
    case AUTHORIZED:
      LineItemQuantityChange change =
orderLineItems.lineItemQuantityChange(orderRevision);
      if (change.newOrderTotal.isGreaterThanOrEqual(orderMinimum)) {
        throw new OrderMinimumNotMetException();
      }
      return singletonList(new OrderRevisionProposed(orderRevision,
change.currentOrderTotal, change.newOrderTotal));

    default:
      throw new UnsupportedStateTransitionException(state);
  }
}

public void apply(OrderRevisionProposed event) {
  this.state = REVISION_PENDING;
}

public List<Event> process(ConfirmReviseOrder command) {
  OrderRevision orderRevision = command.getOrderRevision();
  switch (state) {
    case REVISION_PENDING:
      LineItemQuantityChange licd =
orderLineItems.lineItemQuantityChange(orderRevision);
      return singletonList(new OrderRevised(orderRevision, licd.currentOrderTotal,
licd.newOrderTotal));
    default:
      throw new UnsupportedStateTransitionException(state);
  }
}


public void apply(OrderRevised event) {
  OrderRevision orderRevision = event.getOrderRevision();
  if (!orderRevision.getRevisedLineItemQuantities().isEmpty()) {
    orderLineItems.updateLineItems(orderRevision);
  }
  this.state = AUTHORIZED;
}
```

As you can see, each method has been replaced by a `process()` method and one or more `apply()` methods. The `reviseOrder()` method has been replaced by `process(ReviseOrder)` and `apply(OrderRevisionProposed)`. Similarly, `confirmRevision()` has been replaced by `process(ConfirmReviseOrder)` and `apply(OrderRevised)`.

6.1.3 Handling concurrent updates using optimistic locking

It is not uncommon for two or more requests to simultaneously update the same aggregate. An application that uses traditional persistence often uses optimistic locking to prevent one transaction from overwriting another's changes. Optimistic locking uses a version column to detect whether an aggregate has changed since it was read. The application maps the aggregate root to a table that has a `VERSION` column, which is incremented whenever the aggregate is updated. The application updates the aggregate using an `UPDATE` statement like this:

```
UPDATE AGGREGATE_ROOT_TABLE
SET VERSION = VERSION + 1 ...
WHERE VERSION = <original version>
```

This `UPDATE` statement will only succeed if the version is unchanged from when the application read the aggregate. If two transactions read the same aggregate then the first one that updates the aggregate will succeed. The second one will fail because the version number has changed and so it won't accidentally overwrite the first transaction's changes.

An event store can also use optimistic locking to handle concurrent updates. Each aggregate instance has a version that is read along with the events. When the application inserts events the event store verifies that the version is unchanged. A simple approach is to use the number of events as the version number. Alternatively, as you will see below, an event store could maintain an explicit version.

6.1.4 Event sourcing and publishing events

Strictly speaking, event sourcing simply persists aggregates as events and reconstructs the current state of an aggregate from those events. It is straightforward, however, to also use event sourcing as a reliable event publishing mechanism. Saving an event is an inherently atomic operation. We just need to implement a mechanism to deliver all persisted events to interested consumers.

In chapter {chapter-ipc}, I described a couple of different mechanisms for publishing events as part of a transaction. One mechanism is polling, the other is transaction log tailing. An event sourcing-based application can use one of these mechanisms. The main difference is that it permanently stores events in an `EVENTS` table rather than temporarily saving events in an `OUTBOX` table and then deleting them.

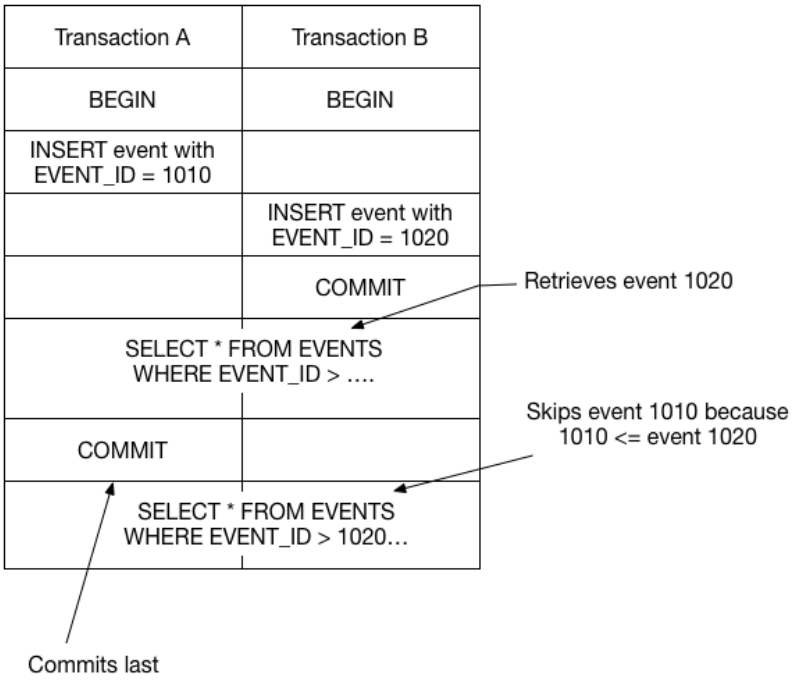
Using a query to poll for new events

If, for example, events are stored in the `EVENTS` table shown in figure 6.2, a consumer can simply poll the table for new events by executing a `SELECT` statement and publish

the events to a message broker. The challenge, however, is determining which events are new. For example, lets imagine that `eventIds` are monotonically increasing. The superficially appealing approach is for a consumer to record the last `eventId` that it has processed. It would then retrieve new events using a query like this: `SELECT * FROM EVENTS where event_id > ? ORDER BY event_id ASC`.

The problem with this approach, however, is that transactions can commit in an order that is different than the order in which they generate events. As a result, consumers can easily skip events. Table 6.6shows an example scenario where the polling mechanism skips an event.

Figure 6.6. A scenario where an event is skipped because its transaction A commits after transaction B. Polling sees eventId=1020 and then later skips eventId=1010



In this scenario, Transaction *A* inserts an event with an `EVENT_ID` of 1010. Next, transaction *B* inserts an event with an `EVENT_ID` of 1020 and then commits. If consumer were now to query the `EVENTS` table it would find event 1020. Later on, after transaction *A* committed and event 1010 became visible, the consumer would ignore it.

One solution to this problem is to add an extra column to the `EVENTS` table, which tracks whether an event has been published. The consumer that polls the table and publishes events would then use the following process:

1. Find unpublished events by executing this SELECT statement: `SELECT * FROM EVENTS where PUBLISHED = 0 ORDER BY event_id ASC`

2. Publish events to the message broker
3. Mark the events as having been published: `UPDATE EVENTS SET PUBLISHED = 1 WHERE EVENT_ID in ?`

This approach ensures that the consumer will not skip events.

Using transaction log tailing to reliably publish events

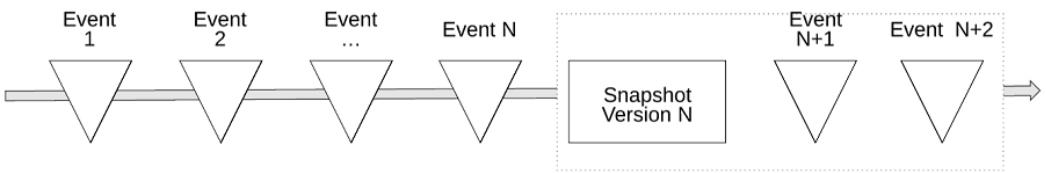
More sophisticated event stores use a different approach that avoids this problem and is also more performant and scalable For example, Eventuate Local, which is an open-source event store, uses transaction log tailing. It reads events inserted into an `EVENTS` table from the MySQL replication stream and publishes them to Apache Kafka. Later on, I describe how Eventuate Local works in more detail.

6.1.5 Using snapshots to improve performance

An `Order` aggregate has relatively few state transitions and so it only has a small number of events. It is efficient to query the event store for those events and reconstruct an `Order` aggregate. Long-lived aggregates, however, can have a large number of events. For example, an `Account` aggregate potentially has a large number of events. Over time, it would become increasingly inefficient to load and fold those events

A common solution is to periodically persist a snapshot of the aggregate's state. The application restores the state of an aggregate by loading the most recent snapshot and only those events that have occurred since the snapshot was created. Figure 6.7 shows an example of using a snapshot.

Figure 6.7. Using a snapshot improves performance by eliminating the need to load all events. An application only needs to load the snapshot and the events that occur after it.



In this example, the snapshot version is *N*. The application only needs to load the snapshot and the two events that follow it in order to restore the state of the aggregate. The previous *N* events are not loaded from the event store.

When restoring the state of an aggregate from a snapshot, an application first creates an aggregate instance from the snapshot and then iterates through the events applying them. For example, the Eventuate Client framework, which I describe below in section XYZ, uses code similar to the following to reconstruct an aggregate:

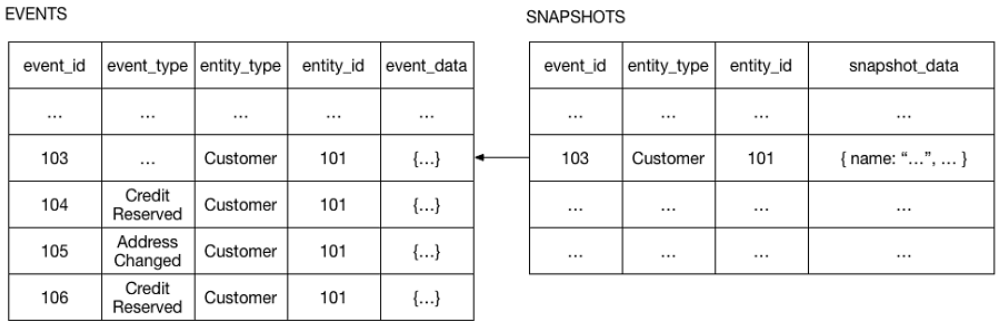
```
Class aggregateClass = ...;
Snapshot snapshot = ...;
Aggregate aggregate = recreateFromSnapshot(aggregateClass, snapshot);
```

```
for (Event event : events) {
  aggregate = aggregate.applyEvent(event);
}
// use aggregate...
```

When using snapshots, the aggregate instance is recreated from the snapshot, rather being creating using its default constructor. If an aggregate has a simple, easily serializable structure then the snapshot can simply be, for example, its JSON serialization. More complex aggregates can be snapshotted by using the Memento pattern[24].

The `Customer` aggregate in the online store example has a very simple structure : the customer's information, their credit limit and their credit reservations. A snapshot of a `Customer` is simply the JSON serialization of its state. Figure 6.8 shows how to recreate a `Customer` from a snapshot corresponding to the state of a `Customer` as of event #103. The `Customer Service` just needs to load the snapshot and the events that have occurred after event #103.

Figure 6.8. The `Customer Service` recreates the `Customer` by deserializing the snapshot's JSON and then loading and applying events #104 through #106.



The `Customer Service` recreates the `Customer` by deserializing the snapshot's JSON and then loading and applying events #104 through #106.

6.1.6 *Idempotent message processing*

Service often consume messages from other applications or other services. A service might, for example, consume domain events published by aggregates or command message sent by a saga orchestrator. As I described in chapter {chapter-ipc}, message consumers must be idempotent since a message broker might deliver the same message multiple times. An message consumer idempotent if it can safely be invoked with the same message multiple times. The Tram framework, for example, implements idempotency by detecting and discarding duplicate messages by recording the *ids* of processed messages in a `PROCESSED_MESSAGES` table. It updates the `PROCESSED_MESSAGES` table during the local ACID transaction used by the business

[24] en.wikipedia.org/wiki/Memento_pattern

logic to create or update aggregates. Event sourcing-based business logic must implement an equivalent mechanism.

Idempotent message processing with an RDBMS-based event store

If an application uses an RDBMS-based event store, it can use an identical approach to detect and discard duplicates messages. It simply inserts the message *id* into `PROCESSED_MESSAGES` table as part of the transaction that inserts into the `EVENTS` table.

Idempotent message processing when using NoSQL event store

An NoSQL-based event store, which has a limited transaction model, however, use a different mechanism to implement idempotent message handling. A consumer must somehow atomically persist events and record the message *id*. Fortunately, there is a simple solution. A message consumer stores the message's *id* in the events that are generated while processing it. It detects duplicates verifying that none of an aggregate's events contain the message *id*.

One challenge with using this approach, however, is that processing a message might not generate any events. The lack of events means there is no record of a message having been processed. A subsequent redelivery and reprocessing of the same message might result in incorrect behavior. For example, consider the following scenario:

1. Message A is processed but does not update an aggregate
2. Message B is processed and the message consumer updates the aggregate
3. Message A is re-delivered again and because there is not record of it having been processed, the message consumer updates the aggregate
4. Message B is processed again, ….

In this scenario the redelivery of events results in a different and possibly erroneous outcome.

One way to avoid this problem is to always publish an event. If an aggregate does not emit an event, an application saves a pseudo event solely to record the message *id*. Event consumers must ignore these pseudo events.

6.1.7 Evolving domain events

Event sourcing, at least conceptually, stores events forever. This is feature is a double-edged sword. On the one hand, it provides the application with an audit log of changes that is guaranteed to be accurate. It also enables an application to reconstruct the historical state of an aggregate. But on the other hand, this feature creates a challenge since the structure of events often changes over time. As a result, an application must potentially deal with multiple versions of events. Lets first look at the different ways that events can change. After that I will describe a commonly used approach to handling changes.

Event schema evolution

Conceptually, an event sourcing application has a schema that is organized into three

levels. The top level of the schema consists of one or more aggregates. The second level defines the events that each aggregate emits. The bottom level defines the structure of the events. Table 6.1 shows the different types of changes that can occur at each level.

Table 6.1. The different ways that an application's events can evolve

Level	Change	Description	Backwards compatible
Schema	Add aggregate	Define a new aggregate type	Yes
	Remove aggregate	Remove an existing aggregate	??
	Rename aggregate	Change the name of an aggregate type	No
Aggregate	Add event	Add a new event Type	Yes
	Remove event	Remove an event type	??
	Rename event	Change the name of an event type	No
Event	Add field	Add a new field	Yes
	Delete field	Delete a field	No
	Rename field	Rename a field	No
	Change type of field	Change the type of a field	No

These changes occur naturally as a service's domain model evolves over time. For example, when a service's requirements change or as its developers gain deeper insight into a domain and improve the domain model. At the schema level, developers add, remove and rename aggregate classes. At the aggregate level, the types of events emitted by a particular aggregate can change. Developers can change the structure of an event type by adding, removing and changing the name or type of a field.

Fortunately, many of these types of changes are backwards compatible changes. For example, adding a field to an event is unlikely to impact consumers. A consumer simply ignores unknown fields. Other changes, however, are not backwards compatible. For example, changing the name of an event or the name of a field requires consumers of that event type to be changed.

Managing schema changes through upcasting

In the SQL database world, changes to a database schema are commonly handled by using schema migrations. Each schema change is represented by a migration, which is a SQL script that changes the schema and migrates the data to a new schema. The schema migrations are stored in version control system and applied to a database using a tool such as Flyway.

An event sourcing application can use a similar approach to handle non-backwards compatible changes. However, instead of migrating events to the new schema version in situ, event sourcing frameworks transform events when they are loaded from the event store. A component, which is commonly called an *upcaster*, updates individual events from an old version to a newer version. As a result, the application code only ever deals with the current event schema. Now that we have looked at how event sourcing works, let's look at its benefits and drawbacks.

6.1.8 Benefits of event sourcing

Event sourcing has both benefits and drawbacks. The benefits are:

- Reliably publishes domain events
- Preserves the history of aggregates
- Mostly avoids the O/R impedance mismatch problem
- Provides developers with a time machine

Lets look at each benefit in more detail.

Reliably publishes domain events

A major benefit of event sourcing is that it reliably publishes events whenever the state of an aggregate changes. It is a good foundation for an event-driven microservice architecture. Also, because each event can store the identity of the that made the change, event sourcing provides an audit log that is guaranteed to be accurate. The stream of events can be used for a variety of other purposes including notifying users, application integration, analytics, and monitoring.

Preserves the history of aggregates

Another benefit of event sourcing is that it stores the entire history of each aggregate. You can easily implement temporal queries that retrieve the past state of an aggregate. To determine the state of an aggregate at a given point in time you simply the fold the events that occurred up until that point. It is straightforward, for example, to calculate the available credit of a customer (TODO better example) at some point in the past.

Mostly avoids the O/R impedance mismatch problem

Event sourcing also mostly avoids the O/R impedance mismatch problem. That is because it persists events rather than aggregates. Events typically have a simple, easily serializable, structure. As described earlier, a service can snapshot a complex aggregate by serializing a memento of its state, which adds a level of indirection between an aggregate and its serialized representation.

Provides developers with a time machine

Another benefit of event sourcing is that it stores a history of everything that has happened in the lifetime of an application. Lets imagine, that the FTGO developers need to implement a new requirement to customers who added an item to their shopping cart and then removed it. A traditional application does not preserve this information and so can only market to customers who add and remove items after the feature has been implemented. In contrast, an event sourcing-based application can immediately market to customers who have done this in a past. It is as if event sourcing provides developers with a time machine for traveling to the past and implement unanticipated requirements.

6.1.9 Drawbacks of event sourcing

Event sourcing is, of course, not a silver bullet. It has the following drawbacks:

- Different programming model that has a learning curve
- Complexity of a messaging-based application
- Evolving events can be tricky
- Querying the event store is challenging

Let's look at each drawback.

Different programming model that has a learning curve

It is a different and unfamiliar programming model so there is a learning curve. In order for an existing application to use event sourcing, you must rewrite its business logic. Fortunately, this is a fairly mechanical transformation, which can be done when you migrate your application to microservices.

Complexity of a messaging-based application

Another drawback of event sourcing is that message brokers usually guarantee at-least once delivery. Event handlers that are not idempotent must detect and discard duplicate events. The event sourcing framework can help by assigning each event a monotonically increasing id. An event handler can then detect duplicate events by tracking of highest seen event ids. This even happens automatically when event handlers update aggregates.

Evolving events can be tricky

Another challenge with event sourcing is that the schema of events (and snapshots!) will evolve over time. Since events are stored forever, aggregates potentially need to fold events corresponding to multiple schema versions. There is a real risk that aggregates become bloated with code to deal with all the different versions. As I mentioned earlier in section XYZ a good solution to this problem is to upgrade events to the latest version when they are loaded from the event store. This approach separates the code that upgrades events from the aggregate. This simplifies the aggregates, since they only need to apply the latest version of the events.

Querying the event store is challenging

Another drawback of event sourcing is that querying the event store can be challenging. Let's imagine, for example, that you need to find customers who have exhausted their credit limit. You cannot simply write `SELECT * FROM CUSTOMER WHERE CREDIT_LIMIT = 0`. There isn't a column containing the credit. Instead, you must use a more complex and potentially inefficient query that has a nested `SELECT`to compute the credit limit by folding events that set the initial credit and adjust it. To make matters worse, a NoSQL-based event store will typically only support primary key-based lookup. Consequently, you must implement queries using an approach called Command Query Responsibility Segregation (CQRS), which I describe in chapter {chapter-queries}.

6.2 Implementing an event store

An application that uses event sourcing stores its events in an event store. An event store is a hybrid of a database and a message broker. It is a database because it has an API for inserting and retrieving an aggregate's events by primary key. An event store is also a message broker since it has an API for subscribing to events.

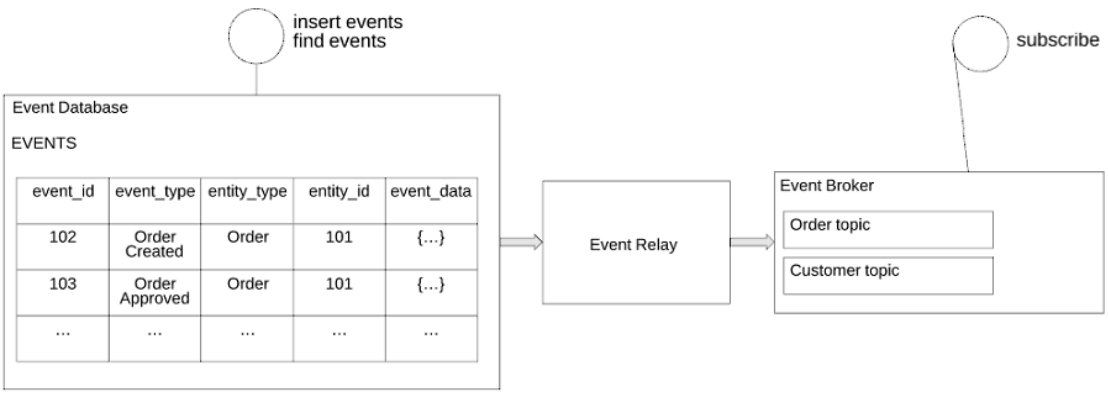
There are a few different ways to implement an event store. One option is to implement your own event store and event sourcing framework. You can, for example, persist events in an RDBMS. A simple albeit low performance way to publish events is for subscribers to poll the `EVENTS` table for events.

Another option is to use a special purpose event store, which typically provides a rich set of features and better performance and scalability. Greg Young, an event sourcing pioneer, has a .NET-based, open-source event store called Event Store. Lightbend, the company formerly known as Typesafe, has a microservices framework called Lagom that is based on event sourcing. My startup, Eventuate, has a event sourcing framework and event store for microservices. Eventuate is available as a cloud service and a Kafka/RDBMS-based open-source project. Lets look at how Eventuate Local, which is an open-source event store, works.

6.2.1 How Eventuate Local works

Eventuate Local is an open-source event store that is built using MySQL and Apache Kafka. Figure 6.9 shows the architecture. Events are stored in MySQL. Applications insert and retrieve aggregate events by primary key. Applications consume events from Apache Kafka. A transaction log tailing mechanism propagates events from MySQL to Apache Kafka.

Figure 6.9. The architecture of Eventuate Local. It consists of an event database (e.g. MySQL), which stores the events, an event broker (e.g. Apache Kafka), which delivers events to subscribers and an event relay, which publishes events stored in the event database to the event broker.



Lets look at the different Eventuate Local components starting with the database schema.

The schema of Eventuate Local's event database

The event database consists of three tables:

- `events` - stores the events
- `entities` - one row per entity
- `snapshots` - stores snapshots

The central table is, of course, the `events` table. The structure of this table is very similar to the table shown in figure 6.2. Here is its definition:

```
create table events (
  event_id varchar(1000) PRIMARY KEY,
  event_type varchar(1000),
  event_data varchar(1000) NOT NULL,
  entity_type VARCHAR(1000) NOT NULL,
  entity_id VARCHAR(1000) NOT NULL,
  triggering_event VARCHAR(1000)
);
```

The `triggering_event` column is used to detect duplicate events/messages. It stores the *id* of the message/event whose processing generated this event.

The `entities` table stores the current version of each entity. It is used to implement optimistic locking. Here is the definition of this table:

```
create table entities (
  entity_type VARCHAR(1000),
  entity_id VARCHAR(1000),
  entity_version VARCHAR(1000) NOT NULL,
  PRIMARY KEY(entity_type, entity_id)
);
```

When an entity is created, a row is inserted into this table. Each time an entity is updated, the `entity_version` column is updated.

The `snapshots` table stores the snapshots of each entity. Here is the definition of this table:

```
create table snapshots (
  entity_type VARCHAR(1000),
  entity_id VARCHAR(1000),
  entity_version VARCHAR(1000),
  snapshot_type VARCHAR(1000) NOT NULL,
  snapshot_json VARCHAR(1000) NOT NULL,
  triggering_events VARCHAR(1000),
  PRIMARY KEY(entity_type, entity_id, entity_version)
)
```

The `entity_type` and `entity_id` columns specify the snapshot's entity. The `snapshot_json` column is the serialized representation of the snapshot and

the `snapshot_type` is its type. The `entity_version`specifies the version of the entity that this is a snapshot of.

The three operations supported by this schema are find, create, and update. The `find` operation queries the `snapshots` table to retrieve the latest snapshot, if any. If a snapshot exists, the `find` operation queries the `events` table to find all events whose `event_id` is greater than the snapshot's `entity_version`. Otherwise, `find` retrieves all events for the specified entity. The `find` operation also queries the `entity` table to retrieve the entity's current version.

The `create` operation inserts a row into the `entity` table and inserts the events into the `events` table. The `update` operation inserts events into the `events` table. It also performs an optimistic locking check by updating the entity version in the `entities` table using this `UPDATE` statement:

```
UPDATE entities SET entity_version = ?
WHERE entity_type = ? and entity_id = ? and entity_version = ?
```

This statement verifies that the version is unchanged since it was retrieved by the `find` operation. It also updates the `entity_version` to the new version. The `update` operation performs these updates within a transaction in order to ensure atomicity. Now that we have looked how Eventuate Local stores an aggregate's events and snapshots, lets now look at how client subscribe to events using Eventuate Local's event broker.

The Eventuate Local's event broker

Services consume events by subscribing to the event broker, which is implemented using Apache Kafka. The event broker has a topic for each aggregate type. A topic is, as described in chapter {chapter-ipc}, a partitioned message channel. This enables consumers to scale horizontally while preserving message ordering. The aggregate *id* is used as the partition key, which preserves the ordering of events published by a given aggregate. To consume an aggregate's events, a service simply subscribes to the aggregate's topic. Lets now look at the event relay, which is the glue between the event database and the event broker,

The Eventuate Local event relay

The event relay propagates events inserted into the event database to the event broker. The mechanism used to accomplish this depends on the database. The MySQL version of the event relay uses the MySQL master/slave replication protocol. The event relay connects to the MySQL server as if it were a slave and reads the MySQL binlog, which is a record of updates made to the database. Inserts into the `EVENTS` table, which correspond to events, are published to the appropriate Apache Kafka topic. The event relay ignores the rest.

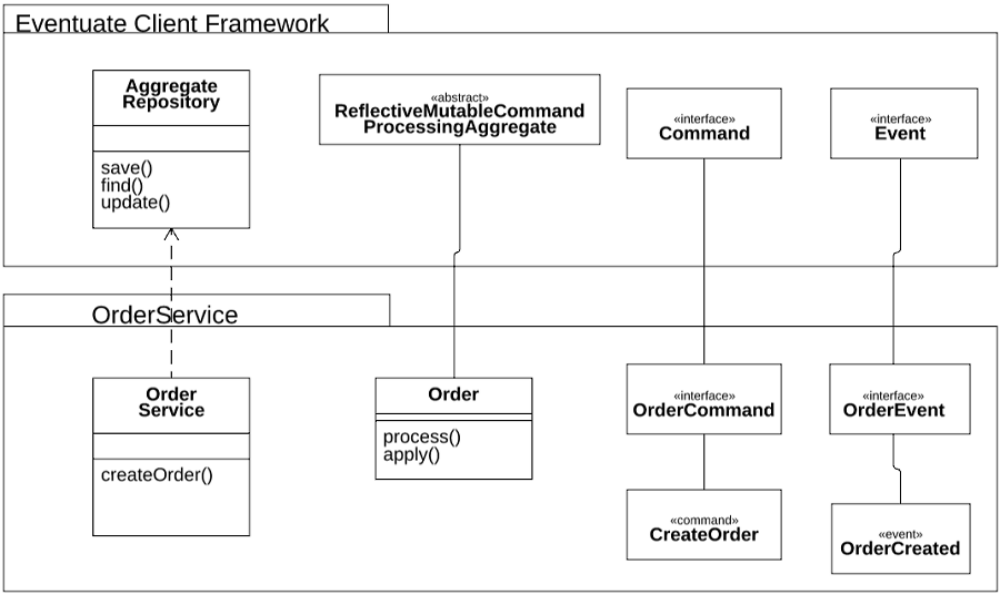
The event relay is deployed as a standalone process. In order to restart correctly, it periodically saves the current position in the binlog - filename and offset - in a special Apache Kafka topic. On startup it first retrieves the last recorded position from the

topic. The event relay then starts reading the MySQL binlog from that position.

6.2.2 The Eventuate client framework for Java

The Eventuate client framework enables developers to write event sourcing-based applications that use the Eventuate Local event store. It is available for a variety of languages including Java and NodeJS. The framework, which is shown in figure 6.10, provides the foundation for developing event sourcing-based aggregates, services and event handlers.

Figure 6.10. The main classes and interfaces provided by the Eventuate client framework for Java



The framework provides base classes for aggregates, commands and events. There is also an `AggregateRepository` class, which provides CRUD functionality. The framework also has an API for subscribing to events. Lets briefly look at each of the types shown in this diagram.

Defining aggregates with the `ReflectiveMutableCommandProcessingAggregate` class

`ReflectiveMutableCommandProcessingAggregate` is the base class for aggregates. It is a generic class that has two type parameters. The first parameter is the concrete aggregate class. The second parameter is the superclass of the aggregate's command classes. As the name suggests, it uses reflection to dispatch command and events to appropriate method. Commands are dispatched to a `process()` method and events to an `apply()` method.

The `Order` class that you saw earlier extends

`ReflectiveMutableCommandProcessingAggregate`. Listing 6.3 shows the `Order` class.

Listing 6.3. The Eventuate version of the `Order` class

```
public class Order extends ReflectiveMutableCommandProcessingAggregate<Order,
OrderCommand> {

  public List<Event> process(CreateOrderCommand command) { ... }

  public void apply(OrderCreatedEvent event) { ... }

  ...
}
```

The two type parameters passed to `ReflectiveMutableCommandProcessingAggregate` are `Order` and `OrderCommand`, which is the base interface for `Order's` commands.

Defining aggregate commands

An aggregate's command classes must extend an aggregate specific base interface, which itself must extend the `Command` interface. For example, the `Order` aggregate's commands extend `OrderCommand`:

```
public interface OrderCommand extends Command {
}

public class CreateOrderCommand implements OrderCommand { ... }
```

The `OrderCommand` interface extends `Command` and the `CreateOrderCommand` command class extends `OrderCommand`.

Defining domain events

An aggregate's event classes must extend the `Event` interface, which is a marker interface with no methods. It is also useful to define a common base interface, which extends `Event`, for all an aggregate's event classes. For example, here is the definition of the `OrderCreated` event:

```
interface OrderEvent extends Event {

}

public class OrderCreated extends OrderEvent { ... }
```

The `OrderCreated` event class extends `OrderEvent`, which is the base interface for the `Order` aggregate's event classes. The `OrderEvent` interface extends `Event`.

Creating, finding and updating aggregates with the `AggregateRepository` class

The framework provides several ways to create, find and update aggregates. The simplest approach, which I describe here, is to use an `AggregateRepository`. `AggregateRepository` is generic class that is parameterized by the aggregate class and the aggregate's base command class. It provides three

overloaded methods:

- `save()` - creates an aggregate
- `find()` - finds an aggregate
- `update()` - updates an aggregate.

The `save()` and `update()` are particularly convenient since they encapsulate the boilerplate code required for creating and updating aggregates. For instance, `save()` takes a command object as a parameter and performs the following steps:

1. Instantiates the aggregate using its default constructor
2. Invokes `process()` to process the command
3. Applies the generated events by calling `apply()`
4. Saves the generated events in the event store.

The `update()` method is similar. It has two parameters, an aggregate *id* and a command and performs the following steps:

1. Retrieves the aggregate from the event store
2. Invokes `process()` to process the command
3. Applies the generated events by calling `apply()`
4. Saves the generated events in the event store.

The `AggregateRepository` class is primarily used by services, which create and update aggregates in response to external requests. For example, listing 6.4 shows how the `OrderService` uses an `AggregateRepository` to create an `Order`.

Listing 6.4. The `OrderService` uses an `AggregateRepository` to create an `Order` aggregate

```
public class OrderService {
  private AggregateRepository<Order, OrderCommand> orderRepository;

  public OrderService(AggregateRepository<Order, OrderCommand> orderRepository) {
    this.orderRepository = orderRepository;
  }

  public EntityWithIdAndVersion<Order> createOrder(OrderDetails orderDetails) {
    return orderRepository.save(new CreateOrder(orderDetails));
  }
}
```

The `OrderService` is injected with an `AggregateRepository` for creating, finding and updating `Orders`. Its `create()` method invokes `AggregateRepository.save()` with a `CreateOrder` command.

Subscribing to events

The Eventuate Client framework also provides an API for writing event handlers. Listing 6.5 shows an event handler for `CreditReserved` events.

The `@EventSubscriber` annotation specifies the *id* of the durable subscription. Events that are published when the subscriber is not running will be delivered when it starts up. The `@EventHandlerMethod` annotation identifies the `creditReserved()` method as an event handler.

Listing 6.5. An event handler for `OrderCreatedEvent`

```
@EventSubscriber(id="orderServiceEventHandlers")
public class OrderServiceEventHandlers {

  @EventHandlerMethod
  public void creditReserved(EventHandlerContext<CreditReserved> ctx) {
    CreditReserved event = ctx.getEvent();
    ...
  }
```

An event handler has an parameter of type `EventHandlerContext`, which contains the event and its metadata.

Now that we have looked at how to write event sourcing-based business logic using the Eventuate client framework, lets look at how to use event sourcing-based business logic with sagas.

6.3 Using sagas and event sourcing together

Let's imagine that you have implemented one or more services using event sourcing. You have probably written services similar to the one shown in listing 6.4. However, if you have read chapter {chapter-sagas} then you know that services often need to initiate and participate in sagas, which are sequences of local transactions used to maintain data consistency across services. For example, the `Order Service` uses a saga to validate an `Order`. The `Restaurant Order Service`, `Consumer Service`, and the `Accounting Service` participate in that saga. Consequently, you must integrate sagas and event sourcing-based business logic.

Event sourcing makes it easy to use choreography-based sagas. The participants simply exchange the domain events emitted by their aggregates. Each participant's aggregates handle events by processing commands and emitting new events. You simply need to write the aggregates and the event handler classes, which update the aggregates.

However, integrating event sourcing-based business logic with orchestration-based sagas is not as easy. That's because the event store's concept of a transaction is quite limited. When using an event store, an application can only create or update a single aggregate and publish the resulting event(s). However, each step of a saga consists of several actions that must be performed atomically:

- Saga creation - a service that initiates a saga must atomically create or update an aggregate and create the saga orchestrator. For example, the `Order Service's createOrder()` method must create an `Order`aggregate and a `CreateOrderSaga`.

- Saga orchestration - a saga orchestrator must atomically consume replies,update its state and send command messages.
- Saga participants - saga participants, such as the `Restaurant Order Service` and the `Order Service`, must atomically consume messages, detect and discard duplicates, create or update aggregates, and send reply messages.

Because of this mismatch between these requirements and the transactional capabilities of an event store, integrating orchestration-based sagas and event sourcing potentially creates some interesting challenges.

A key factor in determining the ease of integration is whether the event store uses an RDBMS or a NoSQL database. The Tram saga framework that I described in chapter {chapter-sagas} and the underlying Tram messaging framework, which I described in chapter {chapter-ipc}, rely on flexible ACID transactions provided by the RDBMS. The saga orchestrator and the saga participants use ACID transactions to atomically update their databases and exchange messages. If the application uses an RDBMS-based event store, such as Eventuate Local, then it can *cheat* and it can invoke the Tram saga framework and update the event store within an ACID transaction. If, however, the event store uses a NoSQL database, which can't participate in the same transaction as the Tram Saga framework, it will have to take a different approach.

Lets take a closer look at the different scenarios and the issues you will need to address. I will cover the following topics:

- Implementing choreography-based sagas
- Creating an orchestration-based saga
- Implementing an event sourcing-based saga participant
- Implementing saga orchestrators using event sourcing

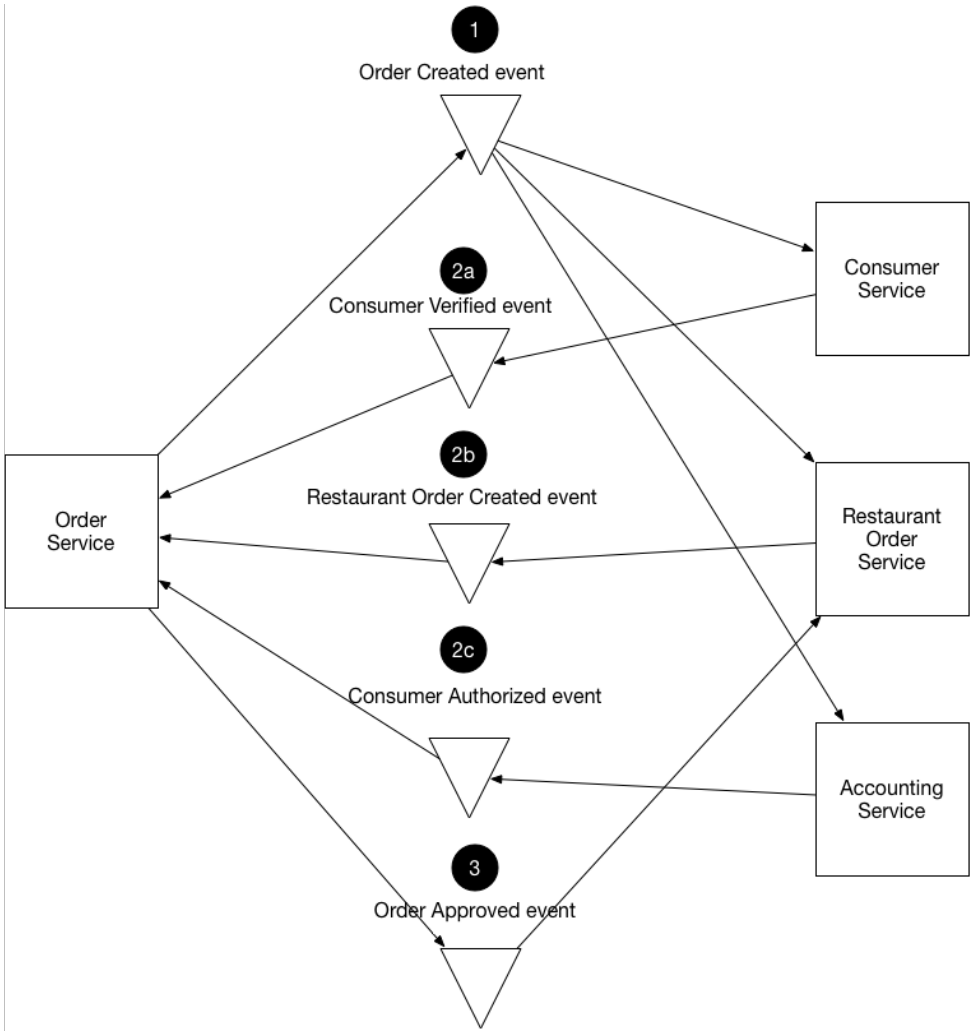
Let's begin by looking at how to implement choreography-based sagas using event sourcing.

6.3.1 Implementing choreography-based sagas using event sourcing

The event-driven nature of event sourcing makes it quite straightforward to implement choreography-based sagas. When an aggregate is updated, it emits an event. An event handler for a different aggregate can consume that event and update its aggregate. The event sourcing framework automatically makes each event handler idempotent.

For example, in chapter {chapter-sagas}, I described how implement the `Create Order saga` using choreography. The `ConsumerService`, `RestaurantOrderService` and the `AccountingService` subscribe to the `OrderService's` events and vice versa. Each service has an event handler similar to the one shown in listing XYZ. The event handler updates the corresponding aggregate, which emits another event. Figure 4.2 shows how saga works.

Figure 6.11. Implementing the `CreateOrder` saga using choreography. The saga participants communicate by exchanging events.



The happy path through this saga is as follows:

1. The `OrderService` creates an `Order` in the `PENDING` state and publishes an `OrderCreated` event.
2. The `OrderCreated` event is received by the following services:
   a. The `ConsumerService`, which verifies that the consumer can place the order, and publishes `ConsumerVerified` event
   b. The `RestaurantOrderService`, which validate the `Order`, creates a `RestaurantOrder` in a `PENDING` state, and publishes `RestaurantOrderCreated` event

   c. The `AccountingService`, which charges the consumer's credit card and publishes `ConsumerAuthorized` event
3. The `OrderService` receives the `ConsumerVerified`, `RestaurantOrderCreated` and `ConsumerAuthorized` events , changes the state of the `Order` to `APPROVED` and publishes an `OrderApproved` event.
4. The `RestaurantOrderService`, receives the `OrderApproved` event and changes the state of the `RestaurantOrder` to `XYZ-TODO`

Event sourcing and choreography-based sagas work very well together. Event sourcing provides the mechanisms that sagas need including messaging-based IPC, message de-duplication, and atomic updating of state and message sending. Despite its simplicity, choreography-based sagas have several drawbacks. Not only are the drawbacks that I describe in chapter {chapter-sagas}, but there is an event sourcing-specific drawback.

The problem is with using events for saga choreography means that events have a dual purpose. Event sourcing uses events to represent state changes. However, using events for saga choreography requires an aggregate to emit an event if even there is no state change. For example, if updating an aggregate would violate a business rule, then the aggregate must emit an "error" event. An even worse problem is when a saga participant cannot create an aggregate. There is no aggregate that can emit an "error" event.

Because of these kinds of issues, it is best to implement more complex sagas using orchestration. In the rest of this section, I explain how to integrate orchestration-based sagas and event sourcing. As you will see, it involves solving some interesting problems. Let's first look at how a service method, such as `OrderService.createOrder()`, creates a saga orchestrator.

6.3.2 Creating an orchestration-based saga

Saga orchestrators are created by service methods. Some service methods simply create a saga orchestrator. Others, such as `OrderService.createOrder()`, do two things: create or update an aggregate and create a saga orchestrator. Both actions must be done atomically as part of the same database transaction. Or, they must be done in a way that guarantees that if it does the first action then the second action will be done eventually. How the service ensures that both of these actions is performed depends on the kind of event store it uses.

Creating an saga orchestrator when using an RDBMS-based event store

If a service uses an RDBMS-based event store, it can simply update the event store and create a saga orchestrator within the same ACID transaction. For example, let's imagine that the `OrderService` uses Eventuate Local and the Tram Saga framework. Its `createOrder()` method would look like this:

```
class OrderService

  @Autowired
```

```
  private SagaManager<CreateOrderSagaData> createOrderSagaManager;

  @Transactional                                                          (1)
  public EntityWithIdAndVersion<Order> createOrder(OrderDetails orderDetails) {
    EntityWithIdAndVersion<Order> order =
        orderRepository.save(new CreateOrder(orderDetails));              (2)

    CreateOrderSagaData data =
        new CreateOrderSagaData(order.getTODO(), orderDetails);           (3)

    createOrderSagaManager.create(data, Order.class, order.getId());

    return order;
  }
...
```

(1) Ensure the `createOrder()` executes within a database transaction
(2) Create the `Order` aggregate
(3) Create the `CreateOrderSaga`

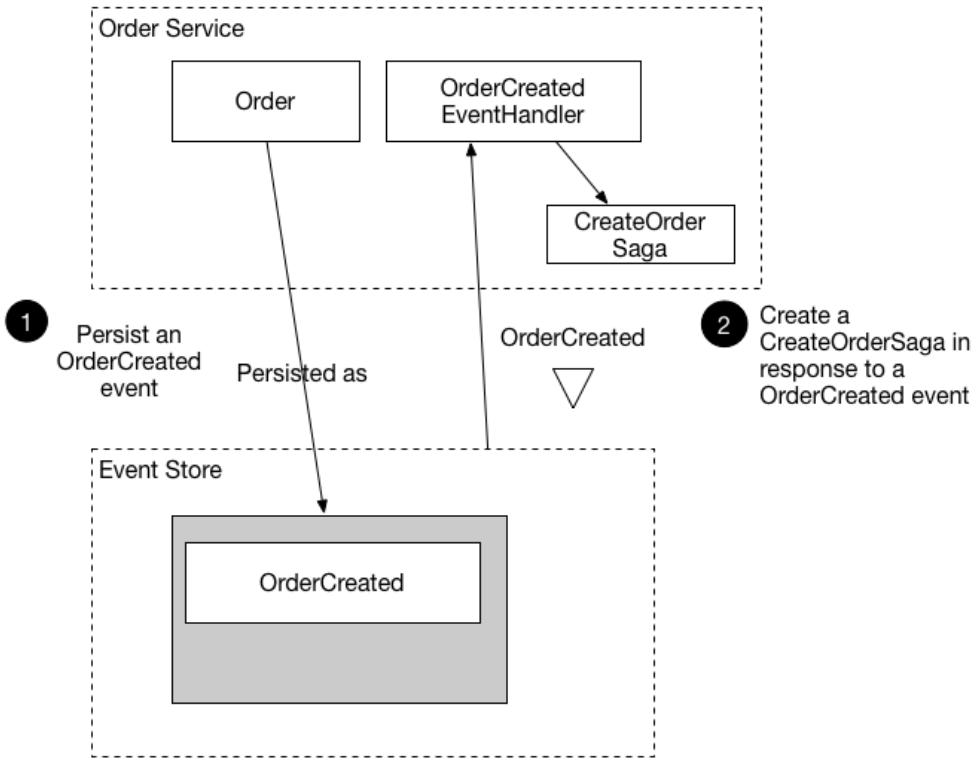
It is a combination of the `OrderService` in listing 6.4, and the `OrderService` described in chapter {chapter-sagas}. Since Eventuate Local uses an RDBMS, it can participate in the same ACID transaction as the Tram Saga framework. If, however, a service uses a NoSQL-based event store, creating a saga orchestrator is not as straightforward.

Creating an saga orchestrator when using a NoSQL-based event store

A service that uses a NoSQL-based event store will most likely be unable to atomically update the event store and create a saga orchestrator. The saga orchestration framework might use an entirely different database. Or, even if it used the same NoSQL database, the application won't be able to create or update two different objects atomically because of the NoSQL database's limited transaction model. Instead, a service must have an event handler that creates the saga orchestrator in response to a domain event emitted by the aggregate.

For example, figure 6.12 shows how the `Order Service` creates an `CreateOrderSaga` using an event handler for the `OrderCreated` event. The `Order Service` first creates an `Order` aggregate and persists it in the event store. The event store publishes the `OrderCreated` event, which is consumed by the event handler. The event handler invokes the Tram Saga framework to create a `CreateOrderSaga`.

Figure 6.12. Using an event handler to reliably create a saga after a service creates an event sourcing-based aggregate



One issue to keep in mind when writing an event handler that creates a saga orchestrator is handling duplicate events. At least once message delivery means that the event handler that creates the saga might be invoked multiple times. It's important to ensure that only one saga instance is created.

A straightforward approach is to derive the *id* of the saga from a unique attribute of the event. There are a couple of different options. One option is to use the *id* of the aggregate that emits the event as the *id* of the saga. This works well for sagas that are created in response to aggregate creation events.

Another option is to use the event *id* as the saga *id*. Since event ids are unique this will guarantee that saga *id* is unique. Also, if an event is a duplicate then the event handler's attempt to create the saga will fail because the *id* already exists. This option is useful when multiple instances of the same saga can exist for a given aggregate instance.

A service that uses an RDBMS-based event store can also use the same event-driven approach create sagas. A benefit of this approach is that it promotes loose coupling since services, such as `OrderService`, no longer explicitly instantiate sagas. Now that we have looked how to reliably create a saga orchestrator, lets look at how event

sourcing-based services can participate in orchestration-based sagas.

6.3.3 Implementing an event sourcing-based saga participant

Let's imagine that you used event sourcing to implement a service that needs to participate in an orchestration-based saga. Not surprisingly, if your service uses an RDBMS-based event store, such as Eventuate Local, then you can easily ensure that it atomically processes saga command messages and sends replies. It can simply update the event store as part of the ACID transaction initiated by the Tram framework. However, you must use an entirely different approach if your service uses an event store that cannot participate in the same transaction as the Tram framework.

There are a couple of different issues that you must address:

- Idempotent command message handling
- Atomically sending a reply messages

Let's first look at how to implement idempotent command message handlers.

Idempotent command message handling

The first problem to solve is how an event sourcing-based saga participant can detect duplicate and discard messages in order to implement idempotent command message handling. Fortunately, this is an easy problem to address using the idempotent message handling mechanism I described earlier. A saga participant simply records the message *id* in the events that are generated when processing the message. Before updating an aggregate, it verifies that it hasn't processed the message before by looking for the message *id* in the events.

Atomically sending a reply messages

The second problem to solve is how an event sourcing-based saga participant can atomically send replies. In principle, a saga orchestrator could subscribe to the events emitted by an aggregate. However, there are two problems with this approach. The first is that an saga command might not actually change the state of an aggregate. In this scenario, the aggregate won't emit an event and so no reply will be sent to the saga orchestrator. The second problem is that this approach requires the saga orchestrator to treat saga participants that use event sourcing different than those that don't. That's because in order to receive domain events the saga orchestrator must subscribe to the aggregate's event channel in addition to its own reply channel.

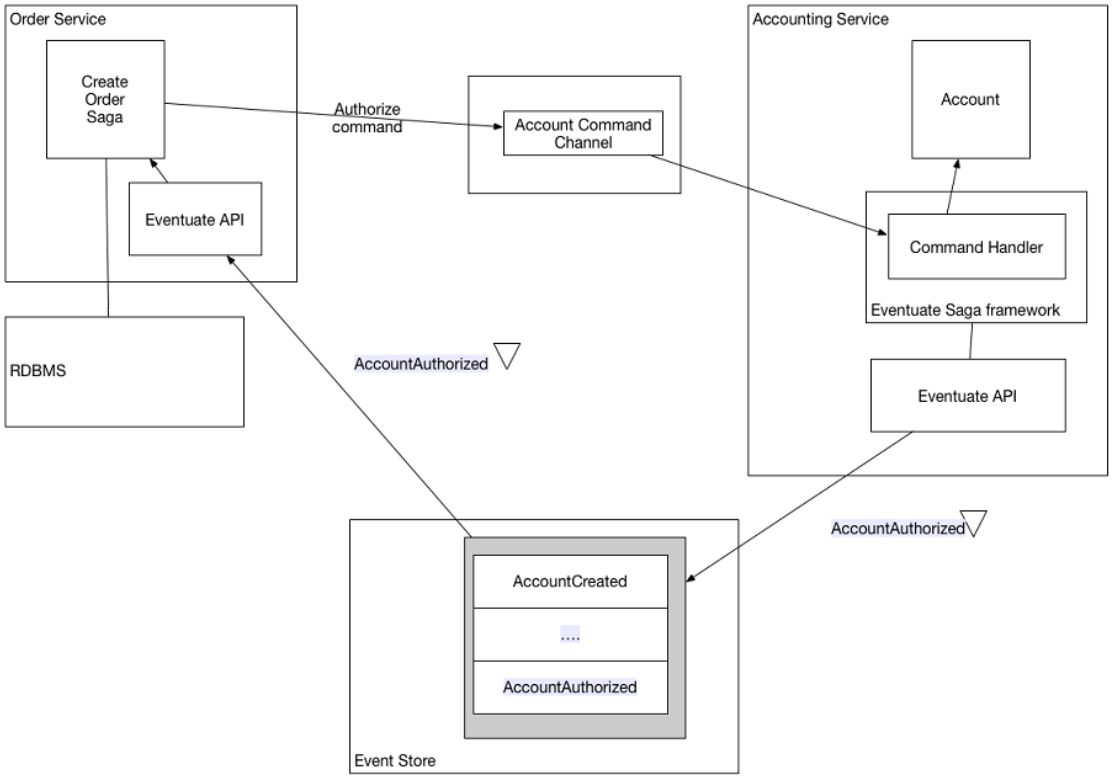
A better approach is for the saga participant to continue to send a reply message to the saga orchestrator's reply channel. However, rather than sending the reply message directly, a saga participant uses a two step process. First, when a saga command handler, creates or updates an aggregate it arranges for an `SagaReplyRequested` pseudo-event to be saved in the event store along with the real events emitted by the aggregate. Second, an event handler for the `SagaReplyRequested` pseudo-event uses the data contained in the event to construct the reply message, which it then writes to the saga orchestrator's reply channel. Let's

look at an example to see how this works.

Example event sourcing-based saga participant

In this example, we take a look at the `Accounting Service`, which is one of the participants of the `CreateOrderSaga`. Figure 6.13 shows how the `Accounting Service` handles the `Authorize Command` sent by the saga. The `Accounting Service` is implemented using the Eventuate Saga framework. The Eventuate Saga framework is an open source framework for writing sagas that use event sourcing. It is built on the Eventuate Client framework.

Figure 6.13. How the event sourcing-based `Accounting Service's` participates in the `CreateOrderSaga`



This diagram shows how the `CreateOrderSaga` and the `AccountingService` interact. The sequence of events is as follows:

1. The `CreateOrderSaga` sends an `AuthorizeAccount` command to the `AccountingService` via a messaging channel. The Eventuate Saga framework's `SagaCommandDispatcher` invokes the `AccountingServiceCommandHandler` to handle the command message.
2. The `AccountingServiceCommandHandler` sends the command to

specified `Account` aggregate.
3. The aggregate emits two events, `AccountAuthorized` and `SagaReplyRequestedEvent`.
4. The `SagaReplyRequestedEventHandler` handles the `SagaReplyRequestedEvent` by sending a reply message to the `CreateOrderSaga`

The `AccountingServiceCommandHandler`, which is shown in listing 6.6, handles the `AuthorizeAccount` command message by calling `AggregateRepository.update()` to update the `Account` aggregate.

Listing 6.6. The `AccountingServiceCommandHandler` handles command message sent by sagas. The `authorize()` method handles an `AuthorizeCommand`.

```
public class AccountingServiceCommandHandler {

  @Autowired
  private AggregateRepository<Account, AccountCommand> accountRepository;

  public void authorize(CommandMessage<AuthorizeCommand> cm) {
    AuthorizeCommand command = cm.getCommand();
    accountRepository.update(command.getOrderId(),
            command,
            replyingTo(cm)
                .catching(AccountDisabledException.class,
                          () -> withFailure(new AccountDisabledReply()))
                .build());
  }

  ...
```

The `authorize()` method invokes an `AggregateRepository` to update the `Account` aggregate. The third argument to `update()`, which is the `UpdateOptions`, is computed by this expression:

```
replyingTo(cm)
    .catching(AccountDisabledException.class,
              () -> withFailure(new AccountDisabledReply()))
    .build()
```

These `UpdateOptions` configure the `update()` method to do the following:

1. Use the *message id* as an idempotency key to ensure that the message is processed exactly once. As mentioned earlier, the Eventuate framework stores the idempotency key in all generated events and enabling it to detect and ignore duplicate attempts to update an aggregate.
2. Add a `SagaReplyRequestedEvent` pseudo event to the list of events saved in the event store. When the TODO-EventHandler receives the `SagaReplyRequestedEvent` pseudo event, it sends a reply to the `CreateOrderSaga's` reply channel.
3. Send an `AccountDisabledReply` reply instead of the default error reply when the aggregate throws an `AccountDisabledException`.

Now that we have looked at how to implement saga participants using event sourcing, lets look at how to implement saga orchestrators.

6.3.4 Implementing saga orchestrators using event sourcing

So far in this section, I have described how event sourcing-based services can initiate and participate in sagas. You can also use event sourcing to implement saga orchestrators. This will enable you to develop applications that are entirely based on an event store.

There are three key design problems you must solve when implementing a saga orchestrator:

1. How to persist a saga orchestrator?
2. How to atomically change the state of the orchestrator and send command messages?
3. How to ensure that a saga orchestrator processes reply messages exactly once?

In chapter {chapter-sagas}, I described how to implement an RDBMS-based saga orchestrator. Let's look at how to solve these problems when using event sourcing.

Persisting a saga orchestrator using event sourcing

A saga orchestrator has a very simple lifecycle. It is created and when it handles replies from saga participants it is updated. We can, therefore, persist a saga using the following events:

- `SagaOrchestratorCreated` - the saga orchestrator has been created
- `SagaOrchestratorUpdated` - the saga orchestrator has been updated

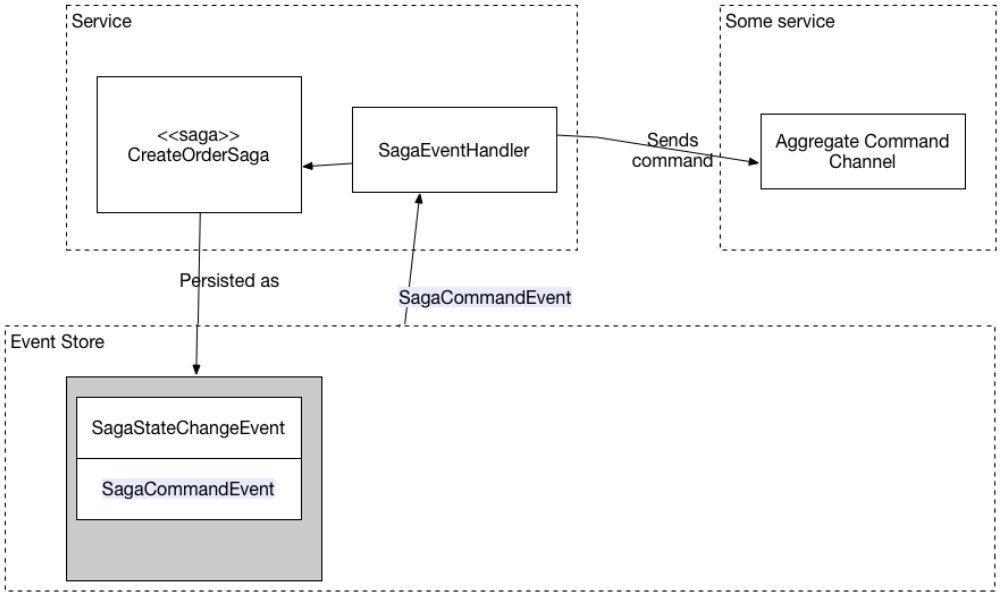
A saga orchestrator emits a `SagaOrchestratorCreated` event when it is created and a `SagaOrchestratorUpdated` event when it has been updated. These events contain the data necessary to recreate the state of the saga orchestrator. For example, the events for the `CreateOrderSaga`, which I described in chapter {chapter-sagas}, would contain a serialized (e.g. JSON) `CreateOrderSagaData`

Sending command messages reliably

Another key design issue is how to atomically update the state of the saga and send a command. As described in chapter {chapter-sagas}, the Tram-based saga implementation accomplishes this by updating the orchestrator and inserting the command message into a `message` table as part of the same transaction. An application that uses an RDBMS-based event store, such as Eventuate Local, can use the same approach. An application that uses a NoSQL-based event store, such as Eventuate SaaS, can use an analogous approach despite having a very limited transaction model. The trick is to persist a `SagaCommandEvent` event, which represents a command to send, and an event handler that subscribes to `SagaCommandEvents` and sends the command message to the appropriate channel. Figure 6.14 shows how this works.

Figure 6.14. How an event sourcing-based saga orchestrator sends sends command to saga

participants



The saga orchestrator uses a two step process to send commands:

1. A saga orchestrator emits a `SagaCommandEvent` for each command that it wants to send. The `SagaCommandEvent` contains all of the data needed to send the command such as the destination channel and the command object. These events are persisted in the event store.
2. An event handler processes these `SagaCommandEvents` and sends command messages to the destination message channel This two step approach guarantees that the command will be sent at least once.

Since the event store provides at least once delivery, an event handler might be invoked multiple times with the same event This will cause the event handler for `SagaCommandEvents` to send duplicate command messages. Fortunately, however, duplicate commands easily be detected and discarded by a saga participant using the following mechanism. The *id* of the `SagaCommandEvent`, which is guaranteed to be unique, is used as the *id* of the command message. As a result, the duplicates messages will have the same *id*. A saga participant that receives a duplicate command message will discard it using the mechanism described earlier.

Processing replies exactly once

A saga orchestrator also needs to detect and discard duplicate reply messages. It can do this using the mechanism I described earlier. The orchestrator stores the reply message's *id* in the events that it emits when processing the reply. It can then easily determine whether a message is a duplicate.

Now that we have described how to solve some key design issues when implementing a saga orchestrator using event sourcing lets look at an example of a saga that is implementing using event sourcing.

As you can see, event sourcing is a good foundation for implementing sagas. This is in addition to the other benefits of event sourcing including the inherently reliable generation of events whenever data changes; reliable audit logging; and the ability to do temporal queries. Event sourcing isn't a silver bullet. There is a significant learning curve. Evolving the event schema is not always straightforward. Despite these drawbacks, however, event sourcing has a major role to play in a microservice architecture. In the next chapter, we will switch gears and look at how to tackle the distributed data management challenge in a microservice architecture: queries. I'll describe how to solve the problem writing queries when data is stored in an event store.

6.4 Summary

- Event sourcing persists an aggregate as sequence of events. Each event represents either the creation of the aggregate or a state change. An application recreates the state of an aggregate by replaying events. Event sourcing preserves the history of a domain object, provides an accurate audit log and reliably publishes domain events.
- Snapshots improves performance by reducing the number of events that must be replayed.
- Events are stored in an event store, which is a hybrid of a database and a message broker. When a service saves an event in an event store, it delivers the event to subscribers.
- Eventuate Local is an open-source event store based on MySQL and Apache Kafka. Developers use the Eventuate client framework to write aggregates and event handlers.
- One challenge with using event sourcing is handling the evolution of events. An application potentially must handle multiple event versions when replaying events. A good solution is to use upcasting, which upgrades events to the latest version when they are loaded from the event store.
- Event sourcing is a simple way to implement choreography-based sagas. Services have event handlers that listen to the events published by event sourcing-based aggregates.
- Event sourcing is a good way to implementing saga orchestrators. As a result, you can write applications that exclusively use an event store.

7 Implementing queries in a microservice architecture

This chapter covers:

- The challenges of querying data in a microservice architecture
- When and how to implement queries using the API composition pattern
- When and how to implement queries using the Command Query Responsible Segregation (CQRS) pattern

The past few chapters have focussed on designing transactional business logic. However, transaction management is not the only data-related challenge in a microservice architecture. Another challenge is implementing query operations, which are either part of the application's public API or used internally to retrieve data displayed by the UI. This is a challenge because queries often need to retrieve data that is scattered amongst the databases owned by multiple services. You can't use distributed queries because, even if it were technically possible, it violates encapsulation. As a result, implementing a query operation is not always a relatively simple matter of designing a SQL query and executing it against a monolithic database.

Consider, for example, the query operations for the FTGO application that I described in chapter 2. Some queries retrieve data that is owned by just one service. For example, the `findConsumerProfile()`query returns data from the `Consumer Service`. The `Consumer Service` has an endpoint for this operation that simply queries its database. Other FTGO query operations, however, such as `findOrder()` and `findOrderHistory()`, return data owned by multiple services. Implementing these query operations is not as straightforward.

I start this chapter by describing two different patterns for implementing query

operations in a microservice architecture:

- The API Composition pattern, which is the simplest and should be used whenever possible. It works by making clients of the services that own the data responsible for invoking the services and combining the results.
- The Command Query Responsibility Segregation (CQRS) pattern, which is more powerful than the API composition pattern but is also more complex. It maintains one or more view databases whose sole purpose is to support queries.

After describing the two patterns, I then describe how to design CQRS views. Finally, I describe the implementation of an example view. Let's start by taking a look at the API composition pattern.

7.1 Querying using the API Composition pattern

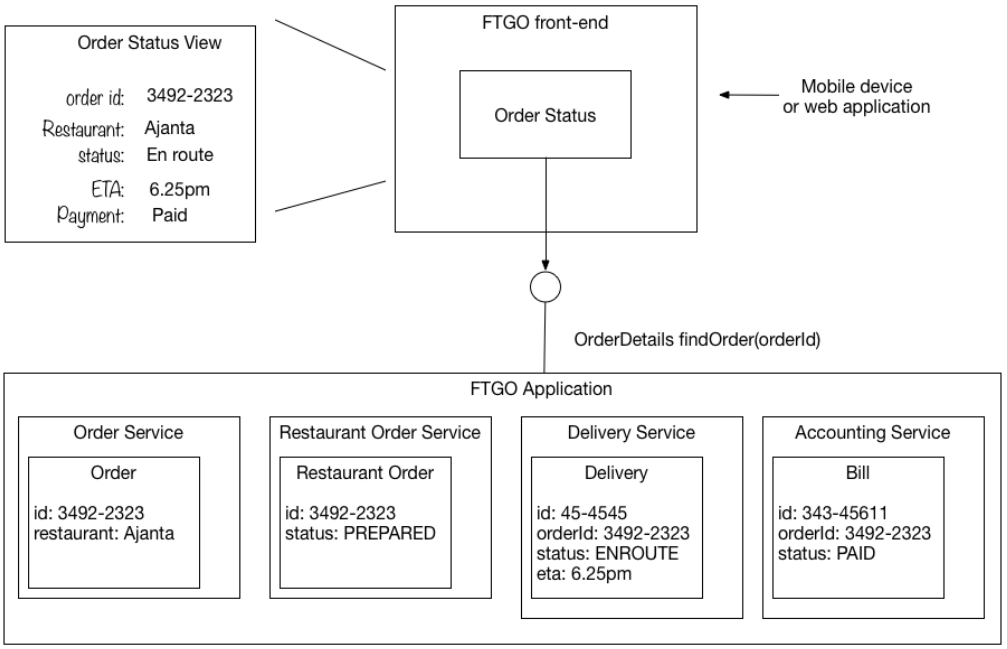
The FTGO application implements numerous query operations. Some queries, as I mentioned earlier, simply retrieve data from a single service. Implementing these queries is usually straightforward although later in this chapter, when I describe the CQRS pattern, I show examples of single service queries that are challenging to implement.

There are also queries that retrieve data from multiple services. In this section, I describe the `findOrder()` query operation, which is an example of query that retrieves data from multiple services, and explain why it is difficult to implement in a microservice architecture. I then describe the API composition pattern and show how you can use it to implement queries such as `findOrder()`.

7.1.1 The *findOrder()* query operation

The `findOrder()` operation retrieves an order by its primary key. It takes an `orderId` as a parameter and returns an `OrderDetails` object, which contains information about the order. This operation is, as shown in figure 7.1, is called by a front-end module, such as a mobile device or a web application, that implements the *Order Status* view.

Figure 7.1. The `findOrder()` operation is invoked by a FTGO front-end module and returns the details of an `Order`



The information displayed by the *Order Status* view includes basic order information including its status, its payment status, the status of the order from the restaurant's perspective, and the delivery status including its location and estimated delivery time if in transit.

A traditional monolithic application easily retrieves the order details by executing a single SELECT statement since the data resides in a single database. In contrast, in the microservice-based version of the FTGO application, the data is scattered around the following services:

- `Order Service` - basic order information including the details and status
- `Restaurant Order Management` - the status of the order from the restaurant's perspective and the estimated time it will be ready for pickup
- `Delivery Service` - the order's delivery status, its estimated delivery time, and its current location
- `Accounting Service` - the order's payment status

Any client that needs the order details must ask all of these services.

7.1.2 An overview of the API composition pattern

One way to implement query operations, such as `findOrder()`, that retrieve data owned by multiple services is to use the API composition pattern. This pattern implements a

query operation by simply invoking the services that own the data and combining the results. Figure 7.2 shows the structure of this pattern. It has two types of participants:

- An *API composer*, which implements the query operation by querying the provider services
- A *Provider service*, which is a service that owns some of the data that the query returns

Figure 7.2. The API Composition pattern consists of an API composer, and two or more provider services. The API composer implements a query by querying the providers and combining the results.

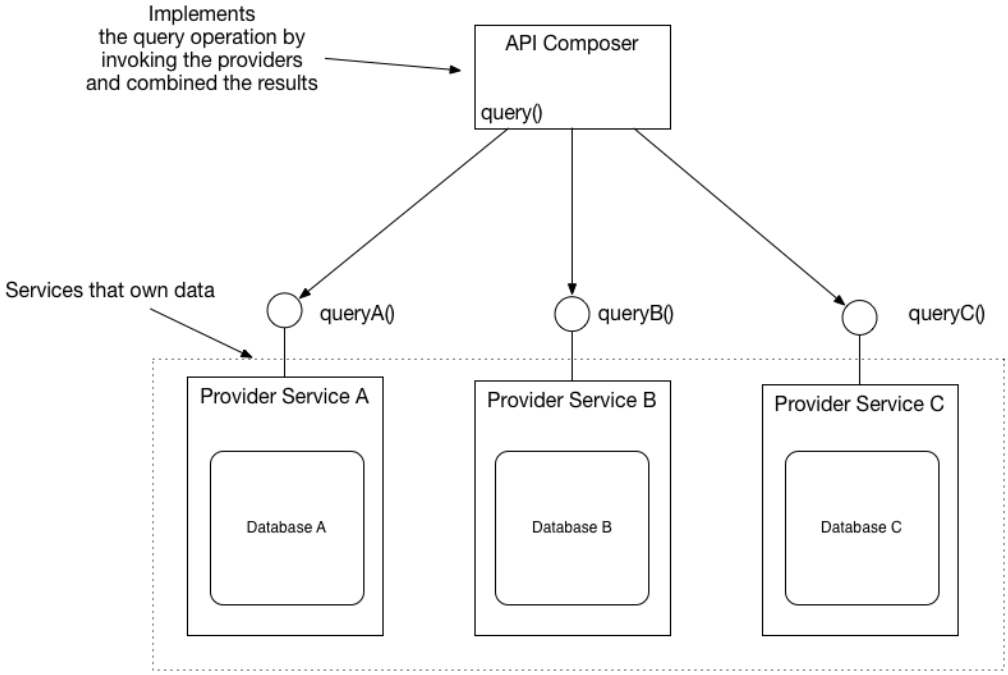


Figure 7.2 shows three provider services. The *API composer* implements the query by retrieving data from the provider services and combining the results. An *API composer* might be a client such as a web application, that needs the data to render a web page. Alternatively, it might be a service, such as an API Gateway that I describe in chapter <, which exposes the query operation as an API endpoint.

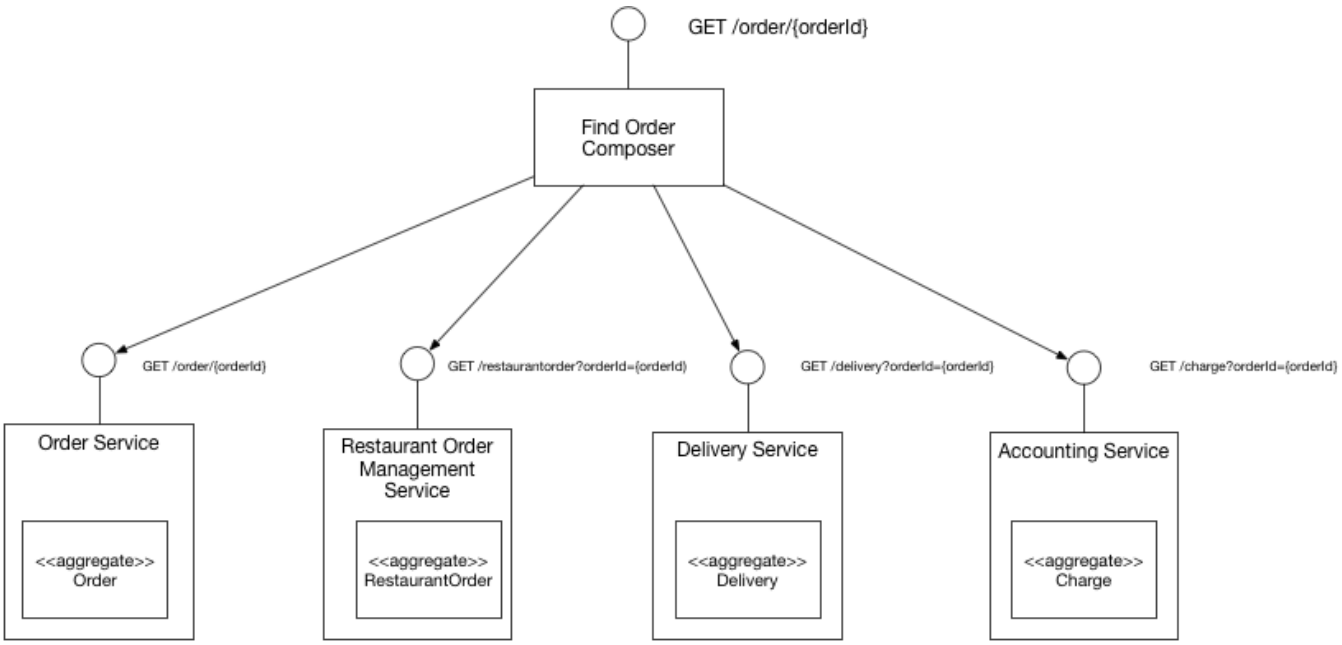
Whether you can use this pattern to implement a particular query operation depends on several factors, including how the data is partitioned, the capabilities of the APIs exposed by the services that own the data, and the capabilities of the databases used by the services. For instance, even if the *Provider services* have APIs for retrieving the required data, the aggregator might need to perform an in-memory join of potentially large datasets, which is very inefficient. Later on you will see examples of query

operations that can't be implemented using this pattern. Fortunately, however, there are many scenarios where is pattern is applicable. To see this pattern in action lets look at an example.

7.1.3 Implementing the `findOrder()` query operation using the API Composition pattern

The `findOrder()` query operation corresponds to a simple primary key-based equijoin query. It is reasonable to expect that each of the *Provider services* has an API endpoint for retrieving the required data by `orderId`. Consequently, the `findOrder()` query operation is an excellent candidate to be implemented by the API Composition pattern. The *API composer* simply invokes the four services and combines the results together. Figure 7.3 shows the design of the `Find Order Composer`.

Figure 7.3. Implementing `findOrder()` using the API Composition pattern



In this example, the *API composer* is a service, which exposes the query as a REST endpoint. The *Provider services* also implement REST APIs. However, the concept is the same if, the services used some other inter-process communication protocol, such as gRPC, instead of HTTP. The `FindOrderAggregator` implements a REST endpoint `GET /order/{orderId}`. It invokes the four services and joins the responses using the *orderId*. Each *Provider service* implements a REST endpoint that returns a response corresponding to a single aggregate. The `OrderService` retrieves its version of an `Order` by primary key and the other services use the `orderId` as a foreign key to retrieve their aggregates.

As you can see, the API composition pattern is quite simple. Lets look at a couple of design issues.

7.1.4 API Composition design issues

When using this pattern, there are a couple of design issues that you must address.

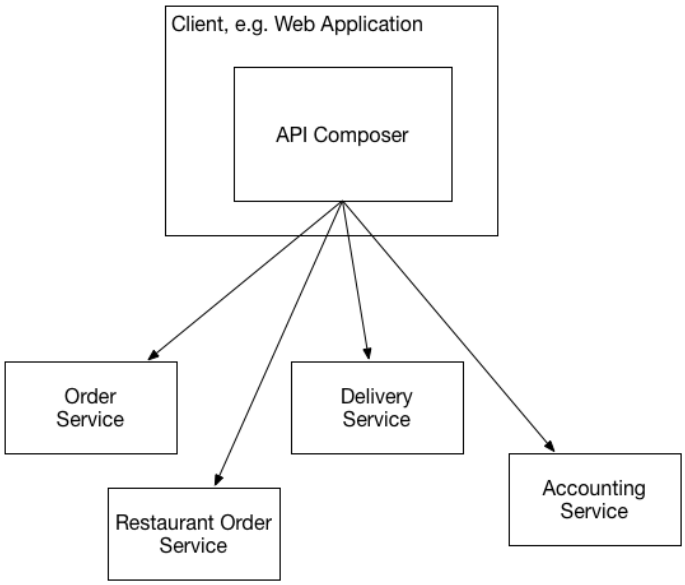
- Deciding which component in your architecture is the query operation's *API composer*.
- How to write efficient aggregation logic

Let's look at each issue.

Who plays the role of the *API composer*?

One decision that you must make is who plays the role of the query operation's *API composer*. You have three options. The first option, which is shown in figure 7.4, is for a client of the services to be the *API composer*.

Figure 7.4. Implementing API composition in a client. The client queries the provider services to retrieve the data.

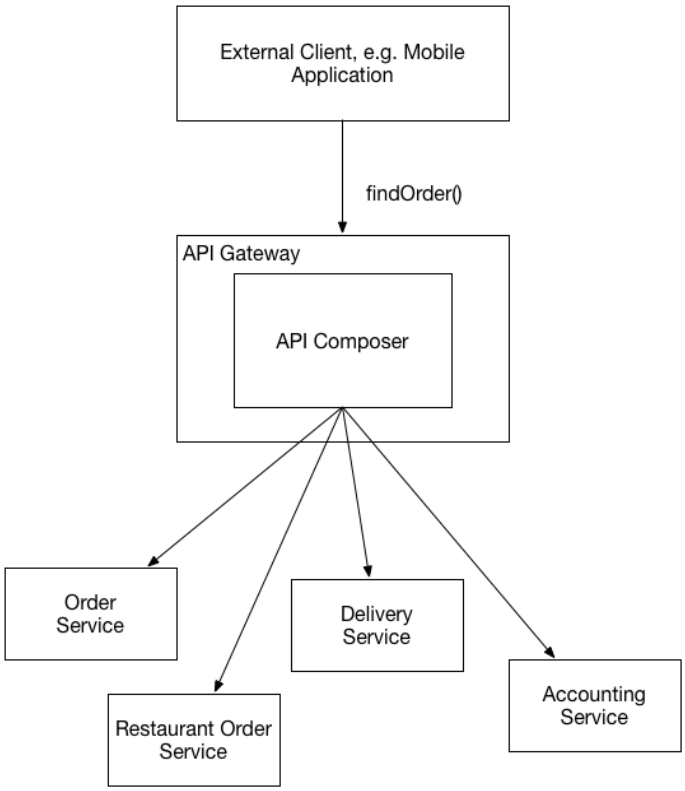


A front-end client, such as a web application, that implements the `Order Status` view and is running on the same LAN could efficiently retrieve the order details using this pattern. However, as you will learn in chapter 8, it is likely this option is not practical for clients that are outside of the firewall and access services via a slower network.

The second option, which is shown in figure 7.5, is for an API gateway, which implements the application's external API, to play the role of an *API composer* for a

query operation.

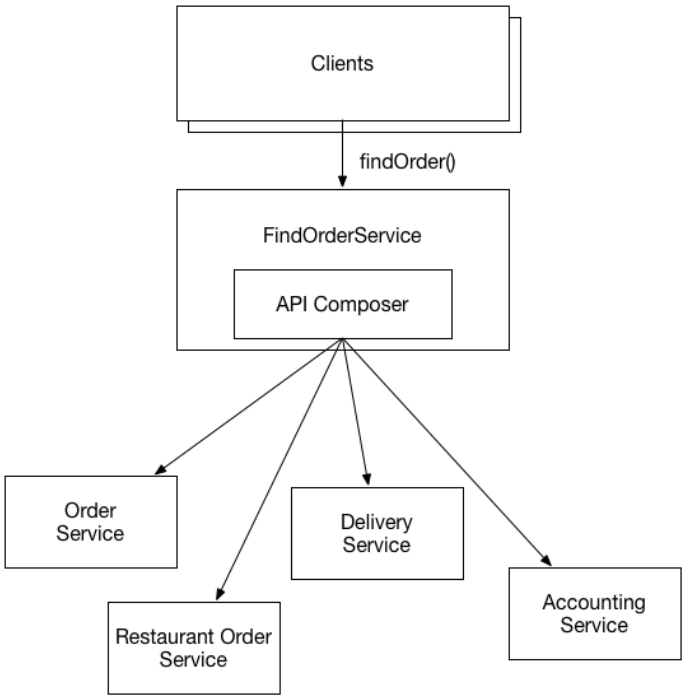
Figure 7.5. Implementing API composition in the API Gateway. The API queries the provider services to retrieve the data, combines the results, and returns a response to the client.



This option makes sense if the query operation is part of the application's external API. Instead of simply routing a request to another service, the API gateway implements the API composition logic. This approach enables a client, such as a mobile device, that is running outside of the firewall to efficiently retrieve data from numerous services with a single API call. I describe the API Gateway in chapter 8.

The third option, which is shown in figure 7.6, is to implement an *API composer* as a standalone service.

Figure 7.6. Implement a query operation used by multiple clients and services as a standalone service



You should use this option for a query operation that is used internally by multiple services. This operation can also be used for externally accessible query operations whose aggregation logic is too complex to be part of an API gateway.

Api composers should use a reactive programming model

When developing a distributed system, an ever-present concern is minimizing latency. Whenever possible, an *API composer* should call provider services in parallel in order to minimize the response time for a query operation. The `Find Order Aggregator` should, for example, invoke the four services concurrently since there are no dependencies between the calls. Sometimes, however, an *API composer* needs the result of one *Provider service* in order to invoke another service. In this case, it will need to invoke some but hopefully not all provider services sequentially.

The logic to efficiently execute a mixture of sequential and parallel service invocations can be complex. In order for an *API composer* to be maintainable as well as performant and scalable it should use a reactive design based on Java CompletableFutures, RxJava Observables or some other equivalent abstraction. I will discuss this topic further in chapter 8 when describing the API gateway pattern.

7.1.5 The benefits and drawbacks of the API composition pattern

This pattern is a simple and intuitive way to implement query operations in a microservice architecture. However, this pattern has some drawbacks:

- Increased overhead
- Risk of reduced availability
- Lack of transactional data consistency

Let's take a look at them.

Increased overhead

One drawback of this pattern is the overhead of invoking multiple services and querying multiple databases. In a monolithic application, a client can retrieve data with a single request, which will often implement a single database query. In comparison, using the API composition pattern involves multiple requests and database queries. As a result, more computing and network resources are required, which increases the cost running the application.

Risk of reduced availability

Another drawback of this pattern is reduced availability. As I described in chapter 3, the availability of an operation declines with the number of services that are involved. Since the implementation of a query operation involves at least three services - the *API composer* and at least two provider services - its availability will be significantly less than that of a single service. For example, if the availability of an individual service is 99.5% then the availability of the `findOrder()` endpoint, which invokes four provider services, is $99.5\%^{(4+1)} = 97.5\%$!

There are couple of strategies that you can use to improve availability. The first strategy is for the *API composer* to return previously cached data when a *Provider service* is unavailable. An *API composer* sometimes caches the data returned by a *Provider service* in order to improve performance. It can also use this cache to improve availability. If a provider is unavailable, the *API composer* can return (albeit potentially stale) data from the cache.

Another strategy is improving availability is for the *API composer* to return incomplete data. For example, let's imagine that the `Restaurant Order Service` is temporarily unavailable. The *API Composer*for the `findOrder()` query operation could simply omit that service's data from the response since the UI can still display useful information. I will cover more details of API design, caching and reliability when I describe the API Gateway pattern in chapter 8.

Lack of transactional data consistency

Another drawback of the API Composition pattern is the lack of data consistency. A monolithic application typically executes a query operation using a single database transaction. ACID transactions - subject to the fine print about isolation levels - ensure that an application has a consistent view of the data even if it executes multiple

database queries. In contrast, the API composition pattern executes multiple database queries against multiple databases. There is a risk, therefore, that a query operation will return inconsistent data.

For example, an `Order` retrieved from the `Order Service` might be in the `CANCELLED` state where as the corresponding `RestaurantOrder` retrieved from the `Restaurant Order Service` might not yet have been cancelled. The *API composer* must resolve this discrepancy, which increases the code complexity. To make matters worse, an *API composer* might not be always able to detect inconsistent data and will return it to the client.

Despite these drawbacks, the API Composition pattern is extremely useful. You can use it to implement many query operations. There are, however, some query operations that can't be efficiently implemented using this pattern. A query operation might, for example, require the *API composer* to perform an in-memory join of large data sets. It is usually better to implement these types of query operations using the Command Query Responsible Segregation pattern. Let's take a look at how this pattern works.

7.2 Using the Command Query Responsible Segregation (CQRS) pattern

Many enterprise applications use an RDBMS as the transactional system of record and a text search database such as ElasticSearch or Solr for text search queries. Some applications keep the databases synchronized by writing to both simultaneously. Others periodically copy data from the RDBMS to the text search engine. Applications with this architecture leverage the strengths of multiple databases: the transactional properties of the RDBMS, and the querying capabilities of the text database.

CQRS is a generalization of this kind of architecture. It maintains one or more view databases - not just text search databases - that implement one or more of the application's queries. To understand why this is useful lets first look at some queries that can't be efficiently implemented using the API Composition pattern. After that I will explain how CQRS works.

7.2.1 Motivations for using CQRS

The API composition pattern is a good way to implement many queries that must retrieve data from multiple services. Unfortunately, it is only a partial solution to the problem of querying in a microservice architecture. That is because, there are multi-service queries the API composition pattern can't implement efficiently.

What's more, there are also single service queries that are challenging to implement. Perhaps, the service's database does not efficiently support the query. Alternatively, it sometimes makes sense for a service to implement a query that retrieves data owned by a different service. Let's take a look at these problems starting with a multi-service query that can't be efficiently implemented using API composition.

Implementing the `findOrderHistory()` query operation

The `findOrderHistory()` operation retrieves a consumer's order history. It has several parameters:

- `consumerId` - identifies the consumer
- `pagination` - page of results to return
- `filter` - filter criteria including the max age of the orders to return; an optional order status; and optional keywords that match the restaurant name and menu items.

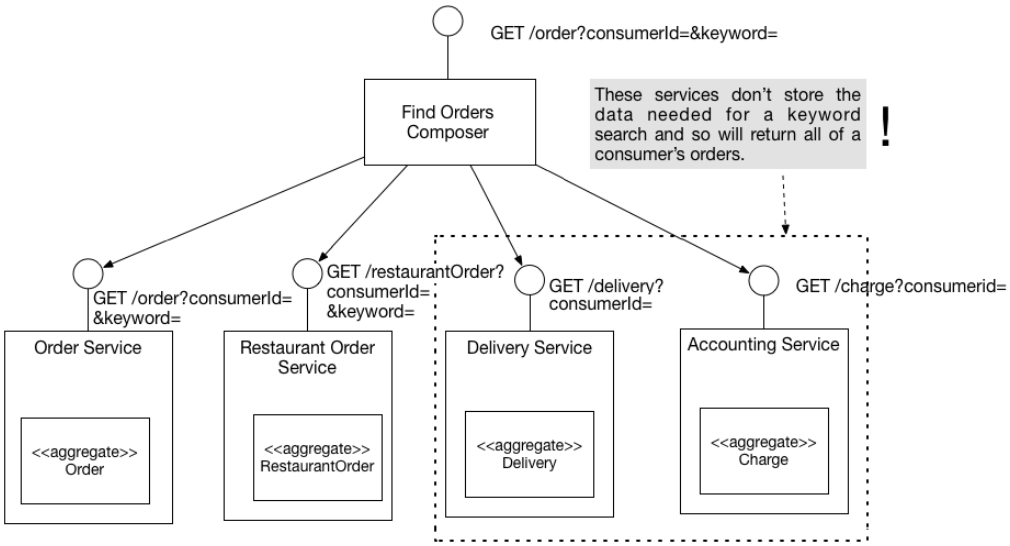
This query operation returns an `OrderHistory` object, which contains a summary of the matching orders sorted by increasing age. It is called by the module that implements the `Order History` view. This view displays a summary of each order, which includes the order number, the order status, the order total, and the estimated delivery time.

On the surface, this operation is similar to the `findOrder()` query operation. The only difference is that it returns multiple orders instead of just one. It might appear that the *API composer* simply has to execute the same query against each *Provider service* and combine the results. Unfortunately, it's not that simple.

One problem is that one or more services might not store an attribute that is used for filtering or sorting. For example, one of the `findOrder()` operation's filter criteria is keyword that matches against a menu items. Only two of the services, the `Order Service` and the `Restaurant Order Service` store an `Order's` menu items. Neither the `Delivery Service` nor the `Accounting Service` store the menu items and so can't filter their data using this keyword. Similarly, neither the `Restaurant Order Service` or the `Delivery Service` can sort by the `orderCreationDate` attribute.

There are two ways an *API composer* could solve this problem. One solution is for the *API composer* to do an in-memory join as shown in figure 7.7. It retrieves all orders for the consumer from the `Delivery Service` and the `Accounting Service` and performs a join with the orders retrieved from the `Order Service` and the `Restaurant Order Service`.

Figure 7.7. API Composition cannot efficiently retrieve a consumer's orders because some providers, such as the `Delivery Service`, don't store the attributes used for filtering.



The drawback of this approach is that it potentially requires the *API composer* to retrieve and join large datasets, which is inefficient.

The other solution is for the *API composer* to retrieve matching orders from the `Order Service` and `Restaurant Order Service` and then request orders from the other services by id. This is only practical, however, if those services have a bulk fetch API. Requesting orders individually will likely be inefficient because of excessive network API traffic.

Queries such as `findOrderHistory()` require the *API composer* to duplicate the functionality of an RDBMS's query execution engine. On the one hand, this potentially moves work from the less scalable database to the more scalable application[EBAY]. But on the other hand, it is less efficient. Also, developers should be writing business functionality - not a query execution engine. Below I show how to apply the CQRS pattern and use a separate datastore, which is designed to efficiently implement the `findOrderHistory()` query operation. But first, let's look at an example of a query operation that is challenging to implement despite being local to a single service.

A challenging single service query: `findAvailableRestaurants()`

As you have just seen, it can be challenging to implementing queries that retrieve data from multiple services. What's more, even queries that are local to a single service can be difficult to implement. There are a couple of reasons why this might be the case. One reason is because, as I describe below, sometimes it's not appropriate for the service that owns the data to implement the query. The other reason is that sometimes a service's database (or data model) doesn't efficiently support the query.

Consider, for example, the `findAvailableRestaurants()` query operation. This query finds the restaurants that are available to deliver to a given address at a given time. The heart of this query is a geospatial search (a.k.a. location-based search) for restaurants that are within a certain distance of the delivery address. It is a critical part of the order process and is invoked by the UI module that displays the available restaurants.

The key challenge of implementing this query operation is performing an efficient geospatial query. How you implement the `findAvailableRestaurants()` query depends on the capabilities of the database that stores the restaurants. For example, it is straightforward to implement the `findAvailableRestaurants()` query using either MongoDB or the Postgres and MySQL geospatial extensions. These databases support geospatial datatypes, indexes and queries. When using one of these databases, the `Restaurant Service` persists a `Restaurant` as a database record that has a `location` attribute. It finds the available restaurants using a geospatial spatial query that is optimized by a geospatial index on the `location` attribute.

If the FTGO application stores restaurants in some other some kind other database then implementing the `findAvailableRestaurant()` query is more challenging. It must maintain a replica of the restaurant data in a form that is designed to support the geospatial query. The application could, for example, use the Geospatial Indexing Library for DynamoDB that uses a table as a geospatial index. Alternatively, the application could store a replica of the restaurant data in an entirely different type of database, a situation very similar to using a text search database for text queries.

The challenge with using replicas is keeping them up to date whenever the original data changes. As you will learn below, CQRS solves the problem of synchronizing replicas.

The need to separate concerns

Another reason why single service queries are challenging to implement is that sometimes the service that owns the data should not be the one that implements the query. The `findAvailableRestaurants()` query operation retrieves data that is owned by the `Restaurant Service`. This service enables restaurant owners to manage their restaurant's profile and menu items. It stores various attributes of a restaurant including its name; address; cuisines; menu and opening hours. Given that this service owns the data, it makes sense, at least on surface, for it to implement this query operation. However, data ownership is not the only factor to consider.

You must also take into account the need to separate concerns and avoid overloading services with too many responsibilities. For example, the primary responsibility of the team that developers `Restaurant Service` is enabling restaurant managers to maintain their restaurants. That is quite different than implementing a high-volume, critical query. What's more, if they were responsible for the `findAvailableRestaurants()` query operation the team would constantly live in fear of deploying a change that prevented consumers from placing orders.

It makes sense for the `Restaurant Service` to merely provide the restaurant data to another service that which implements the `findAvailableRestaurants()` query

operation and is most likely owned by the `Order Service` team. Just as with the `findOrderHistory()` query operation and when needing to maintain geospatial index there is a requirement to maintain an eventually consistent replica of some data in order to implement a query. Lets look at how to accomplish this using CQRS.

7.2.2 Overview of CQRS

The examples I just described in section drawbacks-of-API-composition highlighted three problems that are commonly encountered when implementing queries in a microservice architecture:

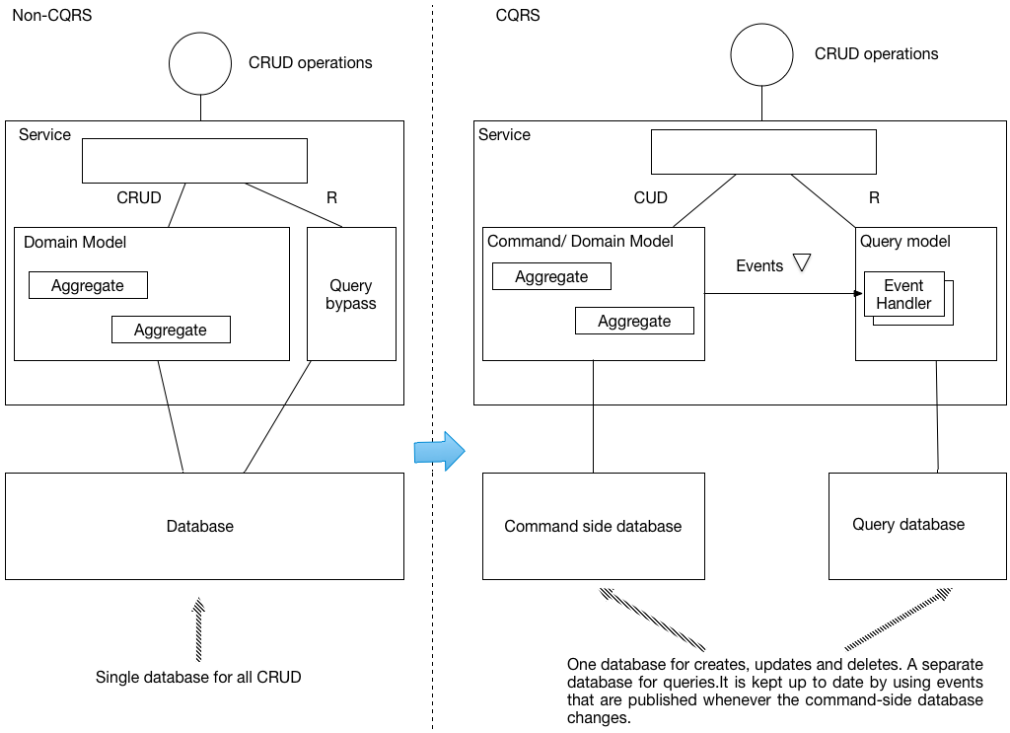
1. Using the API Composition pattern to retrieve data scattered across multiple services results in expensive, inefficient in-memory joins.
2. The service that owns the data, stores the data in a form or in a database that does not efficiently support the required query.
3. The need to separate concerns means that the service that owns the data is not the service that should implement the query operation.

The solution to all three of these problems is to use the CQRS pattern.

CQRS separates commands from queries

CQRS, as the name suggests, is all about segregation, a.k.a. separation of concerns. As figure 7.8 shows, it splits a persistent data model and the modules that use it into two parts: the command side and the query side. The command side modules and data model implement create, update and delete operations (e.g. HTTP POSTs, PUTs, and DELETEs). The query side modules and data model implement queries (e.g. HTTP GETs). The query side keeps its data model synchronized with the command side data model by subscribing to the events published by the command side.

Figure 7.8. On the left is the non-CQRS version of the service and on the right is the CQRS version. CQRS restructures a service into command side and query side modules, which have separate databases.



Both the non-CQRS and CQRS versions of the service have an API consisting of various CRUD operations. In a non CQRS-based service those operations are typically implemented by a domain model, which is mapped to a database. For performance, a few queries might bypass the domain model and access the database directly. A single persistent data model supports both commands and queries.

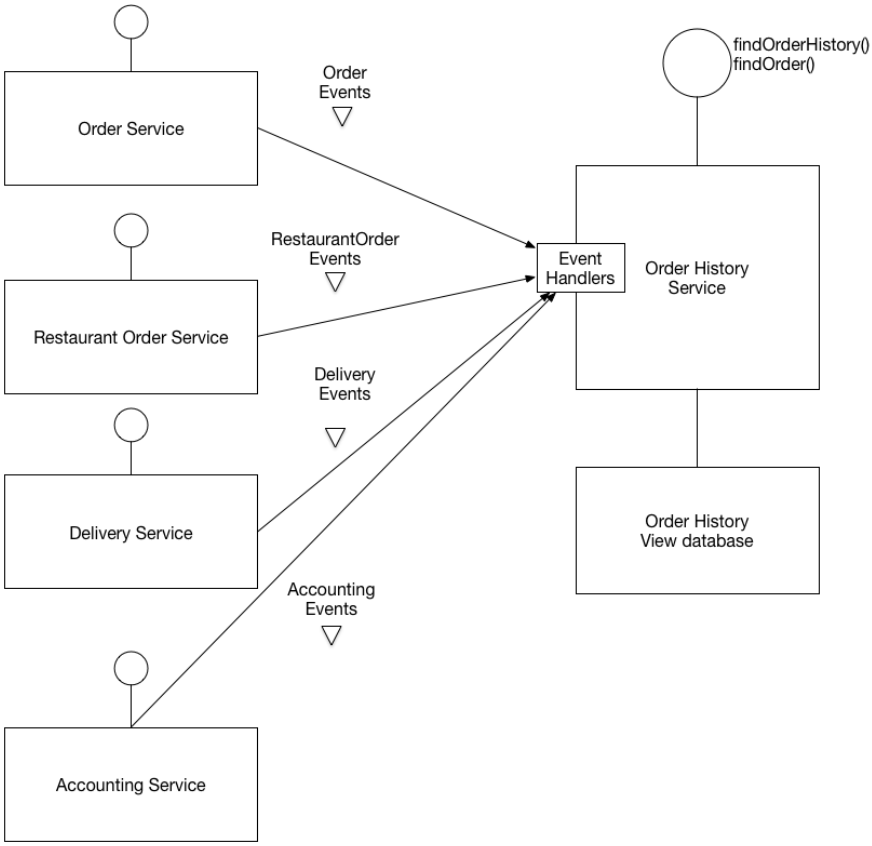
In a CQRS-based service, the command side domain model handles create, update and delete operations and its mapped to its own database. It might also handle simple queries, such as non-join, primary key-based queries. The command side publishes domain events whenever its data changes. These events might be published using a framework such as TRAM, or using event sourcing.

A separate query model handles the non-trivial queries. Its much simpler than the command side since it is not responsible for implementing the business rules. The query side uses whatever kind of database makes sense for the queries that it must support. The query side has event handlers that subscribe to domain events and update the database or databases. There might even be multiple query models, one for each type of query.

CQRS and query-only services

Not only can CQRS be applied within a service but you can also use this pattern to define query services. A query service has an API consisting of only query operations, no command operations. It implements the query operations by querying a database that it keeps up to date by subscribing to events published by one or more other services. Figure 7.9 shows the design of a `Order History Service`. This service, which I describe in more detail below, implements the `findOrderHistory()` query operation.

Figure 7.9. The design of the `Order History Service`, which is a query side service. It implements the `findOrderHistory()` query operation by querying a database, which it maintains by subscribing to events published by multiple other services.



The `Order History Service` has event handlers that subscribe to events published by the `Order Service`, the `Restaurant Order Service`, `Delivery Service` and `Accounting Service`. These event handlers update the `Order History View Database`.

A query side service is good way to implement a view that is built by subscribing to events published by multiple services. This kind of view doesn't belong to any particular service and so it makes sense to implement it as a standalone service. For example, below you will see the `Order History Service`, which is query service that implements the `findOrderHistory()` query operation. This service subscribes to events published by several services including `Order Service`, `Delivery Service` etc.

A query service also a good way to implement a view that replicates of data owned by a single service yet because of the need to separate concerns isn't part of that service. For example, the FTGO developers can define an `Available Restaurants Service`, which implements the `findAvailableRestaurants()` query operation described earlier. It subscribes to events published by the `Restaurant Service` and updates a database designed for efficient geospatial queries.

In many ways, CQRS is a event-based, generalization of the widely used approach of using RDBMS as the system of record and a text search engine, such as Elastic Search, to handle text queries. What's different is that CQRS uses a broader range of database types - not just a text search engine. Also, CQRS query side views are updated in near real-time by subscribing to events. Lets now look at the benefits and drawbacks of CQRS.

7.2.3 Benefits of CQRS

CQRS has both benefits and drawbacks. Let's first look at the benefits, which are:

- Enables the efficient implementation of queries in a microservice architecture
- Enables the efficient implementation of a diverse queries
- Makes querying possible in an event sourcing-based application
- Improves separation of concerns

Enables the efficient implementation of queries in a microservice architecture

One benefit of the CQRS pattern is that it efficiently implements queries that retrieve data that is owned by multiple services. As I described earlier, using the API composition pattern to implement queries sometimes results in expensive, inefficient in-memory joins of large datasets. For those queries, it is more efficient to use an easily queried CQRS view that pre-joins the data from two or more services.

Enables the efficient implementation of a diverse queries

Another benefit of CQRS is that it enables an application or service to efficiently implement a diverse set of queries. Attempting to support all queries using a single persistent data model is often challenging and in some cases impossible. Some NoSQL databases have very limited querying capabilities. And, even when a database has extensions to support a particular kind of query, using a specialized database is often more efficient. The CQRS pattern avoids the limitations of a single datastore by defining one or views, each one of which efficiently implements specific queries.

Enables querying in an event sourcing-based application

CQRS is also overcomes a major limitation of event sourcing. An event store only supports primary-key based queries. The CQRS pattern addresses this limitation by defining one or more views of the aggregates, which are kept up to date, by subscribing to the streams of events that are published by the event sourcing-based aggregates As a result, an event sourcing-based application invariably uses CQRS.

Improves separation of concerns

Another benefit of CQRS is that it separates concerns. A domain model and its corresponding persistent data model doesn't handle both commands and queries. The CQRS pattern defines separate code modules and database schemas for the command and query sides of a service. By separating concerns, the command side and query side are likely to be simpler and easier to maintain.

Moreover, CQRS enables the service that implements a query to be different than the service that owns the data. For example, earlier I described how even though the `Restaurant Service` owns the data that is queried by the `findAvailableRestaurants` query operation it makes sense for another service to implement such as critical, high-volume query. A CQRS query service maintains a view by subscribing to the events publishing by the service or services that own the data.

7.2.4 Drawbacks of CQRS

Even though CQRS has several benefits, it also has significant drawbacks:

- More complex architecture
- Dealing with the replication lag

Let's look at these drawbacks starting with the increased complexity.

More complex architecture

One drawback of CQRS is that it adds complexity. Developers must write the query side services that update and query the views. There is also the extra operational complexity of managing and operating the extra datastores. What's more, an application might use different types of databases, which adds further complexity for both developers and operations.

Dealing with the replication lag

Another drawback of CQRS is dealing with the "lag" between the command side and the query side views. As you might expect, there is delay between when command side publishes an event and that event is processed by the query side and the view updated. A client application that updates an aggregate and then immediately queries a view may see the previous version of the aggregate. It must often be written in a way that avoids exposing these potential inconsistencies to the user.

One solution is for the command side and query side APIs to supply the client with

version information that enables it to tell that the query side is out of date. A client can poll the query side view until it is up to date. Below I will describe how the service APIs can enable a client to do this.

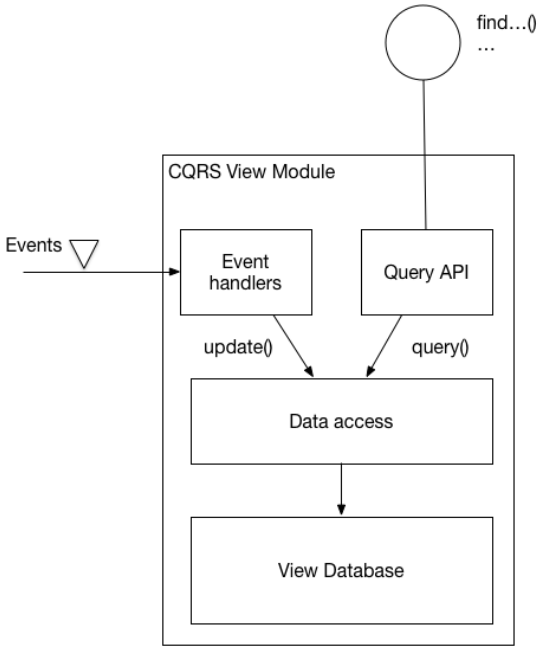
A UI application such as a native mobile application or single page JavaScript application can handle replication lag by updating its local model once the command is successful without issuing a query. It can, for example, update its model using data returned by the command. Hopefully, when a user action triggers a query the view will be up to date. One drawback of this approach is that the UI code might need to duplicate server-side side code in order to update its model.

As you can see, CQRS has both benefits and drawbacks. As I mentioned earlier, you should use the API composition whenever possible and use CQRS only when you must. Now that you have seen the benefits and drawbacks of CQRS, let's now look at how to design CQRS views.

7.3 Designing CQRS Views

A CQRS view module has an API consisting of one more query operations. It implements these query operations by querying a database that it maintains by subscribing to events published by one or more services. As figure 7.10 shows, a view module consists of consists of a view database and three submodules.

Figure 7.10. The design of a CQRS view module. Event handlers update the view database, which is queried by the Query API module.



The *data access* module implements the database access logic. The *event handlers* and *query API* modules use the *data access* module to update and query the database. The event handlers module subscribes to events and updates the database. The query API module implements the query API.

You must make some important design decisions when developing a view module:

- Choose a database and design the schema.
- When designing the data access module, you must address various issues including ensuring that updates are idempotent and handling concurrent updates.
- When implementing a new view in an existing application or changing the schema of an existing you must implement a mechanism to efficiently build or rebuild the view.
- Decide how to enable a client of the view to cope with the replication lag, which I described above.

Lets look at each of these issues.

7.3.1 Choosing a view datastore

A key design decision is the choice of database and the design of the schema. The primary purpose of the database and the data model is to efficiently implement the view module's query operations. It's the characteristics of those queries that are the primary consideration when selecting a database. The database must also, however, efficiently implement the update operations performed by the event handlers.

SQL vs. NoSQL databases

Not that long ago, there was one type of database to rule them all: the SQL-based RDBMS. As the Web grew in popularity, however, various companies discovered that an RDBMS could not satisfy their web scale requirements. That led to the creation of the so-called NoSQL databases. A NoSQL database typically has a limited form of transactions and a less general querying capabilities. For certain use cases, these databases have certain advantages over SQL databases including a more flexible datamodel and better performance and scalability.

A NoSQL database is often a good choice for a CQRS view, which can leverage its strengths and ignore its weaknesses. A CQRS view benefits from the richer data model, and performance of a NoSQL database. It is unaffected by the limitations of a NoSQL database, since it only uses simple transactions and executes a fixed set of queries.

Having said that, sometimes it makes sense to implement a CQRS view using a SQL database. A modern RDBMS running on modern hardware has excellent performance. Developers, database administrators, and IT operations are, in general, much more familiar with SQL databases than they are with NoSQL databases. As mentioned earlier, SQL databases often have extensions for non-relational features such as a geospatial datatypes and queries. Also, a CQRS view might need to use a SQL database in order to support a reporting engine.

As you can see in table 7.1, there are lots of different options to choose from. And to make the choice even more complicated, the differences between the different types of database is starting to blur. For example, MySQL, which is an RDBMS, has excellent support for JSON, which is one of the strengths of MongoDB, a JSON-style document-oriented database.

Table 7.1. Query-side view stores

If you need....	then use....	for example...
PK-based lookup of JSON objects	a document store such as MongoDB or DynamoDB, or a key value store such as Redis.	Implement order history by maintaining a MongoDB Document containing the per-customer
Query-based lookup of JSON objects	a document store such as MongoDB or DynamoDB	Implement customer view using MongoDB or DynamoDB
Text queries	a text search engine such as Elastic Search	Implement text search for orders by maintaining a per-order Elastic Search document
Graph queries	a graph database such as Neo4j	Implement fraud detection by maintaining a graph of customers, orders, and other data
Globally distributed database infrastructure	a NoSQL database that has this built in	e.g. Cassandra
Traditional SQL reporting/BI	an RDBMS	Standard business reports and analytics

Now that I have discussed the different kind of databases that you can use to implement a CQRS view let's look the problem of how to efficiently update a view.

Supporting update operations

As well as efficiently implementing queries, the view data model must also efficiently implement the update operations executed by the event handlers. Usually, an event handler will update or delete a record in the view database using its primary key. For example, below I describe the design of a CQRS view for the `findOrderHistory()` query. It stores each `Order` as a database record and uses the `orderId` the primary key. When this view receives an event from the `Order Service` it can straightforwardly update the corresponding record.

Sometimes, however, it will need to update or delete a record using the equivalent 'foreign key'. Consider, for instance the event handlers for `Delivery*` events. If there is a one-to-one corresponding between a `Delivery` and `Order` then `Delivery.id` might be the same as the `Order.id`. If it is then `Delivery*` event handlers can easily update the correspond database record.

But lets suppose that a `Delivery` has its own primary key or there is one-to-many relationship between an `Order` and a `Delivery`. Some `Delivery*` events, such as `DeliveryCreated` event, will contain the `orderId`. But other events such as a `DeliveryPickedUp` event might not. In this scenario, an event handler for `DeliveryPickedUp` will need to update a record using the `deliveryId` as the

equivalent of a foreign key.

Some types of database efficiently support foreign-key based update operations. For example, if you are using an RDBMS or MongoDB you simply create a index on the necessary columns. MongoDB also has equivalent functionality. However, non-primary key based updates are not straightforward when using other NOSQL Databases. The application will need to maintain some kind of database-specific mapping from a foreign key to primary key in order to determine which record to update. For example, an application that uses DynamoDB, which only supports primary key-based updates and deletes, must first query a DynamoDB index (describe below) to determine the primary keys of the items to update or delete.

7.3.2 Data access module design

The event handlers and the query API module don't access the datastore directly. Instead they use the data access module, which consists of a data access object (DAO) and its helper classes. The DAO has several responsibilities. It implements the update operations invoked by the event handlers and the query operations invoked by the query module. The DAO maps between the data types used by the higher-level code and the database API. It is also must handle concurrent updates and ensure that updates are idempotent. Lets look at these issues starting with how to handle concurrent updates.

Handling concurrency

Sometimes a DAO must handle the possibility multiple concurrent updates to the same database record. If a view subscribes to events publishing by a single aggregate type, there won't be any concurrency issues That is because events published by a particular aggregate instance are processed sequentially. As a result, a record corresponding to an aggregate instance won't be updated concurrently. However, iff a view subscribes to events published by multiple aggregate types then it is possible that multiple events handlers update the same record simultaneously.

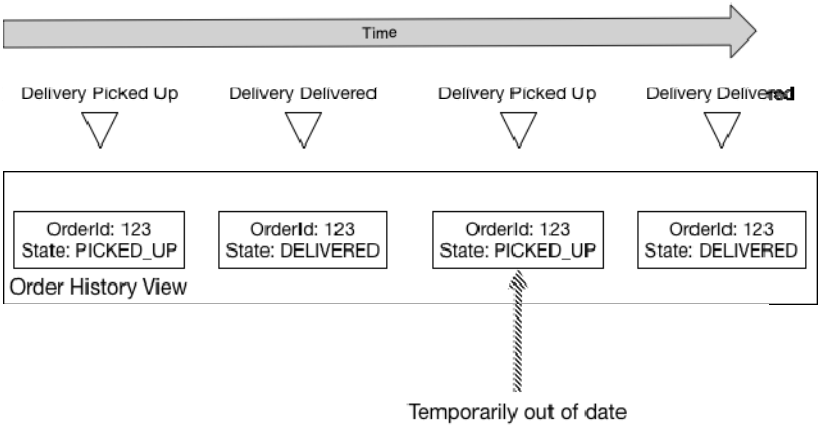
For example, an event handler for an `Order*` event might be invoked at the same time as an event handler for a `Delivery*` event for same order. Both events handlers then simultaneously invoke the DAO to update the database record for that `Order`. A DAO must be written in a way that ensures that this situation is handled correctly. It must not allow one update to overwrite another. If a DAO implements updates by reading a record and then writing the updated record it must use either pessimistic or optimistic locking. Below you will see an example of a DAO that handles concurrent updates by updating database records without reading them first.

Idempotent event handlers

As I mentioned in chapter 3, an event handler might be invoked with the same event more than once. This is generally not a problem if a query side event handler is idempotent. An event handler is an idempotent if handling duplicate events results in the correct outcome. In the worst case, the view datastore will temporarily be out of date. For example, an event handler that maintains the `Order History` view might be

invoked with the following albeit improbable sequence of events shown in figure 7.11: `DeliveryPickedUp`, `DeliveryDelivered`, `DeliveryPickedUp`, `DeliveryDelivered`.

Figure 7.11. The `DeliveryPickedUp` and `DeliveryDelivered` events are delivered twice which causes the order state in view to be temporarily out of date.



After the event handler processes the second `DeliveryPickedUp` event, the `OrderHistory` view temporarily contains the out of date state of the `Order` until the `DeliveryDelivered` is processed. If this behavior is undesirable then the event handler should detect and discard duplicates events like a non-idempotent event handler.

An event handler is not idempotent if duplicate events result in an incorrect outcome. For example, an event handler that increments the balance of a bank account is not idempotent. An non-idempotent event handler must, as I described in chapter 3, detect and discard duplicate events by recording the *ids* of events that it has processed in the view datastore.

In order to be reliable, the event handler must record the event *id* and update the datastore atomically. How to do this depends on the type of the database. If the view database store is a SQL database, the event handler could simply insert processed events into a `PROCESSED_EVENTS` table as part of the transaction that updates the view. If, however, the view datastore is a NoSQL database that has a limited transaction model, the event handler must save the event in the datastore 'record' (e.g. MongoDB document, or DynamoDB table item) that it updates.

It is important to note that the event handler does not need to record the id of every event. If, as is the case with Eventuate, events have a monotonically increasing *id* then each record only needs to store the `max(eventId)` that is received from a given aggregate instance. Furthermore, if the record corresponds to a single aggregate instance then the event handler only need to record `max(eventId)`. Only records that represent joins of events from multiple aggregates must contain a map

from `[aggregate type, aggregate id]` to `max(eventId)`.

For example, below you will see that the DynamoDB implementation the `Order History` view contains items that have attributes for tracking events that look like this:

```
{...
   "Order3949384394-039434903" : "0000015e0c6fc18f-0242ac1100e50002",
   "Delivery3949384394-039434903" : "0000015e0c6fc264-0242ac1100e50002",
}
```

This view is a join of events published by various services services. The name of each of these event tracking attributes is «`aggregateType`»«`aggregateId`» and value is the `eventId`. Later on, I describe how this works in more detail.

Enabling a client application to use an eventual consistent view

As I described earlier, one issue with using CQRS is that a client that updates the command side and then immediately executes a query might not see its own update. The view is eventually consistent because of the unavoidable latency of the messaging infrastructure.

The command and query module APIs can enable the client to detect an inconsistency using the following approach. A command side operation returns a token containing the *id* of the published event to the client. The client then passes the token to a query operation, which returns an error if the view has not been updated by that event. A view module can implement this mechanism using the duplicate event detection mechanism.

7.3.3 Adding and updating CQRS views

CQRS views will be added and updated throughout the lifetime of an application. Sometimes you need to add a new view to support a new query. At other times you might need to recreate a view because the schema has changed or you need to fix a bug in code that updates the view.

Adding and updating views is conceptually quite simple. To create a new view, you simply develop the query side module, setup the datastore and deploy the service. The query side module's event handlers process all of the events and eventually the view will be up to date. Similarly, updating an existing view is also conceptually simple: you simply change the event handlers and rebuild the view from scratch. The problem, however, is that this approach is unlikely to work in practice. Let's look at the issues.

Build CQRS views using archived events

One problem is that message brokers can't store messages indefinitely. Traditional message brokers such as RabbitMQ delete a message once it has been processed by a consumer. Even more modern brokers such as Apache Kafka, which retain messages for a configurable retention period, aren't intended to store events indefinitely. As a result, a view can't be built by simply reading all the needed events from the message broker. Instead, an application must also read older events that have been archived in, for example, AWS S3. This can be done using a scalable, big data technology such as

Apache Spark.

Build CQRS views incrementally

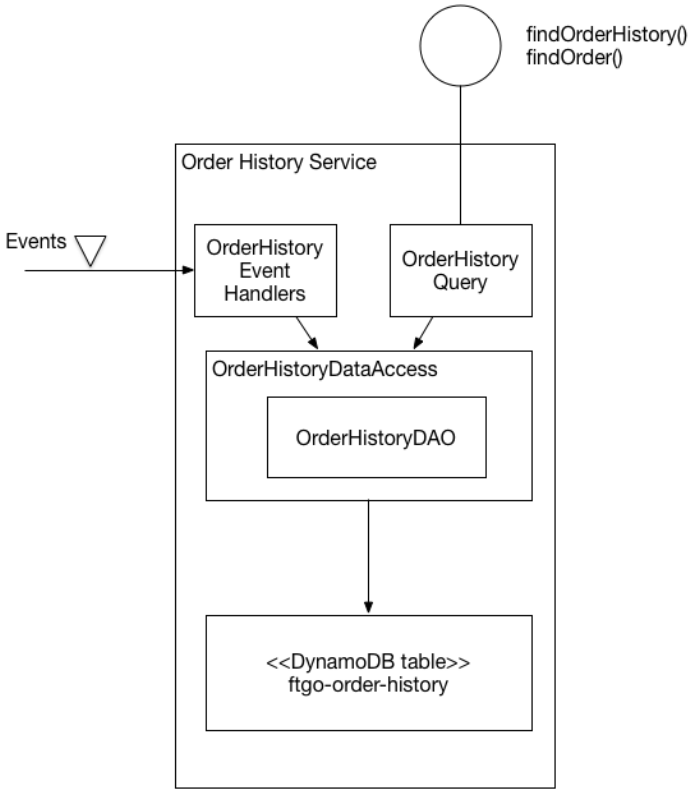
Another problem with view creation is that the time and resources required to process all events keeps growing over time. Eventually, view creating will become too slow and expensive. The solution is to use a two-step incremental algorithm. The first step, periodically computes a snapshot of each aggregate instance based on its previous snapshot and events that have occurred since that snapshot was created. The second step creates a view using the snapshots and any subsequent events.

7.4 Implementing CQRS view with AWS DynamoDB

Now that we have looked at the various design issues that you must address when using CQRS, lets now look at an example. In this section, I describe how to implement a CQRS view for the `findOrderHistory()` operation using DynamoDB. AWS DynamoDB is a scalable, NoSQL database that available as a service on the Amazon cloud. As I describe in more detail below, the DynamoDB data model consists of tables, which contain items that, like JSON objects, are collections of hierarchical name-value pairs. AWS DynamoDB is a fully managed database and you can scale up and down the capacity of a table dynamically.

The CQRS view for the `findOrderHistory()` consumes events from multiple services and so it is implemented as a standalone `Order View Service`. The service has an API that implements two operations, `findOrderHistory()` and `findOrder()`. Even though `findOrder()` can be implemented using API Composition, this view provides this operation for free. Figure 7.12 shows the design of the service. The `Order History Service` is structured as a set of modules, each of which implements a particular responsibility in order to simplify development and testing.

Figure 7.12. The design of the OrderHistoryService



The responsibility of each module is as follows:

- `OrderHistoryEventHandlers` - subscribes to events published by the various services and invokes the `OrderHistoryDAO`
- `OrderHistoryQuery API` module - implements the REST endpoints described above
- `OrderHistoryDataAccess` - contains the `OrderHistoryDAO`, which defines the methods that update and query the `ftgo-order-history` DynamoDB table, and its helper classes
- `ftgo-order-history` DynamoDB table - the table that stores the orders

Lets look at the design of the event handlers, the DAO and the DynamoDB table in more detail.

7.4.1 `OrderHistoryEventHandlers` module

This module consists of the event handlers that consume events and update the DynamoDB table. As listing 7.1 shows, the event handlers are simple methods. Each method is a simple one-liner that invokes a `OrderHistoryDao` method with arguments

that are derived from the event.

Listing 7.1. The `OrderHistoryEventHandlers` class defines several one-line event handler methods

```
public class OrderHistoryEventHandlers {

  private OrderHistoryDao orderHistoryDao;

  public OrderHistoryEventHandlers(OrderHistoryDao orderHistoryDao) {
    this.orderHistoryDao = orderHistoryDao;
  }

  @DomainEventHandler
  public void handleOrderCreated(DomainEventEnvelope<OrderCreated> dee) {
    orderHistoryDao.addOrder(dee.getEvent().getOrder(), makeSourceEvent(dee));
  }

  @DomainEventHandler
  public void handleDeliveryPickedUp(DomainEventEnvelope<DeliveryPickedUp>
                                                dee) {
   orderHistoryDao.notePickedUp(dee.getEvent().getOrderId(),
           makeSourceEvent(dee));
  }

  ...
```

Each event handler has a single parameter of type `DomainEventEnvelope`, which contains the event and some metadata describing the event. For example, the `handleOrderCreated()` method is invoked to handle an `OrderCreated` event. It calls `orderHistoryDao.addOrder()` Similarly, the `handleDeliveryPickedUp()` method is invoked to handle a `DeliveryPickedUp` event. It calls `orderHistoryDao.notePickedUp()`.

Both methods call the helper method `makeSourceEvent()`, which constructs a `SourceEvent` containing the type and id of the aggregate that emitted the event and event id. Below you will see that the `OrderHistoryDao` uses the `SourceEvent` to ensure that update operations are idempotent. Lets now look at the design of the DynamoDB table and after that I will examine `OrderHistoryDao`.

7.4.2 Data modeling and query design with DynamoDB

Like many NoSQL databases, DynamoDB has data access operations that are much less powerful than those that are provided by an RDBMS. Consequently, you must carefully design how the data is stored. In particular, the queries often dictate the design of the schema.

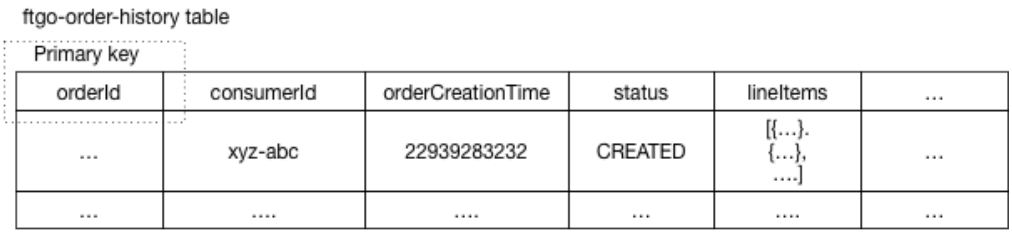
The structure of the `ftgo-order-history` table

The DynamoDB storage model consists of tables, which contain items, and indexes, which provide alternative ways to access a table's items and are described below. An item is a collection of named attributes. An attribute value is either a scalar value such

as a string; a multi-valued collection of strings; or a collection of named attributes. While an item is the equivalent to a row in an RDBMS, it is a lot more flexible and can store an entire aggregate.

This flexibility enables the `OrderHistoryDataAccess` module to store each `Order` as a single item in a DynamoDB table called `ftgo-order-history`. Each field of the `Order` class is mapped to an item attribute as shown in figure 7.13. Simple fields such as `orderCreationTime` and `status` are mapped to single value item attributes. The `lineItems` field is mapped to an attribute that is a list of maps, one map per time line. It can be considered to be a JSON array of objects.

Figure 7.13. Preliminary structure of the DynamoDB OrderHistory table.



An important part of the definition of a table is its primary key. A DynamoDB application inserts, updates and retrieves a table's items by primary key. It would seem to make sense for the primary key to `orderId.` This enables the `Order History Service` to insert, update, and retrieve an order by `orderId`. However, before finalizing this decision let's first explore how a table's primary key impacts the kinds of data access operations it supports.

Defining an index for the `findOrderHistory` query

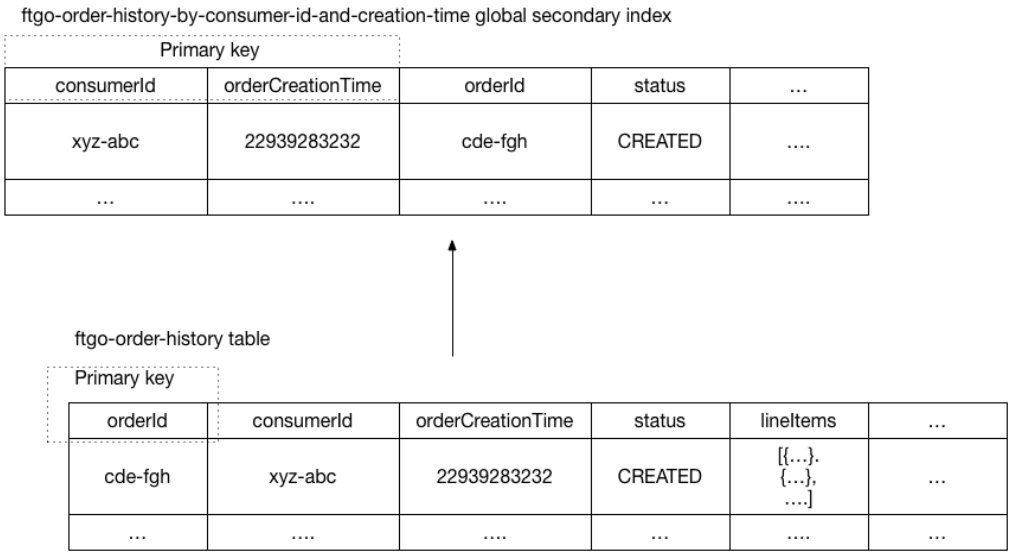
This table definition supports primary key-based reads and writes of `Orders`. However, it doesn't support a query such as `findOrderHistory()` that returns multiple matching orders sorted by increasing age. That's because a DynamoDB table that has a primary key that is single scalar attribute does not support queries that return multiple items.

The DynamoDB `query()` operation requires a primary key to consists of two scalar attributes. The first attribute is a partition key. The partition key is so called because it hashed and used to select item's storage partition. The second attribute, which is optional is the sort key. A `query()` operation returns those items that have the specified partition key; have a sort key in the specified range; and that match the optional filter expression. The query returns items in the order specified by the sort key.

The `findOrderHistory()` query operation returns a consumer's orders sorted by increasing age. It therefore requires a primary key that has the `consumerId` as the partition key and the `orderCreationDate`as the sort key. However, it doesn't make sense for `(consumerId, orderCreationDate)` to be the primary key of the `ftgo-order-history` table since it is not unique.

The solution is for the `findOrderHistory()` query operation to use what DynamoDB calls a secondary index on the `ftgo-order-history` table. This index has `(consumerId, orderCreationDate)` as its non-unique key. Like an RDBMS index, a DynamoDB index is automatically updated whenever its table is updated. But unlike a typical RDBMS index, a DynamoDB index can have non-key attributes. Non-key attributes improves performance since they are returned by the query and so the application doesn't have to fetch them from the table. Also, as you will see below, they can also be used for filtering, Figure 7.14 shows the structure of the table and this index.

Figure 7.14. The design of the OrderHistory table and index



The index is part of the definition of the `ftgo-order-history` table and is called `ftgo-order-history-by-consumer-id-and-creation-time`. The index's attributes include the primary key attributes, `consumerId` and `orderCreationTime`, attributes and non-key attributes including `orderId` and `status`.

The `ftgo-order-history-by-consumer-id-and-creation-time` enables the `OrderHistoryDaoDynamoDb` to efficiently retrieve a consumer's orders sorted by increasing age. Lets now look at how to retrieve only those orders that match the filter criteria.

Implementing the search criteria

The `findOrderHistory()` query operation has `filter` parameter that specifies the search criteria. One filter criteria is the maximum age of the orders to returns. This is easy to implement since the DynamoDB `Query` operation's *key condition expression* supports a range restriction on the sort key. The other filter criteria correspond to non-key attributes and can be implemented using a *filter expression*,

which is a boolean expression. A DynamoDB query operation returns only those items that satisfy the filter expression. For example, to find `Orders` that are `CANCELLED`, the `OrderHistoryDaoDynamoDb` uses a query expression `orderStatus = :orderStatus`, where `:orderStatus` is a placeholder parameter.

The keyword filter criteria is more challenging to implement. It selects orders whose restaurant name or menu items match the one of the specified keywords. The `OrderHistoryDaoDynamoDb` enables the keyword search by tokenizing the restaurant name and menu items and storing the set of keywords in a set-valued attribute called `keywords`. It finds the orders that match the keywords by using a filter expression that uses the `contains()` function, e.g `contains(keywords, :keyword1) OR contains(keywords, :keyword2)` where `:keyword1` and `:keyword2` are placeholders for the specifies keywords.

Paginating query results

Some consumers will have a large number of orders. It makes sense, therefore, for the `findOrderHistory()` query operation to use pagination. The DynamoDB query operation has an operation `pageSize`parameter, which specifies the maximum number of items to return. If there are more items then the result of the query has a non-null `LastEvaluatedKey` attribute. A DAO can retrieve the next page of items by invoking the query with `exclusiveStartKey` parameter set to the `LastEvaluatedKey`.

As you can see, DynamoDB doesn't support position-based pagination. Consequently, the `Order History Service` returns an opaque pagination token to its client. The client uses this pagination token to request the next page of results.

Now that I have described how to query DynamoDB for orders and let's look at how to insert and update them.

Updating orders

DynamoDB supports two operations for adding and updating orders, `PutItem()` and `UpdateItem()`. The `PutItem()` operation creates or replaces an entire item by its primary key. In the theory, the `OrderHistoryDaoDynamoDb` could use this operation to insert and update orders. One challenge, however, with using `PutItem()` is ensuring that simultaneous updates to the same item are handled correctly. The `OrderHistoryDaoDynamoDb` must ensure that one update is not overwritten by another update.

The solution is to use optimistic locking. When `OrderHistoryDaoDynamoDb` invokes `PutItem()` it must that the order had not been changed it was loaded. On the one hand, this is fairly easy to implement since DynamoDB's supports optimistic locking. But on the other hand, an even simpler and more efficient approach is to use the `UpdateItem()` operation.

The `UpdateItem()` operation updates individual attributes of the item, creating the item if necessary. Since different event handlers update different attributes of the `Order` item, it makes sense to use `UpdateItem`. What's more this operation is more

efficient since there is no need to first retrieve the order from the table. One challenge with updating the database in response to events is, as I mentioned earlier, detecting and discarding duplicate events. Lets look at how to do that when using DynamoDB.

Detecting duplicate events

All of the `Order History Service's` event handlers are idempotent. Each one simply sets one or more attributes of the Order item. The `Order History Service` could, therefore, simply ignore the issue of duplicate events. The downside of ignoring the issue, however, is that `Order` item will sometimes be temporarily out of date. That is because an event handler that receives a duplicate event will set an Order item's attributes to previous values. The `Order` item won't have the correct values until later events are redelivered.

As described earlier, one way to prevent data from becoming out of date is to detect and discard duplicate events. The `OrderHistoryDaoDynamoDb` can detect duplicate events by recording in each item the events that have caused it to be updated. It can then use the `UpdateItem()` operation's conditional update mechanism to only update an item if an event is not a duplicate.

A conditional update is only performed if a *condition expression* is true. A *condition expression* tests whether an attribute exists or has a particular value. The `OrderHistoryDaoDynamoDb` DAO can track events received from each aggregate instance using an attribute whose name is `«aggregateType»«aggregateId»` and whose value is highest received event id. An event is a duplicate if the attribute exists and its value is less than or equal to the event id. The `OrderHistoryDaoDynamoDb` DAO uses this condition expression:

```
attribute_not_exists(«aggregateType»«aggregateId») OR «aggregateType»«aggregateId»
< :eventId
```

This *condition expression* only allows the update if the attribute does not exists or the `eventId` is greater than the last processed event id.

For example, let's suppose that an event handler receives an `DeliveryPickup` event whose id is `123323-343434` from a `Delivery` aggregate whose id is `3949384394-039434903`. The name of the tracking attribute is `Delivery3949384394-039434903`. There event should be considered a duplicate if the value of this attribute is greater than or equal to `123323-343434`. The `query()` operation invoked by the event handler updates the Order item using this condition expression:

```
 attribute_not_exists(Delivery3949384394-039434903) OR Delivery3949384394-039434903
< :eventId
```

Now that I have described the DynamoDB data model and query design let's take a look at the `OrderHistoryDaoDynamoDb`, which defines the methods that update and query the `ftgo-order-history` table.

7.4.3 The `OrderHistoryDaoDynamoDb` class

The `OrderHistoryDaoDynamoDb` class implements method that read and write items in the `ftgo-order-history` table. Its update methods are invoked by `OrderHistoryEventHandlers` and its query methods are invoked by `OrderHistoryQuery API` Let's take a look some example methods starting with the `addOrder()` method.

The `addOrder()` method

The `addOrder()` method adds an order to the `ftgo-order-history` table. It has two parameters, `order` and `sourceEvent`. The `order` parameter is the `Order` to add, which is obtained from the `OrderCreated`event. The `sourceEvent` parameter contains the eventId and the type and id of the aggregate that emitted the event. It is used to implement the conditional update. Listing 7.2 is the source code for this method.

Listing 7.2. The `OrderHistoryDaoDynamoDb` class implements a method for creating or updating an `Order` in the `ftgo-order-history` table

```
public class OrderHistoryDaoDynamoDb ...

@Override
public boolean addOrder(Order order, Optional<SourceEvent> eventSource) {
 UpdateItemSpec spec = new UpdateItemSpec()
         .withPrimaryKey("orderId", order.getOrderId())                  (1)
         .withUpdateExpression("SET orderStatus = :orderStatus, " +      (2)
                 "creationDate = :cd, consumerId = :consumerId, lineItems =" +
                 " :lineItems, keywords = :keywords, restaurantName = " +
                 ":restaurantName")
         .withValueMap(new Maps()                                        (3)
                 .add(":orderStatus", order.getStatus().toString())
                 .add(":cd", order.getCreationDate().getMillis())
                 .add(":consumerId", order.getConsumerId())
                 .add(":lineItems", mapLineItems(order.getLineItems()))
                 .add(":keywords", mapKeywords(order))
                 .add(":restaurantName", order.getRestaurantName())
                 .map())
         .withReturnValues(ReturnValue.NONE);
 return idempotentUpdate(spec, eventSource);
}
```

(1) the primary key of the Order item to update.
(2) the update expression that updates the attributes
(3) the values of the placeholders in the update expression.

The `addOrder()` method creates an `UpdateSpec`, which is part of the AWS SDK and describes the update operation. After creating the `UpdateSpec` calls `idempotentUpdate()`, which is a helper method that performs the update after adding a condition expression that guards against duplicate updates.

The `notePickedUp()` method

The `notePickedUp()` method, which is shown in listing 7.3, is called by the event handler for the `DeliveryPickedUp` event. It changes the `deliveryStatus` of the `Order` item to `PICKED_UP`.

Listing 7.3. The `OrderHistoryDaoDynamoDb` class implements a `notePickedUp()` method, which is called by an event handler method

```
public class OrderHistoryDaoDynamoDb ...

@Override
public void notePickedUp(String orderId, Optional<SourceEvent> eventSource) {
 UpdateItemSpec spec = new UpdateItemSpec()
         .withPrimaryKey("orderId", orderId)
         .withUpdateExpression("SET #deliveryStatus = :deliveryStatus")
         .withNameMap(Collections.singletonMap("#deliveryStatus",
                 DELIVERY_STATUS_FIELD))
         .withValueMap(Collections.singletonMap(":deliveryStatus",
                 DeliveryStatus.PICKED_UP.toString()))
         .withReturnValues(ReturnValue.NONE);
 idempotentUpdate(spec, eventSource);
}
```

This method is similar to `addOrder()`. It creates an `UpdateItemSpec` and invokes `idempotentUpdate()`. Let's look at this method.

The `idempotentUpdate()` method

Listing 7.4 shows the `idempotentUpdate()` method, which updates the item after possibly adding a condition expression to the `UpdateItemSpec` that guards against duplicate updates.

Listing 7.4. The `OrderHistoryDaoDynamoDb` class defines an `idempotentUpdate()` method, which ignores updates triggered by duplicate events

```
public class OrderHistoryDaoDynamoDb ...

private boolean idempotentUpdate(UpdateItemSpec spec, Optional<SourceEvent>
        eventSource) {
 try {
  table.updateItem(eventSource.map(es -> es.addDuplicateDetection(spec))
          .orElse(spec));
  return true;
 } catch (ConditionalCheckFailedException e) {
  // Do nothing
  return false;
 }
}
```

If the `sourceEvent` is supplied it invokes `SourceEvent.addDuplicateDetection()` to

modify the `UpdateItemSpec`. This method adds the condition expression that was described earlier. Now that we have looked at the code that updates the table, let's look at the query method.

The `findOrderHistory()` method

The `findOrderHistory()` method, which is shown in listing 7.5, retrieves the consumer's orders by querying the `ftgo-order-history` using the `ftgo-order-history-by-consumer-id-and-creation-time`. It has two parameters, `consumerId`, which specifies the consumer, and `filter`, which specifies the search criteria. This method creates `QuerySpec`, which like `UpdateSpec` is part of the AWS SDK, from its parameters, queries the index and transforms the returned items into an `OrderHistory` object.

Listing 7.5. The `OrderHistoryDaoDynamoDb` class implements a `findOrderHistory()` class, which retrieves a consumers orders that match the search criteria

```
public class OrderHistoryDaoDynamoDb ...

@Override
public OrderHistory findOrderHistory(String consumerId, OrderHistoryFilter
       filter) {

 QuerySpec spec = new QuerySpec()
         .withScanIndexForward(false)                                    ❶
         .withHashKey("consumerId", consumerId)
         .withRangeKeyCondition(new RangeKeyCondition("creationDate")    ❷
                               .gt(filter.getSince().getMillis()));

 filter.getStartKeyToken().ifPresent(token ->
       spec.withExclusiveStartKey(toStartingPrimaryKey(token)));

 Map<String, Object> valuesMap = new HashMap<>();

 String filterExpression = Expressions.and(                              ❸
         keywordFilterExpression(valuesMap, filter.getKeywords()),
         statusFilterExpression(valuesMap, filter.getStatus()));

 if (!valuesMap.isEmpty())
  spec.withValueMap(valuesMap);

 if (StringUtils.isNotBlank(filterExpression)) {
  spec.withFilterExpression(filterExpression);
 }

 System.out.print("filterExpression.toString()=" + filterExpression);

 filter.getPageSize().ifPresent(spec::withMaxResultSize);                ❹

 ItemCollection<QueryOutcome> result = index.query(spec);

 return new OrderHistory(
         StreamSupport.stream(result.spliterator(), false)
```

```
            .map(this::toOrder).collect(toList()),
          Optional.ofNullable(result
                .getLastLowLevelResult()
                .getQueryResult().getLastEvaluatedKey())
            .map(this::toStartKeyToken));
}
```

1. specifies that query must return the orders in order of increasing age.
2. the maximum age of the orders to return
3. construct a filter expression and placeholder value map from the `OrderHistoryFilter`.
4. limit the number of results if the caller has specified a page size

After building a `QuerySpec`, this method then executes a query and builds an `OrderHistory` from the returned items.

The `findOrderHistory()` method implements pagination by serializing the value returned by `getLastEvaluatedKey()` into a JSON token. If a client specifies a start token in the `OrderHistoryFilter` then `findOrderHistory()` serializes it and invokes `withExclusiveStartKey()` to set the start key

As you can see, you must address numerous issues when implementing a CQRS view including picking a database, designing the data model that efficiently implements updates and queries, handling concurrent updates and dealing with duplicate events. The only complex part of the code is the DAO since it must properly handle concurrency and ensure that updates are idempotent.

7.5 Summary

- Implementing queries that retrieve data from multiple services is challenging because each service's data is private
- There are two ways to implement these kinds of query: the API composition pattern and the Command Query Responsibility Segregation (CQRS) pattern
- The API composition pattern, which gathers data from multiple services, is the simplest way to implement queries and should you should use it whenever possible
- A limitation of the API composition pattern is that some complex queries require inefficient in-memory joins of large datasets.
- The CQRS pattern, which implements queries using view databases, is more powerful but is more complex to implement.
- A CQRS view module must handle concurrent updates as well as detect and discard duplicate events
- CQRS improves separation of concerns by enabling a service to implement a query that returns data owned by a different service
- Clients must handle the eventual consistency of CQRS views

8 External API patterns

This chapter covers:

- The challenge of designing APIs that support a diverse set of clients
- How to use the API gateway and Backends for front ends patterns
- How to design and implement an API gateway
- Using reactive programming to simplify API composition

Let's imagine that you have used the microservice architecture and structured your application's business logic as a collection of services. The sole reason for these services to exist, of course, is to be consumed by clients. It is very likely that a diverse set of clients, some inside the firewall and some outside, will consume your services. Examples of the kinds of clients that your application must support include server-side web applications, which are the application's presentation tier, iPhone and Android mobile applications, JavaScript applications running in a browser, IoT devices, and third-party applications.

An important design decision that you must make is what kind of API your application provides to these clients. In a monolith application, the API is simply the monolith's API. But in a microservices-based application, there is not just one API. Instead, each service has its own API.

The task of designing the API is made even more challenging because different clients require different data. Also, different clients access the services over different kinds of networks. The clients within the firewall use a high performance LAN and the clients outside of the firewall use the Internet or mobile network, which has lower performance. Consequently, as you will learn below, it often doesn't make sense to

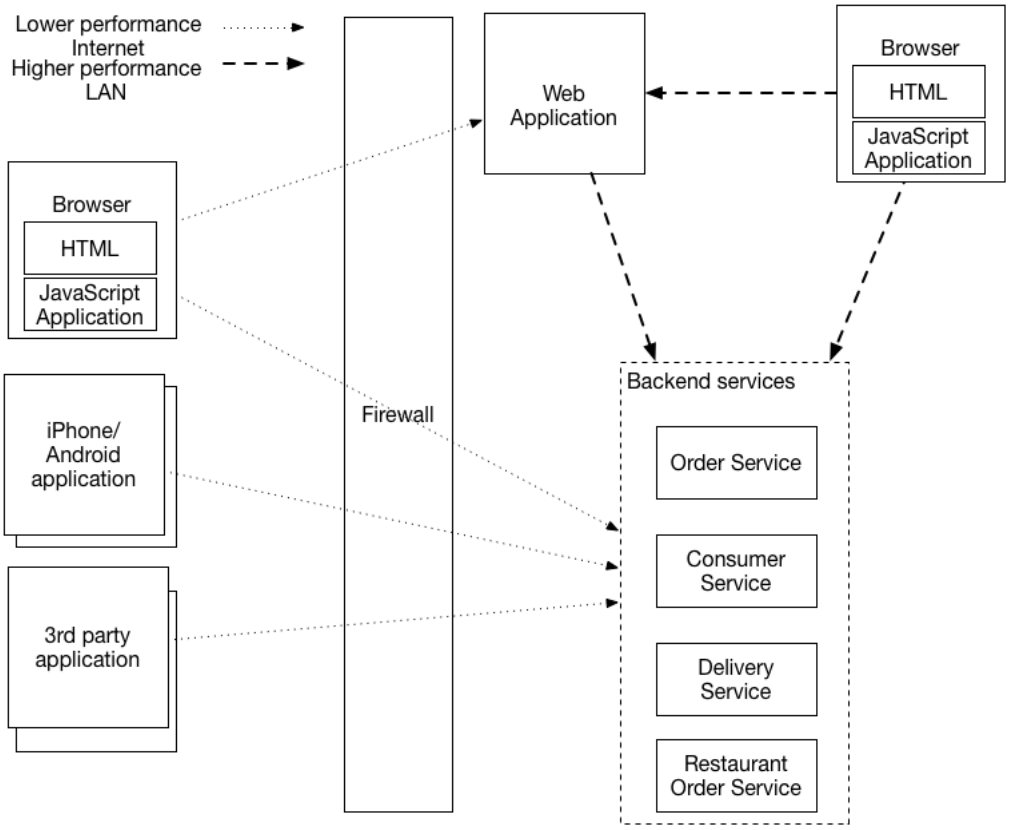
have a single, one-size fits all API.

This chapter begins by describing various external API design issues. I then describe the External API patterns: the API gateway pattern and the Backends for front ends pattern. After that I describe how to design and implement an API gateway.

8.1 External API design issues

In order to explore the various API-related issues, let's consider the FTGO application. As figure 8.1 shows, this application's services are consumed by a variety of clients.

Figure 8.1. The FTGO application's services and their clients. There are several different types of clients. Some are inside the firewall, others are outside. Those outside of the firewall access the services over the lower performance Internet/mobile network. Those clients inside the firewall use a higher performance LAN.



Four kinds of clients consume the services' API:

1. Web applications: the `Consumer web application`, which implements the browser-based UI for consumers, the `Restaurant web application`, which

implements the browser-based UI for restaurants, and the `Admin web application`, which implements the internal administrator UI.
2. JavaScript applications running in the browser.
3. Mobile applications, one for consumers and the other for couriers.
4. Applications written by third-party developers.

The web applications and the browser-side JavaScript for the administration application run inside the firewall and so access the services over a high bandwidth, low latency LAN. The other clients run outside of the firewall and so access the services over the lower bandwidth, higher latency Internet or mobile network.

One approach to API design is for clients to simply invoke the services directly. On the surface, this sounds quite straightforward. After all, that's how clients invoke the API of a monolithic application. However, this approach is rarely used in a microservice architecture because of the following drawbacks:

- The fine-grained service APIs require clients to make multiple requests to retrieve the data that they need, which is inefficient and can result in a poor user experience
- The lack of encapsulation caused by clients knowing about each service and its API makes it difficult to change the architecture and the APIs.
- Services might use IPC mechanisms that aren't convenient or practical for clients to use, especially those clients outside of the firewall.

To learn more about these drawbacks let's take a look at how the FTGO mobile application for consumers retrieves data from the services.

8.1.1 API design issues for the FTGO mobile client

Consumers use the FTGO mobile client to place and manage theirs orders. Let's imagine that you are developing the mobile client's `View Order` view, which displays an order. As I described in chapter 7, the information displayed by this view includes basic order information including its status; its payment status; the status of the order from the restaurant's perspective; and the delivery status including its location and estimated delivery time if in transit.

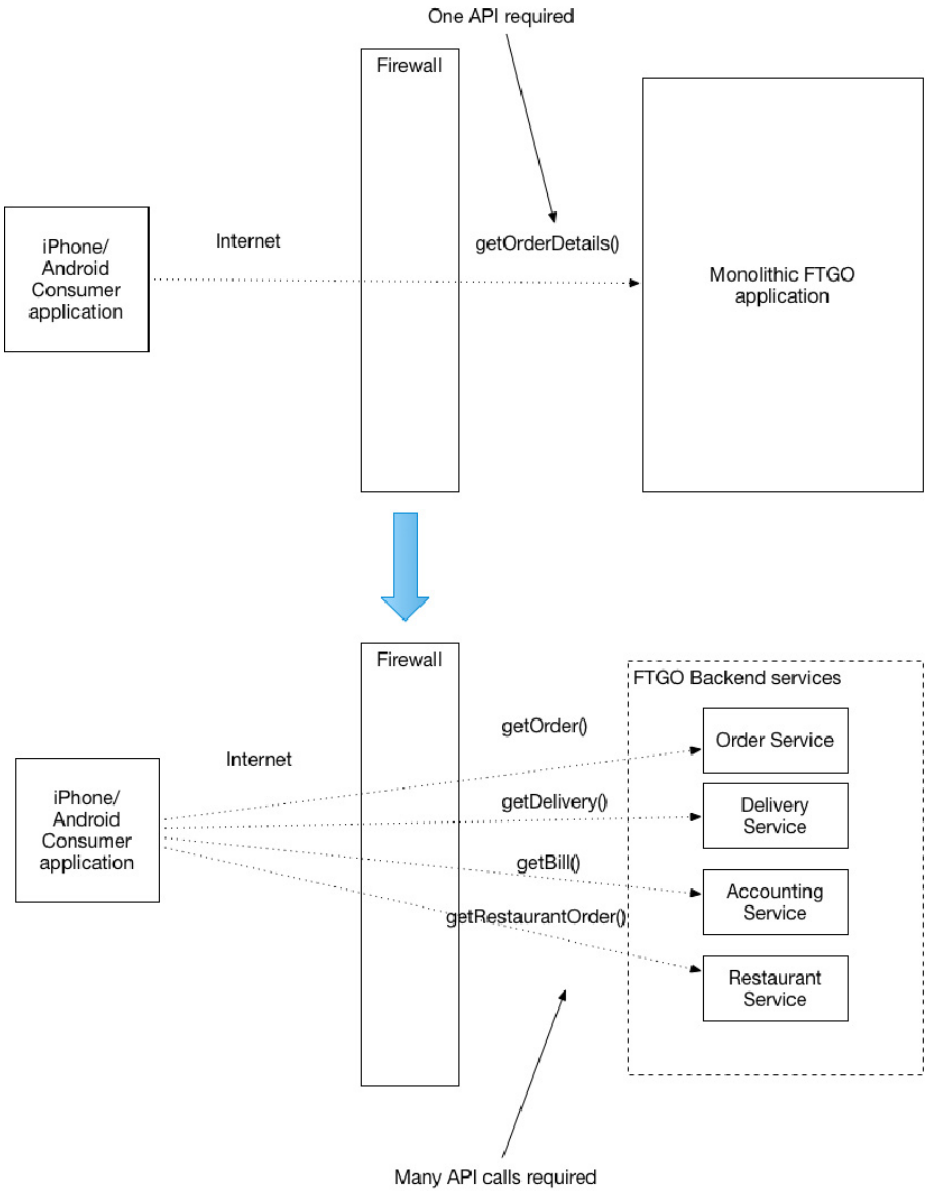
The monolithic version of the FTGO application has an API endpoint that returns the order details. The mobile client retrieves the information needs by making a single request. In contrast, in the microservices version of FTGO application the order details are, as described previously, scattered across several services including:

- `Order Service` - basic order information including the details and status
- `Restaurant Order Service` - the status of the order from the restaurant's perspective and the estimated time it will be ready for pickup
- `Delivery Service` - the order's delivery status, its estimated delivery time, and its current location
- `Accounting Service` - the order's payment status

If the mobile client invokes the services directly, then it must, as figure 8.2 shows,

make multiple calls to retrieve this data.

Figure 8.2. A client can retrieve the order details from the monolithic FTGO application with a single request. But the client must make multiple requests to retrieve the same information in a microservices architecture



In this design, the mobile application is playing the role of *API Composer*. It invokes

multiple services and combines the results. While this approach seems reasonable, it has several serious problems.

Poor user experience due to the client making multiple requests

The first problem is that the mobile application must sometimes make multiple requests to retrieve the data it wants to display to the user. The chatty interaction between the application and the services can make the application seem unresponsive, especially when it uses the Internet or a mobile network. The Internet has much lower bandwidth and higher latency than a LAN. Mobile networks are even worse. The latency of a mobile network (and Internet) is typically 100x greater than a LAN.

The higher latency might not be a problem when retrieving the order details since the mobile application minimize the delay by executing the requests concurrently. The overall response time is no greater than that of a single request. But in other scenarios, a client may need to execute requests sequentially, which will result in a poor user experience.

What's more, poor user experience due to network latency is not the only issue with a chatty API. It requires the mobile developer to write potentially complex API composition code. This work is a distraction from their primary task of creating a great user experience. Also, because each network request consumes power, a chatty API reduces the battery life of the mobile device.

Lack of encapsulation requires front end developers to change their code in lock step with the backend

Another drawback of a mobile application directly accessing the services is the lack of encapsulation. As an application evolves, the developers of a service sometime change an API in a way that breaks existing clients. They might even change how the system is decomposed into services. Developers may add new services and split or merge existing services. If, however, knowledge about the services is baked into a mobile application it can be difficult to change the services' APIs.

Unlike when updating a server-side application, it takes hours or perhaps even days to roll out a new version of a mobile application. Apple or Google must approve the upgrade and make it available for download. Users might not download the upgrade immediately, if ever. Also, you might not want to force reluctant users to upgrade. The strategy of exposing service APIs to a mobile creates a significant obstacle to evolving those APIs.

Services might use client-unfriendly IPC mechanisms

Another challenge with a mobile application directly calling services is that some services might use protocols that are not easily consumed by a client. Client applications that run outside of the firewall typically use protocols such as HTTP and WebSockets. However, as I described in chapter 3, service developers have many protocols to choose from, not just HTTP. Some of application's services might use gRPC while others might use the AMQP messaging protocol. These kinds of protocols

work well internally but might not be easily consumed by a mobile client. Some of them are not even firewall friendly.

8.1.2 API design issues for other kinds of clients

I picked the mobile client because it is a great way to demonstrate the drawbacks of clients accessing services directly. However, the problems created by exposing services to clients are not specific to just mobile clients. Other kinds of clients, especially those outside of the firewall, also encounter these problems. As I described earlier, the FTGO application's services are consumed by web applications, browser-based JavaScript applications and third-party applications. Lets take a look at the API design issues with these clients.

API design issues for web applications

Traditional, server-side web applications, which handle HTTP requests from browsers and return HTML pages, run within the firewall and access the services over a LAN. Network bandwidth and latency are not obstacles to implementing API composition in a web application. Also, web applications can use non-web friendly protocols to access the services. The teams that develop web applications are part of the same organization often work in close collaboration with the teams writing the backend services and so a web application can easily be updated whenever the backend services are changed. Consequently, its feasible for a web application to access the backend services directly.

API design issues for browser-based JavaScript applications

Modern browser applications use some amount of JavaScript. Even if the HTML is primarily generated by a server-side web application, it's common for JavaScript running the browser to invoke services. For example, all of the FTGO application web applications - the `Consumer`, `Restaurant` and FTGO Admin browsers-based - contain JavaScript that invokes the backend services. The `Consumer` web application dynamically refreshes the `Order Details` page using JavaScript that invokes the service APIs.

On the one hand, browser-based JavaScript applications are easy to update when service APIs change. But on the other hand, JavaScript applications that access the services over the Internet have the same problems with network latency as mobile applications. And to make matters worse, browser-based UIs, especially those for the desktop, are usually more sophisticated and so will need to compose more services than the mobile applications. It's likely that the `Consumer` and `Restaurant` applications, which access the services over the Internet, will not be able composing service APIs efficiently.

Designing APIs for third-party applications

FTGO, like many other organizations, exposes an API to third-party developers. They can use the FTGO API to write applications that place and manage orders. These third-party applications access the APIs over the Internet and so API composition is likely to be inefficient. However, the inefficiency of API composition is a relatively minor

problem compared to the much larger challenge of designing an API that is used by third-party applications. That's because third-party developers need an API that is stable.

Very few organizations can simply force third-party developers to upgrade to a new API. Organizations that have an unstable API risk losing developers to a competitor. Consequently, you must carefully manage the evolution of an API that is used by third-party developers. You typically have to maintain older versions for a long time, possibly forever.

This requirement is a huge burden for an organization. It is not practical to make the developers of the backend services responsible for maintaining long term backwards compatibility. Rather than exposing services directly to third-party developers, organizations should have a separate public API that is developed by a separate team. As you will learn below, the public API is implemented by an architectural component known as an API gateway. Let's look at how an API gateway works.

8.2 The API gateway pattern

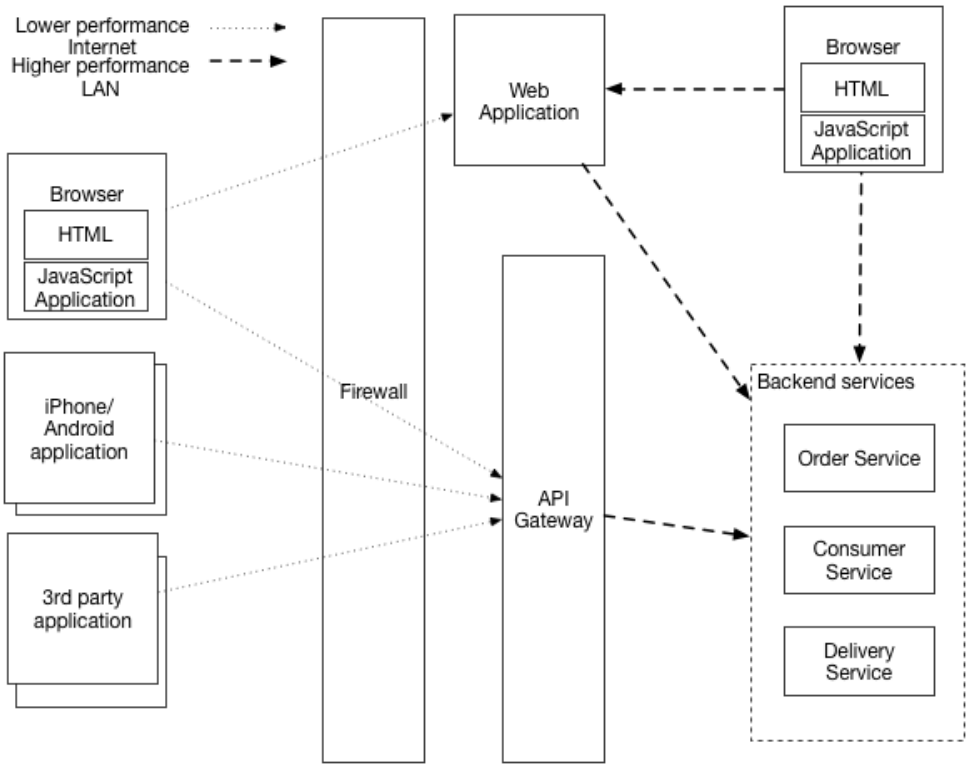
As you have just seen, there are numerous drawbacks with services accessing services directly. It is often not practical for a client to perform API composition over the Internet. The lack of encapsulation makes it difficult for developers to change service decomposition and APIs. Services sometimes use communication protocols that are not suitable outside of the firewall. Consequently, a much better approach is to use what is known as an API gateway.

An API gateway is a service that is the entry point into the application from the outside world. It is responsible for request routing, API composition, and other functions, such as authentication. In this section, I describe the API gateway pattern. I discuss its benefits and drawbacks. I also describe various design issues that you must address when developing an API gateway. Lets take a look at the API gateway pattern.

8.2.1 Overview of the API gateway pattern

Earlier in this chapter in section XYZ, I described the drawbacks of clients, such as the FTGO mobile application, making multiple requests in order to display information to user. A much better approach is for a client to make a single request to what is known as an API gateway. An API gateway is a service that is the single entry point into an application from outside the firewall. It is similar to the Facade pattern from object-oriented design. Like a facade, an API gateway encapsulates the application's internal architecture and provides an API to clients. It might also have other responsibilities such authentication, monitoring, and rate limiting. Figure 8.3 shows the relationship between the clients, the API gateway and the services.

Figure 8.3. The API gateway is the single entry point into the application from outside the firewall



The API gateway is responsible for request routing, API composition and protocol translation. All requests from *external* clients first go to the API gateway. It routes some requests to the appropriate service. The API gateway handles other requests using the API Composition pattern and invoking multiple services and aggregating the results. It might also translate between client-friendly protocols such as HTTP and WebSockets and client-unfriendly protocols that are used by the services.

Request routing

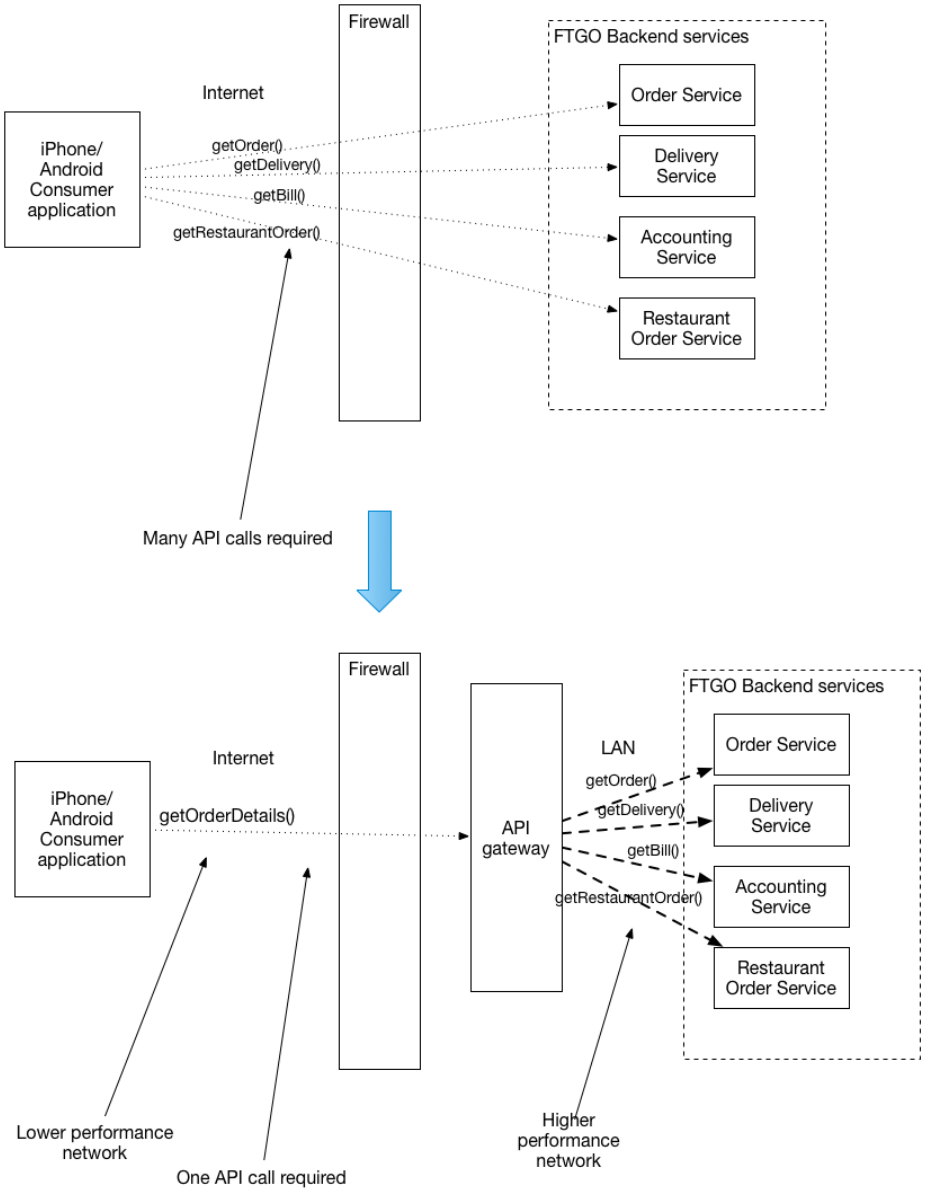
One of the key functions of an API gateway is request routing. An API gateway implements some API operations by simply routing requests to the corresponding service. When it receives a request, the API gateway consults a routing map that specifies which service to route the request to. A routing map might, for example, map an HTTP method and path to the HTTP URL of a service. This function is identical to the reverse proxying features provided by web servers such as NGINX.

API Composition

An API gateway typically does more than simply reverse proxying. It might also implement some API operations using API composition. The FTGO API gateway, for

example, implements the `Get Order Details` API operation using API composition. As figure 8.4 shows, the mobile application makes one request to the API gateway, which fetches the order details from multiple services.

Figure 8.4. An API gateway often does API Composition, which enables a client such as mobile device to efficiently retrieve data using a single API request.



The FTGO API gateway provides a coarse-grained API that enables mobile clients to retrieve the data they need with a single request. For example, the mobile client simply makes a single `getOrderDetails()` request to the API gateway.

Protocol translation

An API gateway might also perform protocol translation. The API gateway might provide a RESTful API to external clients even though the application services use a mixture of protocols internally include REST and gRPC. When needed, the implementation of some API operations translate between the RESTful external API and the internal gRPC-based APIs.

The API gateway provides each client with client-specific API

An API gateway could provide a single one-size-fits-all (OSFA) API. The problem with a single API is that different clients often have different requirements. For instance, a third-party application might require the `Get Order Details` API operation to return the complete `Order` details while a mobile client only needs a subset of the data. One way to solve this problem is to give clients the option of specifying in a request which fields and related objects the server should return. This approach is adequate for a public API that must serve a broad range of third-party application. However, it often doesn't give clients the control that they need.

A better approach is for the API gateway to provide each client with its own API. For example, the FTGO API gateway can implement provide the FTGO mobile client with an API that is specifically designed to meet its requirements. It might even have different APIs for the Android and iPhone mobile applications. The API gateway will also implement a public API for third-party developers to use. Later on, I'll describe the Backends for front ends pattern that takes this concept of an API-per-client even further by defining a separate API gateway for each client.

Implementing edge functions

Although an API gateway's primary responsibilities are API routing and composition it may also implement what are known as edge functions. An edge function is, as the name suggests, a request processing function that is implemented at the edge of an application. Examples of edge functions that an application might implement include:

- authentication - verifying the identity of the client making the request
- authorization - verifying that the client is authorized to perform that particular operation
- rate limiting - limiting how many requests per second from either a specific client and/or from all clients
- caching - cache responses to reduce the number of requests made to the services
- metrics collection - collect metrics on API usage for billing analytics purposes
- request logging - log requests

There are three different places in your application that you could implement these

edge functions. The first option is to implement these edge functions in the backend services. This might make sense for some functions such as caching, metrics collection and possibly authorization. However, it is generally more secure if the application authenticates requests on the edge before they reach the services.

The second option is implement these edge functions in an edge service that is upstream from the API gateway. The edge service is the first point of contact for an external client. It authenticates the request and performs other edge processing before passing it to the API gateway.

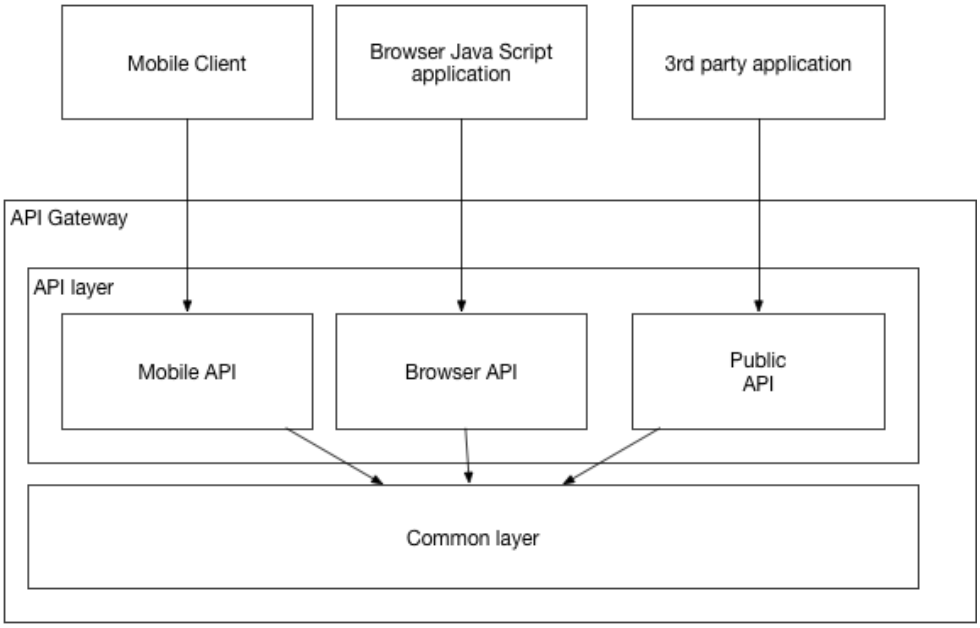
An important benefit of using a dedicated edge service is that it separates concerns. The API gateway focusses on API routing and composition Another benefit is that it centralizes responsibility for critical edge functions such as authentication. This particularly valuable when, as I describe below, an application has multiple API gateways that are possibly written using a variety of languages and frameworks. The drawback of this approach is that increases network latency because of the extra hop. It also adds to the complexity of the application.

As a result, its often convenient to use the third option and implement these edge functions, especially authorization, in the API gateway itself. There is one less network hop, which improves latency. There are also fewer moving parts, which reduces complexity.

API gateway architecture

An API gateway has a layered, modular architecture. Its architecture, which is shown in figure 8.5, consists of two layers, the API layer and a common layer. The API layer consists of one or more independent API modules. Each API module implements an API for a particular client. The common layer implements shared functionality including edge functions such as authentication.

Figure 8.5. An API gateway has a layered modular architecture. The API for each client is implemented by a separate module. The common layer implements functionality that is common to all APIs such as authentication



In this example, the API gateway has three API modules:

- Mobile API, which implements the API for the FTGO mobile client
- Browser API, which implements the API to the JavaScript application running in the browser
- Public API, which implements the API for third-party developers.

An API module implements each API operation in one of two ways. Some API operations map directly to single service API operation. An API module implements these operation by routing requests to the corresponding service API operation. It might implement these API operations using a generic routing module that reads a configuration file describing the routing rules.

An API module implements other, more complex API operations using API composition. The implementation of this API operation consist of custom code. Each API operation implementation handles requests by invoking multiple services and combining the results.

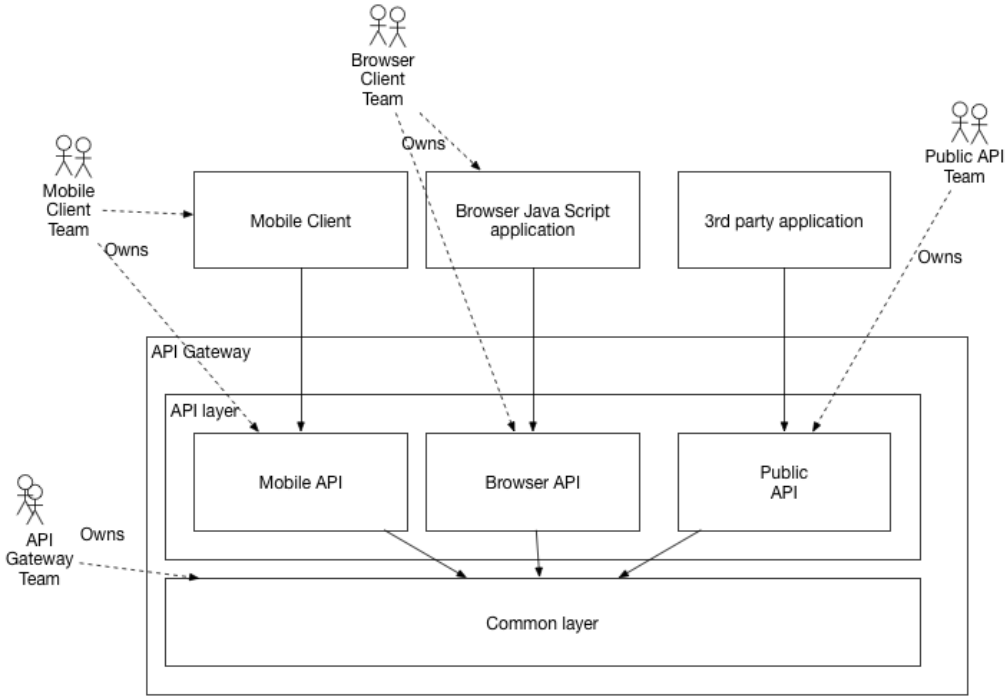
API gateway ownership model

An important question that you must answer is who is responsible for the development of the API gateway and its operation? There are a few different options. One option is for a separate team to be responsible for the API gateway. The drawback of this option

is that it is similar to SOA, where an Enterprise Service Bus (ESB) team was responsible for all ESB development. If a developer working on the mobile application needs access to a particular service they must submit a request to the API gateway team and wait for them to expose the API. This kind of centralized bottleneck in the organization is very much counter to the philosophy of the microservice architecture, which promotes loosely coupled autonomous teams.

A better approach, which has been promoted by Netflix, is for the client teams - the mobile, web and public API teams - to own the API module that exposes their API. An API gateway team is responsible for developing the `Common` module and for the operational aspects of the gateway This ownership model, which is shown in figure 8.6, gives the teams control over their APIs.

Figure 8.6. A client team owns their API module. As they change the client, they can change the API module and not ask the API group to make the changes.



When a team needs to change their API, they simply check-in the changes to the source repository for the API gateway. Of course, in order to work well the API gateway's deployment pipeline must be fully automated. Otherwise, the client teams will often be blocked waiting for the API gateway team to deploy the new version.

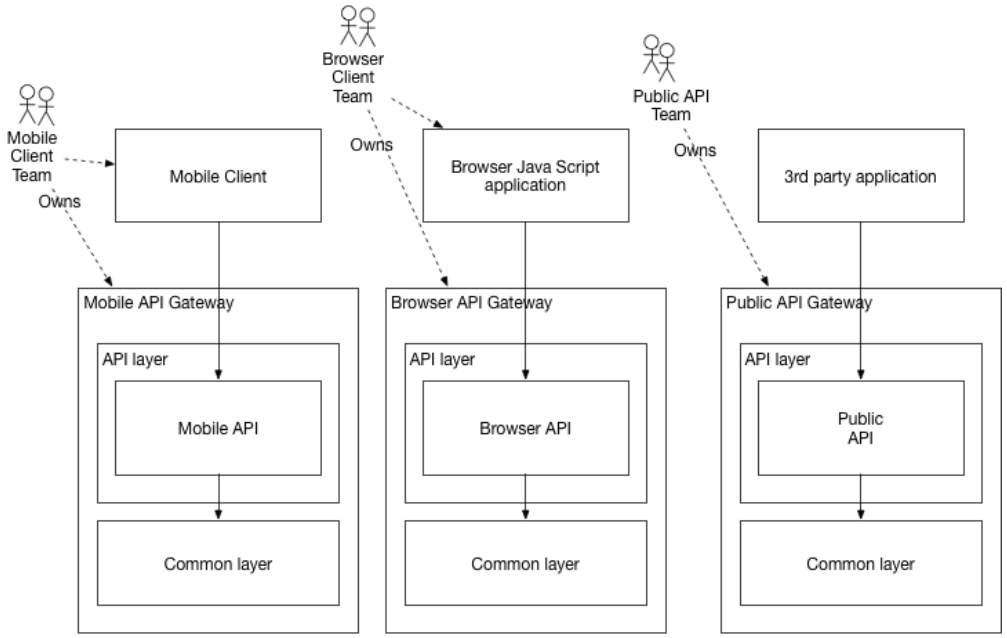
Using the Backends for front ends pattern

One concern with an API gateway is that responsibility for it is blurred. Multiple teams

contribute to the same code base. An API gateway team is responsible for its operation. While not as bad as, as a SOA ESB, this blurring of responsibilities is counter to the microservice architecture philosophy of "if you build, it you own it".

The solution is to have an API gateway for each client, the so-called Backend for Frontend (BFF) pattern. As figure 8.7 shows, each API module becomes its own standalone API gateway that is developed and operated by a single client team.

Figure 8.7. The Backend for front end pattern defines a separate API gateway for each client. Each client team owns their API gateway. An API gateway team owns the common layer.



The Public API team own and operate their API gateway, the mobile team own and operate theirs, and so on. In theory, different API gateways could be developed using different technology stacks. However, this risks duplicating code for common functionality such as the code that implements edge functions. Ideally, all API gateways all use the same technology stack. The common functionality is a shared library implemented by the API gateway team.

As well as clearly defining responsibilities, the BFF pattern has other benefits. The API modules are isolated from one another, which improves reliability. One misbehaving API cannot readily impact other APIs. It also improves observability since different API modules are different processes. Another benefit of the BFF pattern is that each API is independently scalable. The BFF pattern also reduces startup time since each API gateway is a smaller, simpler application.

8.2.2 Benefits and drawbacks of an API gateway

As you might expect, the API gateway pattern has both benefits and drawbacks.

Benefits

A major benefit of using an API gateway is that it encapsulates internal structure of the application. Rather than having to invoke specific services, clients simply talk to the gateway. The API gateway provides each client with a client-specific API. This reduces the number of round trips between the client and application. It also simplifies the client code.

Drawbacks

The API gateway pattern also has some drawbacks. It is yet another highly available component that must be developed, deployed and managed. There is also a risk that the API gateway becomes a development bottleneck. Developers must update the API gateway in order to expose their services's API. It is important that the process for updating the API gateway is as lightweight as possible. Otherwise, developers will be forced to wait in line in order to update the gateway. Despite these drawbacks, however, for most real world applications it makes sense to use an API gateway. If necessary, you can use the Backends for front ends patterns to enable the teams to develop and deploy their APIs independently.

8.2.3 Netflix as an example of an API gateway

A great example of an API gateway is the Netflix API. The Netflix streaming service is available on hundreds of different kinds of devices including televisions, blueray players, smart phones, etc. Initially, Netflix attempted to have a one-size-fits-all [1] style API for their streaming service. However, they soon discovered that it didn't work well because of the diverse range of devices and their different needs. Today, they use an API gateway that implements a separate API for each device. The client device team develops and owns the API implementation.

In the first version of the API gateway, each client team implemented their API using Groovy scripts that perform routing and API composition. Each script invoked one or more service APIs using Java client libraries provided by the service teams. On the one hand, this works well and client developers have written over thousands of scripts. The Netflix API gateway handles billions of requests per day and on average each API calls fans out to 6-7 backend services. But on the other hand, Netflix has found this monolithic architecture to be somewhat cumbersome.

As a result, Netflix is now moving to an API gateway architecture that is similar to the Backends for front ends pattern. In this new architecture, client teams write API modules using NodeJS. Each API module runs its own Docker container. The scripts don't, however, invoke the services directly. Instead, they invoke a second "API gateway", which exposes the service APIs using Netflix Falcor. Netflix Falcor is an

[1] http://www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15

API technology, which does declarative, dynamic API composition, and enables a client to invoke multiple services using a single request. This new architecture has a number of benefits. The API modules are isolated from one another, which improves reliability and observability Also, client API module is independently scalable.

8.2.4 API gateway design issues

Now that we looked at the API gateway pattern and its benefits and drawbacks let's now look at various API gateway design issues. There are several issues to consider when designing an API gateway:

- Performance and scalability
- Writing maintainable code by using reactive programming abstractions
- Handling partial failure
- Being a good citizen in the application's architecture

Lets look at each one.

Performance and scalability

An API gateway is the application's front door. All external requests must first pass through the gateway. While most companies don't operate at the scale of Netflix, which handles billions of requests per day, the performance and scalability of the API gateway is usually very important. A key design decision that affects performance and scalability is whether the API gateway should use synchronous or asynchronous I/O.

In the synchronous I/O model, each network connection is handled by a dedicated thread. This is a simpler programming model and works reasonably well. For example, it is the basis of the widely used Java EE servlet framework, although this framework provides the option of completing a request asynchronously. One limitation of synchronous I/O, however, is that operating system threads are heavyweight and so there is a limit on the number of threads and hence concurrent connections that an API gateway can have.

The other approach is to use the asynchronous (a.k.a. non-blocking) I/O model. In this model, a single event loop thread dispatches I/O requests to event handlers. There are a variety of asynchronous I/O technologies to choose from. On the JVM you can use one of the NIO-based frameworks such Netty, Vertx, Spring Reactor, or JBoss Undertow. One popular non-JVM option is NodeJS, which is a platform built on Chrome's JavaScript engine.

Non-blocking I/O is much more scalable since there isn't the overhead of using multiple threads. The drawback, however, is that the asynchronous, callback-based programming model is much complex. The code is more difficult to write, understand and debug. Event handlers must return quickly to avoid blocking the event loop thread.

Also, whether using non-blocking I/O has a meaningful overall benefit depends on the characteristics of the API gateway's request processing logic. Netflix had mixed results

[2] when they rewrote Zuul, which is their edge server, to use NIO. On the one hand, as you would expect, using NIO reduced the cost of each network connection since there is no longer a dedicated thread for each one. Also, a Zuul cluster that ran I/O intensive logic - i.e. request routing - had a 25% increase in throughput and a 25% reduction in CPU utilization. But on the other hand, a Zuul cluster that ran CPU intensive logic - e.g. decryption and compression - showed no improvement.

Use reactive programming abstractions

As mentioned earlier, API composition consists of invoking multiple backend services. Some backend service requests depend entirely on the client request's parameters. Others, however, might depend on the results of other service requests. One approach is for an API endpoint handler method to simply call the services in the order determined by the dependencies. For example, listing 8.1 shows the handler for `findOrder()` request that is written this way. It calls each of the four services, one after the other.

Listing 8.1. Fetching the order details by calling the backend services sequentially

```
@RestController
public class OrderDetailsController {
@RequestMapping("/order/{orderId}")
public OrderDetails getOrderDetails(@PathVariable String orderId) {

  OrderInfo orderInfo = orderService.findOrderById(orderId);

  RestaurantOrderInfo restaurantOrderInfo = restaurantOrderService
          .findRestaurantOrderByOrderId(orderId);

  DeliveryInfo deliveryInfo = deliveryService
          .findDeliveryByOrderId(orderId);

  BillInfo billInfo = accountingService
          .findBillByOrderId(orderId);

  OrderDetails orderDetails =
       OrderDetails.makeOrderDetails(orderInfo, restaurantOrderInfo,
                                     deliveryInfo, billInfo);

  return orderDetails;
}
...
```

The drawback of calling the services sequentially is that the response time is the sum of the service response times. In order to minimize response time, the composition logic should whenever possible invoke services concurrently. In this example there are no dependencies between the service calls. All services should be invoked concurrently, which significantly reduces response time. The challenge is to write concurrent code that is maintainable.

[2] https://medium.com/netflix-techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c

That is because the traditional way to write scalable, concurrent code is to use callbacks. Asynchronous, event-driven I/O is inherently callback-based. Even a Servlet API-based API composer that invokes services concurrently typically uses callbacks. It could execute requests concurrently by calling `ExecutorService.submitCallable()`. The problem, however, is that this method returns a `Future`, which has a blocking API. A more scalable approach is for an API Composer to call `ExecutorService.submit(Runnable)` and for each `Runnable` to invoke a callback with the outcome of the request. The callback accumulates results and once all of them have been received it sends back the response to the client.

Writing API composition code using the traditional asynchronous callback approach quickly leads you to callback hell. The code will be tangled, difficult to understand and error-prone especially when composition requires a mixture of parallel and sequential requests. A much better approach is to write API composition code in a declarative style using a reactive approach. Examples of reactive abstractions for the JVM include:

- Java 8 `CompletableFutures`
- Project Reactor `Monos`
- RxJava (Reactive Extensions for Java) `Observables`, which was created by Netflix specifically to solve this problem in their API gateway
- Scala `Futures`.

A NodeJS-based API gateway would use JavaScript promises or RxJS, which is reactive extensions for JavaScript. Using one of these reactive abstractions will enable you to write concurrent code that is simple and easy to understand. Later in this chapter, I show an example of this style of coding using Project Reactor `Monos` and version 5 of the Spring Framework.

Handle partial failures

As well as being scalable, an API gateway must also be reliable. One way to achieve reliability is to run multiple instances of the gateway behind a load balancer. If one instance fails, the load balancer will route requests to the other instances.

Another way to ensure that an API gateway is reliable is to properly handle failed requests and requests that have unacceptably high latency. When an API gateway invokes a service there is always a chance that the service is slow or unavailable. An API gateway may wait a very long time, perhaps indefinitely, for a response, which consumes resources and prevents it from sending a response to its client. An outstanding request to a failed service might even consume a limited, precious resource such a thread and ultimately result in the API gateway being unable to handle any other requests. The solution, is as I described in chapter 3, for an API gateway use the Circuit Breaker pattern when invoking services.

Being a good citizen in the architecture

Later in chapter 10, which describes how to deploy services, I'll cover patterns for service discovery and observability. The service discovery patterns enable a service

client, such as an API gateway, to determine the network location of a service instance so that it can invoke it. The observability patterns enable a developers to monitor the behavior of an application and troubleshoot problems. An API gateway like other services in the architecture must implement the patterns that have been selected for the architecture.

8.3 Implementing an API gateway

Let's now look at how to implement an API gateway. As I described earlier, the responsibilities of an API gateway are:

- Request routing: mapping `(method, resource)` to a service
- API composition: mapping `(method, resource)` to a request handler, which combines the results of invoking multiple services
- Edge functions, most notably authentication.
- Protocol translation: translating between client-friendly protocols and the client-unfriendly protocols used by services
- Being a good citizen in the application's architecture

There are a couple of different ways to implement an API gateway:

- Using an off-the-shelf (OTS) API gateway product/service - this option requires the little or no development but is the least powerful. For example, an OTS API gateway typically does not support API composition
- Developing your own API gateway using either an API gateway framework or a web framework as the starting point - this is most powerful approach. However, it requires some development

Lets look at these options starting with using an off-the-shelf API gateway product or service.

8.3.1 Using an off-the-shelf API gateway product/service

Several off-the-self services and product implement API gateway features. Let's first look at a couple of services that are provided by AWS. After that I will discuss some products that you download, configure and run yourself.

AWS API gateway

The AWS API gateway, which is one of the many services provided by Amazon Web Services, is a service for deploying and managing APIs. An AWS API gateway API is a set of REST resources, each of which supports one or more HTTP methods. You configure the API gateway to route each `(Method, Resource)` to a backend service A backend service is either an AWS Lambda Function, application-defined HTTP service, or an AWS Service If necessary, you can configure the API gateway to transform request and response using a template-based mechanism. The AWS API gateway can also authenticate requests.

The AWS API gateway fulfills some of the requirements for an API gateway that I

described earlier. It implements routing and authentication. The API gateway is provided by AWS so you are not responsible for installation and operations. You simply configure the API gateway and AWS handles everything else including scaling.

Unfortunately, the AWS API gateway has several drawbacks and limitations, which cause it to not fulfill other requirements. It does not provide a way to implement API composition. The AWS API gateway only supports HTTP(S) with a heavy emphasis on JSON. It can only route requests to HTTPS endpoints that are publicly accessible. As a result, the AWS API gateway is best suited to routing to services that are deployed as AWS Lambda Functions, which I describe later in chapter 10.

AWS Application Load Balancer

Another AWS service that provides API gateway-like functionality is the AWS Application Load Balancer[3], which is a load balancer for HTTP, HTTPS, WebSocket and HTTP/2 When configuring an Application Load Balancers, you define routing rules that route requests to backend services, which must be running on AWS EC2 instances.

Like the AWS API gateway, the AWS Application Load Balancer meets some of the requirements for an API gateway. It implements basic routing functionality. It is hosted so you are not responsible for installation or operations. Unfortunately, it is quite limited. It does not implement HTTP method based routing. Nor does it implement API composition or authentication. As a result, the AWS Application Load Balancer does not meet the requirements for an API gateway.

Using an API gateway product

Another option is to use an API gateway product such as Kong or Traefik. These are open source packages that you install and operate yourself. Kong is based on the NGINX HTTP server and Traefik is written in GoLang. Both products let you configure flexible routing rules that use the HTTP method, headers and path to select the backend service. Kong let's you configure plugins that implement edge functions such as authentication. Traefik can even integrate with some service registries, which I described in chapter 3.

Although these products offer powerful routing capabilities they have some drawbacks. You must install, configure and operate them yourself. They don't support API composition. If you want the API gateway to perform API composition you must develop you own API gateway.

8.3.2 Developing your own API gateway

a step-by-step description of how to develop an API Gateway. Along the way, you can teach about each class, what it does, and how to use it.

Developing an API gateway is not particularly difficult. It is basically a web application that proxies requests to other services. You can build one using your

[3] https://aws.amazon.com/blogs/aws/new-aws-application-load-balancer/

favorite web framework. There are, however, two key design problems that you'll need to solve:

1. Implement a mechanism for defining routing rules in order to minimize the complex coding.
2. Correctly, implementing the HTTP proxying behavior including how HTTP headers handled.

Consequently, a better starting point for developing an API gateway is to use a framework that is designed for that purpose. Its built-in functionality significantly reduces the amount of code that you need to write. Lets take a look at Netflix Zuul, which is an open source project by Netflix, and after that I'll describe the Spring Cloud Gateway, which is a open source project from Pivotal.

Using Netflix Zuul

Netflix developed the Zuul[4] framework to implement edge functions such as routing, rate limiting and authentication. The Zuul framework has the concept of filters, which are reusable request interceptors that are similar to Servlet Filters or NodeJS Express middleware. Zuul handles an HTTP request by assembling a chain of applicable filters that then transform the request, invoke backend services, and transform the response before its sent back to the client. Although you can use Zuul directly, it is far easier to use Spring Cloud Zuul, which is an an open-source project from Pivotal. Spring Cloud Zuul builds on Zuul and through convention-over-configuration makes it remarkably easy to develop a Zuul-based server,

Spring Cloud Zuul is potentially a great foundation for an API gateway. Zuul handles the routing and edge functionality. You can extend Zuul by defining Spring MVC controllers that implement API composition. However, a major limitation of Zuul is that it can only implements path-based routing. It is incapable of, for example, routing `GET /orders` to one service and `POST /orders` to a different service. Consequently, Zuul does not support the query architecture that I described in 7.

About Spring Cloud Gateway

None of the options I've described so far meet all of the requirements. In fact I had given up in my search for an API gateway framework and had started developing an API gateway based on Spring MVC. But then I discovered the Spring Cloud Gateway project, which is an API gateway framework that is built on top of several frameworks including:

- Spring framework 5 - the latest version of the Spring Framework
- Spring Boot 2 - the latest version of Spring Boot designed to work with Spring framework 5
- Spring WebFlux, which is reactive web framework that is part of Spring framework 5, and is built on Project Reactor
- Project Reactor, which is a NIO-based reactive framework for the JVM, and

[4] https://github.com/Netflix/zuul

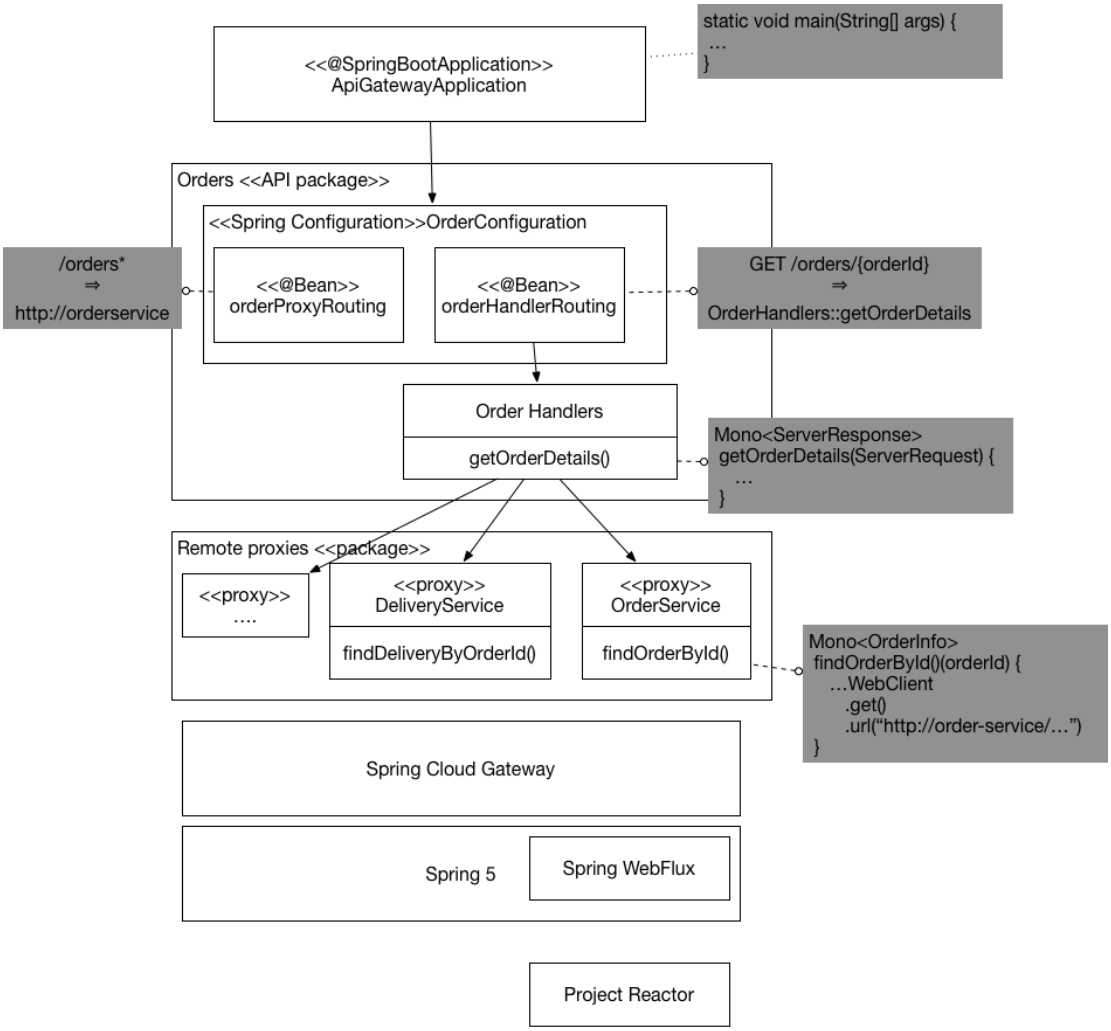
provides the `Mono` abstraction that I use below

Spring Cloud Gateway provides a simple yet comprehensive way to

1. route requests to backend services
2. implement requests handlers that perform API composition
3. handle edge functions such as authentication.

Figure 8.8 shows the key parts of an API gateway that is built using this framework.

Figure 8.8. The architecture of an API gateway built using Spring Cloud Gateway



The API gateway consists of the following packages:

- `ApiGatewayMain` package - defines the Main program for the API gateway.

- One or more API packages - an API package implements a set of API endpoints. For example, the `Orders` package implements the `Order` related API endpoints.
- Proxy package - consists of proxy classes that are used by the API packages to invoke the services

The `OrderConfiguration` class defines the Spring beans responsible routing Order-related requests. A routing rule can match against some combination of the HTTP method, the headers and the path. The `orderProxyRoutes` `@Bean` defines rules that map API operations to backend service URLs. For example, it routes paths beginning with `/orders` to the `Order Service`.

The `orderHandlers` `@Bean` defines rules that override those defined by `orderProxyRoutes`. These rules map API operations to handler methods, which are the Spring WebFlux equivalent of Spring MVC controller methods. For example, `orderHandlers` maps the operation `GET /orders/{orderId}` to the `OrderHandlers::getOrderDetails()` method.

The `OrderHandlers` class implements various request handler methods, such as `OrderHandlers::getOrderDetails()`. This method uses API composition to fetch the order details, which I described earlier. The handle methods invoke backend services using remote proxy classes, such as `OrderService`, This class defines methods for invoking the `OrderService`. Let's take a look at the code starting with the `OrderConfiguration` class.

The `OrderConfiguration` class

The `OrderConfiguration` class, which is shown in listing 8.2, is a Spring `@Configuration` class. It defines the Spring `@Beans` that implement the `/orders` endpoints. The `orderProxyRouting` and `orderHandlerRouting` `@Beans` use the Spring Webflux routing DSL to define the request routing. The `orderHandlers` `@Bean` implements the request handlers that perform API composition.

Listing 8.2. The `OrderConfiguration` class defines the Spring beans that implement the `/orders` endpoints

```
@Configuration
@EnableConfigurationProperties(OrderDestinations.class)
public class OrderConfiguration {

  @Bean
  public RouteLocator orderProxyRouting(OrderDestinations orderDestinations) {
    return Routes.locator()
            .route("orders")
            .uri(orderDestinations.orderServiceUrl)
            .predicate(path("/orders").or(path("/orders/*")))          (1)
            .and()
            ...
            .build();
  }
```

```
  @Bean
  public RouterFunction<ServerResponse>
            orderHandlerRouting(OrderHandlers orderHandlers) {
    return RouterFunctions.route(GET("/orders/{orderId}"),          ❷
                      orderHandlers::getOrderDetails);
  }

  @Bean
  public OrderHandlers orderHandlers(OrderService orderService,
                                RestaurantOrderService restaurantOrderService,
                                DeliveryService deliveryService,
                                AccountingService accountingService) {
    return new OrderHandlers(orderService, restaurantOrderService,   ❸
                             deliveryService, accountingService);
  }

}
```

❶ By default, route all requests whose path begins with `/orders` to the URL `orderDestinations.orderServiceUrl`

❷ Route a `GET /orders/{orderId}` to `orderHandlers::getOrderDetails`

❸ The `@Bean`, which implements the custom request handling logic

`OrderDestinations`, which is shown in listing 8.3, is a Spring `@ConfigurationProperties` class that enables the external configuration of backend service URLs.

Listing 8.3. A Spring `@ConfigurationProperties` class that enables the external configuration of backend service URLs

```
@ConfigurationProperties(prefix = "order.destinations")
public class OrderDestinations {

  @NotNull
  public String orderServiceUrl;

  public String getOrderServiceUrl() {
    return orderServiceUrl;
  }

  public void setOrderServiceUrl(String orderServiceUrl) {
    this.orderServiceUrl = orderServiceUrl;
  }
  ...
}
```

You can, for example, specify the `URL` of the `Order Service` either as the `order.destinations.orderServiceUrl` property in a properties file or as an operating system environment variable `ORDER_DESTINATIONS_ORDER_SERVICE_URL`.

The `OrderHandlers` class

The `OrderHandlers` class, which is shown in listing 8.4, defines the request handler

methods that implement custom behavior including API composition. This class is injected with several proxy classes, which make requests to backend services.

Listing 8.4. The `OrderHandlers` class implement custom request handling logic. The `getOrderDetails()` method, for example, performs API composition to retrieve information about an order.

```
public class OrderHandlers {

  private OrderService orderService;
  private RestaurantOrderService restaurantOrderService;
  private DeliveryService deliveryService;
  private AccountingService accountingService;

  public OrderHandlers(OrderService orderService,
                       RestaurantOrderService restaurantOrderService,
                       DeliveryService deliveryService,
                       AccountingService accountingService) {
    this.orderService = orderService;
    this.restaurantOrderService = restaurantOrderService;
    this.deliveryService = deliveryService;
    this.accountingService = accountingService;
  }

  public Mono<ServerResponse> getOrderDetails(ServerRequest serverRequest) {
    String orderId = serverRequest.pathVariable("orderId");

    Mono<OrderInfo> orderInfo = orderService.findOrderById(orderId);

    Mono<Optional<RestaurantOrderInfo>> restaurantOrderInfo =
       restaurantOrderService
            .findRestaurantOrderByOrderId(orderId)
            .map(Optional::of)                                          (1)
            .onErrorReturn(Optional.empty());                           (2)

    Mono<Optional<DeliveryInfo>> deliveryInfo =
        deliveryService
            .findDeliveryByOrderId(orderId)
            .map(Optional::of)
            .onErrorReturn(Optional.empty());

    Mono<Optional<BillInfo>> billInfo = accountingService
            .findBillByOrderId(orderId)
            .map(Optional::of)
            .onErrorReturn(Optional.empty());

    Mono<Tuple4<OrderInfo, Optional<RestaurantOrderInfo>,               (3)
                Optional<DeliveryInfo>, Optional<BillInfo>>> combined =
           Mono.when(orderInfo, restaurantOrderInfo, deliveryInfo, billInfo);

    Mono<OrderDetails> orderDetails =                                   (4)
        combined.map(OrderDetails::makeOrderDetails);

    return orderDetails.flatMap(person -> ServerResponse.ok()           (5)
            .contentType(MediaType.APPLICATION_JSON)
            .body(fromObject(person)));
```

```
  }


}
```

1. Transform a `RestaurantOrderInfo` into an `Optional<RestaurantOrderInfo>`
2. If the service invocation failed return `Optional.empty()`
3. Combine the four values into a single value, a `Tuple4`
4. Transform the `Tuple4` into a `OrderDetails`
5. Transform the OrderDetails` into a `ServerResponse`

The `getOrderDetails()` method implements API Composition to fetch the order details. It is written in a scalable, reactive style using the `Mono` abstraction, which is provided by Project Reactor. A `Mono`, which is an richer kind of Java 8 `CompletableFuture`, contains the outcome of an asynchronous operation, which is either a value or an exception. It has a rich API for transforming and combining the values returned by asynchronous operations. You can use `Monos` to write concurrent code in a style that is simple and easy to understand. In this example, the `getOrderDetails()` method invokes the four services in parallel and combine the results to create an `OrderDetails` object.

The `getOrderDetails()` method takes a `ServerRequest`, which is the Spring Webflux representation of an HTTP request, as a parameter and does the following:

1. It extracts the `orderId` from the path.
2. It invokes the four services asynchronously via their proxies, which return `Monos`. In order to improve availability, `getOrderDetails()` treats the results of all services except the `OrderService` as optional. If a `Mono` returned by an optional service contains an exception then the call to `onErrorReturn()` transforms it into a `Mono` containing an empty `Optional`.
3. It combines the results asynchronously using `Mono.when()`, which returns a `Mono<Tuple4>` containing the 4 values
4. It transforms the `Mono<Tuple4>` into an `Mono<OrderDetails>` by calling `OrderDetails::makeOrderDetails`
5. It transforms the `OrderDetails` into a `ServerResponse`, which is the Spring WebFlux representation of the JSON/HTTP response

As you can see, because `getOrderDetails()` uses `Monos` it concurrently invokes the services and combines the results without using messy, difficult to read callbacks. Let's take a look one of the service proxies that return the results of a service API call wrapped in a `Mono`.

The `OrderService` class

The `OrderService` class, which is shown in listing 8.5, is a remote proxy for the `Order Service`. It invokes the `Order Service` using a `WebClient`, which is the Spring Webflux reactive HTTP client.

Listing 8.5. The `OrderService` class is a remote proxy for the `Order Service`

```
@Service
public class OrderService {

  private OrderDestinations orderDestinations;

  private WebClient client;

  public OrderService(OrderDestinations orderDestinations, WebClient client) {
    this.orderDestinations = orderDestinations;
    this.client = client;
  }

  public Mono<OrderInfo> findOrderById(String orderId) {
    Mono<ClientResponse> response = client
            .get()
            .uri(orderDestinations.orderServiceUrl + "/orders/{orderId}",
                 orderId)
            .exchange();                                                    (1)
    return response.flatMap(resp -> resp.bodyToMono(OrderInfo.class));      (2)
  }

}
```

(1) Invoke the service
(2) Convert the response body to an `OrderInfo`

The `findOrder()` method retrieves the `OrderInfo` for an order. It uses the `WebClient` to make the HTTP request to the `Order Service` and deserializes the JSON response to an `OrderInfo`. `WebClient` has a reactive API and the response is wrapped in a `Mono`. The `findOrder()` method uses `flatMap()` to transform the `Mono<ClientResponse>` into a `Mono<OrderInfo>`. As the name suggests, the `bodyToMono()`method returns the response body as a `Mono`.

The `ApiGatewayApplication` class

The `ApiGatewayApplication` class, which is shown in listing 8.6, implements the API gateway's `main()` method. It is a standard Spring Boot main class.

Listing 8.6. The main() method for the API gateway

```
@SpringBootConfiguration
@EnableAutoConfiguration
@EnableGateway
@Import(OrdersConfiguration.class)
public class ApiGatewayApplication {

  public static void main(String[] args) {
    SpringApplication.run(ApiGatewayApplication.class, args);
  }
}
```

The `@EnableGateway` annotation imports the Spring configuration for the Spring Cloud

Gateway framework.

Spring Cloud Gateway is an excellent framework for implementing a API gateway. It enables you to configure basic proxying using a simple, concise routing rules DSL. It is also straightforward to route requests to handler methods that perform API composition and protocol translation. Spring Cloud Gateway is built using the scalable, reactive Spring framework 5 and Project Reactor frameworks.

8.4 Summary

- You application's external clients usually access the application's services via an API gateway. An API gateway provides client with a custom API and is responsible for request routing, API composition, protocol translation and implementing edge functions, such as authentication.
- Your application can have either a single API gateway or it can use the Backends for Frontends pattern, which defines an API Gateway for each type of client. The main advantage of the the Backends for Frontends is that it gives the client teams greater autonomy since they develop, deploy and operate their own API gateway.
- Spring Cloud Gateway is good, easy to use foundation for developing an API gateway. It routes requests based using any request attribute including the method and the path. Spring Cloud Gateway routes a request either directly to a backend service or a custom handler method. It is built using the scalable, reactive Spring framework 5 and Project Reactor frameworks. You can write your custom request handlers in a reactive style using, for example, Project Reactor's `Mono` abstraction.