

With examples in Java

Microservices Patterns

Chris Richardson



MANNING



MEAP Edition
Manning Early Access Program
Microservices Patterns
With examples in Java
Version 10

Copyright 2018 Manning Publications

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

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*
- 2 Decomposition strategies*
- 3 Inter-process communication in a microservice architecture*
- 4 Managing transactions with sagas*
- 5 Designing business logic in a microservice architecture*
- 6 Developing business logic with event sourcing*
- 7 Implementing queries in a microservice architecture*
- 8 External API patterns*
- 9 Testing microservices*
- 10 Developing production ready services*
- 11 Deploying microservices*
- 12 Refactoring to microservices*

I

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 become 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.

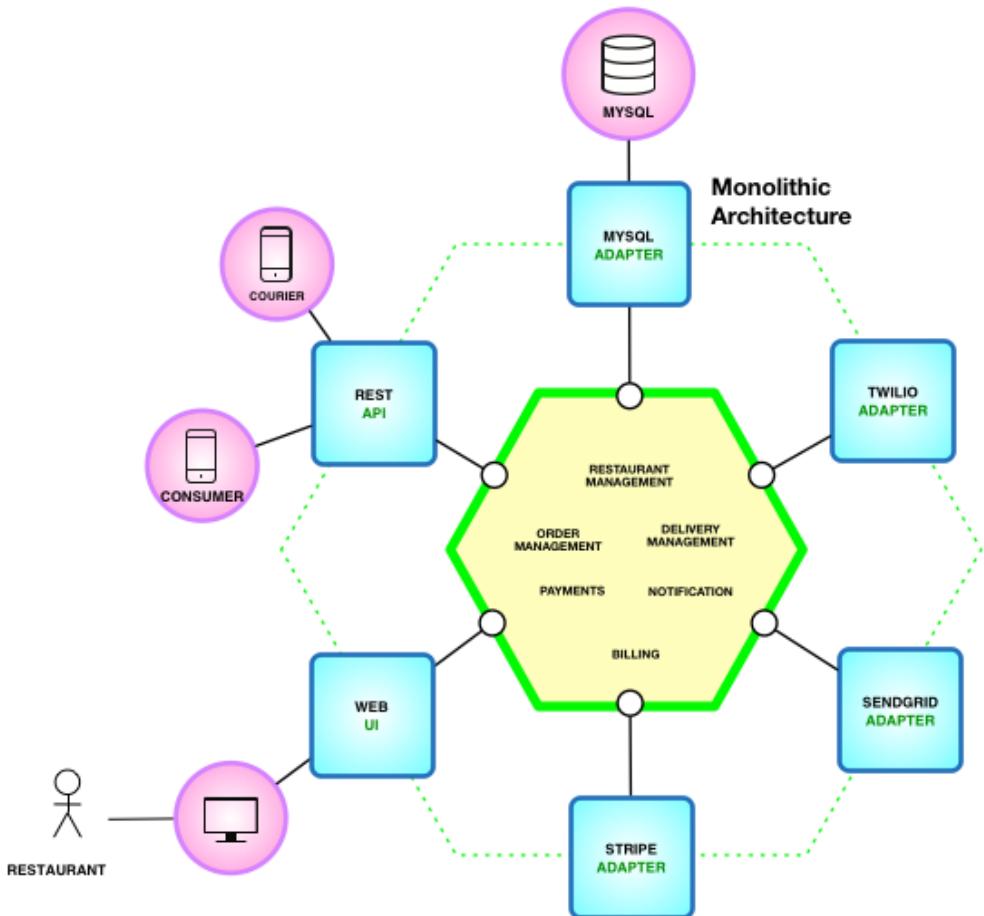
1.1.1 What the FTGO application does

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

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 distant 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 a 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 quality of service requirements, a.k.a. non-functional requirements, 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.

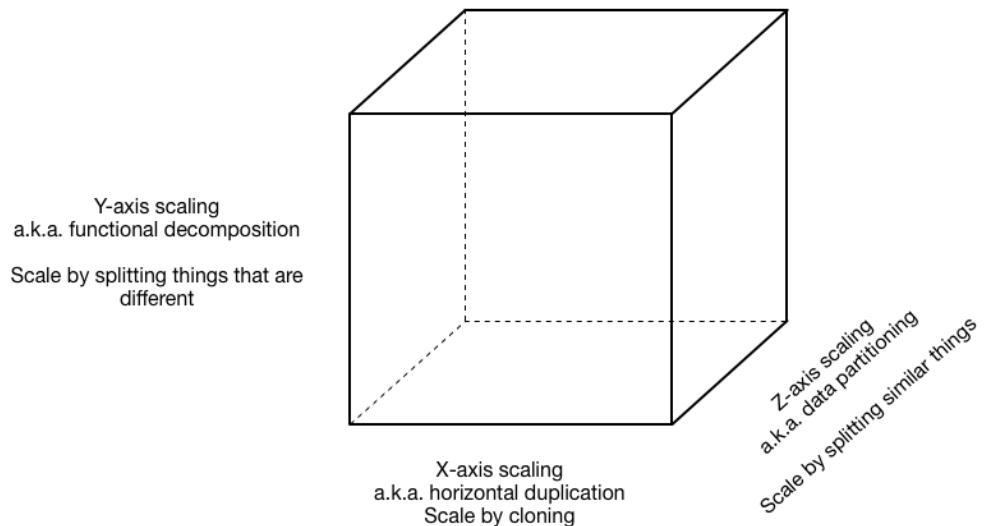
On the one hand, a disciplined team can slow down the pace of their descent towards monolithic hell. They can work hard to maintain the modularity of their application. They can write comprehensive automated tests. But on the other hand, they cannot avoid the issues of a large team working on a single 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.

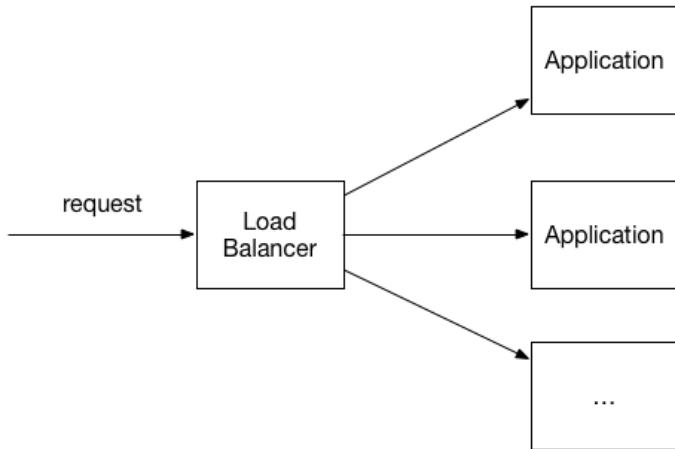


The model defines three ways to scale an application: X, Y and Z.

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

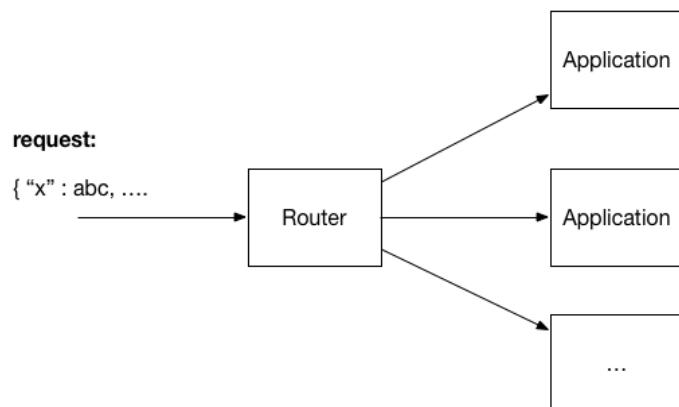


The load balancer distributes requests amongst the N identical instances of the application. This is a great way of improving the capacity and the availability of an application.

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

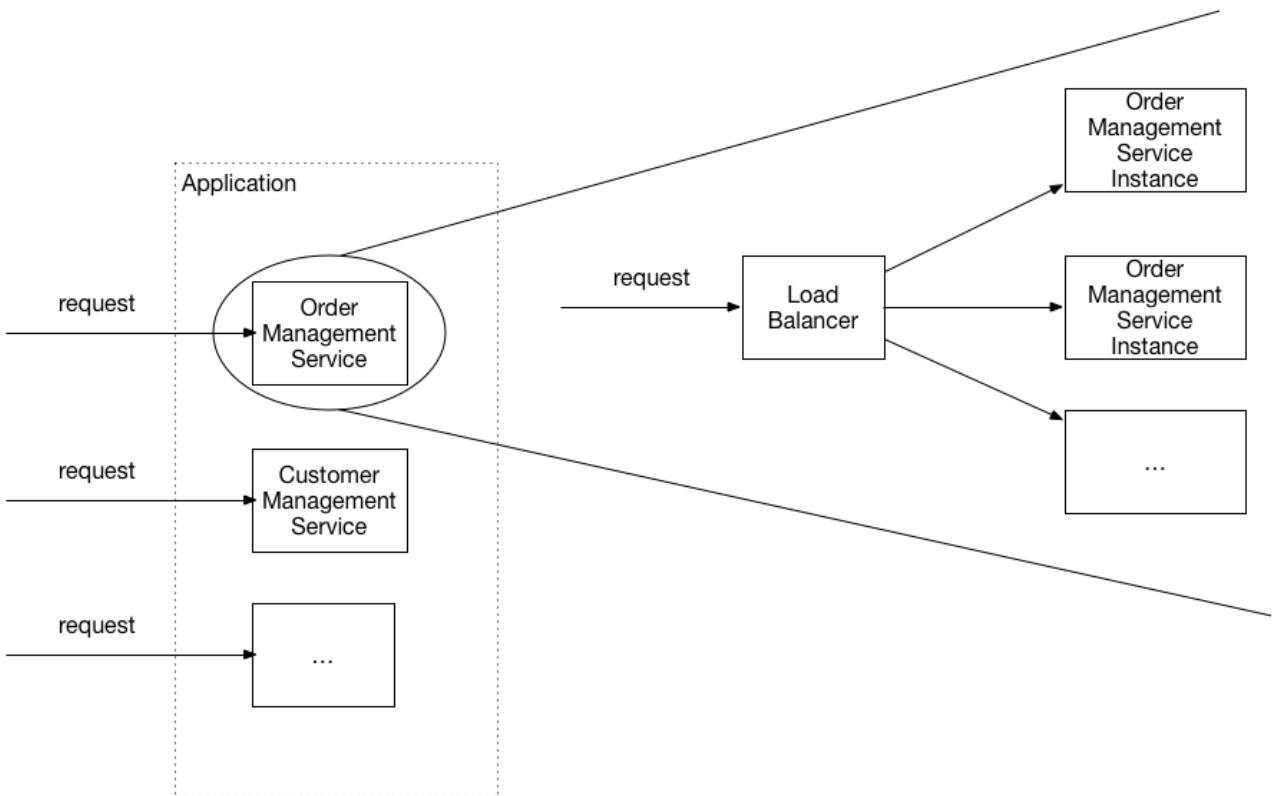


In this example, each application instance is responsible for a subset of the users. The router uses the *userId* specified by the request *Authorization* header to select one of the N identical instances of the application.

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 API, which is an impermeable boundary that is difficult to violate. You can't simply bypass the API and access an internal class as you can with a Java package. As a result, it's much easier to preserve the modularity of the application over time. There are other benefits of using services as the building blocks including the ability to deploy and scale them independently.

1.2.3 Each service has its own database

A key characteristic of the microservice architecture is that the services are loosely coupled and communicate only via APIs. 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!

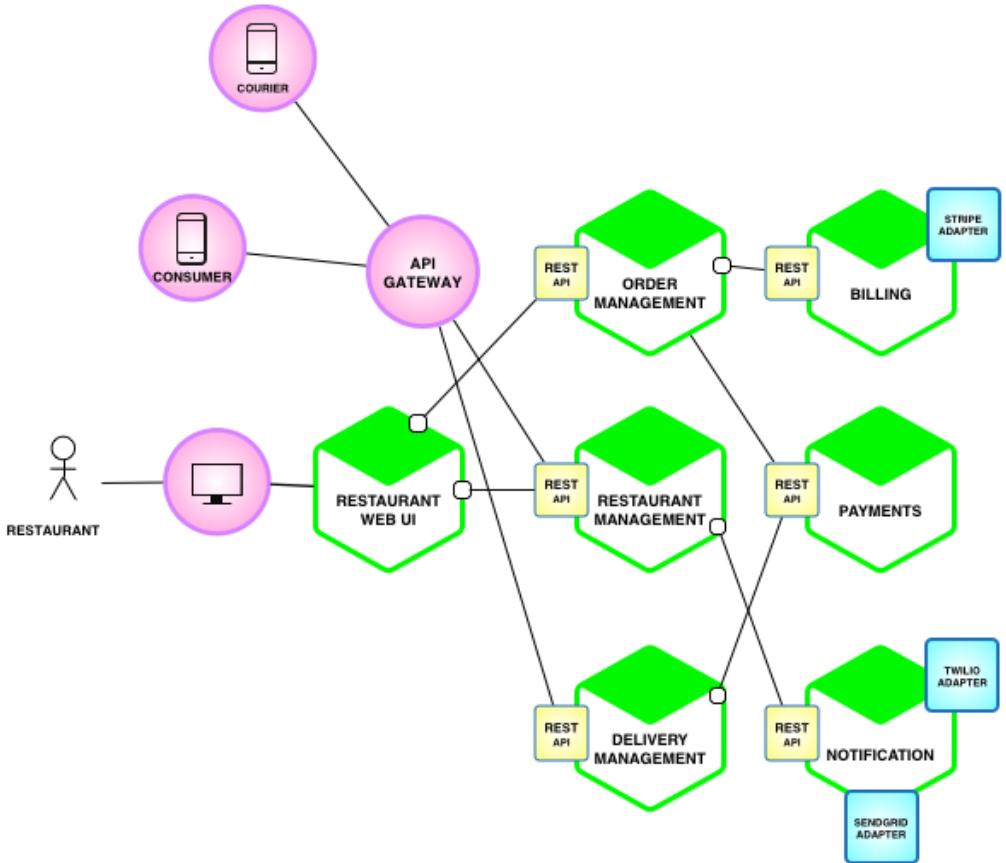
The requirement for each service to have its own database does not mean that it has its own database server. You do not, for example, have to spend 10x more on Oracle RDBMS licenses. In chapter 2, we will explore this topic in depth.

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 8, 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 11 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 2, 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 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:

- Enables the continuous delivery and deployment of large, complex applications.
- 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.

Enables the continuous delivery and deployment of large, complex applications.

The most important benefit of the microservice architecture is that it enables continuous delivery and deployment of large, complex applications. As I describe later in section ["Beyond microservices: process and organization"](#), continuous delivery/deployment is part of DevOps, which is an increasingly popular software development methodology. High-performing DevOps organizations typically deploy changes into production with very few production issues.

There are three ways that the microservice architecture enables continuous delivery/deployment. First, microservice architecture has the testability required by continuous delivery/deployment. Automated testing is a key practice of continuous delivery/deployment. Since each service in a microservice architecture is relatively small, automated tests are much easier to write and faster to execute.

Second, the microservices architecture has the deployability required by continuous delivery/deployment. 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. As a result, it's much easier to deploy changes frequently into production.

Finally, the microservice architecture enables development teams to be autonomous. You can structure the engineering organization as a collection of small (e.g. two pizza) 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. As a result, the development velocity is much higher.

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 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.

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 somewhat of an art. To make matters 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 10.

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 11, 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 dichotomy¹, 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

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

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.

1.4.2 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.

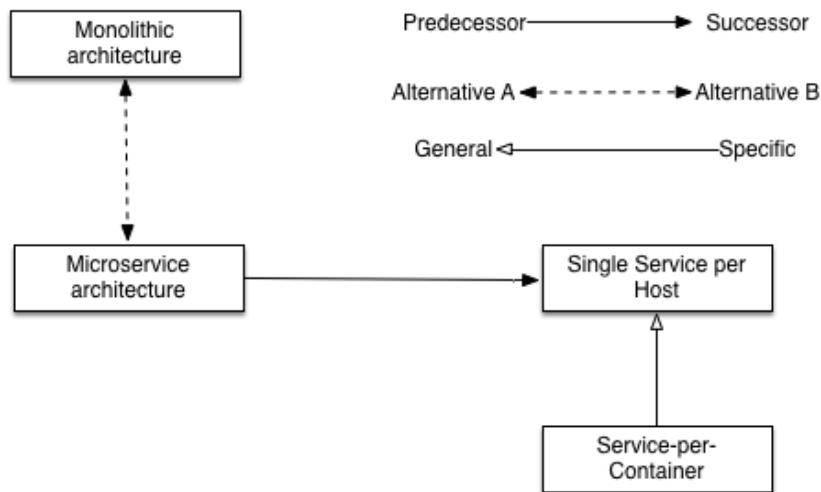
Related patterns

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 10 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 10 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 keynoted² 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

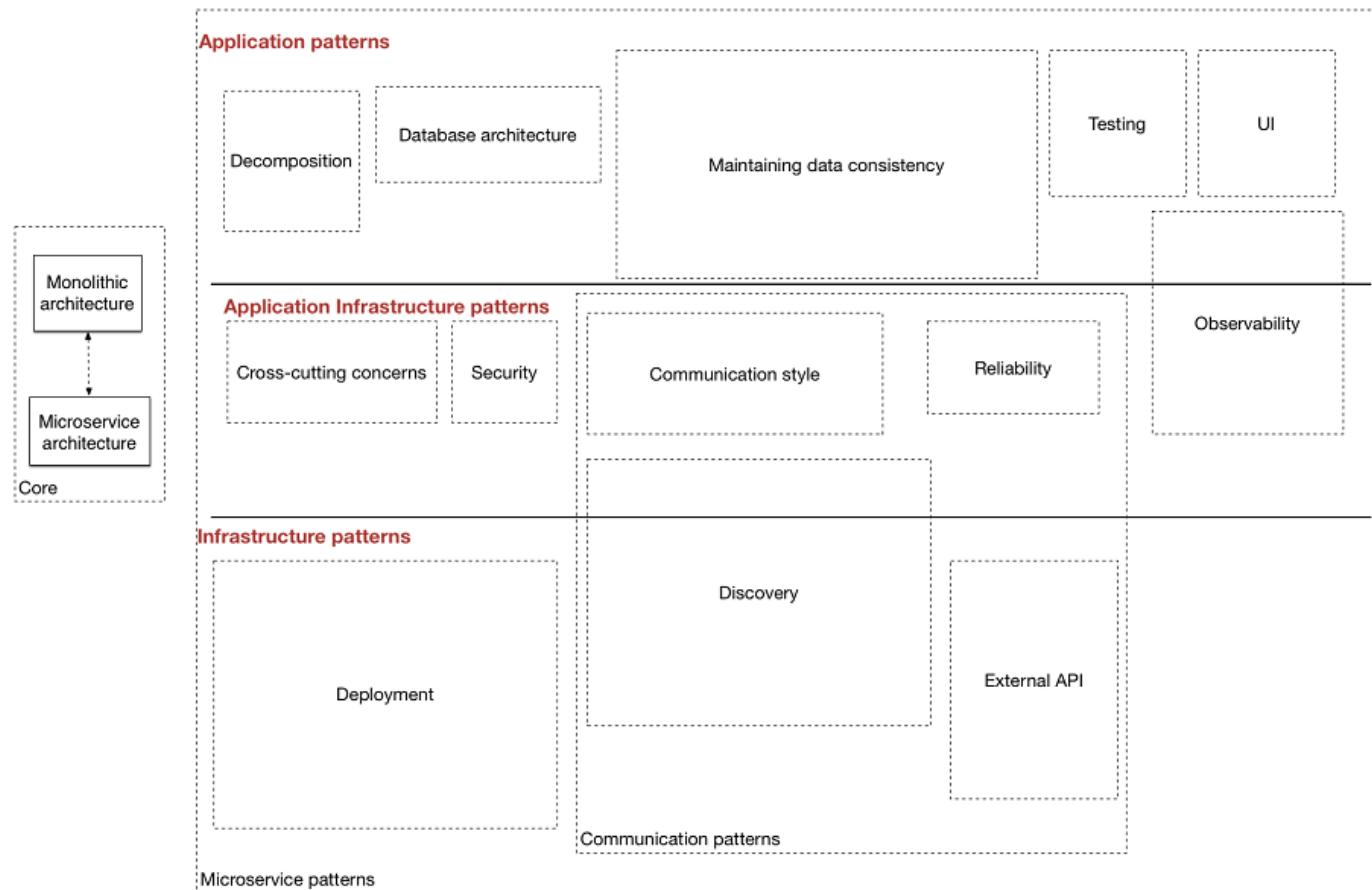
The microservice architecture pattern language is a collection of patterns that help you architecture and design an application using the microservice architecture. The pattern language first helps you to first decide whether to use the microservice architecture. It describes the monolithic architecture 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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

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:

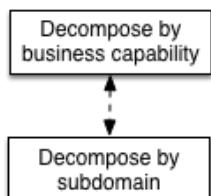
- infrastructure patterns - these are patterns that solve problems that are mostly infrastructure issues that are outside of development
- application infrastructure - these are patterns for infrastructure issues that also impact development
- application patterns - these are patterns that solve problems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

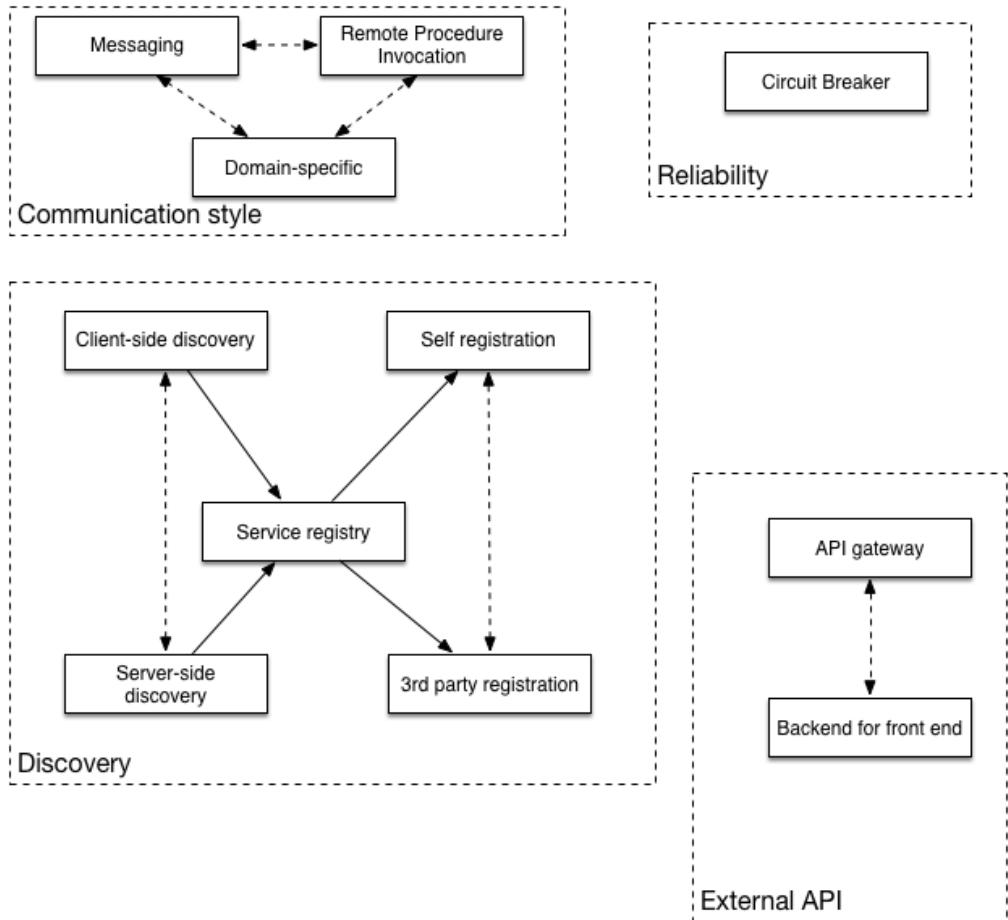
Figure 1.9. The decomposition patterns



In chapter 2 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 3 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 10 we look at the options.

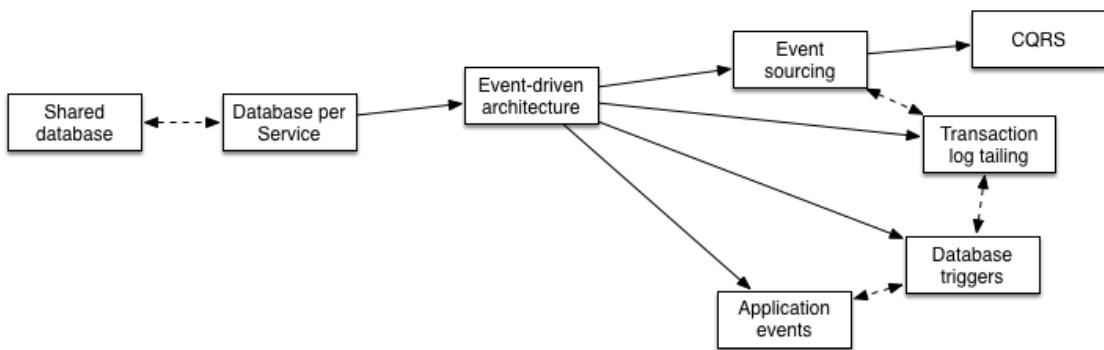
Not only, must you decide how your services communicate amongst themselves but you also have to figure out how clients of your application communicate with the services. One option, is for clients to communicate directly with the individual services. Another option is for clients to make requests via an API Gateway pattern, which routes requests and aggregates data from multiple services. In chapter 8 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

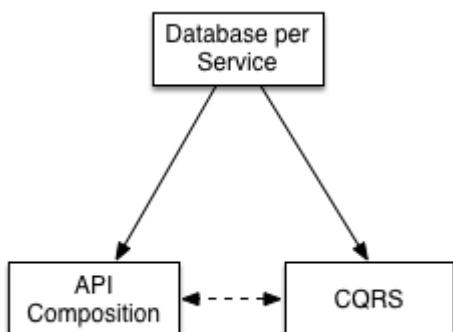


In chapters 4, 5 and 6 we look at these patterns in more detail.

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



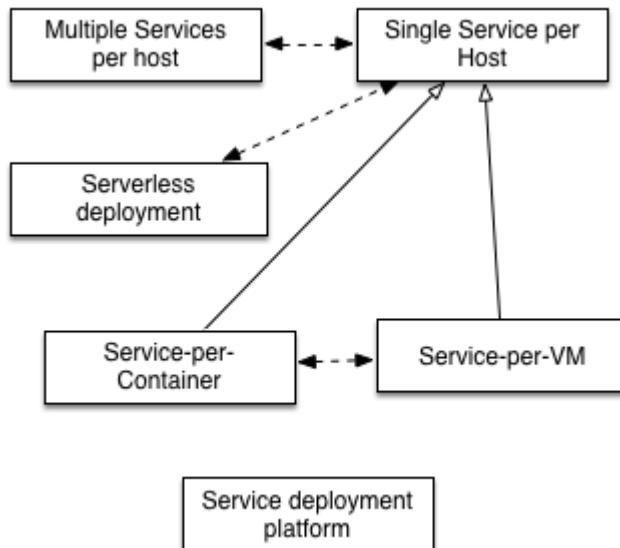
Sometimes you can use the API Composition pattern, which invokes the APIs of one or

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 7 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 10 we look at the different deployment options.

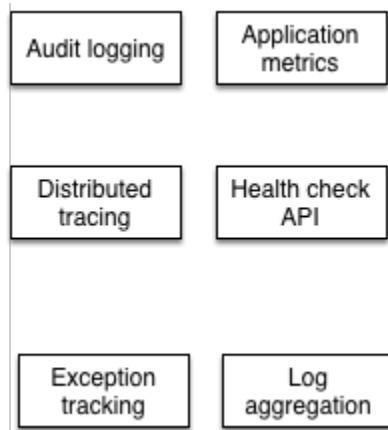
Observability patterns

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

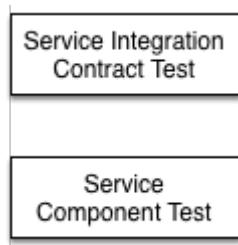


In chapter {chapter-prod-ready} I describe the Observability patterns that make it easier to understand and troubleshoot a microservices-based application.

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



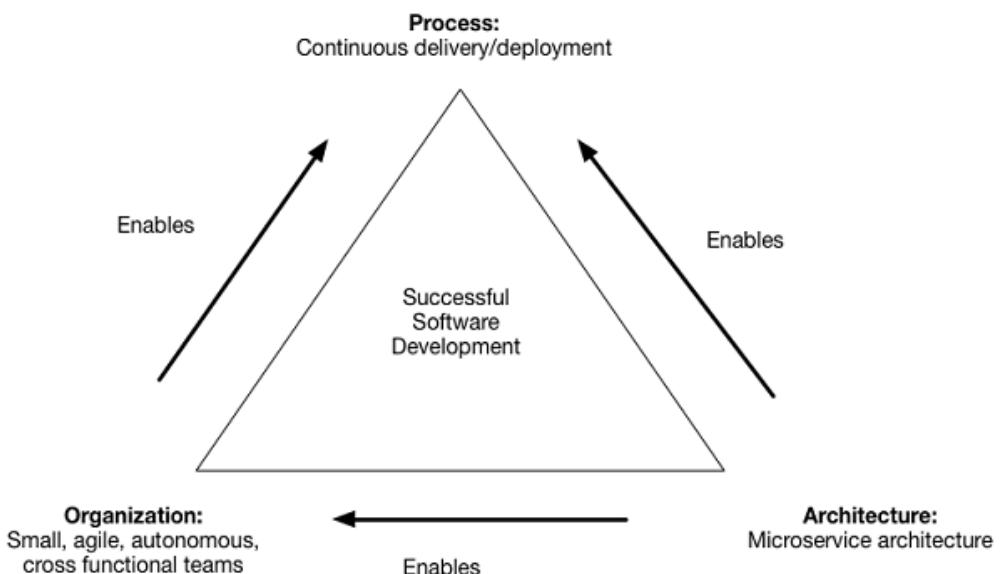
In chapter 9, I describe these testing patterns that enable a microservices application to be tested more easily.

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. As I described earlier in section ["Enables the continuous delivery and deployment of large, complex applications."](#), 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 continuous delivery/deployment, which is a part of DevOps.

Jez Humble³ defines continuous delivery as follows:

Continuous Delivery is the ability to get changes of all types—including new features, configuration changes, bug fixes and experiments—into production, or into the hands of users, safely and quickly in a sustainable way.

A key characteristic of continuous delivery is that software is always releasable. Continuous deployment takes continuous delivery one step further is the practice of automatically deploying releasable code into production. High performing organizations that practice continuous deployment⁴ deploy multiple times per day into production, have far fewer production outages, and recover quickly from any that do occur. As I described earlier in section ["Enables the continuous delivery and](#)

³ continuousdelivery.com/

⁴ 2017 State of DevOps report

[deployment of large, complex applications.”](#), the microservice architecture directly supports continuous delivery/deployment.

Move fast without breaking things

The goal of continuous delivery/deployment (and, more generally, DevOps) is to rapidly yet reliably deliver software. Four useful metrics for assessing software development are:

- Deployment frequency - how often is software deployed into production
- Lead time - the time from a developer checking in a change to that change being deployed
- Meantime to recover - time to recover from a production problem
- Change failure rate - the percentage of changes that result in a production problem

In a traditional organization the deployment frequency is low, and the lead time is high. Stressed out developers and operations people typically stay up late into the night fixing last minute issues during the maintenance window. In contrast, a DevOps organization releases software frequently, often multiple times per day, with far fewer production issues. Amazon, for example, deployed changes into production every 11.6 seconds in 2014⁵ and Netflix had a lead time of 16 minutes for one software component⁶.

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 teammates. 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

⁵ Jon Jenkins, Velocity conference 2011, www.youtube.com/watch?v=dxk8b9rSKOo

⁶ Netflix TechBlog, medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15

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 given 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 *abilities* 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

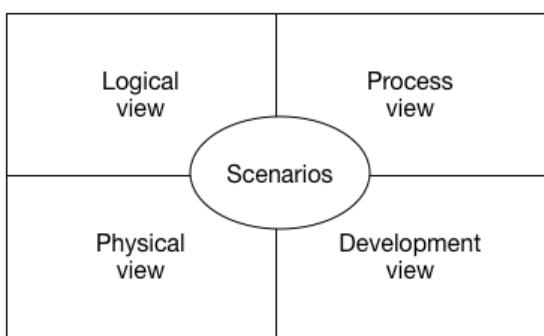
The microservice architecture is obviously a kind of architecture. But what is it exactly and why does it matter? To answer that these question lets first look at what is meant by architecture. Len Bass and colleagues define architecture as follows:

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.

-- Bass et al, *Documenting Software Architectures*

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 example, shows how the classes collaborate. Similarly, a scenario in the process view, shows how the processes collaborate.

The 4+1 view model is an excellent way to describe an application's architecture. Each view describes an important aspect of the architecture. The scenarios illustrate how the elements of a view collaborate. Let's 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 quality of service requirements. These are also known as quality attributes and are the so called *-ilities*. The quality of service 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

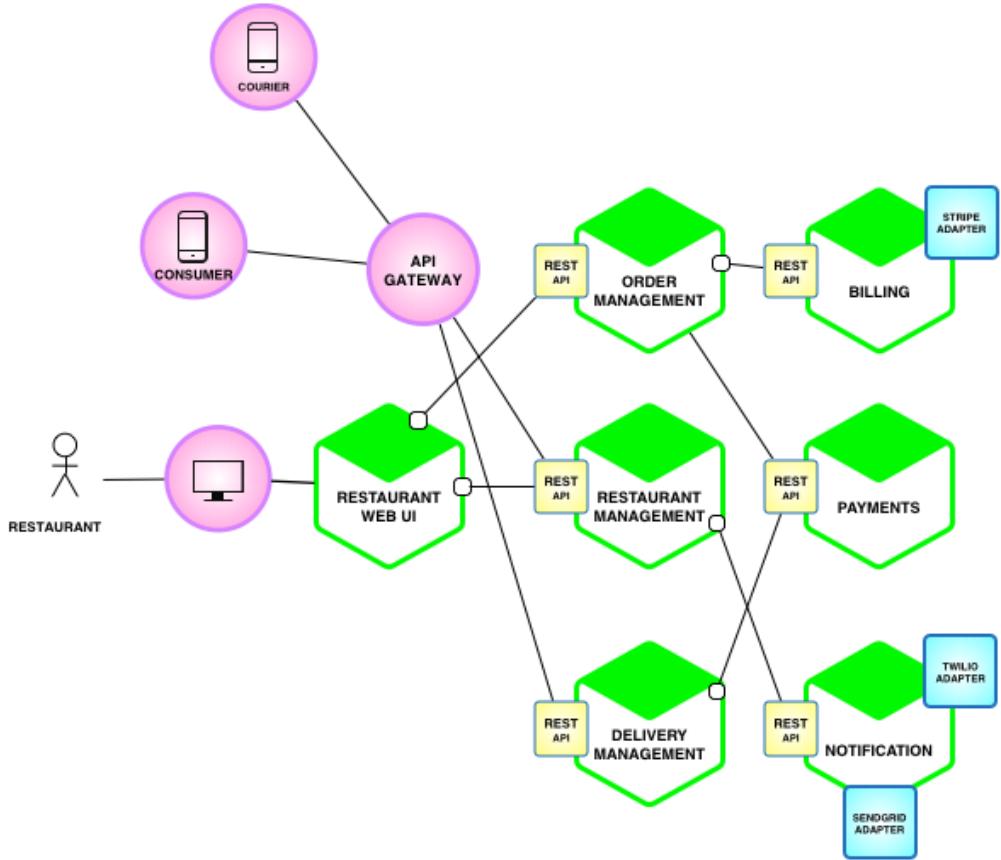
The microservice architecture is what is known as an architectural style. David Garlan and Mary Shaw define an architectural style as follows:

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 3, 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, let's 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:

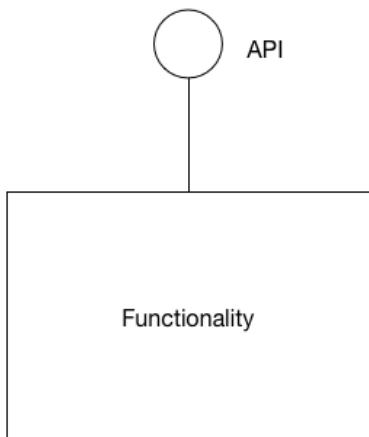
a mechanism to access an underlying capability

– OASIS

[en.wikipedia.org/wiki/Service_\(systems_architecture\)](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. The service API implements operations and publishes events. There are two types of operations, commands and queries. A command perform actions and updates data. A query simply retrieves data. A service might also publish events, which are consumed by its clients.

Figure 2.3. Abstract view of a service. Its implementation is encapsulated by its API. The API implements operations and publishes events.



A service's clients invoke the API's operations and subscribe to its events. The service's implementation is hidden, unlike in a monolithic architecture where a developer can easily 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 10, 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 be 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. Let's now look at how to define an application's microservice architecture.

2.1.2 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, I describe a simple process for defining an application's architecture. It captures what needs to be done but in reality the process is likely to be iterative and involve a lot of creativity.

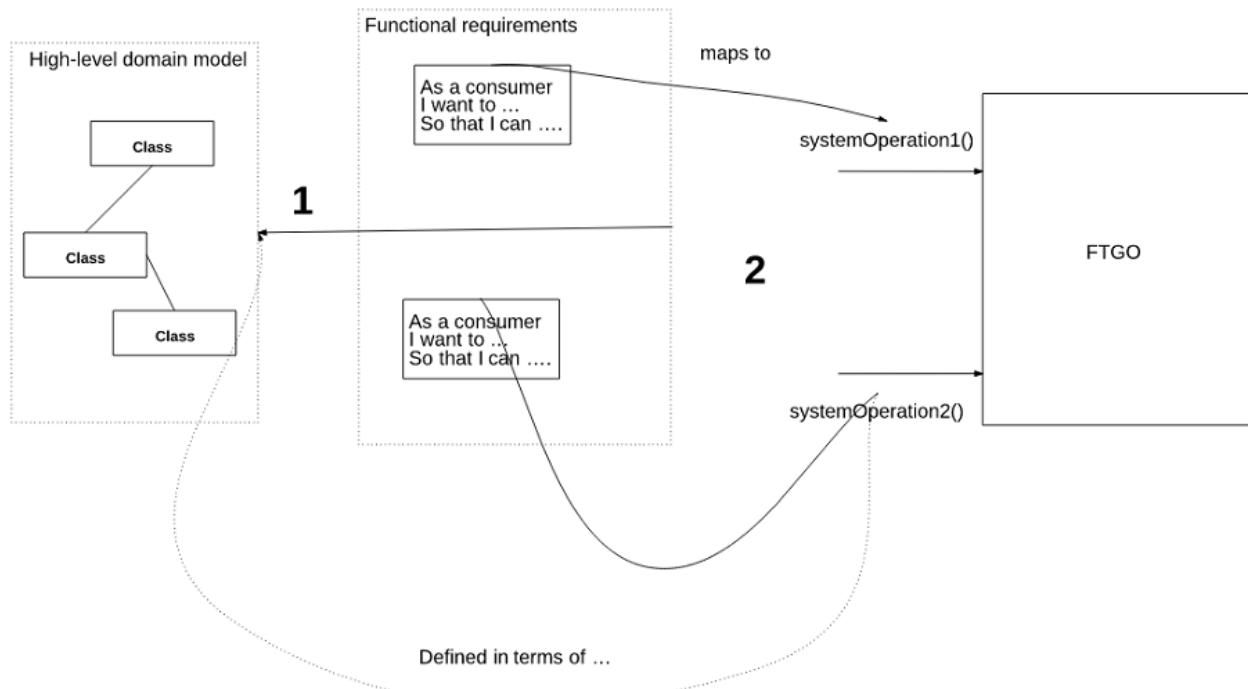
An application's raison d'être is to handle requests, and so we will first identify the key requests. However, instead of describing the requests in terms of specific IPC technologies such as REST or messaging, I use the more abstract notion of system operation. Once we have identified the system operations, 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. Let's 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 than 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:

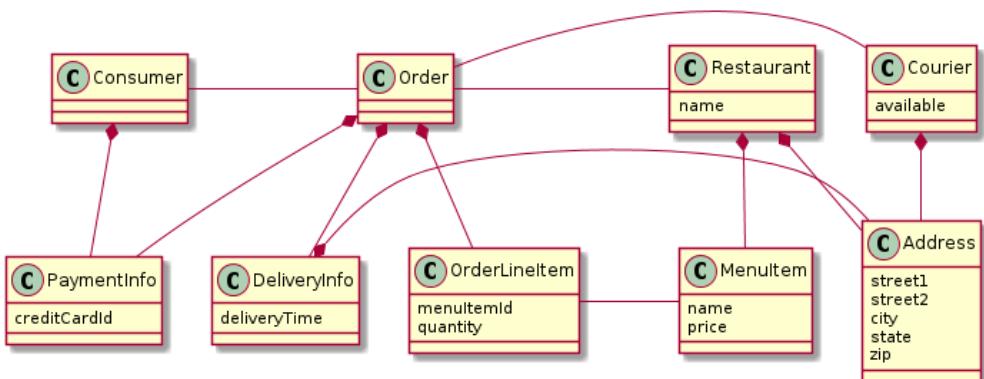
```
Given a consumer
  And a restaurant
When the consumer places an order for that restaurant
  with a delivery address/time that can be served by that restaurant
Then consumer's credit card is authorized
  And an order is created in the PENDING_ACCEPTANCE state
  And the order is associated with the 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, it's 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. Let's 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 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> the order.status is PENDING_ACCEPTANCE a courier is available to deliver the order
Post-conditions	<ul style="list-style-type: none"> 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` checkbusiness 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.

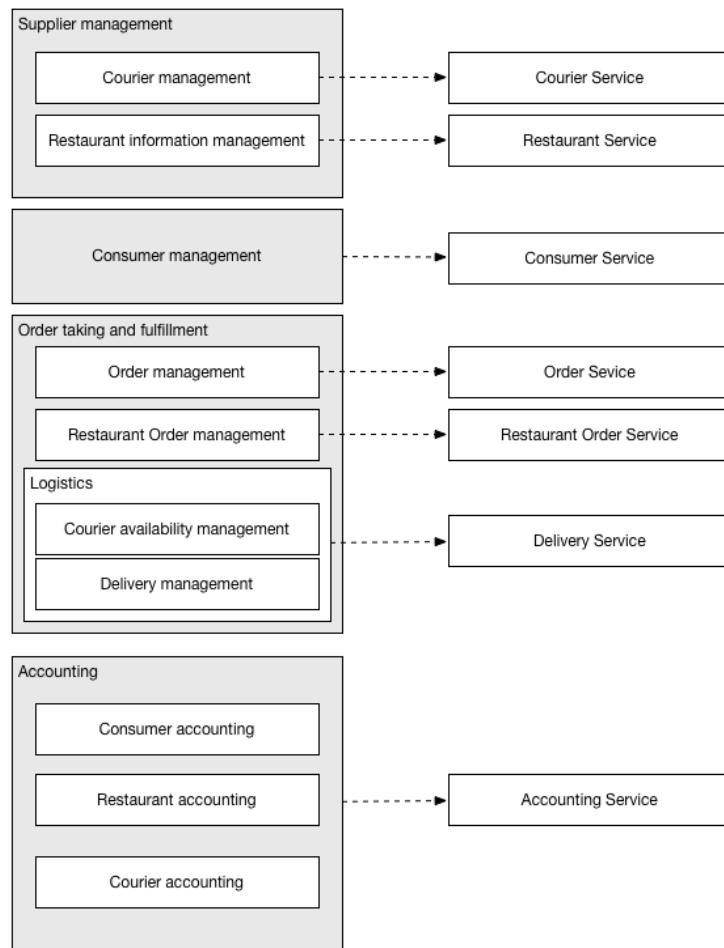
On interesting aspect of this capability hierarchy is that there are two restaurant related

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	<ul style="list-style-type: none"> • <code>createOrder()</code> • <code>findAvailableRestaurants()</code>
Restaurant Order Service	<ul style="list-style-type: none"> • <code>acceptOrder()</code> • <code>noteOrderReadyForPickup()</code>
Delivery Service	<ul style="list-style-type: none"> • <code>noteUpdatedLocation()</code> • <code>noteOrderPickedUp()</code> • <code>noteOrderDelivered()</code>

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

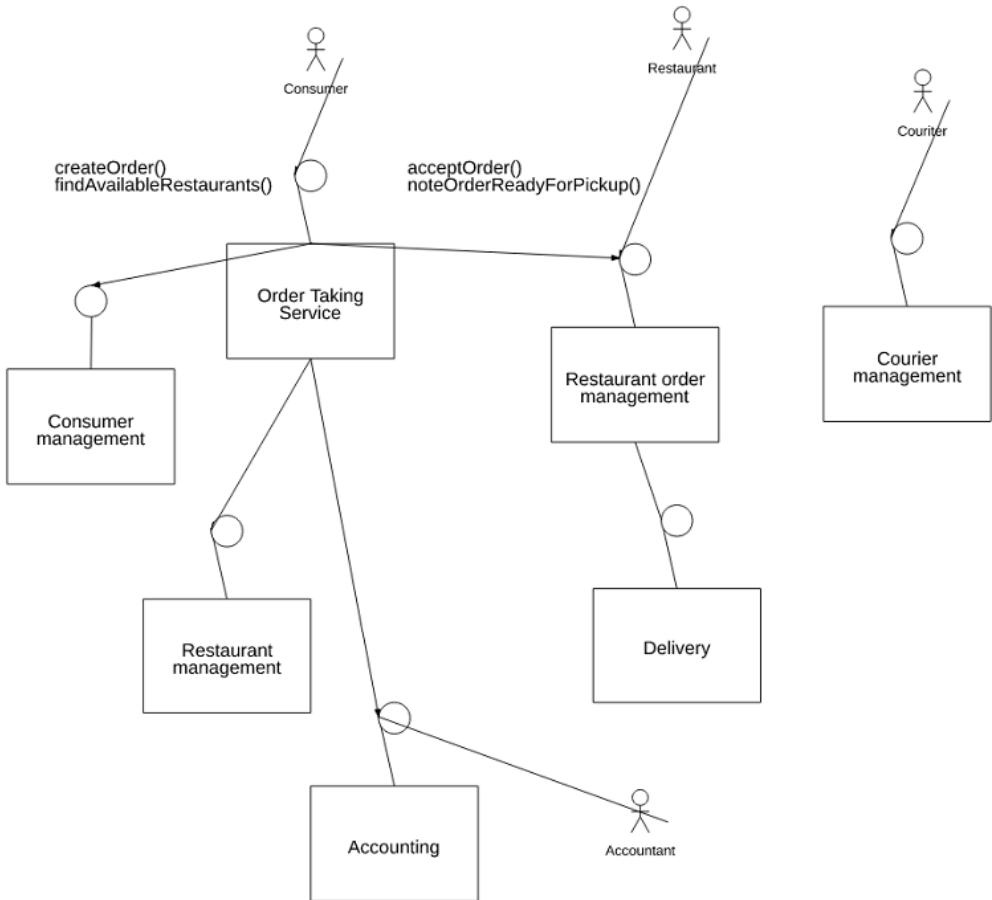
Determining how services collaborate

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 3, which covers inter-process communication, and chapter 4, 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 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

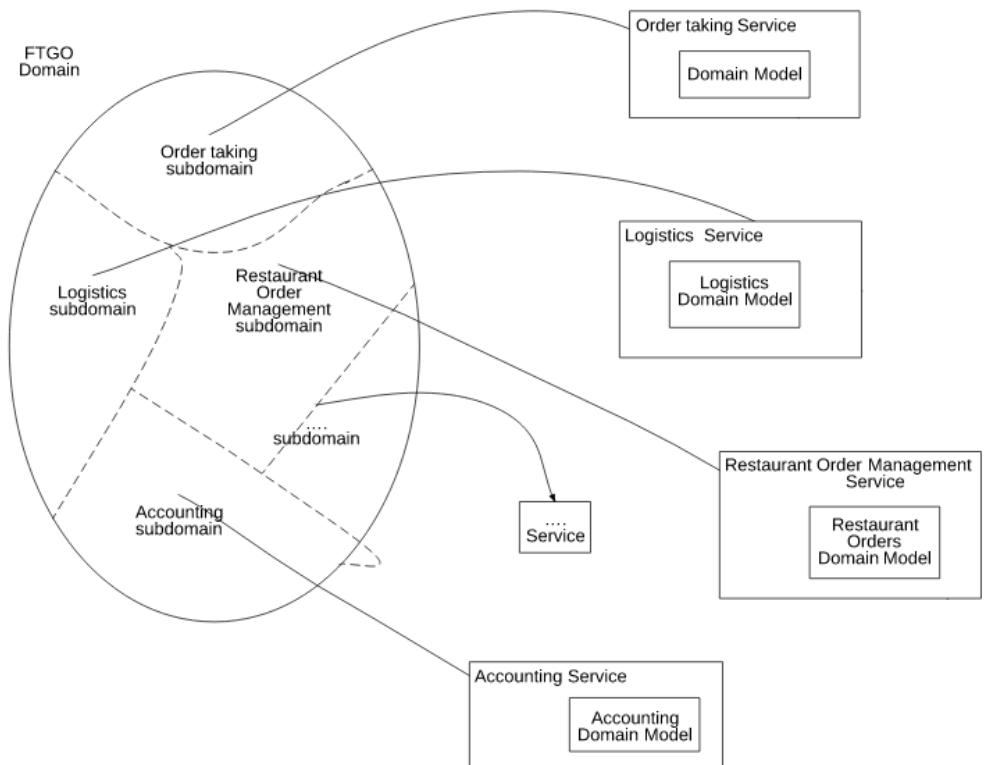
DDD is an approach for building complex software applications that is centered on the development of an object-oriented, domain model. A domain model captures knowledge about a domain in a form that can be used to solve problems within that domain. It defines the vocabulary used by the team - what DDD 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.

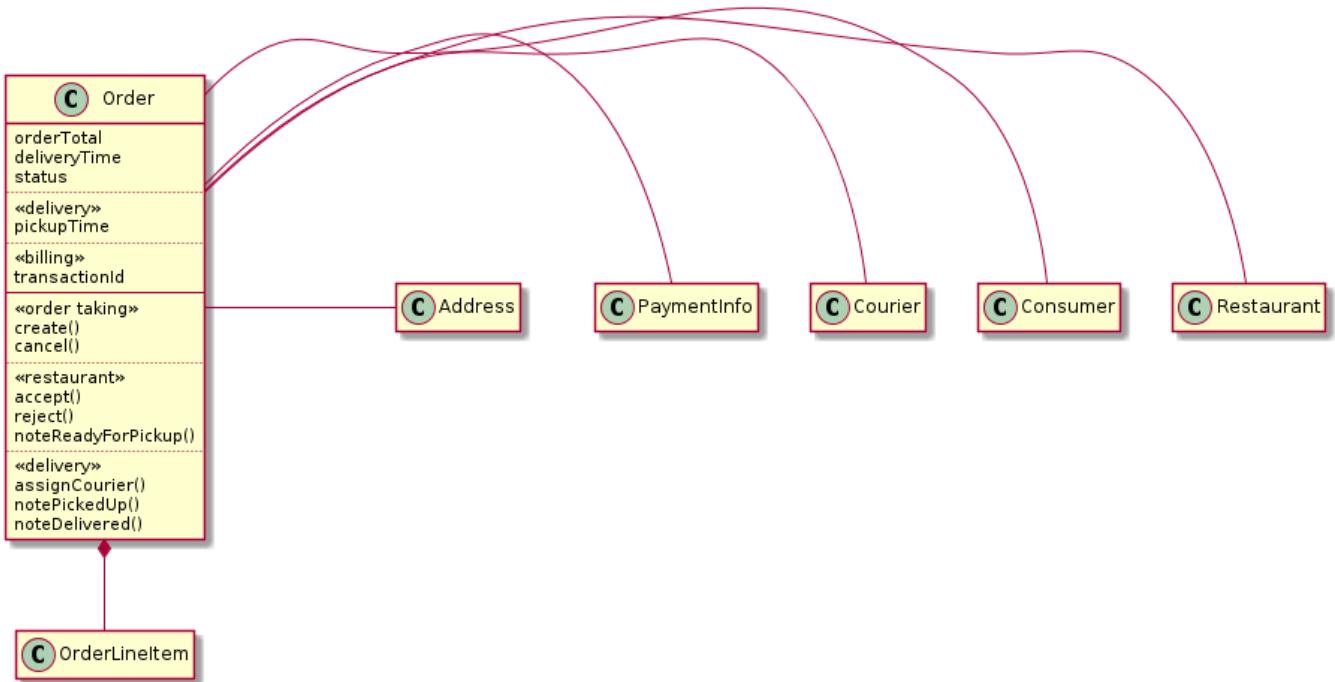
Eliminating God classes

God classes⁷ 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 mapped to a database table with many columns. Most applications have at least one of 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 any business logic that uses it into services.

⁷ wiki.c2.com/?GodClass

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 2.9 shows 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 to 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.

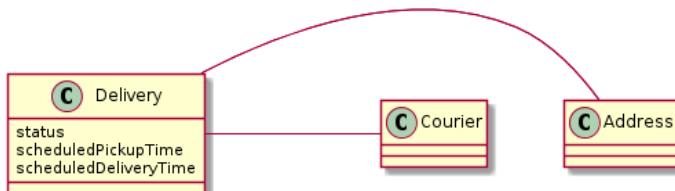
One solution is to package the `Order` class into a library and create a central `Order` database. All services that process orders use this library and access the access database. The trouble with this approach is that it violates one of the key principles of the microservice architecture and results 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.

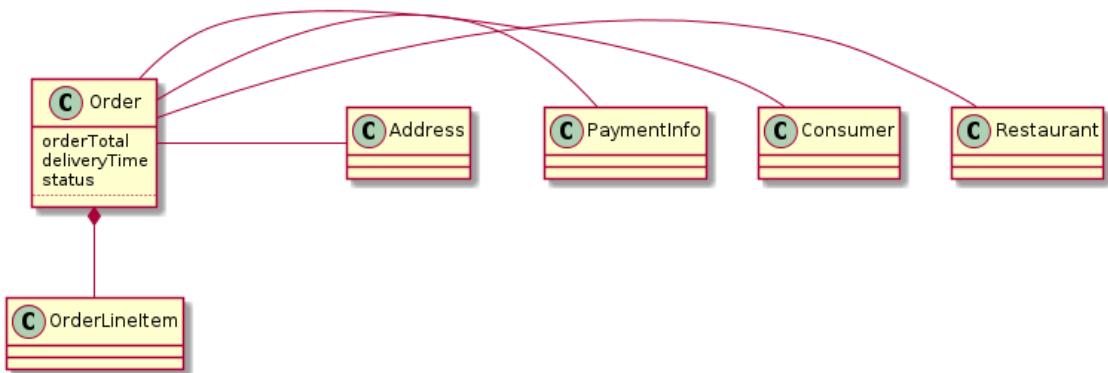
Figure 2.11. The restaurant order subdomain model



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.

SRP and CCP are two of the eleven principles developed by Bob Martin. They are particularly useful when developing a microservice architecture. The remaining nine principles are used when designing classes and packages. For more information about SRP, CCP and the other OOD principles please see this article on Bob Martin's website, the Principles of Object Oriented Design⁸. 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

⁸ butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

- 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

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 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, it's 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 approach⁹ 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 you 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.

⁹ www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10

© Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

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 Principle¹⁰. 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 specification¹¹ 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

¹⁰ en.wikipedia.org/wiki/Robustness_principle

¹¹ semver.org/

as follows:

- 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 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

Before selecting an IPC mechanism for a service, it is useful to first think about how the style of interaction between a service and its clients. Thinking first about the interaction style will help you focus on what is required. It helps you avoid getting mired in the details of a particular IPC technology. Also, as you will see in chapter 9, it helps you select the appropriate integration testing strategy.

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 even 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 is 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 footnote:<http://www.w3.org/XML/Schema>. Over time the developer community has come to realize that JSON also needs a similar mechanism. One option is to use JSON Schema¹².

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 Buffers¹³ and Avro¹⁴. 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 post¹⁵ 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

¹² json-schema.org/

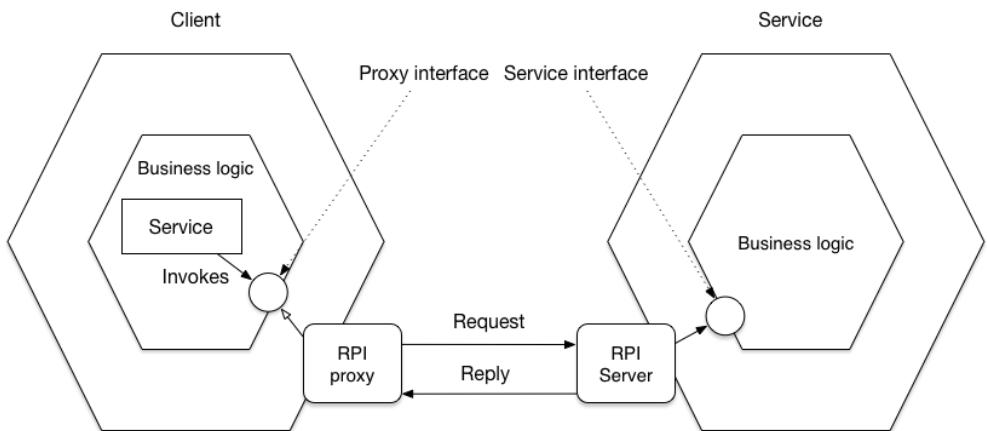
¹³ developers.google.com/protocol-buffers/docs/overview

¹⁴ avro.apache.org/

¹⁵ martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

request to a service and 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. Figure 3.1 shows how RPI works. The business logic in the client invokes a *proxy interface*, which is implemented by an *RPI proxy* adapter class. The *RPI proxy* makes a request to the service. The request is handled by a *RPI server* adapter class, which invokes the service's business logic via an interface. It then sends back a reply to the *RPI proxy*, which returns the result to the client's business logic.

Figure 3.1. The client's business logic invokes an interface, which is implemented by an RPI proxy adapter class. The RPI proxy class makes a request to the service. The RPI serveradapter class handles the request by invoking the servicee's business logic.



The *proxy interface* usually encapsulates the underlying communication protocol. 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 that uses RPI must use a service discovery mechanism. Let's first take a look at REST.

3.3.1 Using REST

Today, it is fashionable to develop APIs in the RESTful¹⁶ 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.

¹⁶ en.wikipedia.org/wiki/Representational_state_transfer

– <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 post¹⁷, 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 REST¹⁸ 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 client code¹⁹.

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 originally have an IDL. Fortunately, the developer community has eventually rediscovered the

¹⁷ roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

¹⁸ martinfowler.com/articles/richardsonMaturityModel.html

¹⁹ www.infoq.com/news/2009/04/hateoas-restful-api-advantages

value of an interface definition language for RESTful APIs. The main REST IDL is the Open API Specification²⁰, 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.

The challenge of fetching multiple resources in a single request

REST resources are usually oriented around business objects, such as `Consumer` and `Order`. Consequently, a common problem when designing a REST API is how to enable the client to retrieve multiple related objects in a single request. For example, let's imagine that a REST client wanted to retrieve an `Order` and the `Order`'s `Consumer`. A pure REST API would require the client to make at least two requests, one for the `Order` and another for its `Consumer`. A more complex scenario would require even more roundtrips and suffer from excessive latency.

One solution to this problem is for an API to allow the client to retrieve related resources when it gets a resource. For example, a client could retrieve an `Order` and its `Consumer` using `GET /orders/order-id-1234?expand=consumer`. The query parameter specifies the related resources to return with the `Order`. This approach works well in any scenarios but its often insufficient for more complex scenarios. This has led to the increasingly popularity of alternative API technologies, such as GraphQL²¹ and Netflix Falcor²², that are designed to support efficient data fetching.

The challenge of mapping operations to HTTP verbs.

Another common problem REST API design problem is how to map the operations that you want to perform on a business object to an HTTP verb. An REST 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. This has led to the growing popularity of alternatives to REST, such as gRPC, which I'll discuss shortly in section "[Using gRPC](#)". But first, let's first look at the benefits and drawbacks of REST

Benefits and drawbacks of REST

There are numerous benefits to using REST:

- 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

²⁰ www.openapis.org

²¹ graphql.org/

²² netflix.github.io/falcor/

- It doesn't require an intermediate broker, which simplifies the system's architecture.

There are some drawbacks to using REST:

- It only supports the request/reply style of communication.
- 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 I describe in section "[Using service discovery](#)" 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.
- Fetching multiple resources in a single request is challenging
- It's sometimes difficult to map multiple update options to HTTP verbs.

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

3.3.2 Using gRPC

gRPC²³ 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

²³ www.grpc.io/

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

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

A good approach to follow is the one described by Netflix²⁴. 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 Hystrix²⁵ 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 library²⁶ 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

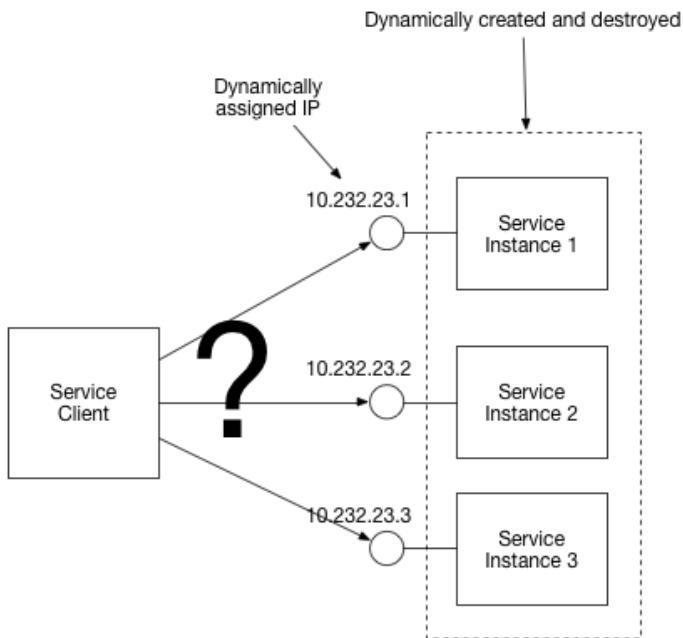
²⁴ techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

²⁵ github.com/Netflix/Hystrix

²⁶ github.com/App-vNext/Polly

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 3.2, a modern application is much more dynamic.

Figure 3.2. 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.

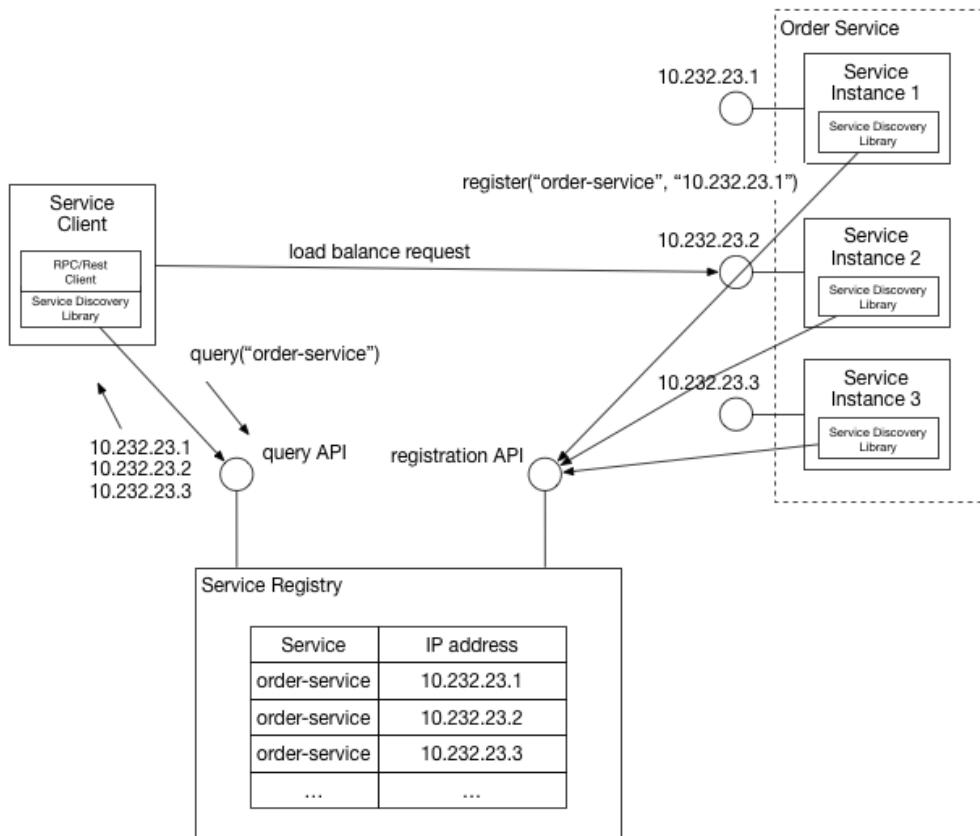
The service discovery mechanism updates the service registry when service instances start and stop. When a client invokes a service, the service discovery mechanism queries the service registry to obtain a list of available service instances and routes the request to one of them.

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 10, 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 3.3 shows how the services and the clients interact with the registry.

Figure 3.3. 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 describe in more detail in

chapter {chapter-prod-ready}. 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.

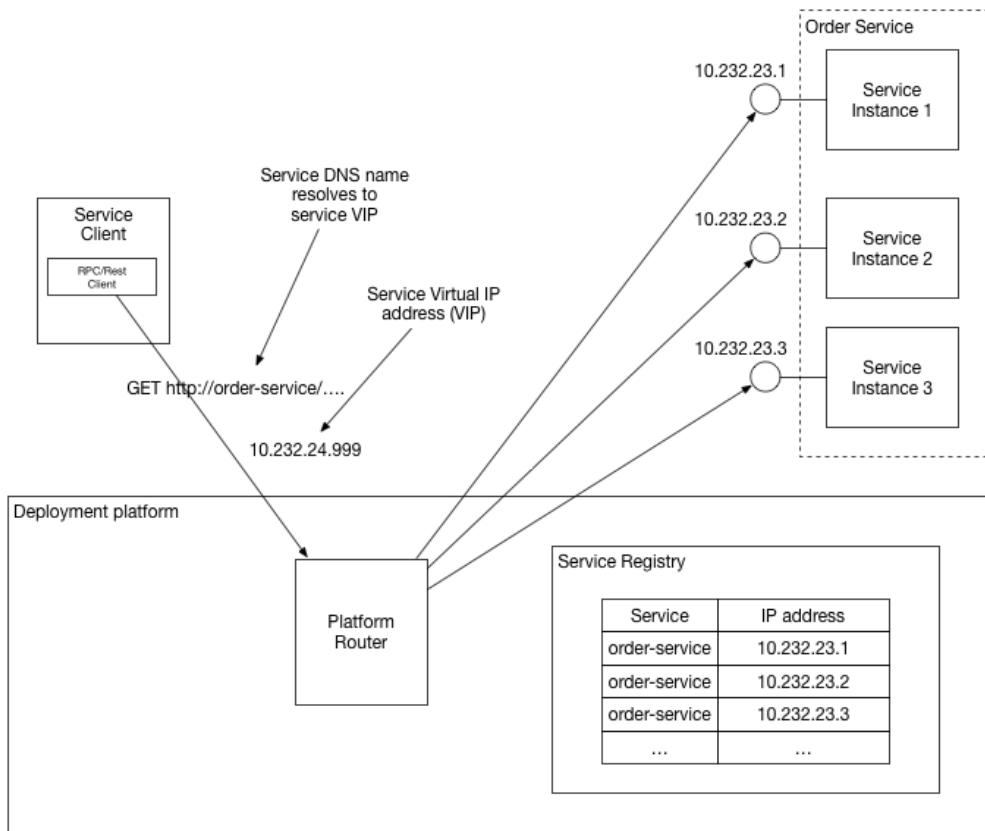
Application-level service discovery has been popularized by Netflix and Pivotal. Netflix developed and open-sourced several components: Eureka, which is a highly available service registry; the Eureka Java client; and Ribbon, which is a sophisticated HTTP client that supports the Eureka client. Pivotal 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 10 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 3.4 shows how this works.

Figure 3.4. The platform is responsible for service registration, discovery and request routing



The deployment platform includes a service registry, which tracks 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 message²⁷ consists of a header (a.k.a. metadata) and a message body. The header contains information such as:

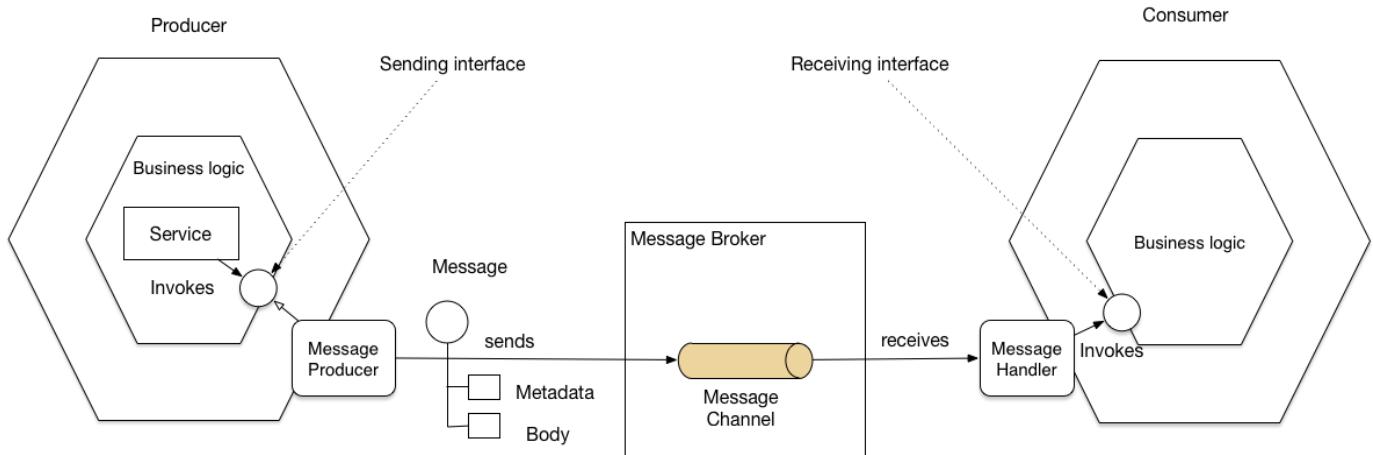
- a unique message id - generated by either the sender or the messaging infrastructure
- an optional return address - the name of the message channel that a reply should be written to

The body is the data being sent and has either a text or binary format. As figure 3.5 shows, messages are exchanged over channels²⁸. The business logic in the producer invokes a *sending interface*, which often encapsulates the underlying communication mechanism. The *sending interface* is implemented by a *message producer* adapter class, which sends a message to a consumer via a message channel. A message channel is an abstraction of the messaging mechanism provided by the message broker. A *message handler* adapter class in the producer is invoked to handle the message. It invokes a *receiving interface* implemented by the consumer's business logic.

²⁷ www.enterpriseintegrationpatterns.com/Message.html

²⁸ www.enterpriseintegrationpatterns.com/MessageChannel.html

Figure 3.5. The business logic in the producer invokes an API, which is implemented by message producer adapter. The message producer sends a message to a consumer via message channel, which is an abstraction of the messaging mechanism provided by the message broker. A message handler adapter in the producer is invoked to handle the message. It invokes an API implemented by the consumer's business logic.



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-point²⁹ and publish-subscribe³⁰. 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 a 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

²⁹ www.enterpriseintegrationpatterns.com/PointToPointChannel.html

³⁰ www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html

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 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 Kafka³¹, 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.

³¹ kafka.apache.org/

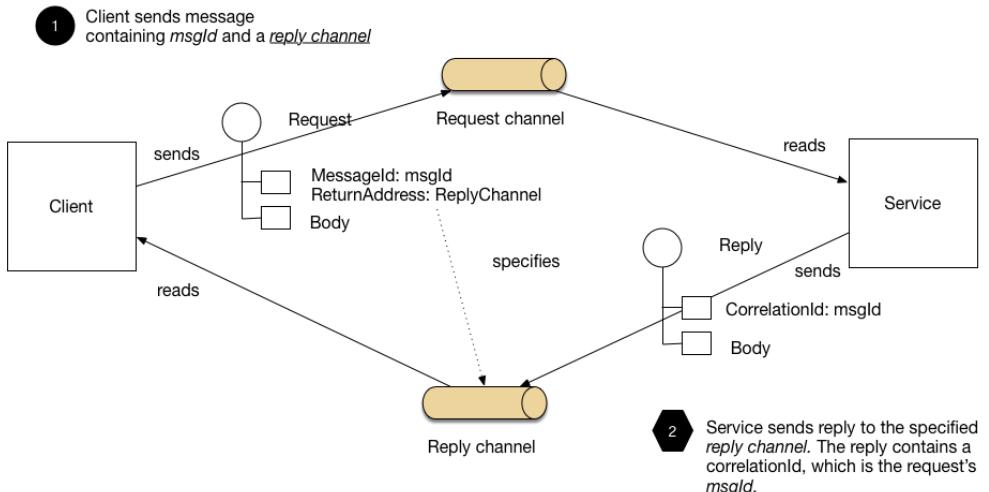
©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

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 3.6 shows how the client and the service interact.

Figure 3.6. 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

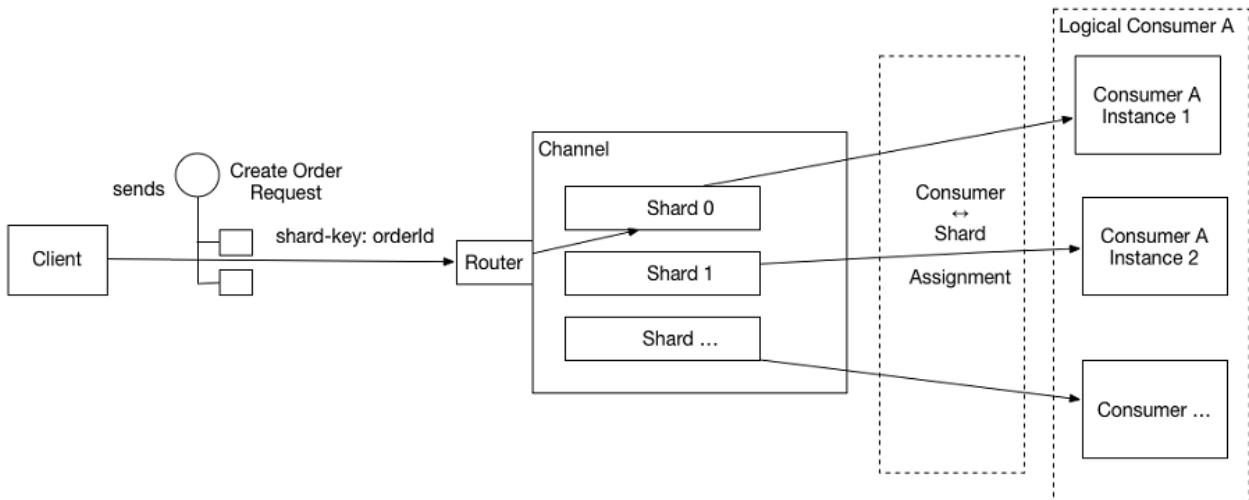
Another messaging design issue is scaling out a consumer. It is a common requirement to have multiple instances of a service into order to process messages concurrently. Moreover, even a single service instance will probably use threads to concurrently process multiple messages. Using multiple threads and service instances to concurrently process messages increases the throughput of the application. The challenge, however, with processing messages concurrently is ensuring that 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 3.7 shows how this works.

Figure 3.7. 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 Idempotent message consumers

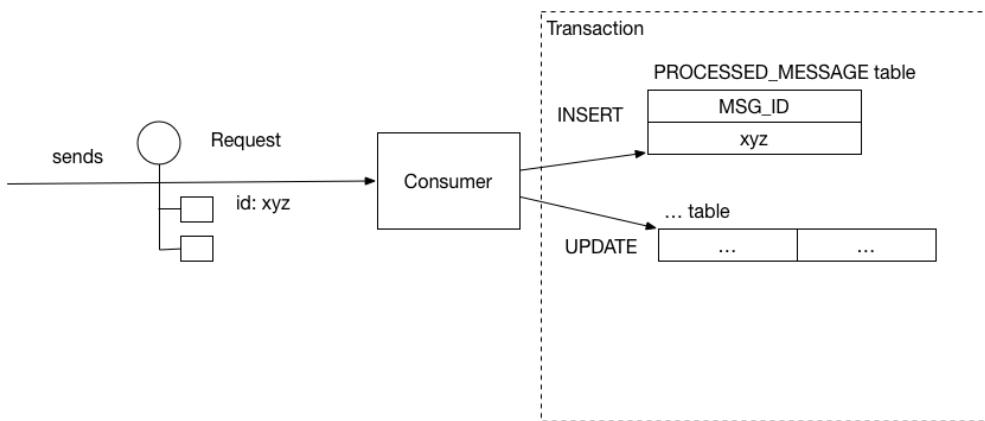
Another challenge with using a messaging system is dealing with duplicate messages. Ideally, a message broker should deliver a message only once but guaranteeing exactly once messaging is usually too costly. Instead, most message brokers promise to deliver at least once. When the system is working normally 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 message consumer must become idempotent by detecting and discarding 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 3.8 shows this approach.

Figure 3.8. 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_MESSAGE` 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

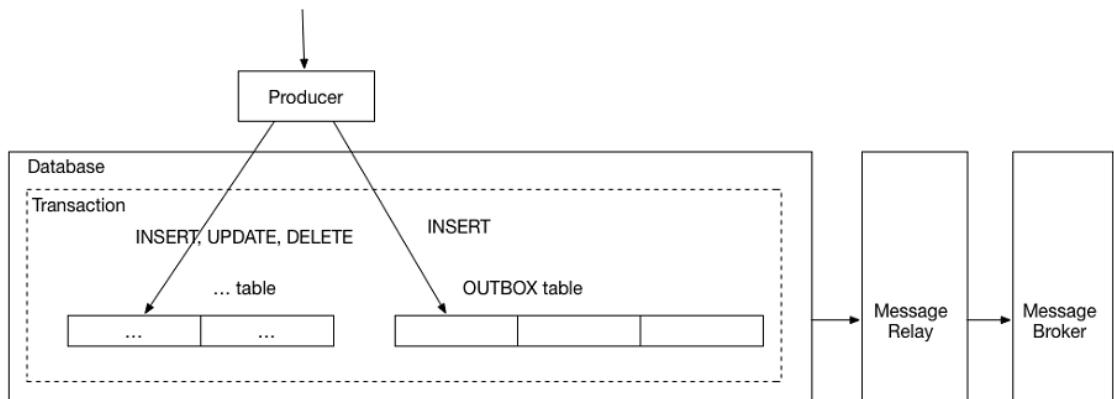
3.5 Using transactional messaging to atomically update the database and publish messages

As I mentioned earlier, an application must atomically update the database and publish messages. Without this guarantee, an application might, for example, crash after updating the database and before publishing a message, which would result in bugs. The challenge is how to send messages as part of a database transaction. For the reasons that I describe in chapter 4, modern applications cannot use distributed transactions to atomically update the database and publish messages. In this section, I describe an alternative approach that reliably publishes message without using distributed transactions. You will also learn about the Eventuate 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 3.9 shows how this works.

Figure 3.9. 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:

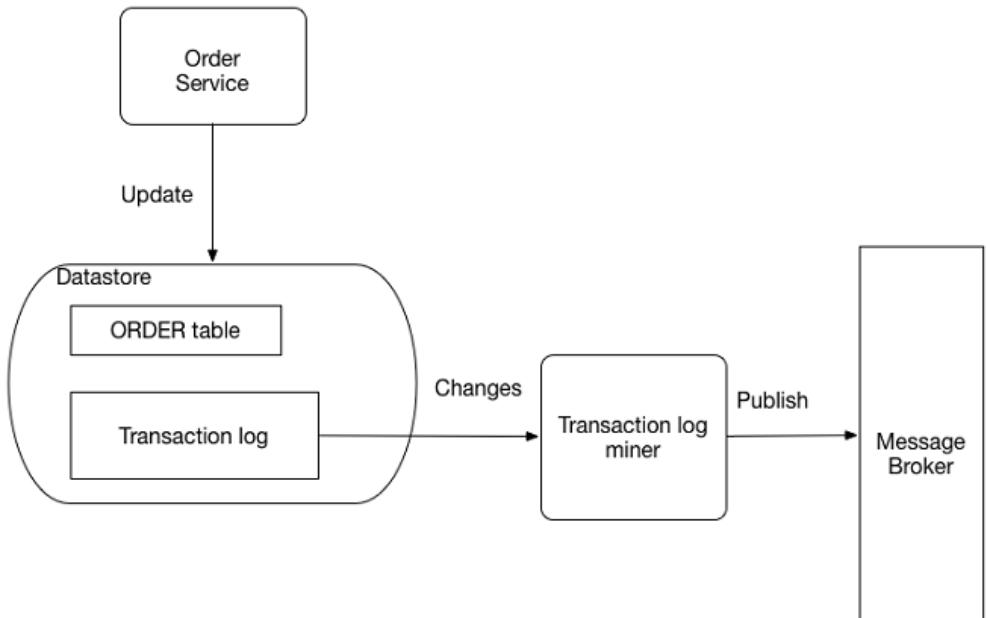
```
BEGIN
DELETE FROM OUTBOX WHERE ID in (....)
COMMIT
```

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 3.10 shows how this approach works.

Figure 3.10. 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 Databus³² - 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 streams³³] - 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 Tram³⁴ - my 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 let's now look at a framework that does this.

³² github.com/linkedin/databus

³³ docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html

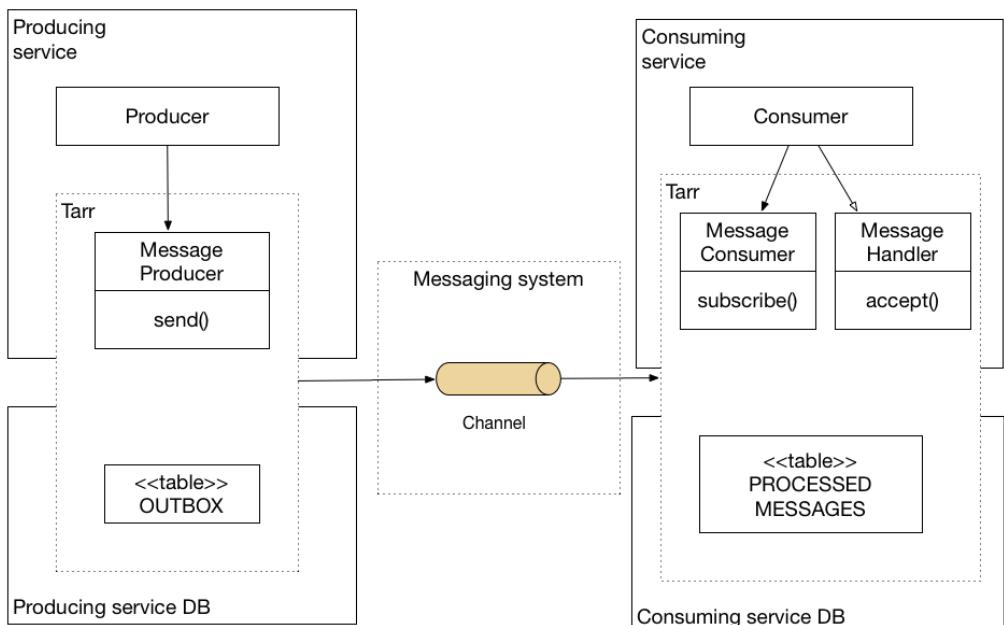
³⁴ github.com/eventuate-tram/eventuate-tram-core

3.5.2 Transactional messaging using the Eventuate Tram framework

Transaction log tailing is an excellent way to atomically publish events as part of a database transaction. In theory, you could implement transaction tailing in your application. However, using a transaction messaging framework enables you to focus on your application's business logic. Also, a good way to learn about a technology is to study a framework.

Eventuate Tram is an example of a framework that implements transactional messaging for Java applications that use a relational database. Figure 3.11 shows the design of the framework's core. It provides an API for sending and receiving messages as part of a JDBC transaction.

Figure 3.11. The Eventuate Tram framework enables a service to publish and consume messages as part of a database transaction



The Eventuate 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 to publish messages to message channels. A consumer service uses the `MessageConsumer` interface to subscribe to messages.

The Eventuate Tram framework's two main database tables are the `OUTBOX` and `PROCESSED_MESSAGE` tables. It uses the `OUTBOX` table to reliably publish messages 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.

```
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 to
- `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 Eventuate 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 Eventuate 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 Eventuate Tram framework invokes the `MessageHandler`. If a failure occurs at any stage, the message broker redelivers the message to Eventuate Tram framework.

Publishing and subscribing to domain events

The Eventuate 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 5. 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 Eventuate 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 `MessageProducer` 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 Eventuate 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 5. 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 Eventuate Tram framework also implements request/reply, which I described earlier in section "["Implementing request/reply"](#)". 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 Eventuate Tram framework's `CommandDispatcher` class which dispatches command messages to handlers

```
public class CommandDispatcher {

    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 4. It defines a `commandHandlers()` method, which returns a `CommandHandlers` that maps command types to the corresponding handler method.

Listing 3.6.

```
public class OrderCommandHandlers {

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

    public Message approveOrder(CommandMessage<ApproveOrderCommand> cm) {
        ApproveOrderCommand command = cm.getCommand();
        ...
    }
}
```

1

}

- 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 Eventuate Tram framework implements transactional messaging for Java applications. 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

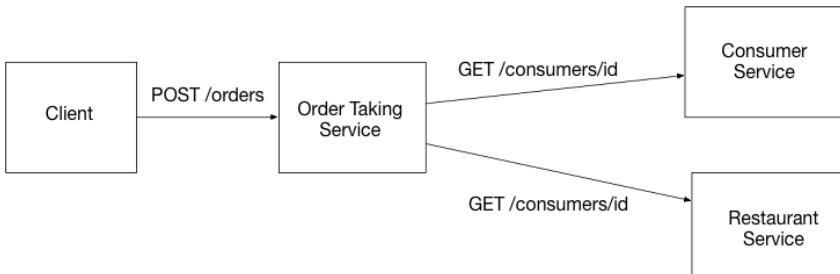
As you have just seen, there are a variety of IPC mechanisms with different trade-offs. One particular trade-off is how your choice of IPC mechanism impacts availability. In this section, you will learn that synchronous communication with other services as part of request handling reduces application 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 consider the scenario shown in figure 3.12. The Order Service has a REST API for creating an Order. It invokes the Consumer Service and the Restaurant Service to validate the Order. Both of those services also have REST APIs.

Figure 3.12. 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 Service
2. Order Service retrieves consumer information by making an HTTP GET /consumers/id request to the Consumer Service
3. Order 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 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 service. This problem exists even if services communicate using request/reply style interaction over asynchronous messaging. For example, the availability of the Order 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.

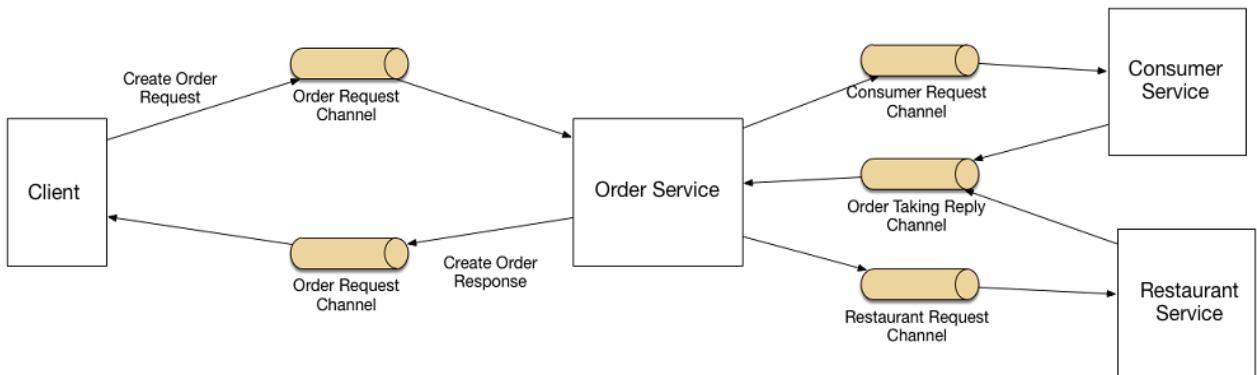
3.6.2 Eliminating synchronous interaction

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 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 Service. This service then asynchronously exchanges messages with other services and eventually sends a reply message to the client. Figure 3.13 shows the design.

Figure 3.13. The FTGO application has higher availability if its services communicate using asynchronous messaging instead of synchronous calls



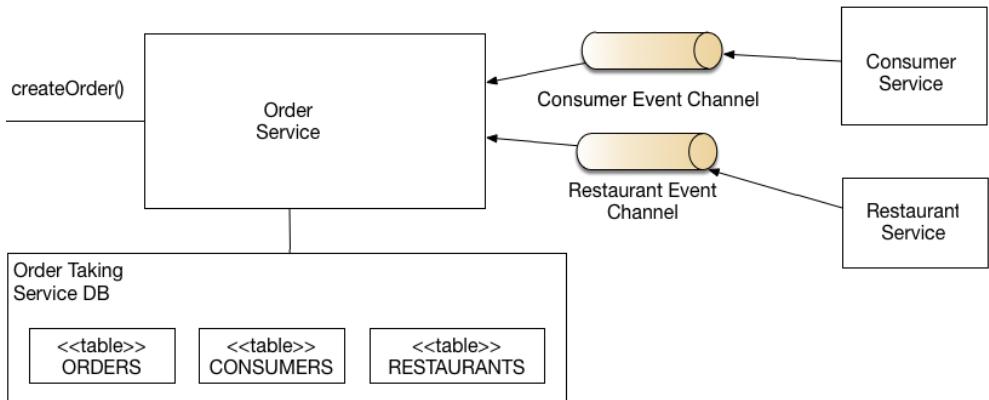
The client and the services communicate asynchronously by sending messages via messaging channels. No participant in this interaction is ever blocked waiting for a response.

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 use 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 by the services that own the data it is updated. For example, the Order Service could maintain a replica of data from the Consumer and Restaurant Service's. This enables the Order Service to validate a request and create an order without having to interact with other services. Figure 3.14 shows the design.

Figure 3.14. The Order Service is self-contained because it has replicas of the consumer and restaurant data



The Consumer and Restaurant Service's publish events whenever their data changes. The `Order 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 require the replication of large amounts of data, which is inefficient. For example, it is not practical for the Order Service to maintain a replica of the data owned by the Consumer and Restaurant Service's. 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. Let's look at how that works.

Finish processing after returning a response

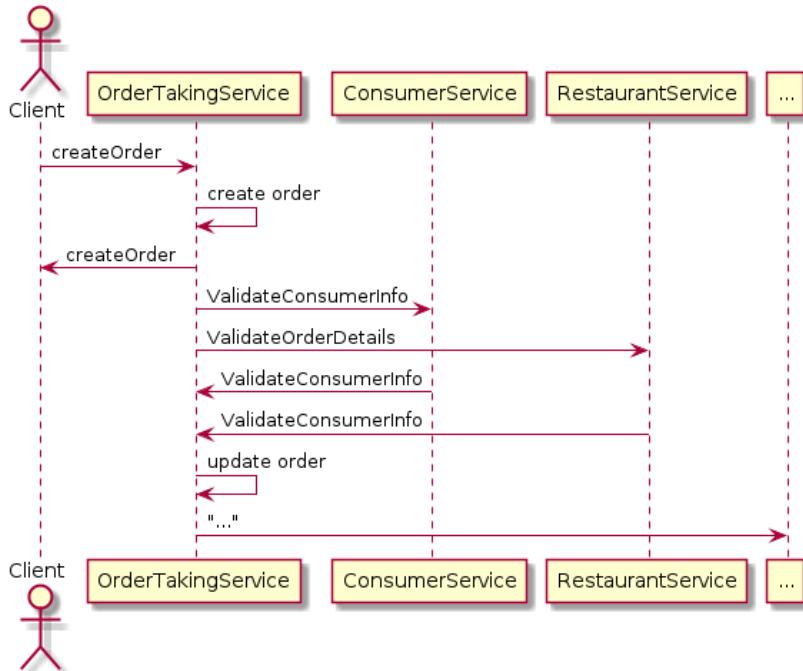
Another way to eliminate synchronous communication while during request processing is for a service to handle a request as follows:

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 Service used this approach then it creates an order in an "unvalidated" state and then validates the order asynchronously by exchanging messages with other services. Figure 3.15 shows what happens when the `createOrder()` operation is invoked.

Figure 3.15. 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 Service creates an Order in a NEEDS_VALIDATION state
2. The Order Service returns a response containing the order id
3. The Order Service sends a ValidateConsumerInfo message to the Consumer Service
4. The Order 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 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 OrderDetailsValidatedmessage to the Order Service
7. The Order Service receives the ConsumerValidated and OrderDetailsValidated and changes the state of the order to unauthorized
8. ...

After the Order has been validated, the Order 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 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 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 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

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 I 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 2 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 programmatic 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 2, 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)³⁵. XA uses two-phase commit (2PC) to ensure that all

³⁵ en.wikipedia.org/wiki/X/Open_XA

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

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 is 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 theorem³⁶, 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

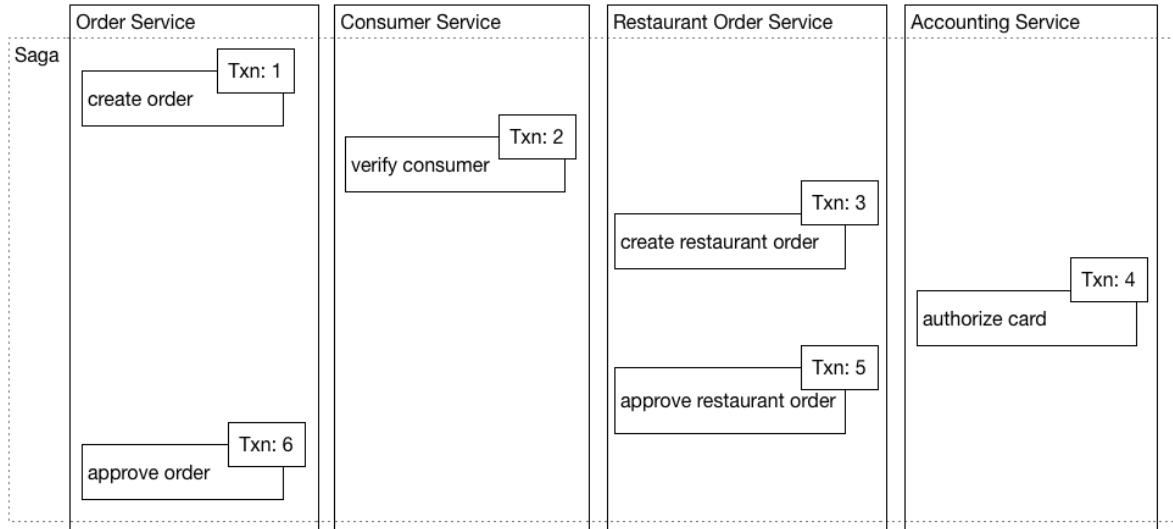
Sagas are a mechanism for maintaining data consistency that avoids the problems of distributed transactions. A saga is a sequence of local transactions. Each local transaction updates data within a single service. It uses the familiar ACID transaction frameworks and libraries mentioned earlier. The local 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

³⁶ 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 `PENDING` 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.

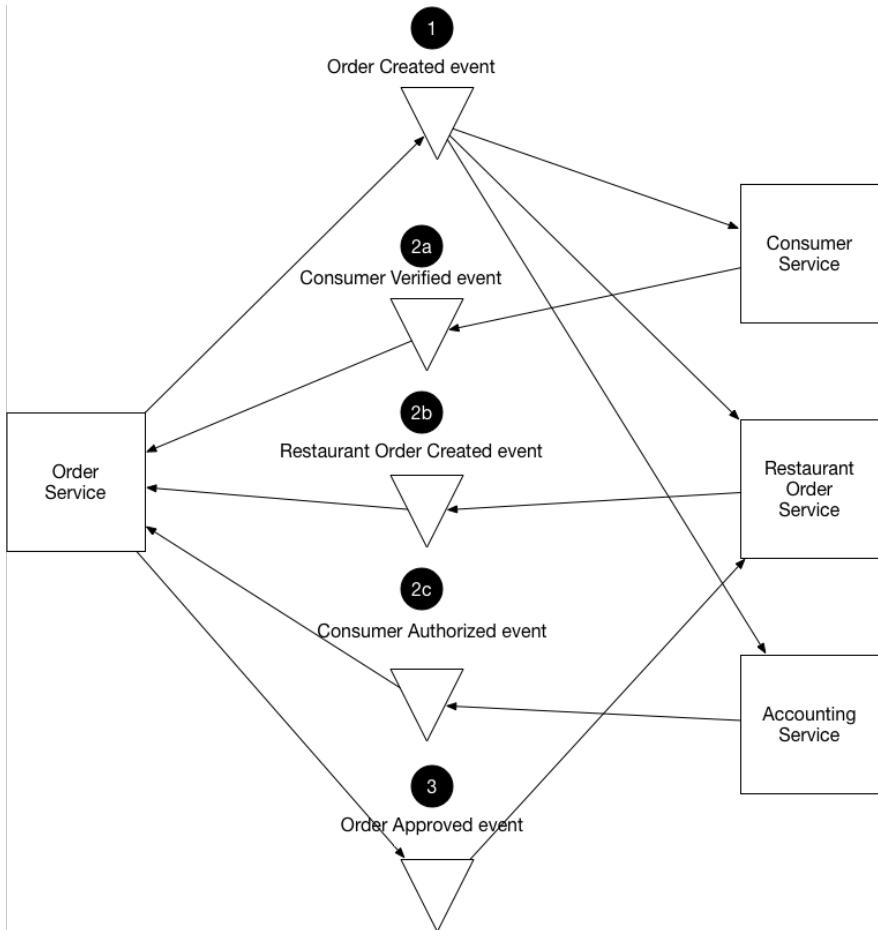
4.2.1 Choreography-based sagas

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:

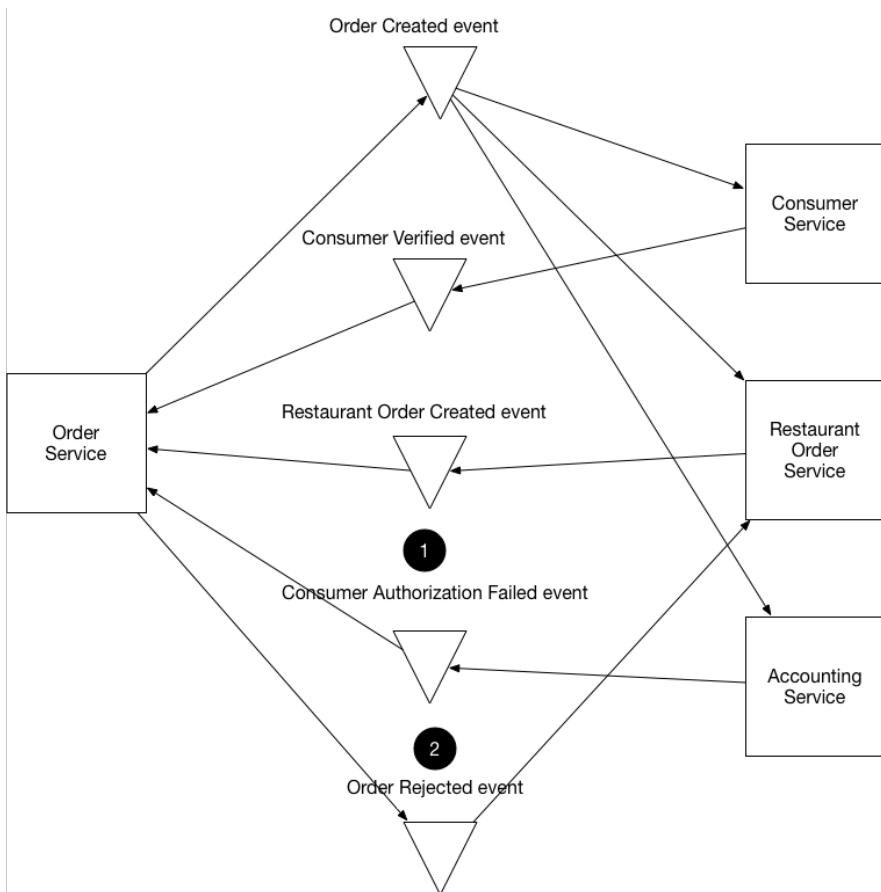
1. The `OrderService` creates an `Order` in the `CREATE_PENDING` state and publishes an `OrderCreated` event.
2. The `OrderCreated` event is received by the following services:
 - a. The `ConsumerService`, which verifies that the consumer can place the order, and publishes `ConsumerVerified` event
 - 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. The sequence of events in the CreateOrder saga when the authorization of the consumer's credit card fails.



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 Createdevent, 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 Eventuate 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.

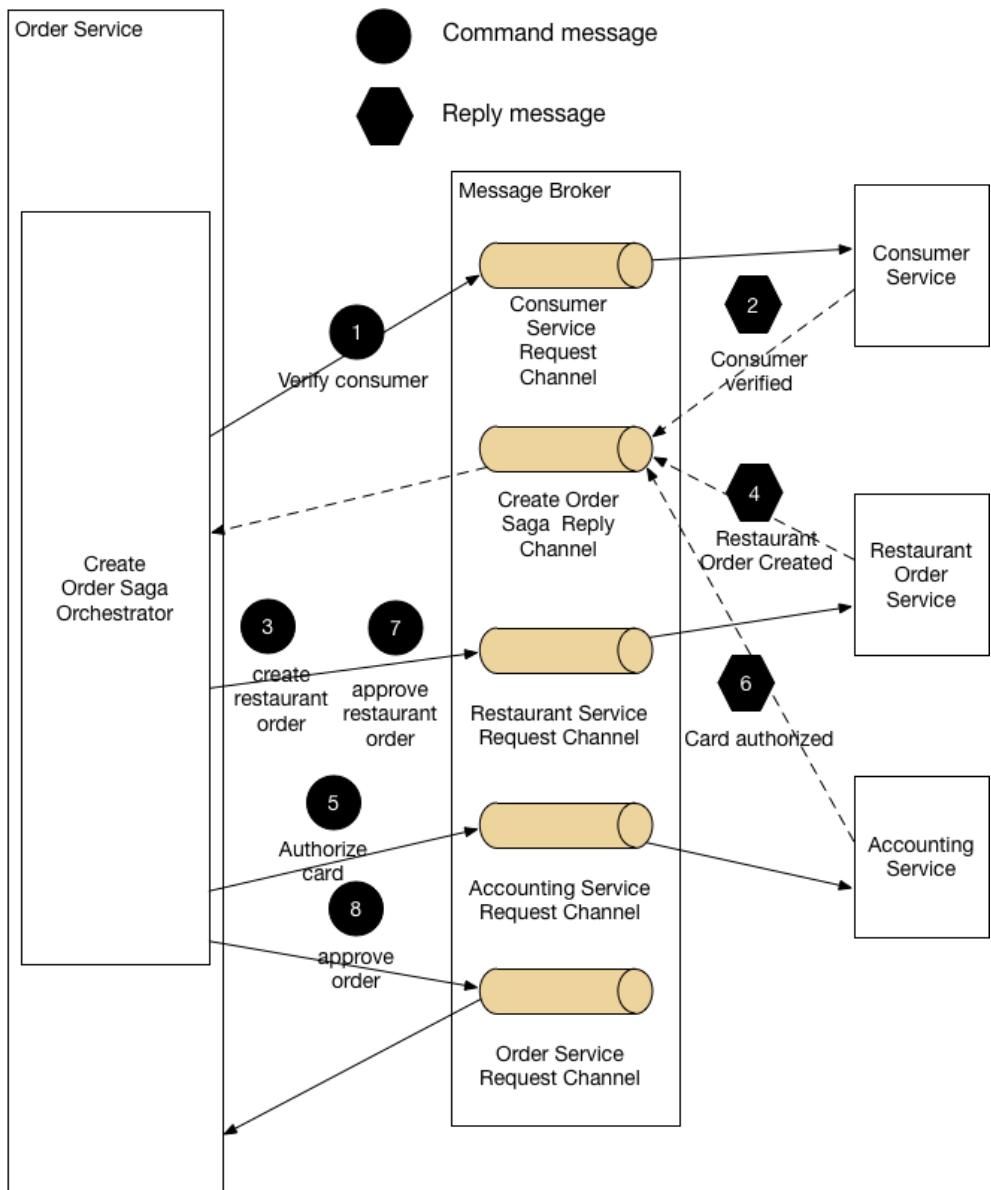
4.2.2 Orchestration-based sagas

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.

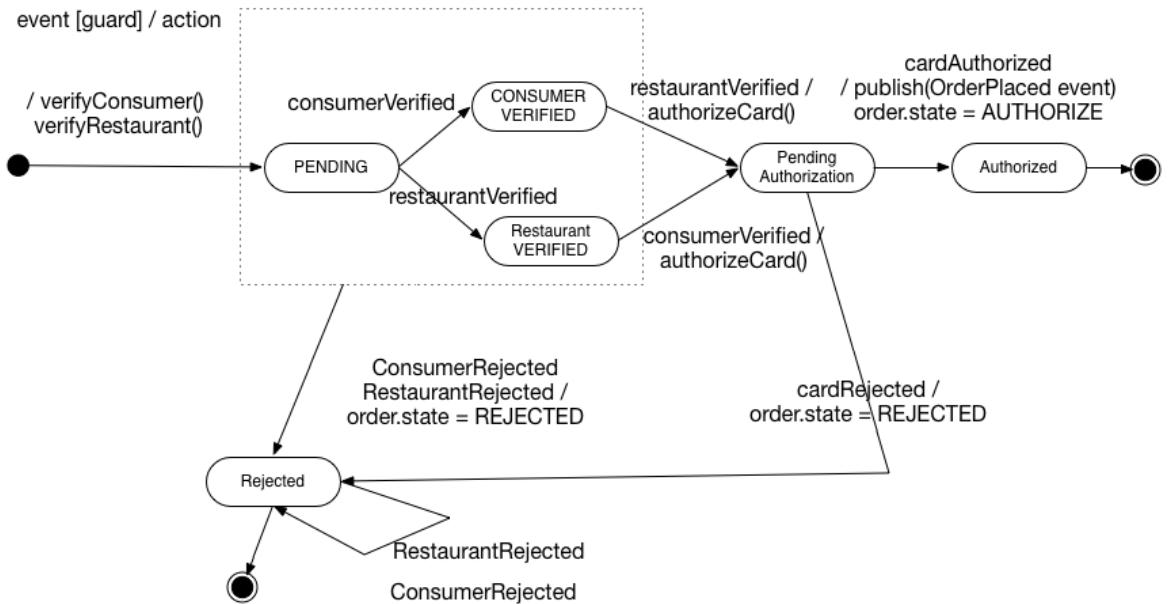
Diagrams such as figure 4.4 each depict one scenario for a saga. However, a saga is likely to have numerous scenarios. For example, the Create Order saga has four scenarios. In addition to the happy path, the saga can fail due to an failure in the Consumer Service, Restaurant Order Service or Accounting 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 3, an application cannot update a database and send a message using 2PC. Instead, sagas must use an approach such as the one provided by the Eventuate Tram framework. The Eventuate 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 Eventuate 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 Eventuate 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 Eventuate Tram framework reads the request from the participant's command channel. Next, it begins a transaction. Within the transaction, the Eventuate 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 Eventuate 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 as 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 Eventuate Tram framework to process reply messages from saga participants. It executes a message processing loop that reads 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 Eventuate Tram framework invokes a message broker API to reads the message from the saga orchestrator's channel. Next, within a transaction, the Eventuate 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 Eventuate Tram framework inserts the message into the OUTBOX table. Finally, Eventuate 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 $(n + 1)^{\text{th}}$ 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 $T_1, \dots, T_n, C_n, \dots, C_1$ as is shown in figure 4.6. In this example, T_{n+1} fails, which requires steps T_1, \dots, 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.

T_1	\dots	T_n	T_{n+1} FAILS	C_n	\dots	C_1
-------	---------	-------	--------------------	-------	---------	-------

The saga executes the compensation transactions in reverse order of the forward transactions: C_n, \dots, C_1 . The mechanics of sequencing the C_i s is not really any different than sequencing the T_i s. 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 to 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.

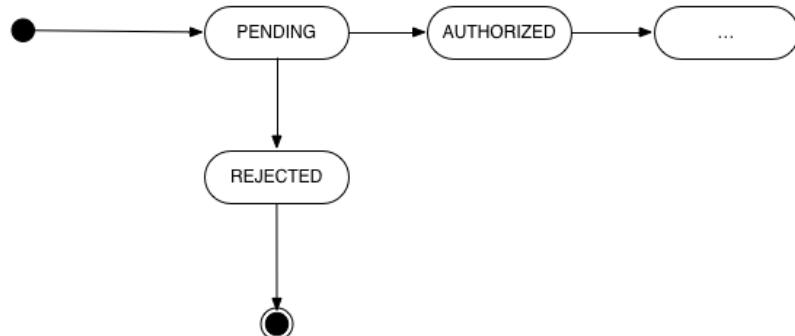
Figure 4.7. The state model for an Order in a monolithic application



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.



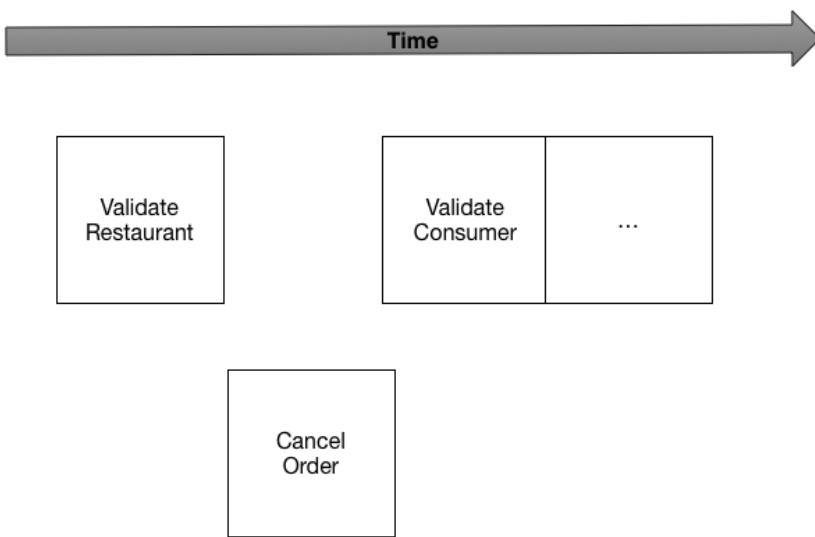
The CREATE_PENDING state is a state that does not exist (or is at least not visible) in a

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 `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 somehow 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`.

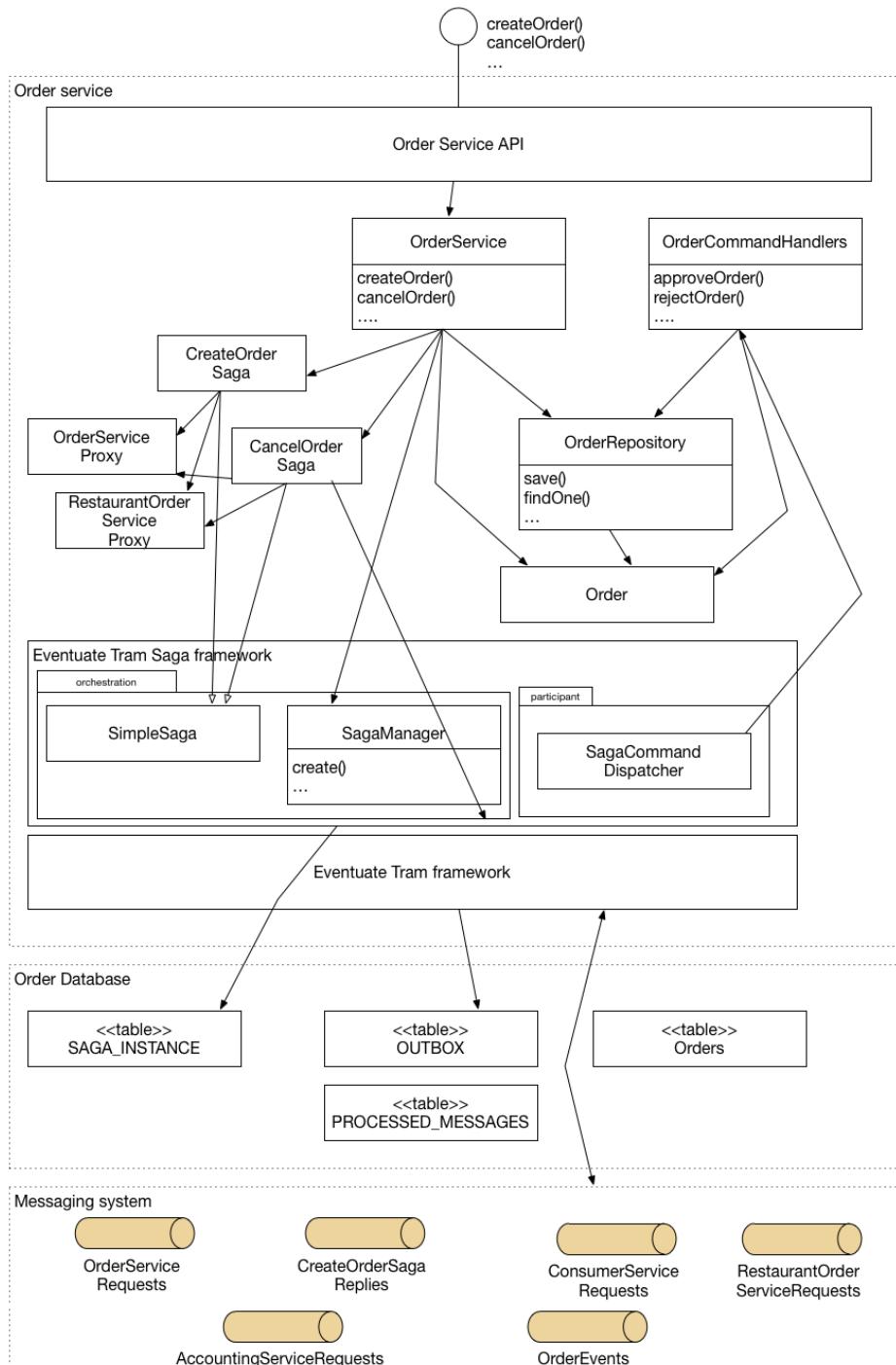
This approach has several benefits and drawbacks. One benefit is that it ensures that the sagas are decoupled from one another. The `Create Order` and `Cancel Order` sagas have no knowledge of each another. Another benefit is that effectively hides any intermediate states. The `Cancel Order` saga is 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. An application must 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



©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

Licensed to Joydeep Ghosh <joydeep.ghosh1977@outlook.com>

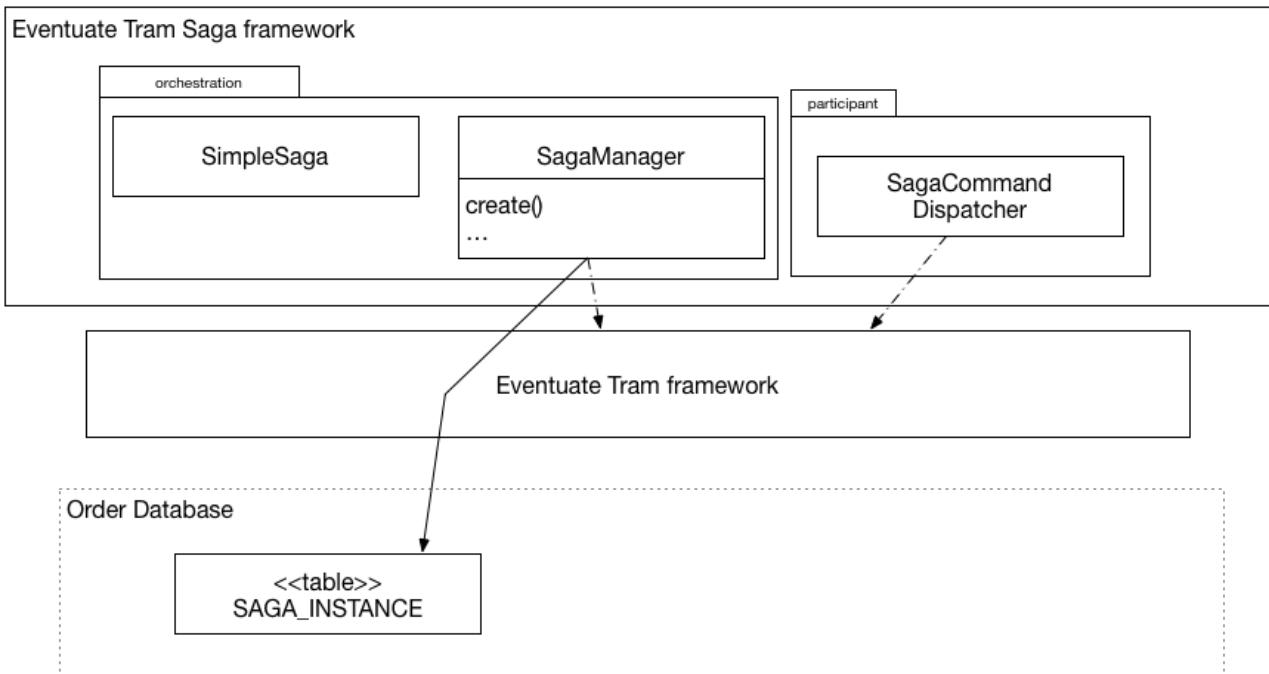
Some parts of the Order Service have a familiar design. Just as in a traditional application, the core of the business logic is implemented by the `OrderService`, `Order`, and `OrderRepository` classes. In this chapter, I am only going to briefly describe these classes. I describe them in more detail in chapter 5.

What's less familiar about the Order Service are the saga-related classes. The `OrderService` defines several saga orchestrators including `CreateOrderSaga` and `CancelOrderSaga`. There are also various proxy classes, such as `RestaurantOrderServiceProxy` and `OrderServiceProxy`, which define the messaging APIs of the services that participate in the sagas. They are used by the sagas to route messages to participants. The `OrderService` also defines a `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 Eventuate Tram saga framework.

4.4.1 About the Eventuate Tram saga framework

The Order Service's sagas are implemented using the Eventuate Tram saga framework. The framework has two main packages, which are shown in figure 4.11. The `orchestration` package provides classes for writing saga orchestrator, and the `participant` package provides classes for writing saga participants.

Figure 4.11. The Eventuate Tram Saga framework provides classes for writing saga orchestrators and 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;

    public CreateOrderSaga(OrderServiceProxy orderService,
                          ConsumerServiceProxy consumerService,
                          RestaurantOrderServiceProxy restaurantOrderService,
                          AccountingServiceProxy accountingService) {
        this.sagaDefinition =
            step()
                .withCompensation(orderService.reject,
                                   this::makeRejectOrderCommand)
            .step()
                .invokeParticipant(consumerService.validateOrder,
                                   this::makeValidateOrderByConsumer)
            .step()
                .invokeParticipant(restaurantOrderService.create,
                                   this::makeCreateRestaurantOrderCommand)
            ...
    }

    @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 two parameters. The first is a `CommandEndpoint`, which represents a destination that can be sent a command message. `CommandEndpoint` is a generic class parameterized by the command type and so enforces static typing of command messages. It specifies the the

channel to send the command to and the expected reply types. The second is a Function that takes the saga's data as a parameter and returns a Command of the type specified by `CommandDestination`. The `SagaManager` invokes the function to get the command message, and sends it to the destination specified by the `CommandEndpoint`.

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 Eventuate 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.

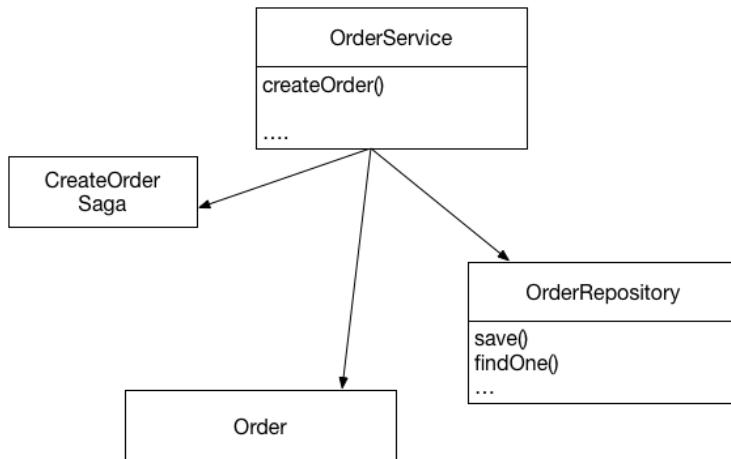
The Saga participants package

The participant package provides an API for writing saga participants. It builds on the Eventuate 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 domain objects. Let's look at the saga framework in action and see an example of a saga orchestrator and participant.

4.4.2 The OrderService class

The `OrderService` class is a domain service, which is called by the service's API layer. It's responsible for creating and managing orders. Figure 4.12 shows the `OrderService` and some of its collaborators. The `OrderService` creates and updates `Orders`, invokes the `OrderRepository` to persist `Orders` and creates sagas, including a `CreateOrderSaga`.

Figure 4.12. The OrderService creates and updates Orders, invokes the OrderRepository to persist Orders and creates sagas, including a CreateOrderSaga.



I'll describe this class in more detail in chapter 5. For now I'll focus on the `createOrder()` method. Listing 4.2 shows the `OrderService`'s `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

```

@FunctionalInterface
public interface OrderService {
    Order createOrder(OrderDetails orderDetails);
}

@ValueObject
public class Order {
    private Long id;
    private String name;
    private String address;
    private String zipCode;
    private String city;
    private String country;
    private Double total;
    private Set lineItems;

    public static Order createOrder(OrderDetails orderDetails) {
        return new Order(
            null,
            orderDetails.getName(),
            orderDetails.getAddress(),
            orderDetails.getZipCode(),
            orderDetails.getCity(),
            orderDetails.getCountry(),
            orderDetails.getTotal(),
            orderDetails.getLineItems()
        );
    }
}

@ValueObject
public class OrderDetails {
    private String name;
    private String address;
    private String zipCode;
    private String city;
    private String country;
    private Double total;
    private List<LineItem> lineItems;
}

@ValueObject
public class LineItem {
    private String name;
    private Double price;
    private Integer quantity;
}

@ValueObject
public class CreateOrderSagaData {
    private Long orderId;
    private OrderDetails orderDetails;
}

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

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

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

        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 Eventuate Tram saga framework.

Listing 4.3. The definition of the `CreateOrderSaga`

```

public class CreateOrderSaga implements SimpleSaga<CreateOrderSagaData> {

    private SagaDefinition<CreateOrderSagaData> sagaDefinition;

    public CreateOrderSaga(OrderServiceProxy orderService,
                          ConsumerServiceProxy consumerService,
                          RestaurantOrderServiceProxy restaurantOrderService,
                          AccountingServiceProxy accountingService) {
        this.sagaDefinition =
            step()
                .withCompensation(orderService.reject,
                                   this::makeRejectOrderCommand)
            .step()
                .invokeParticipant(consumerService.validateOrder,
                                   this::makeValidateOrderByConsumer)
            .step()
                .invokeParticipant(restaurantOrderService.create,
                                   this::makeCreateRestaurantOrderCommand)
                .onReply(CreateRestaurantOrderReply.class,
                        this::handleCreateRestaurantOrderReply)
                .withCompensation(restaurantOrderService.cancel,
                                   this::makeCancelCreateRestaurantOrder)
            .step()
                .invokeParticipant(accountingService.authorize,
                                   this::makeAuthorizeCommand)
    }
}
```

```

        .step()
            .invokeParticipant(restaurantOrderService.confirmCreate,
                                this::makeConfirmCreateRestaurantOrder)
        .step()
            .invokeParticipant(orderService.approve,
                                this::makeApproveOrderCommand)
        .build();
    }

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

```

The CreateOrderSaga's constructor creates the saga definition and stores it in the `sagaDefinition` field.

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 ...

public CreateOrderSaga(..., RestaurantOrderServiceProxy restaurantOrderService,
                      ...) {
    ...
    .step()
        .invokeParticipant(restaurantOrderService.create,          1
                            this::makeCreateRestaurantOrderCommand)
        .onReply(CreateRestaurantOrderReply.class,                 2
                this::handleCreateRestaurantOrderReply)
        .withCompensation(restaurantOrderService.cancel,          3
                          this::makeCancelCreateRestaurantOrder)

    ...
}

private CreateRestaurantOrder
makeCreateRestaurantOrderCommand(CreateOrderSagaData data) {
    return new CreateRestaurantOrder(data.getOrderDetails().getRestaurantId(),
                                    data.getOrderId(),
                                    makeRestaurantOrderDetails(data.getOrderDetails()));
}

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

private CancelCreateRestaurantOrder
makeCancelCreateRestaurantOrder(CreateOrderSagaData data) {

```

```
    return new CancelCreateRestaurantOrder(data.getOrderId());
}
```

- 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 that the saga must create the command message for the forward transaction by calling `makeCreateRestaurantOrderCommand()` and send it to the channel specified by `restaurantOrderService.create`. The `makeCreateRestaurantOrderCommand()` method creates a `CreateRestaurantOrder`. Similarly, the call to `withCompensation()` specifies that the saga must create a command message for the compensating transaction by calling `makeCancelCreateRestaurantOrder()` and send it to the channel specified by `restaurantOrderService.create`. The `makeCancelCreateRestaurantOrder()` 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.

The `RestaurantOrderServiceProxy`, which is shown in listing 4.5 defines the command message endpoints for the Restaurant Order Service. There are three endpoints: `create`, which creates a `RestaurantOrder`; `confirmCreate`, which confirms the creation; and `cancel`, which cancels a `RestaurantOrder`.

Listing 4.5. RestaurantOrderServiceProxy defines the command message endpoints for the Restaurant Order Service

```
public class RestaurantOrderServiceProxy {

    public final CommandEndpoint<CreateRestaurantOrder> create =
        CommandEndpointBuilder
            .forCommand(CreateRestaurantOrder.class)
            .withChannel(
                RestaurantOrderServiceChannels.restaurantOrderServiceChannel)
            .withReply(CreateRestaurantOrderReply.class)
            .build();

    public final CommandEndpoint<ConfirmCreateRestaurantOrder> confirmCreate =
        CommandEndpointBuilder
            .forCommand(ConfirmCreateRestaurantOrder.class)
            .withChannel(
                RestaurantOrderServiceChannels.restaurantOrderServiceChannel)
            .withReply(Success.class)
            .build();
    public final CommandEndpoint<CancelCreateRestaurantOrder> cancel =
        CommandEndpointBuilder
            .forCommand(CancelCreateRestaurantOrder.class)
```

```

        .withChannel(
            RestaurantOrderServiceChannels.restaurantOrderServiceChannel)
        .withReply(Success.class)
        .build();

}

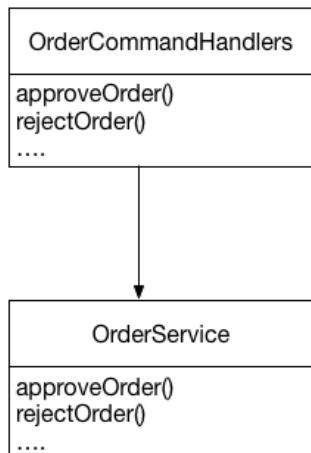
```

Each `CommandEndpoint` specifies the command type, the channel to send the command message to and the expected reply types. Proxy classes, such as `RestaurantOrderServiceProxy`, are not strictly necessary. A saga could simply send command messages directly. However, using proxy and `CommandEndpoint` classes have two important benefits. First, they implement static typing, which reduces the chance of a saga sending the wrong message to a service. Second, a proxy class is a well-defined API for invoking a service that makes the code easier to understand and test. For example, in chapter 9 I describe how to write tests for the `RestaurantOrderServiceProxy` that verify that the Order Service correctly invokes the Restaurant Order Service. Without the `RestaurantOrderServiceProxy`, it would be impossible to write such a narrowly scoped test.

4.4.4 *The OrderCommandHandlers class*

The Order Service participates in the sagas that it creates. For example, the `CreateOrderSaga` invokes the Order Service to either approve or reject an Order. The `OrderCommandHandlers` class, which is shown in figure 4.13, handles the command messages sent by these sagas. Each of its methods updates an Order, publishes domain events, and sends a reply message to the saga.

Figure 4.13. OrderCommandHandlers implements command handlers for the commands that are sent by the various Order Service sagas



The handler methods, which are shown in listing 4.6, are invoked by the Eventuate Tram framework, which implements transactional messaging. Each method invokes the `OrderService` to update an Order.

Listing 4.6.

```

public class OrderCommandHandlers {

    @Autowired
    private OrderService orderService;

    public CommandHandlers commandHandlers() {
        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().getOrderId();
        orderService.approveOrder(orderId);
        return withSuccess();
    } ②  
③

    public Message rejectOrder(CommandMessage<RejectOrderCommand> cm) {
        long orderId = cm.getCommand().getOrderId();
        orderService.rejectOrder(orderId);
        return withSuccess();
    } ④
}

```

- 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 by invoking the `OrderService`. 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 4.7 is an excerpt of this class.

Listing 4.7. 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());
}

@Bean
public RestaurantOrderServiceProxy restaurantOrderServiceProxy() {
    return new RestaurantOrderServiceProxy();
}

@Bean
public OrderServiceProxy orderServiceProxy() {
    return new OrderServiceProxy();
}

...
}

```

This class defines several Spring @Beans including `orderService`, `createOrderSagaManager`, `createOrderSaga`, `orderCommandHandlers` and `orderCommandHandlersDispatcher`. It also defines Spring @Beans for the various proxy classes including `restaurantOrderServiceProxy` and `orderServiceProxy`.

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



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 3, 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'être 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.

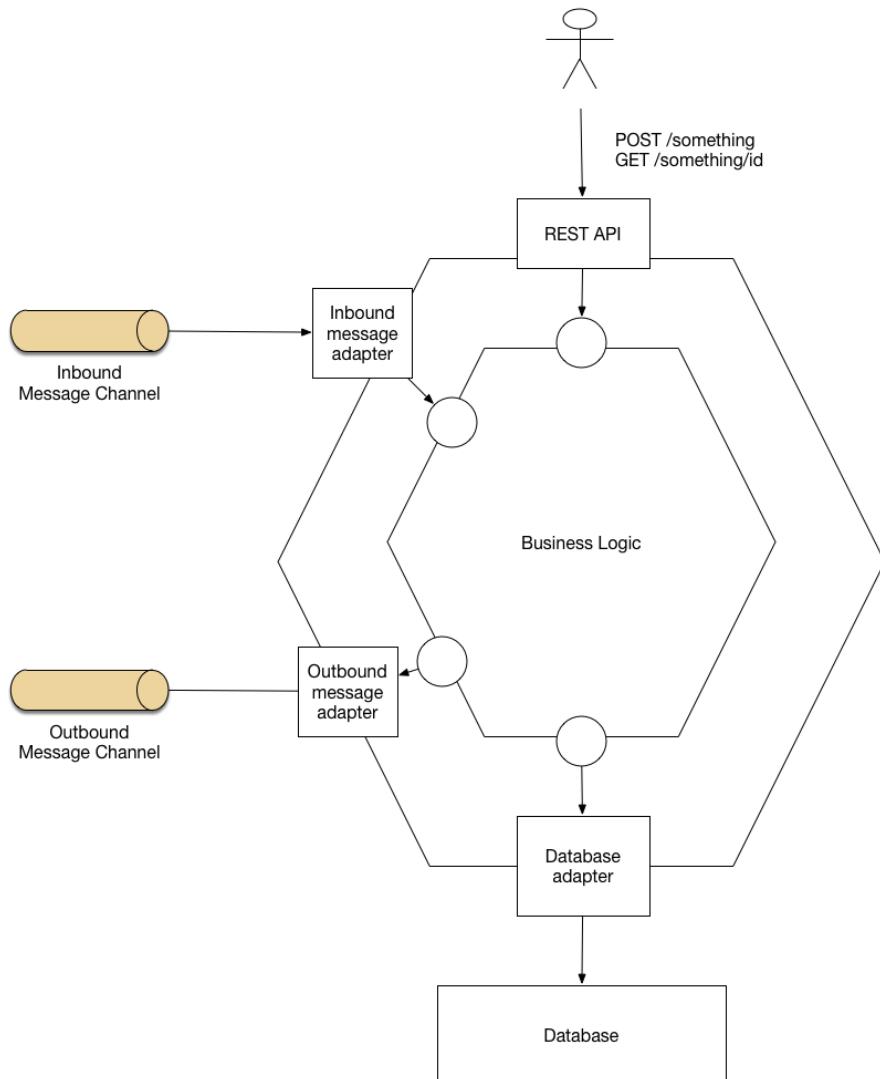
Developing complex business logic is always challenging. What's more, the microservice architecture presents some distinctive challenges. As I described in chapter 4, a microservices-based application often uses sagas to maintain consistency across multiple services. Furthermore, as you will learn below, 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. These messages are commands and notifications from clients and events published by other services.
- Database adapter, which accesses the database
- Outbound message adapter, publishes messages to a message broker. These messages are events, commands to invoke services, and replies to commands sent by clients.

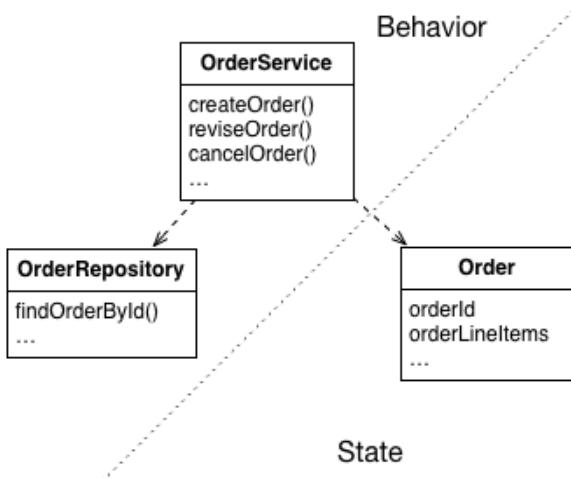
Sitting at the core of the service is the business logic, which is typically the most complex part of the service. It is invoked by the inbound adapters in response to requests from clients. The business logic typically invokes the outbound adapters to access the database and publish messages.

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.

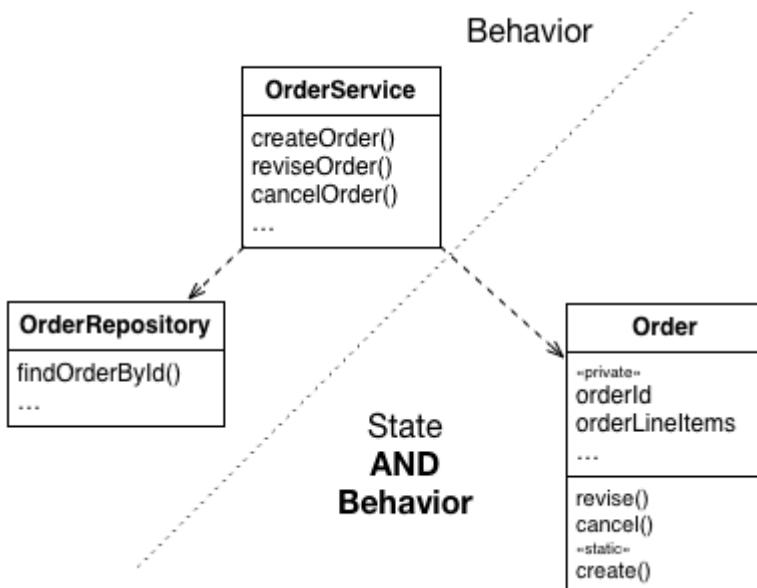
This style of design is highly procedural, and relies on few of the capabilities of object-oriented programming (OOP) languages. This what you would create if you were writing the application in C or another non-OOP language. Nevertheless, you should not be ashamed to use a procedural design when it 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 2, 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:

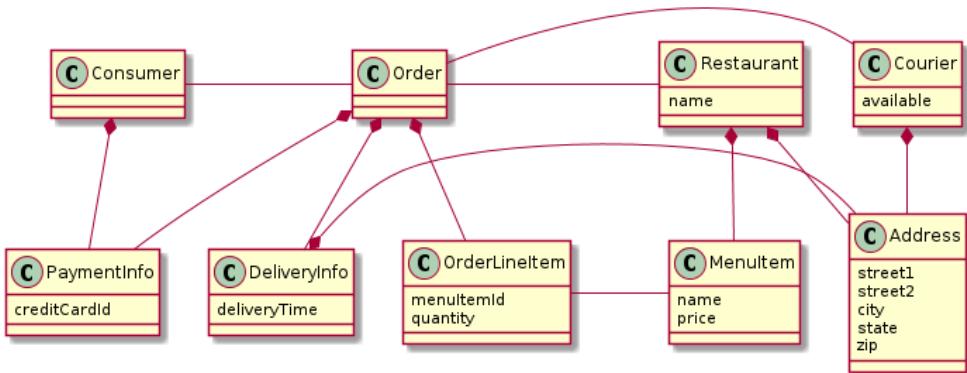
- Entity - an object that has a persistent identity. Two entities whose attributes have the same values are still different objects. In a Java EE application, classes that are persisted using JPA `@Entity` are usually DDD entities.
- Value object - an object that is a collection of values. Two value 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 5.4 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.

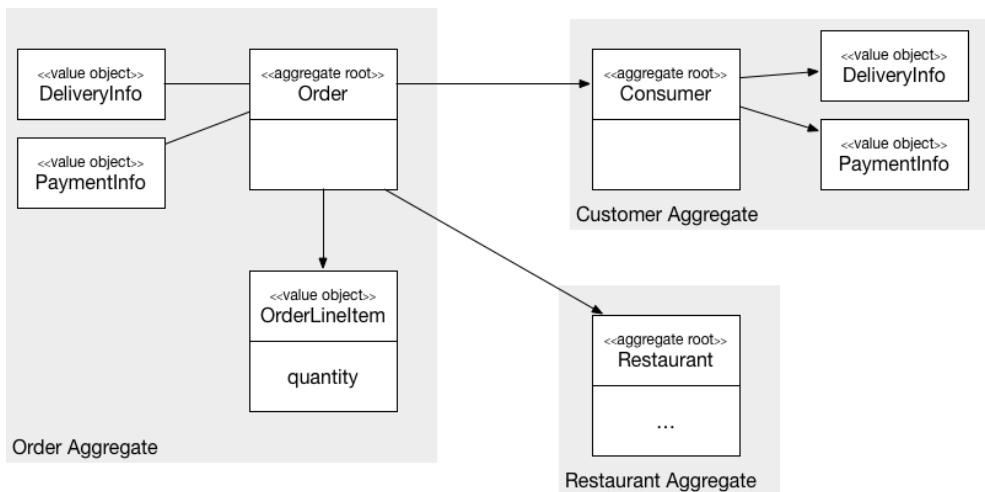
In this scenario, Sam reduces the order total by \$X and Mary reduces it by \$Y. As a result, the Order is no longer valid even though the application verified that the order was still satisfied the order minimum after each consumer's update. As you can see, directly updating part of a business object 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 2 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 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

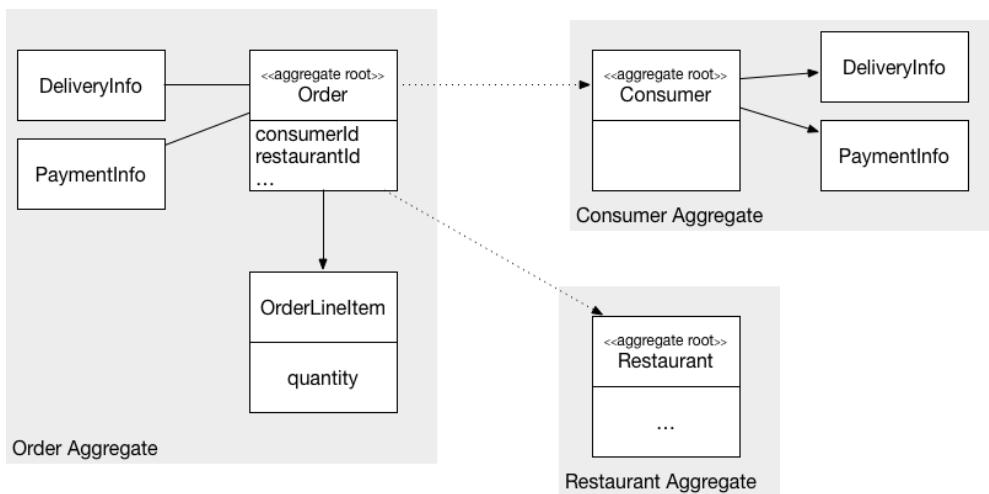
The example above illustrated the perils of directly updating `OrderLineItems` directly. The goal of the first aggregate rule is to eliminate this problem. It requires that the root entity is the only part of an aggregate that can be referenced by classes outside of the aggregate. A client can only update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

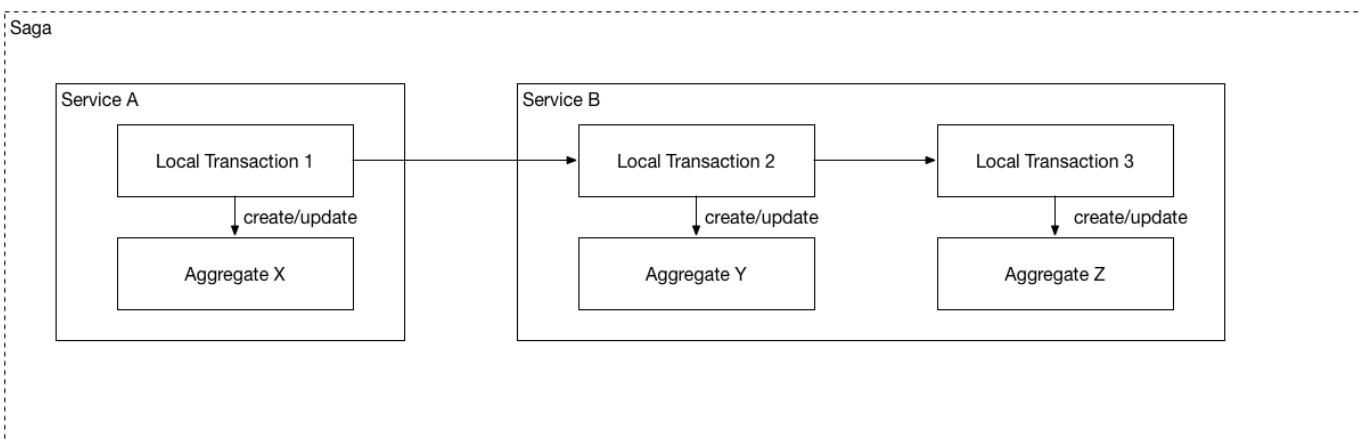
the database by sharding aggregates is relatively straightforward.

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 4, 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 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.

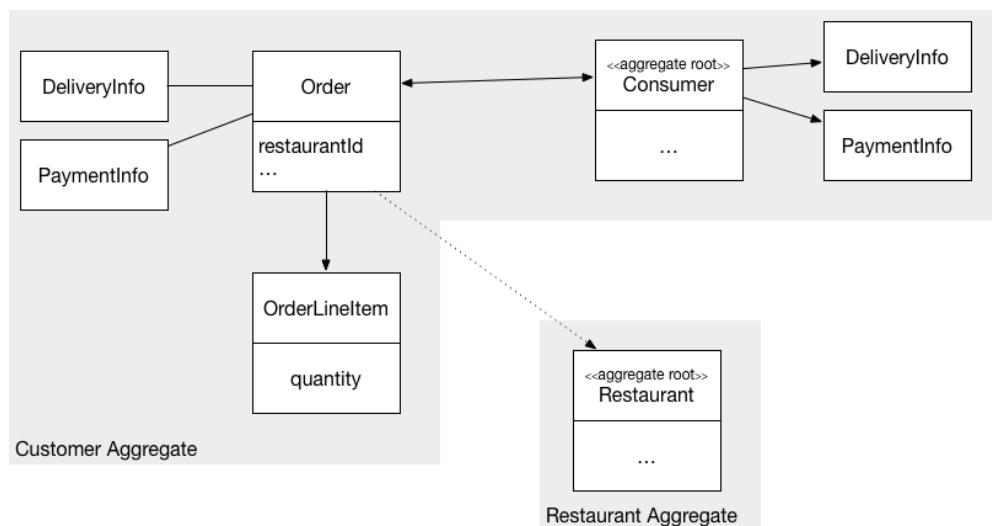
Or is there? It turns out that aggregate boundaries are not set in stone. When developing a domain model, you get to choose where the boundaries lie. But like a 20th century colonial power drawing national boundaries you need to be careful.

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.

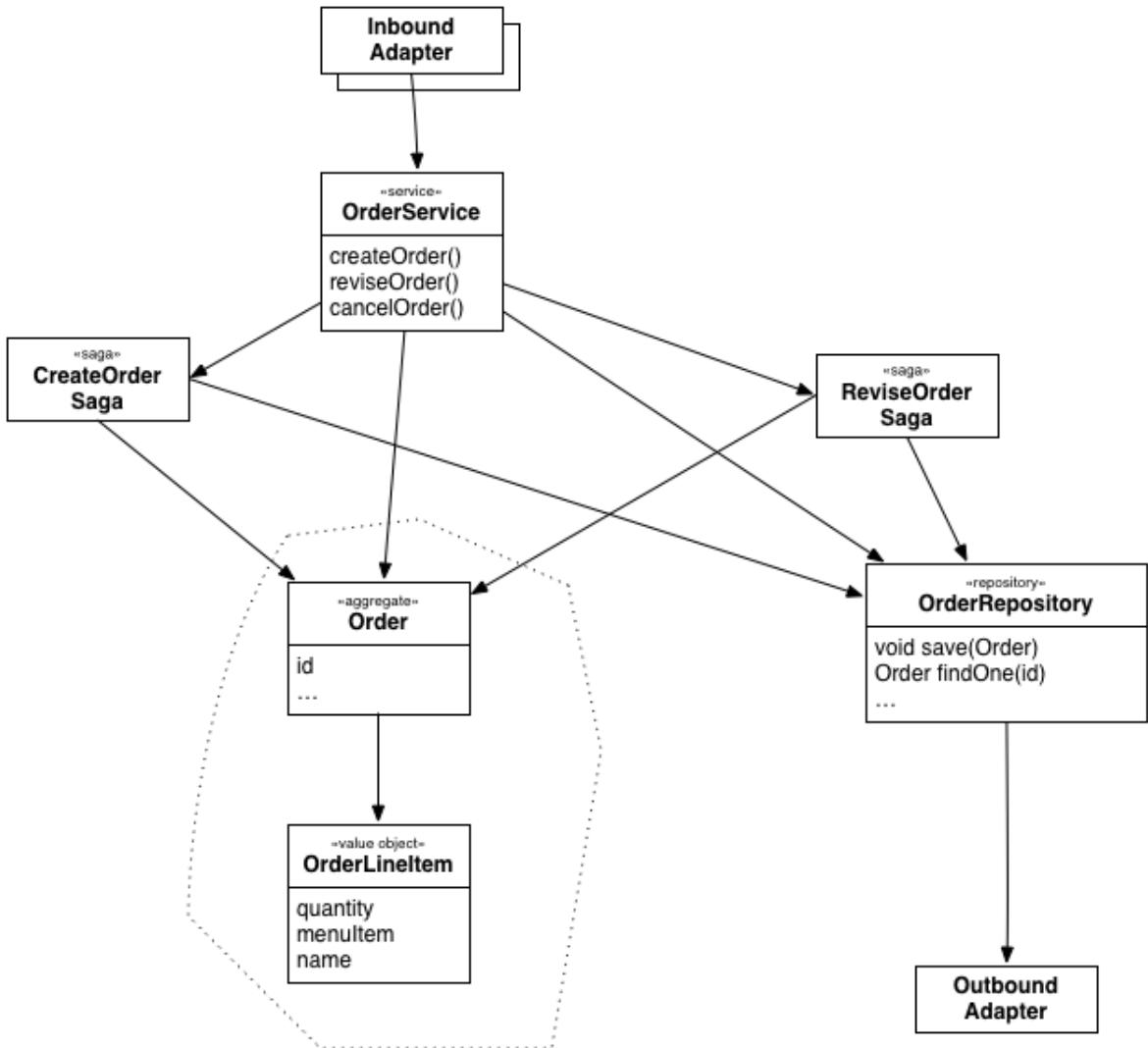
Another drawback of this approach in a microservice architecture is that it is an obstacle to decomposition. The business logic for `Orders` and `Consumers`, for example, must be collocated in the same service, which makes the service larger. Because of these issues, it is best to make aggregates as 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 4. 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 a 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. In a microservice architecture, it is common for a service to maintain a replica of data that is owned by a different service. Domain events are a convenient way to keep those replicas up to date. 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 {}

interface OrderDomainEvent extends DomainEvent {}

class OrderCreated implements OrderDomainEvent {}

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 domain event. `OrderDomainEvent` is a marker interface that identifies the `DomainEvent` as an event published by the `Order` aggregate. 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. The `OrderCreated` event class shown above captures the essence of what has happened. However, your event consumer might need the order details when processing an `OrderCreated` event. One option is for it to retrieve that information from the `OrderService`. The drawback of an event consumer querying the service for the aggregate is that it incurs the overhead of a service request.

An alternative approach known as event enrichment is for events to contain information that consumers need. It simplifies event consumers since they no longer need to request that data from the service that published the event. 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 OrderEvent {
    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 when processing an `OrderCreated` event.

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.

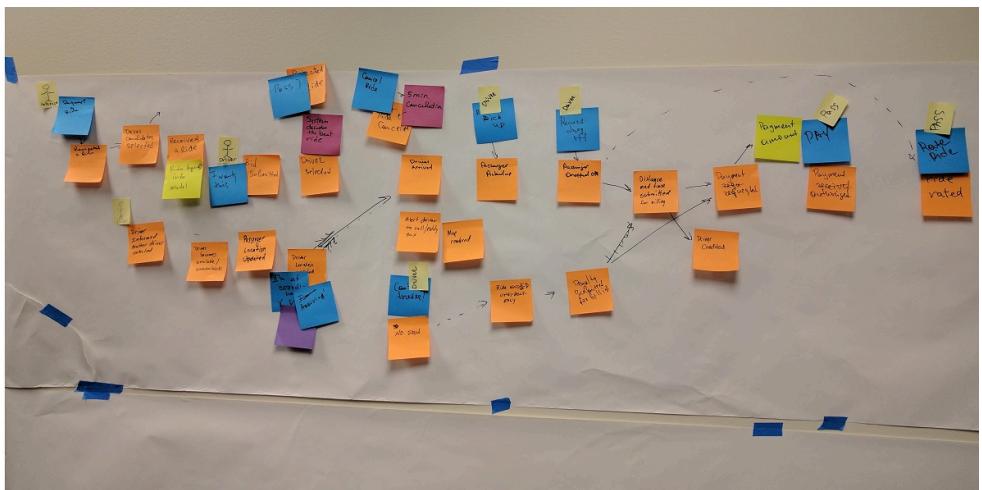
Event storming consists of three main steps:

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 triggers - ask the domain experts to identify the trigger of each event, which is one of the following:
 - o 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 3. 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 be 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 `accept()` method first invokes the `RestaurantOrderRepository` to load the `RestaurantOrder` from the database. It then updates the `RestaurantOrder` by calling `accept()`. The `RestaurantOrderService` then publishes events returned by `RestaurantOrder` 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 3, 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 Eventuate Tram framework, which I described in chapter 3, implements such a mechanism. It inserts 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 Eventuate Tram framework's DomainEventPublisher interface

```
public interface DomainEventPublisher {
```

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

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

A service could call the `DomainEventPublisher` publisher directly. However, one drawback of doing so is that it doesn't ensure that a service only publishes valid events. The `RestaurantOrderService`, for example, should only publish events that implement `RestaurantOrderDomainEvent`, which is the marker interface for the `RestaurantOrder` aggregate's events. A better option is for services to implement a subclass of `AbstractAggregateDomainEventPublisher`, which is shown in listing 5.7. `AbstractAggregateDomainEventPublisher` is an abstract class that provides a type-safe interface for publishing domain events. It's a generic class that has two type parameters, `A`, which is the aggregate type and `E`, which is the marker interface type for the domain events. A service publishes events by calling the `publish()` method, which has two parameters: an aggregate of type `A` and a list of events of type `E`.

Listing 5.7. `AbstractAggregateDomainEventPublisher` as an abstract superclass of type-safe domain event publishers.

```
public class AbstractAggregateDomainEventPublisher<A, E extends DomainEvent> {
    private Function<A, Object> idSupplier;
    private DomainEventPublisher eventPublisher;
    private Class<A> aggregateType;

    protected AbstractAggregateDomainEventPublisher(DomainEventPublisher
eventPublisher,
                                                    Class<A> aggregateType,
                                                    Function<A, Object> idSupplier) {
        this.eventPublisher = eventPublisher;
        this.aggregateType = aggregateType;
        this.idSupplier = idSupplier;
    }

    public void publish(A aggregate, List<E> events) {
        eventPublisher.publish(aggregateType, idSupplier.apply(aggregate),
(List<DomainEvent>) events);
    }
}
```

The `publish()` method retrieves the aggregate's id and invokes `DomainEventPublisher.publish()`. Listing 5.8 shows the `RestaurantOrderDomainEventPublisher`, which publish

Listing 5.8. `RestaurantOrderDomainEventPublisher` provides a type-safe interface for publishing `RestaurantOrder` aggregates' domain events.

```
public class RestaurantOrderDomainEventPublisher extends
AbstractAggregateDomainEventPublisher<RestaurantOrder, RestaurantOrderDomainEvent>
```

```
{
    public RestaurantOrderDomainEventPublisher(DomainEventPublisher eventPublisher) {
        super(eventPublisher, RestaurantOrder.class, RestaurantOrder::getId);
    }
}
```

This class publishes events that are a subclass of `RestaurantOrderDomainEvent` for the `RestaurantOrder` aggregate.

5.3.6 Consuming domain events

Domain events are ultimately published as messages to a message broker, such as Apache Kafka. A consumer could use the broker's client API directly. However, it is more convenient to use a higher-level API such as the Eventuate Tram framework's `DomainEventDispatcher`, which I described in chapter 3. 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 5.9. The Eventuate Tram framework's `DomainEventDispatcher` class which dispatches event messages to event handler methods

```
public class RestaurantOrderEventConsumer {
    @Autowired
    private RestaurantService restaurantService;

    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
        long id = Long.parseLong(de.getAggregateId());
        RestaurantMenu revisedMenu = de.getEvent().getRevisedMenu();
        restaurantService.reviseMenu(id, revisedMenu);
    }
}
```

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

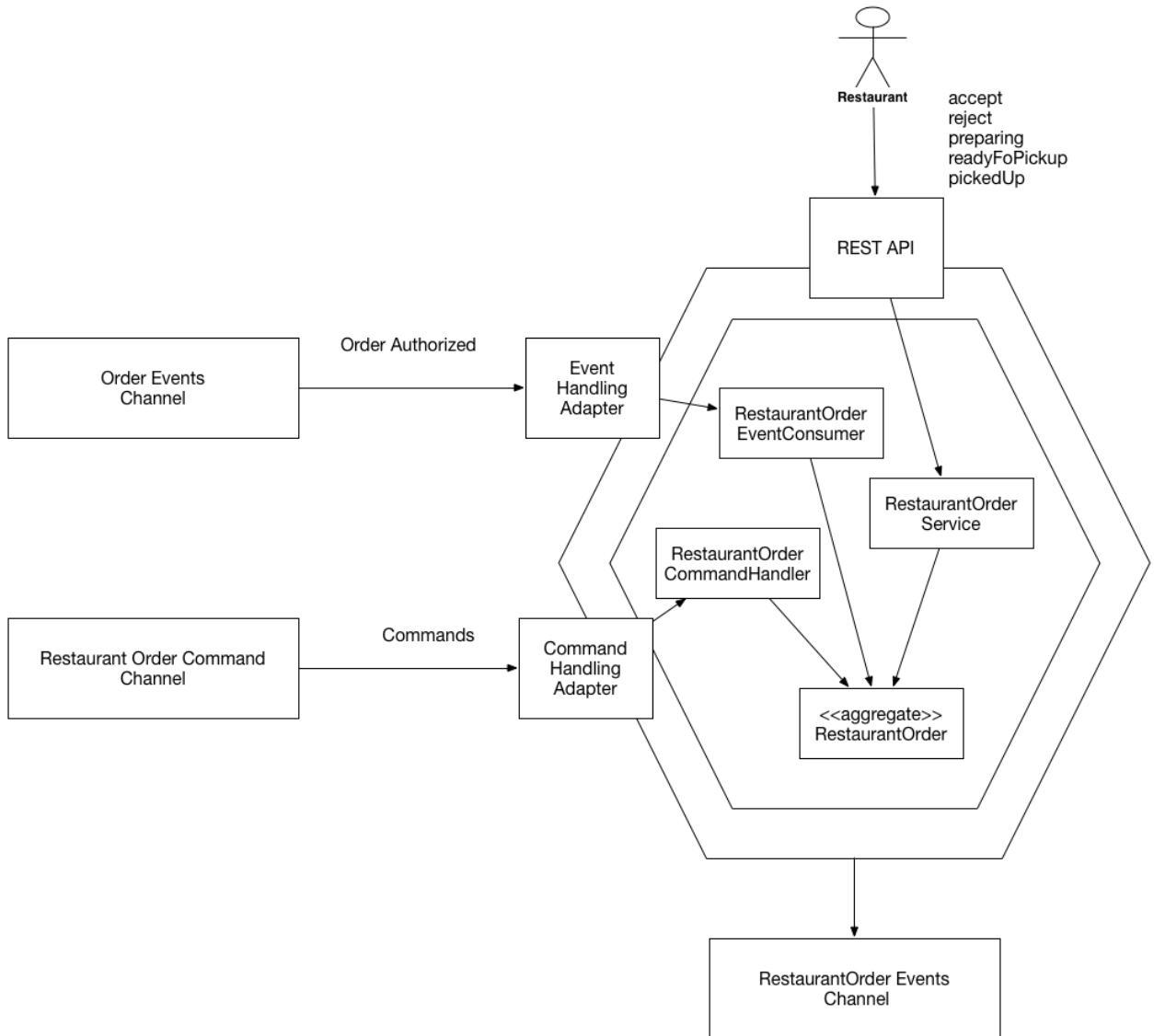
The `reviseMenu()` method handles `RestaurantMenuRevised` events. It calls `restaurantService.reviseMenu()`, which updates the restaurant's menu. That method returns a list of domain events, which are published by the event handler. Now that we

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 5.11 shows 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 2, 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 5.10 shows an excerpt of the code for this class.

Listing 5.10. 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 5.11 shows some of its methods.

Listing 5.11. Some of the RestaurantOrder's methods

```
public class RestaurantOrder {

    public static ResultWithAggregateEvents<RestaurantOrder,
    RestaurantOrderDomainEvent> create(Long id, RestaurantOrderDetails details) {
        return new ResultWithAggregateEvents<>(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<RestaurantOrderDomainEvent> 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 specified aggregate, calls the corresponding method on the aggregate root and publishes any domain events. Listing 5.12 shows an excerpt of this class, which shows the accept() method.

Listing 5.12. 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 RestaurantOrderDomainEventPublisher domainEventPublisher;

    public void accept(long orderId, LocalDateTime readyBy) {
        RestaurantOrder restaurantOrder =
            restaurantOrderRepository.findOne(orderId);
        List<RestaurantOrderDomainEvent> 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 5.13 shows an excerpt of this class including some example command handler methods.

Listing 5.13. An excerpt of the RestaurantOrderCommandHandler, which handles command messages sent by sagas

```
public class RestaurantOrderServiceCommandHandler {
```

```

    @Autowired
    private RestaurantOrderService restaurantOrderService;

    public CommandHandlers commandHandlers() {
        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 {
            RestaurantOrder restaurantOrder =
                restaurantOrderService.createRestaurantOrder(restaurantId,
                    restaurantOrderId, restaurantOrderDetails);
            CreateRestaurantOrderReply reply =
                new CreateRestaurantOrderReply(restaurantOrder.getId());
            return withSuccess(reply);
        } catch (RestaurantDetailsVerificationException e) {
            return withFailure();
        }
    }

    private Message confirmCreateRestaurantOrder
        (CommandMessage<ConfirmCreateRestaurantOrder> cm) {
        Long restaurantOrderId = cm.getCommand().getRestaurantOrderId();
        restaurantOrderService.confirmCreateRestaurantOrder(restaurantOrderId);
        return withSuccess();
    }

    ...

```

- ➊ Map command messages to message handlers
- ➋ Invoke RestaurantOrderService to create the RestaurantOrder
- ➌ Send back a successful reply
- ➍ Send back a failure reply
- ➎ Confirm the order

The responsibilities of each method are as follows:

- `createRestaurantOrder()` - creates a `RestaurantOrder` in the `CREATE_PENDING` state
- `confirmCreateRestaurantOrder()` - confirms the creation of

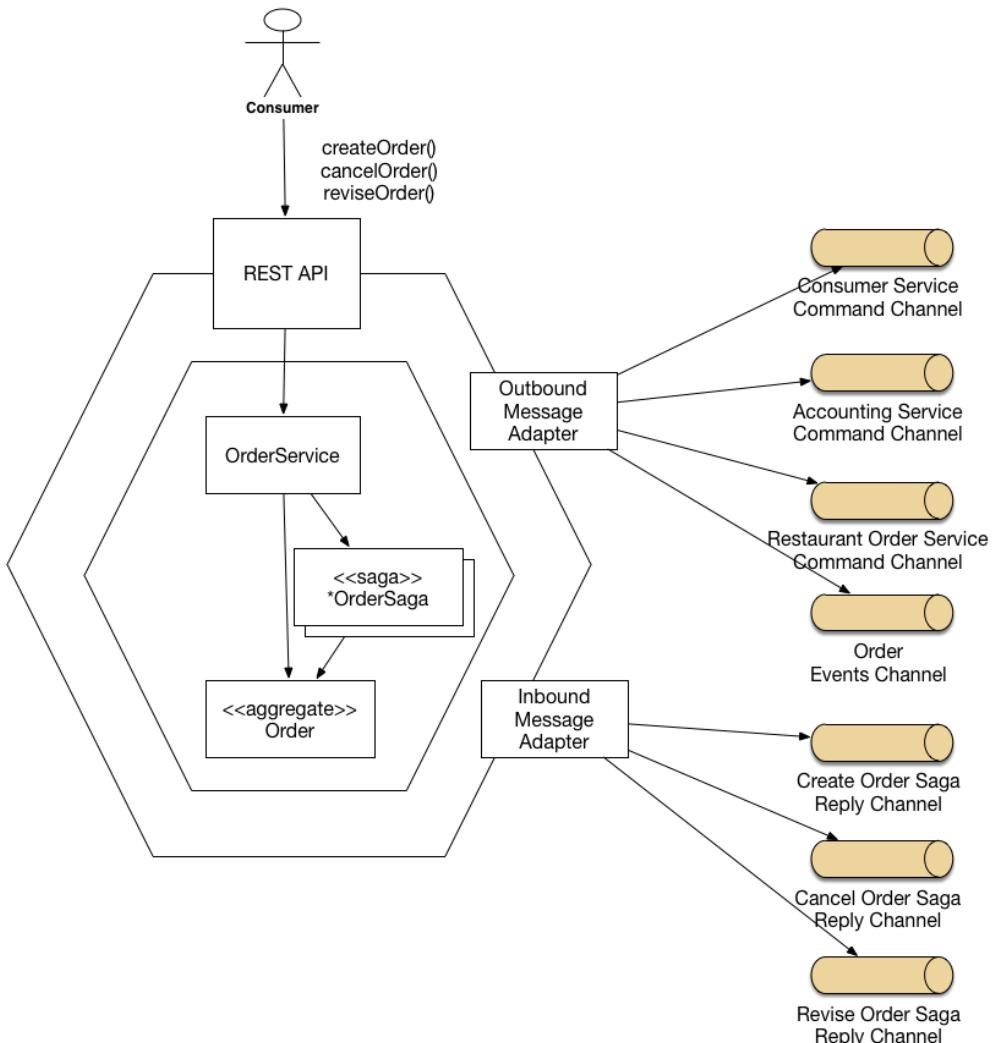
the RestaurantOrder

The `createRestaurantOrder()` 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. It has a REST API for managing orders. It exchanges messages and events with other services via several message channels.



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

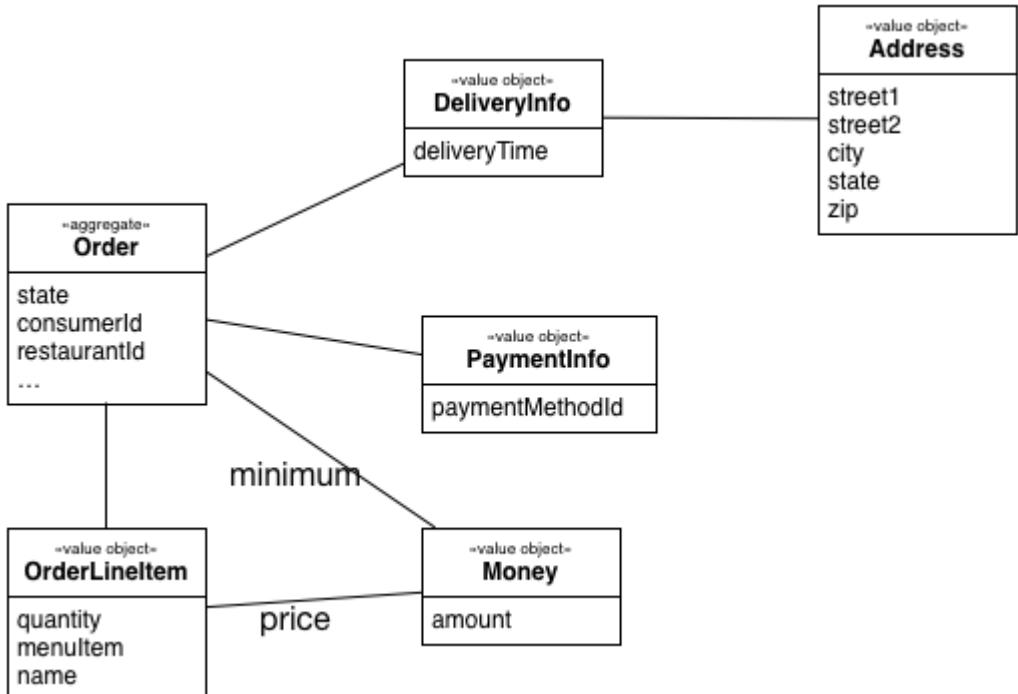
The Order aggregate represents an order placed by a consumer. Lets first look at the

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 5.14 shows the code.

Listing 5.14. 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;

```

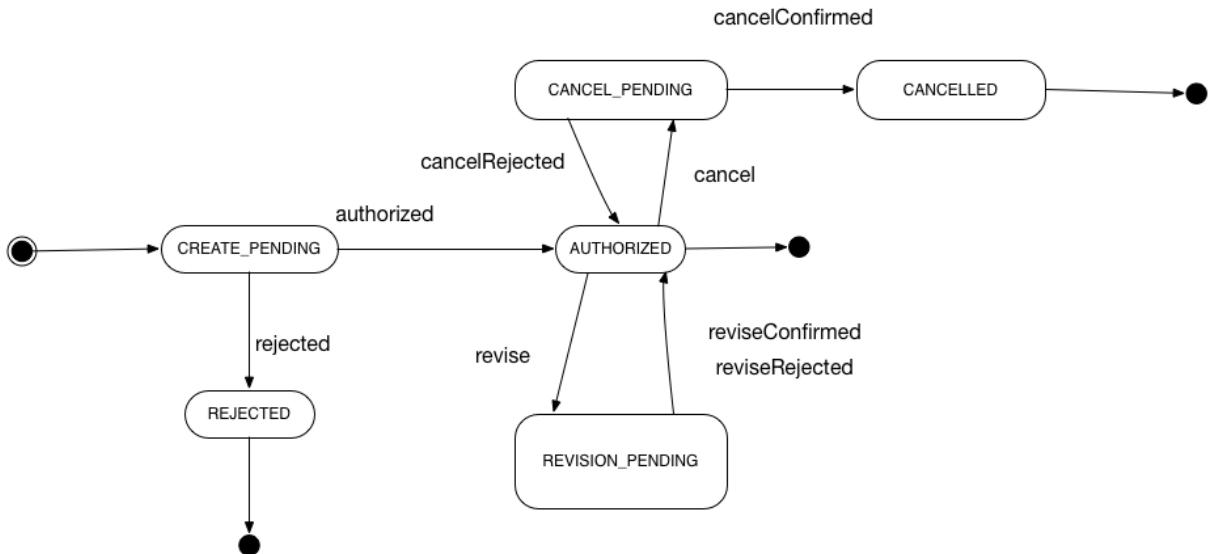
This class is persisted with JPA and is mapped to the ORDERS table. The id field is the primary key. The version field is used for optimistic locking. The state of an Order is represented by the OrderState enumeration. The DeliveryInformation and PaymentInformation fields are mapped using the @Embedded 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 4, 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 5.15 shows the Order's methods that are involved in creating an Order.

Listing 5.15. The methods of the Order class that involved in the order creation process

```

public class Order { ...

    public static ResultWithAggregateEvents<Order, OrderDomainEvent>
createOrder(OrderDetails orderDetails) {
    Order order = new Order(orderDetails);
    List<DomainEvent> events =
        singletonList(new OrderCreatedEvent(orderDetails));
    return new ResultWithAggregateEvents<>(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 5.16 shows these methods.

Listing 5.16. The Order method for revising an Order

```

class Order ...

public List<OrderDomainEvent> 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<OrderDomainEvent> 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 5.17 shows an except of this class. The `OrderService` is injected with various dependencies including the `OrderRepository`, `OrderDomainEventPublisher` and several saga managers. It defines several methods including `createOrder()` and `ReviseOrder()`.

Listing 5.17.

```

@Transactional
public class OrderService {

```

```

@Autowired
private OrderRepository orderRepository;

@Autowired
private SagaManager<CreateOrderSagaState, CreateOrderSagaData>
    createOrderSagaManager;

@Autowired
private SagaManager<ReviseOrderSagaState, ReviseOrderSagaData>
    reviseOrderSagaManagement;

@Autowired
private OrderDomainEventPublisher orderAggregateEventPublisher;

public Order createOrder(OrderDetails orderDetails) {

    Restaurant restaurant = restaurantRepository.findOne(restaurantId);
    if (restaurant == null)
        throw new RuntimeException("Restaurant not found: " + restaurantId);

    List<OrderLineItem> orderLineItems =
        makeOrderLineItems(lineItems, restaurant);
    ResultWithDomainEvents<Order, OrderDomainEvent> orderAndEvents =
        Order.createOrder(consumerId, restaurantId, orderLineItems);
    Order order = orderAndEvents.result;

    orderRepository.save(order); ①

    orderAggregateEventPublisher.publish(order, orderAndEvents.events); ② ③

    OrderDetails orderDetails =
        new OrderDetails(consumerId, restaurantId, orderLineItems,
                        order.getOrderTotal());
    CreateOrderSagaData data = new CreateOrderSagaData(order.getId(),
        orderDetails);

    createOrderSagaManager.create(data, Order.class, order.getId()); ④

    return order;
}

public Order reviseOrder(Long orderId, Long expectedVersion,
                        OrderRevision orderRevision) {
    public Order reviseOrder(long orderId, OrderRevision orderRevision) {
        Order order = orderRepository.findOne(orderId); ⑤
        if (order == null) {
            logger.error("Cannot find order: {}", orderId);
            return null;
        }
        ReviseOrderSagaData sagaData = new ReviseOrderSagaData(order.getConsumerId(),
orderId, null, orderRevision);
        reviseOrderSagaManager.create(sagaData);
        return order;
    }
}

```

① Create the Order aggregate

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

Licensed to Joydeep Ghosh <joydeep.ghosh1977@outlook.com>

- ② Persist the Order in the database
- ③ Publishes domain events
- ④ Create the CreateOrderSaga
- ⑤ Retrieve the Order
- ⑥ Create the ReviseOrderSaga

The `createOrder()` method first creates and persists an `Order` aggregate. It then publishes the domain events emitted by the aggregate. Finally, it creates a `CreateOrderSaga`. The `reviseOrder()` retrieves the `Order` and then creates a `ReviseOrderSaga`.

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 is 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.

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 4, a good way to maintain data consistency in a microservice architecture. But first, let's 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.

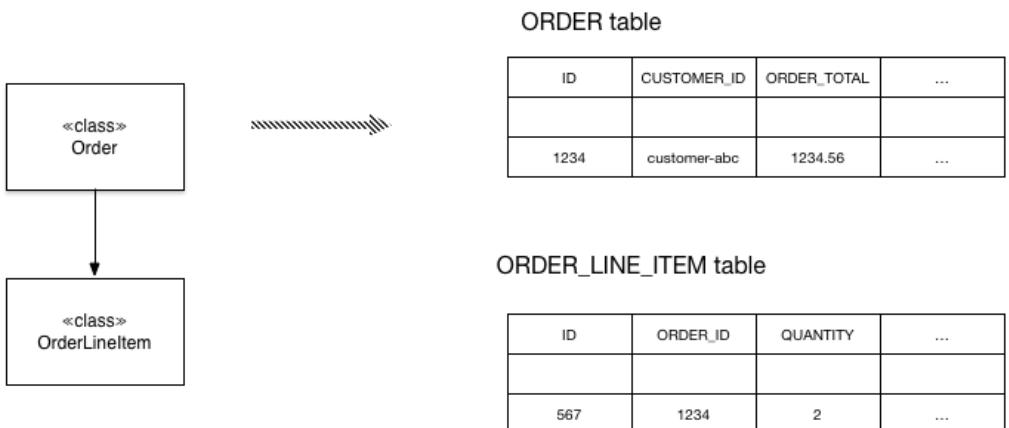
Event sourcing has several important benefits. It preserves the history of aggregates, which is valuable for auditing and regulatory purposes. Also, it reliably publishes domain events, which is particularly useful in a microservice architecture. Event sourcing also has drawbacks. There is a learning 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 7.

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 Orderaggregate 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 classes to rows in those tables. For example, figure 6.1 shows how the Order aggregate, which I described in chapter 5, 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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

Licensed to Joydeep Ghosh <joydeep.ghosh1977@outlook.com>

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"³⁷. 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.

Event publishing is bolted on to the business logic

Another limitation of traditional persistence is that it usually doesn't support

³⁷ blogs.tedneward.com/post/the-vietnam-of-computer-science/

publishing domain events. Domain events, which I described in chapter 5, are events that are published 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.

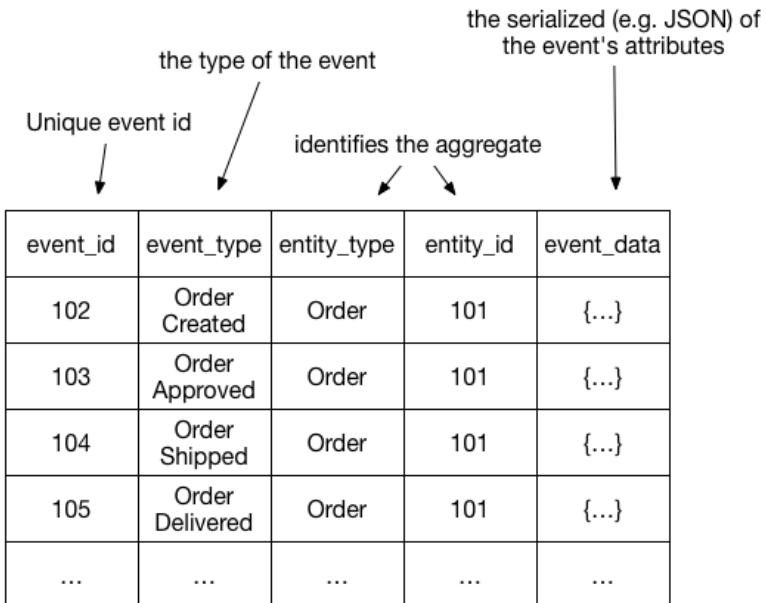
6.1.2 Overview of event sourcing

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 [“The trouble with traditional persistence”](#), 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.



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 “[The Eventuate client framework for Java](#)”, uses code similar to the following to reconstruct an aggregate:

```
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.

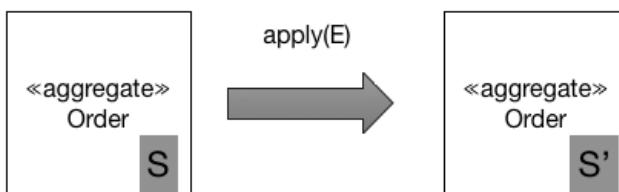
Events represent state changes

In chapter 5, 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.



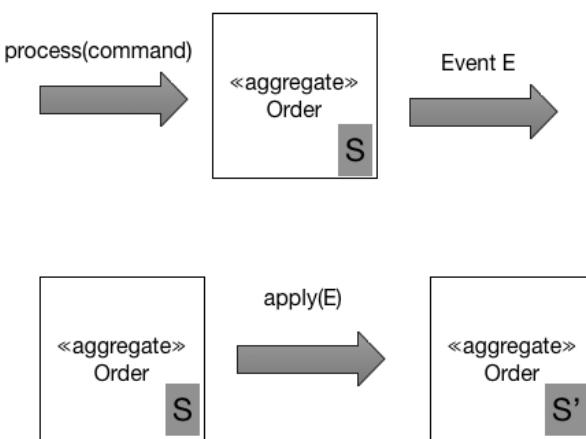
Some events, such as the Order Shipped event, contain little or no data and just represent the state transition. The `apply()` method handles an Order Shipped event by

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

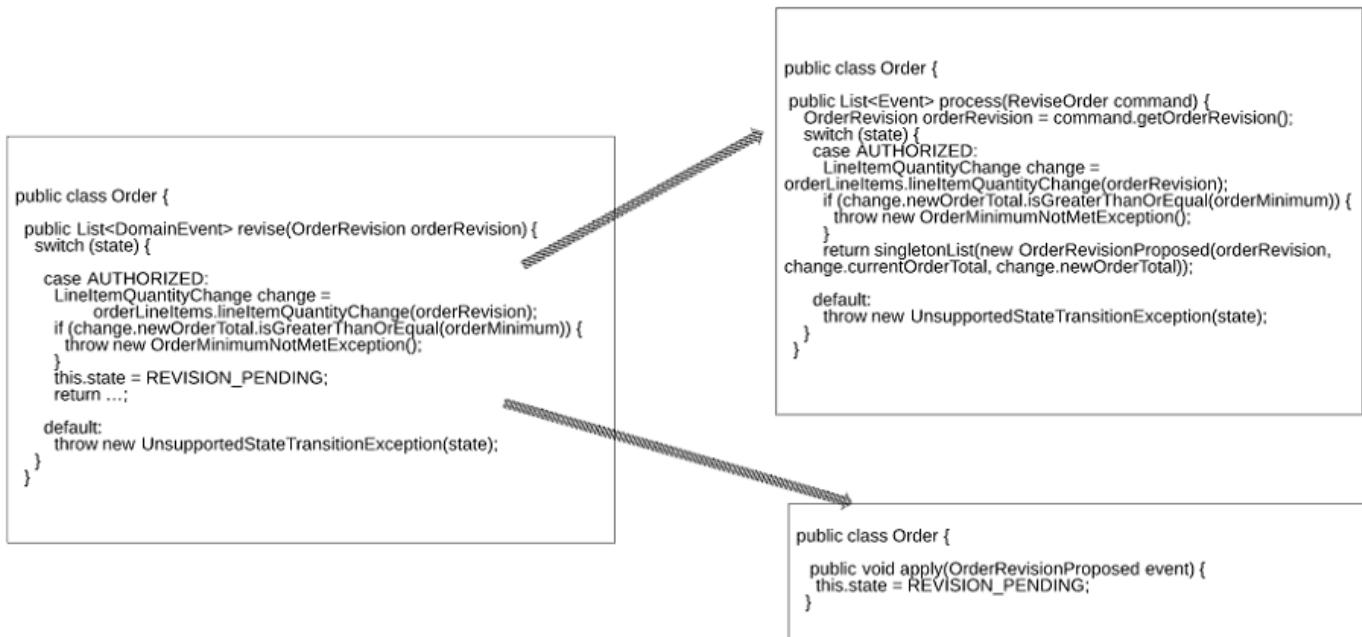


The requirement to generate events and apply them requires a restructuring - albeit mechanical - of the business logic. Event sourcing refactors a command method into two or more methods. The first method takes a command object parameter, which represents the request, and determines what state changes 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 in section "[The Eventuate client framework for Java](#)", 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, let's 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 5. 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.

List 6.1. The Order aggregate's fields and the methods that creates it

```
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(CheckoutOrder 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 3, 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.

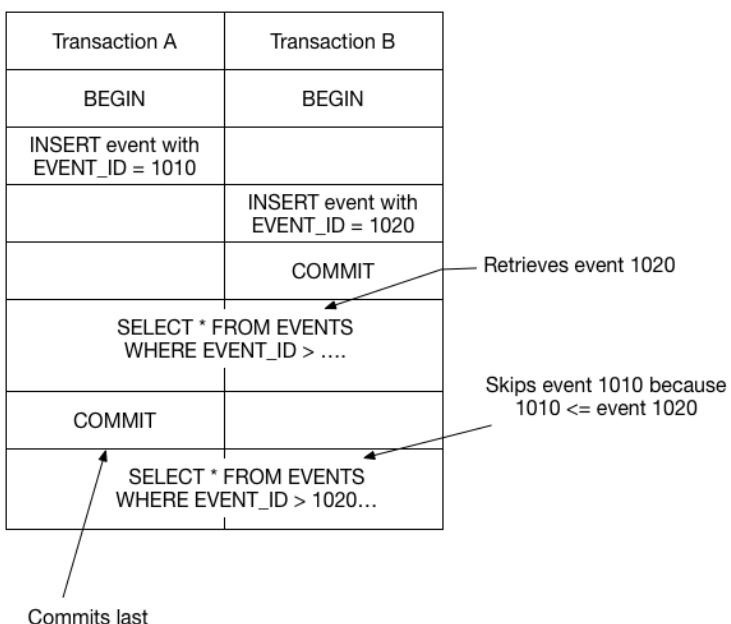
Using a query to poll for new events

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, let's imagine that event IDs are monotonically increasing. The superficially appealing approach is for a consumer to record the last event ID 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 accidentally skip events. Table 6.6 shows 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 a 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

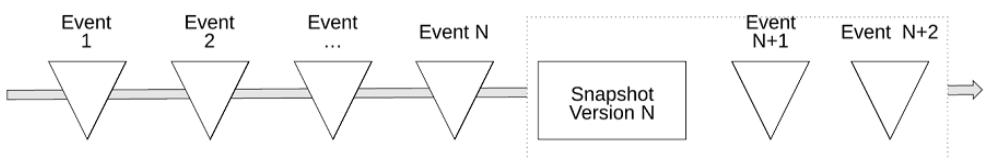
More sophisticated event stores use a different approach that avoids this problem and is also more performant and scalable. For example, Eventuate Local, which is an open-source event store, uses transaction log tailing. It reads events inserted into an EVENTS table from the MySQL replication stream 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 "[The Eventuate client framework for Java](#)", 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 than being created 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 pattern³⁸.

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 diagram illustrates the process of recreating a Customer aggregate. It consists of two tables: 'EVENTS' and 'SNAPSHOTS'. An arrow points from the 'EVENTS' table to the 'SNAPSHOTS' table, indicating the flow of data from events to a snapshot.

EVENTS					SNAPSHOTS			
event_id	event_type	entity_type	entity_id	event_data	event_id	entity_type	entity_id	snapshot_data
...
103	...	Customer	101	{...}	103	Customer	101	{ name: "...", ... }
104	Credit Reserved	Customer	101	{...}
105	Address Changed	Customer	101	{...}
106	Credit Reserved	Customer	101	{...}

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

Services 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 3, message consumers must be idempotent since a message broker might deliver the same message multiple times. A message consumer is idempotent if it can safely be invoked with the same message multiple times. The Eventuate 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 logic to create or update aggregates. Event sourcing-based business logic must implement an equivalent mechanism.

³⁸ en.wikipedia.org/wiki/Memento_pattern

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 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 user who 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 fold the events that occurred up until that point. It is straightforward, for example, to calculate the available credit of a customer 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 the 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 [“Managing schema changes through upcasting”](#) 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 7.

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. There are several to chose from:

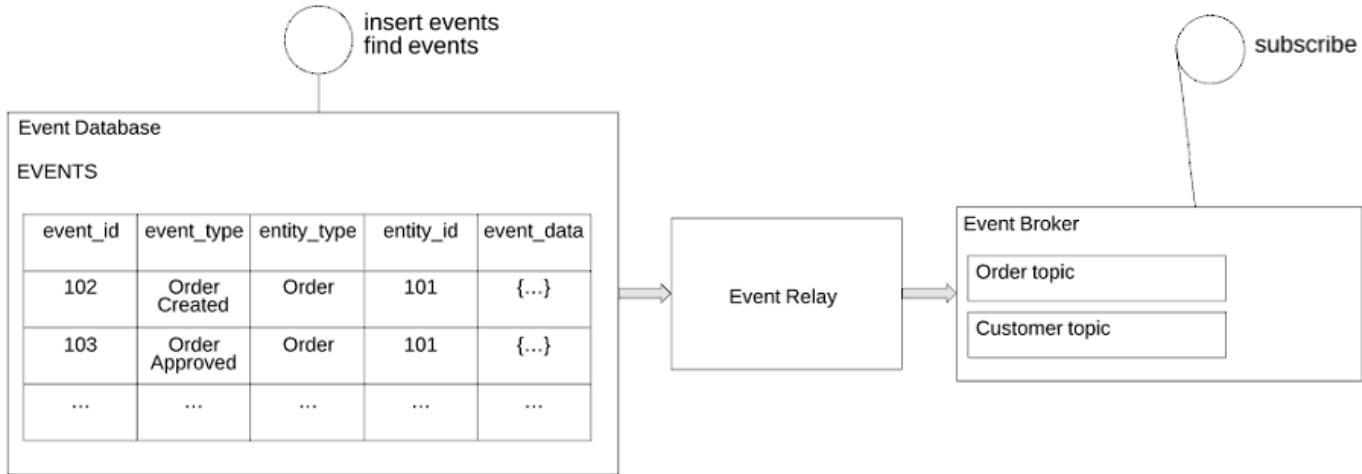
- Event Store - a .NET-based, open-source event store developed by Greg Young, an event sourcing pioneer
- Lagom - a microservices framework developed by Lightbend, the company formerly known as Typesafe
- Axon - an opensource Java framework for developing event-driven applications that use event sourcing and CQRS.
- Eventuate - developed by my startup, Eventuate. There are two versions of Eventuate: Eventuate SaaS, which is a cloud service and Eventuate Local, which is an Apache Kafka/RDBMS-based open-source project.

Although these frameworks differ in the details they the core concepts remain the same. Since Eventuate is the framework that I am most familiar with that's the one I am going to describe here. Let's 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 3, 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

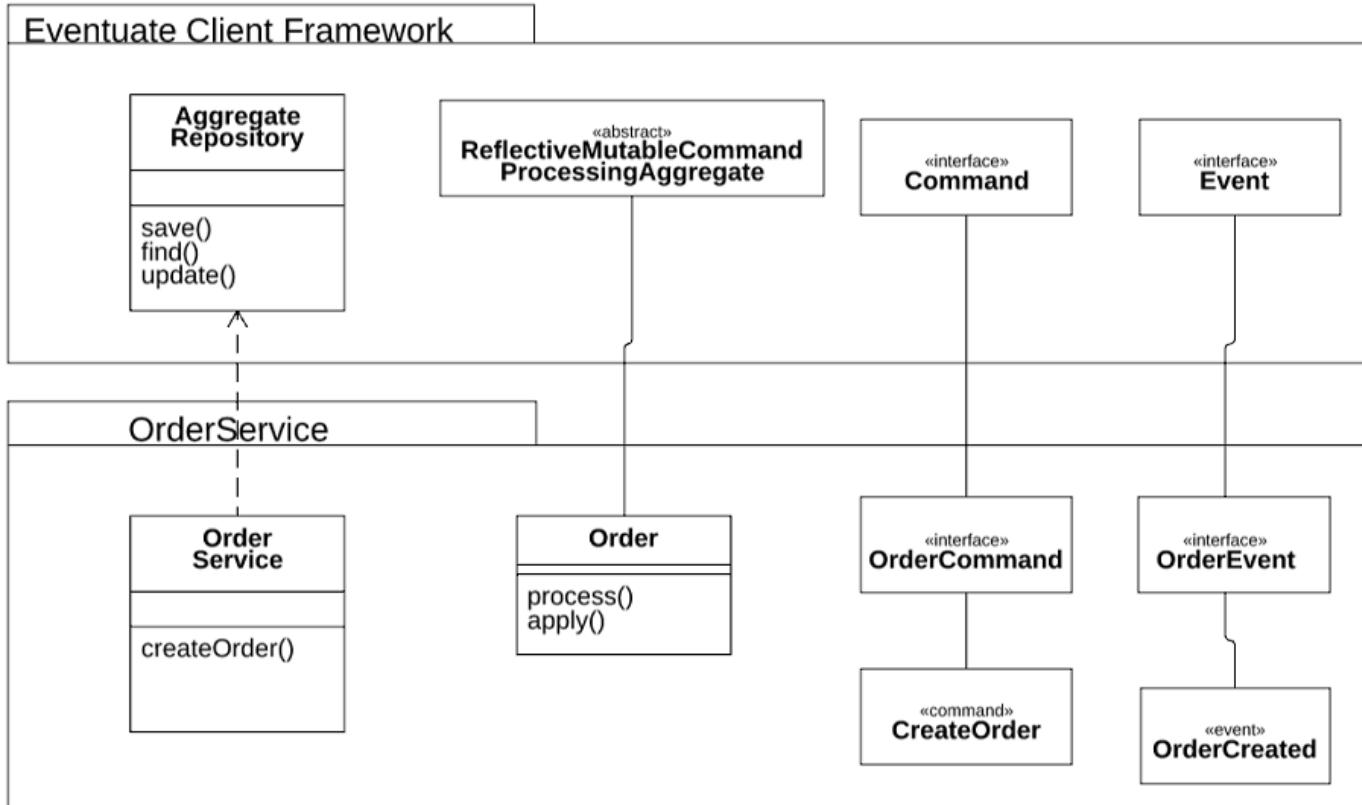
The event relay propagates events inserted into the event database to the event broker. The mechanism used to accomplish this depends on the database. The MySQL version of the event relay uses the MySQL master/slave replication protocol. The event relay connects to the MySQL server as if it were a slave 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 a 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, let's 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 4 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 `Orderaggregate` 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 Eventuate Tram saga framework that I described in chapter 4 and the underlying Tram messaging framework, which I described in chapter 3, 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 Eventuate 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 Eventuate 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*

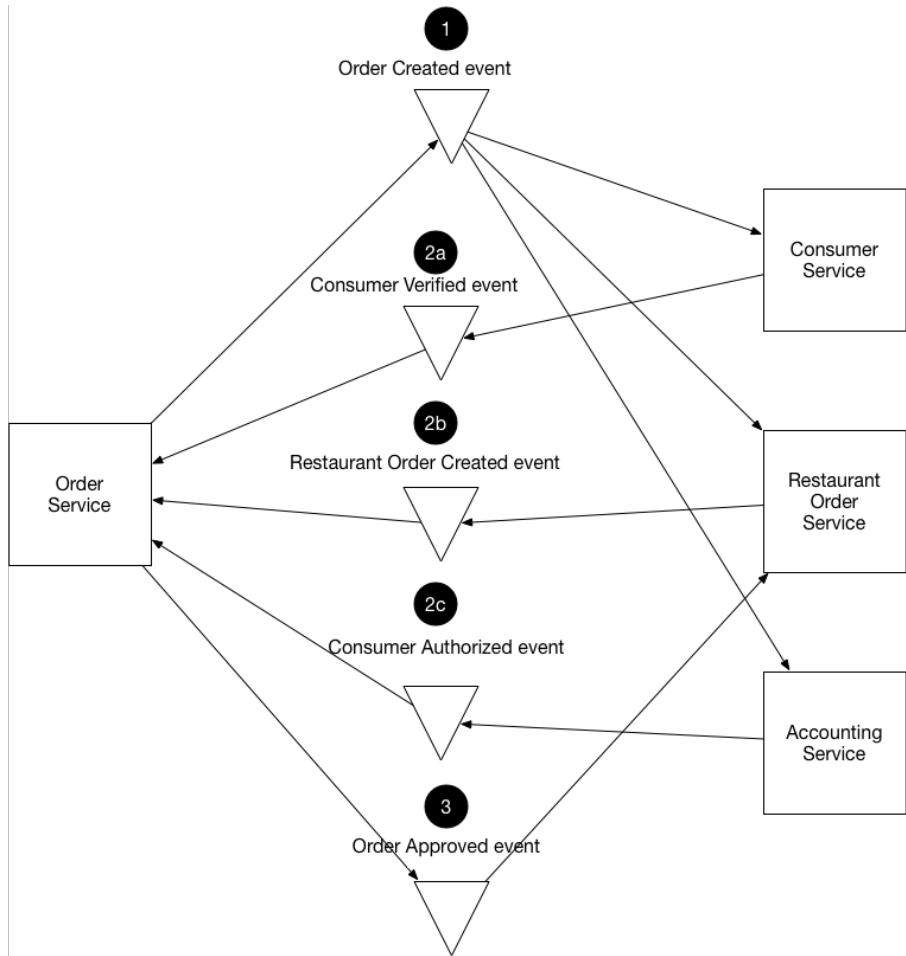
The event-driven nature of event sourcing makes it quite straightforward to implement choreography-based sagas. When an aggregate is updated, it emits an event. An event handler for a different aggregate can consume that event and update its aggregate. The event sourcing framework automatically makes each event handler idempotent.

For example, in chapter 4, I described how to 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 6.5. The event handler updates the corresponding aggregate, which emits another event. Figure 6.11 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 CREATED

Event sourcing and choreography-based sagas work very well together. Event sourcing provides the mechanisms that sagas need including messaging-based IPC, message deduplication, 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 4, 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 Eventuate Tram saga framework. Its `createOrder()` method would look like this:

```
class OrderService

    @Autowired
    private SagaManager<CreateOrderSagaData> createOrderSagaManager;

    @Transactional
    public EntityWithIdAndVersion<Order> createOrder(OrderDetails orderDetails) {
```

```

EntityWithIdAndVersion<Order> order =
    orderRepository.save(new CreateOrder(orderDetails));          ②

CreateOrderSagaData data =
    new CreateOrderSagaData(order.getId(), 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`

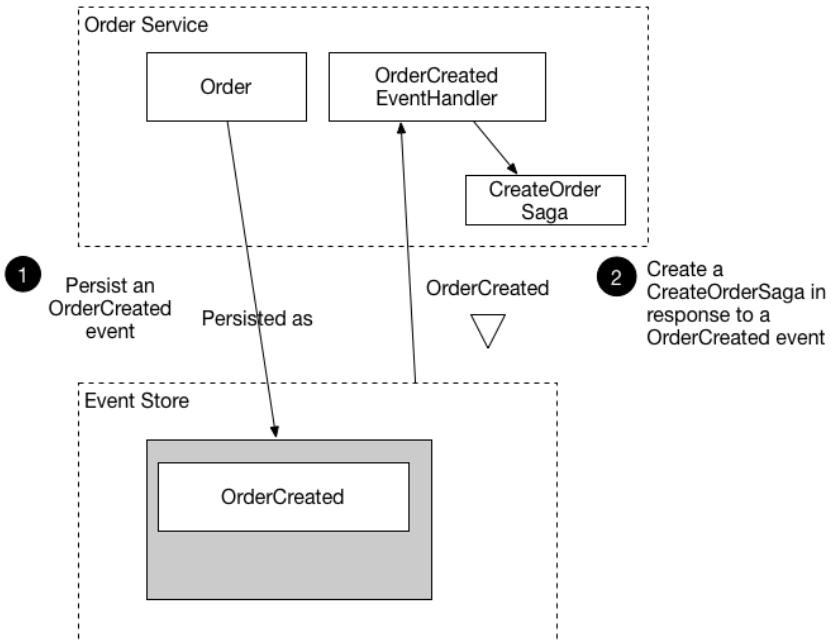
It is a combination of the `OrderService` in listing 6.4, and the `OrderService` described in chapter 4. Since Eventuate Local uses an RDBMS, it can participate in the same ACID transaction as the Eventuate Tram saga framework. If, however, a service uses a NoSQL-based event store, creating a saga orchestrator 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 Eventuate 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, let's 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 Eventuate 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 Eventuate 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

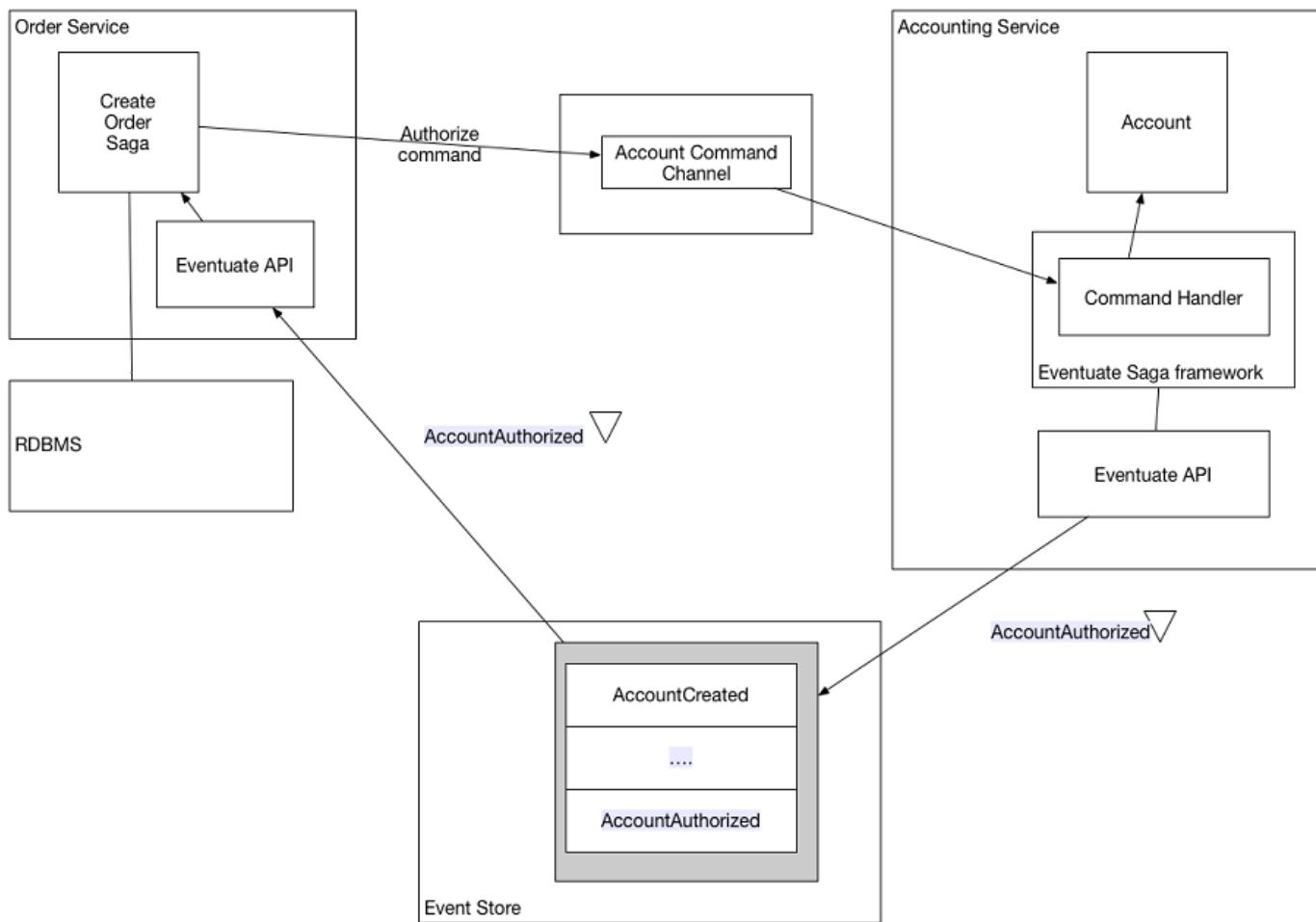
The second problem to solve is how an event sourcing-based saga participant can atomically send replies. In principle, a saga orchestrator could subscribe to the events emitted by an aggregate. However, there are two problems with this approach. The first is that a saga command might not actually change 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 `SagaReplyRequestedEventHandler` 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, let's 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 4, 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

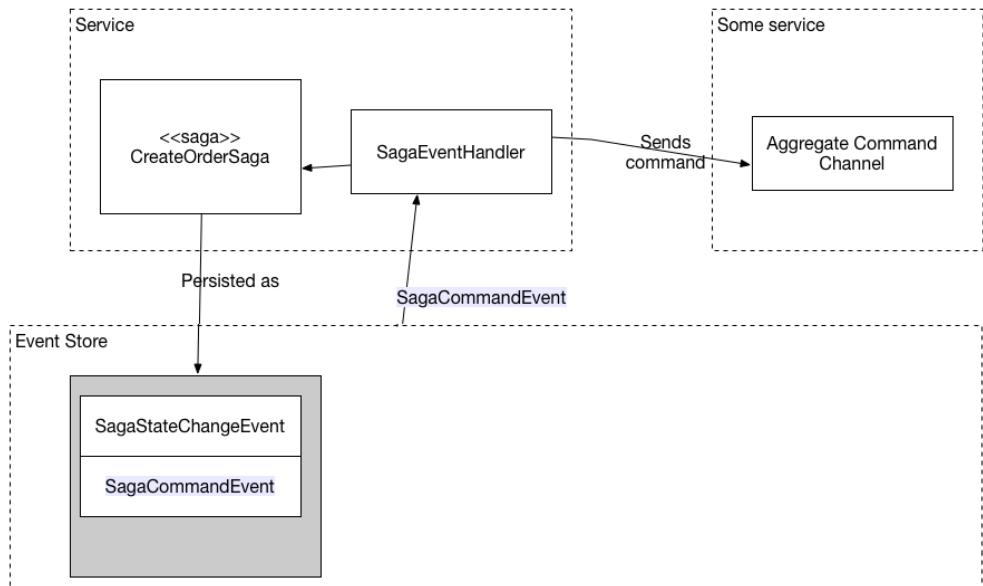
A saga orchestrator emits a `SagaOrchestratorCreated` event when it is created and a `SagaOrchestratorUpdated` event when it has been updated. These events contain the data necessary to recreate the state of the saga orchestrator. For example, the events for the `CreateOrderSaga`, which I described in chapter 4, 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 4, 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 `SagaCommandEvent`s 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 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 `SagaCommandEvent`s 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 `SagaCommandEvent`s 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 let's look at an example of a saga that is implemented 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 implement saga orchestrators. As a result, you can write applications that exclusively use an event store.



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**

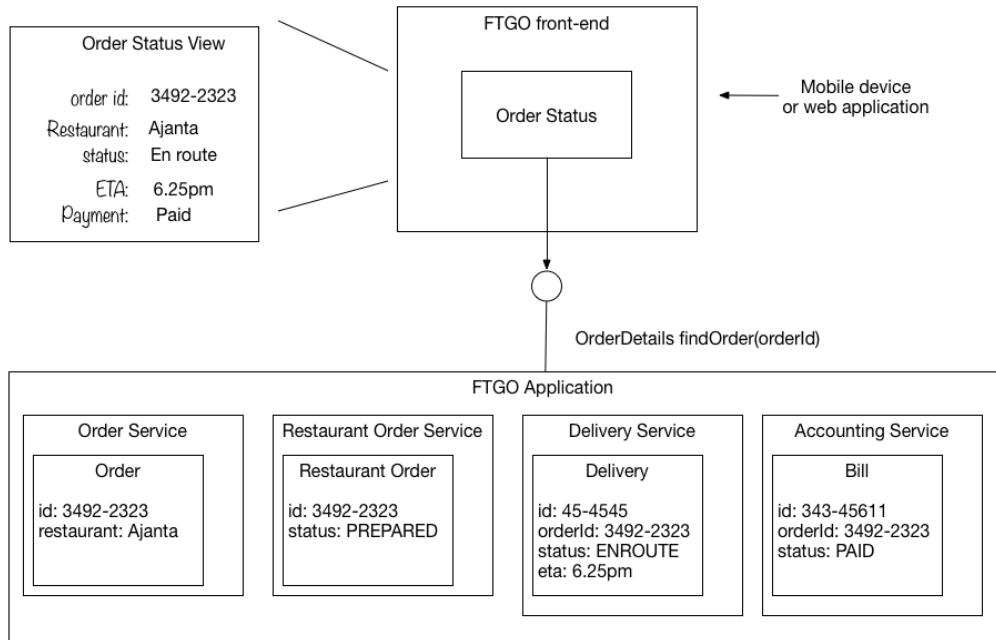
The FTGO application implements numerous query operations. Some queries, as I mentioned earlier, simply retrieve data from a single service. Implementing these queries is usually straightforward although later in this chapter, when I describe the CQRS pattern, I show examples of single service queries 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, 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.

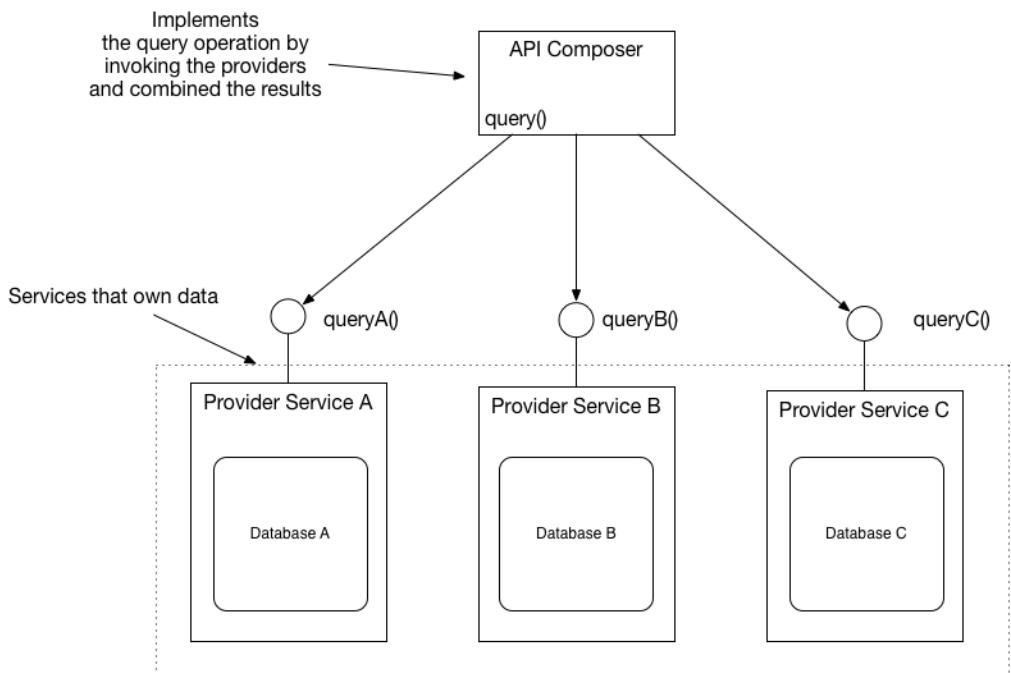


Figure 7.2 shows three provider services. The *API composer* implements the query by retrieving data from the provider services and combining the results. An *API composer* might be a client such as a web application, that needs the data to render a web page. Alternatively, it might be a service, such as an API Gateway that I describe in chapter <, which exposes the query operation as an API endpoint.

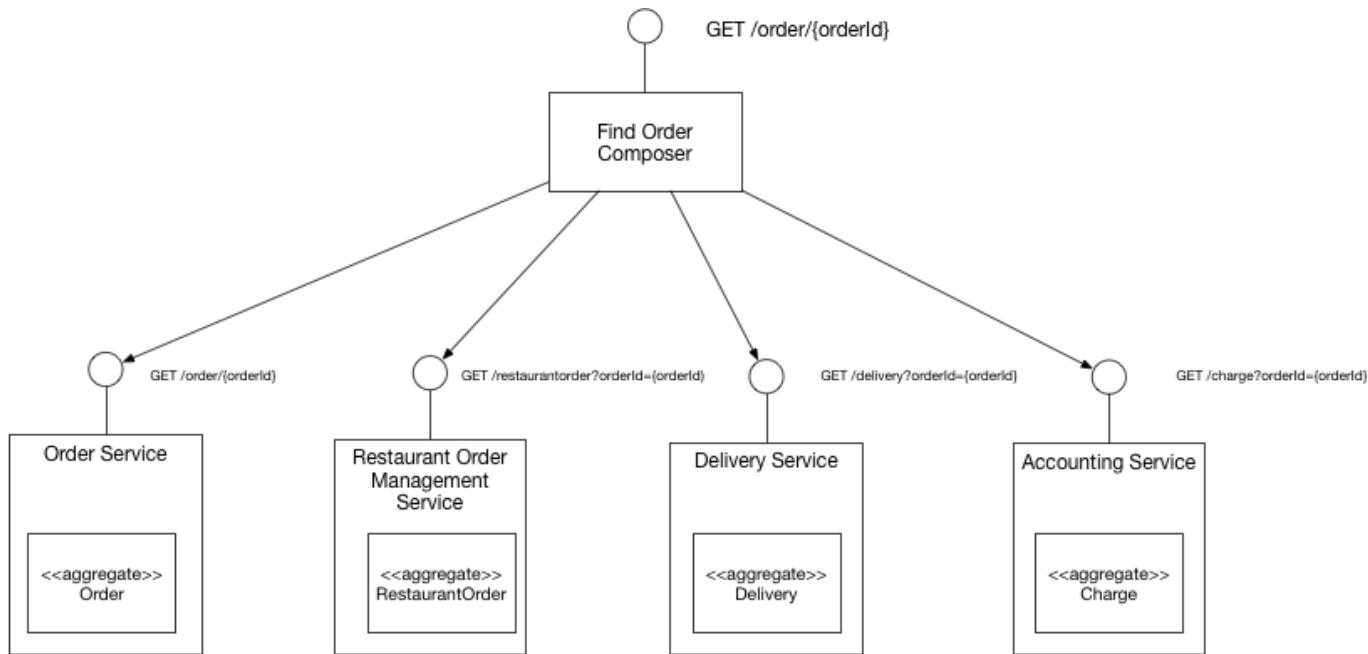
Whether you can use this pattern to implement a particular query operation depends on several factors, including how the data is partitioned, the capabilities of the APIs exposed by the services that own the data, and the capabilities of the databases used by the services. For instance, even if the *Provider 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 this pattern is applicable. To see this pattern in action let's 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.

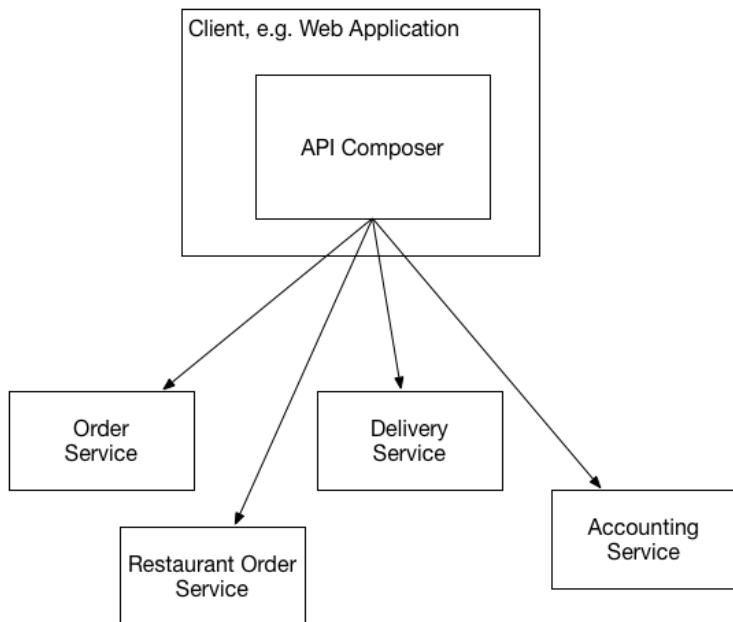
- Deciding which component in your architecture is the query operation's *API composer*.
- How to write efficient aggregation logic

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*.

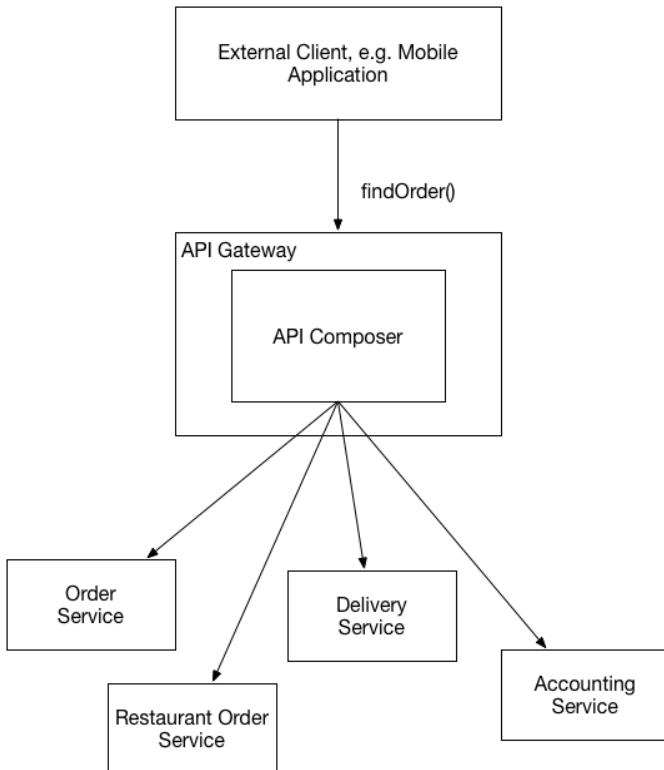
Figure 7.4. Implementing API composition in a client. The client queries the provider services to retrieve the data.



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.

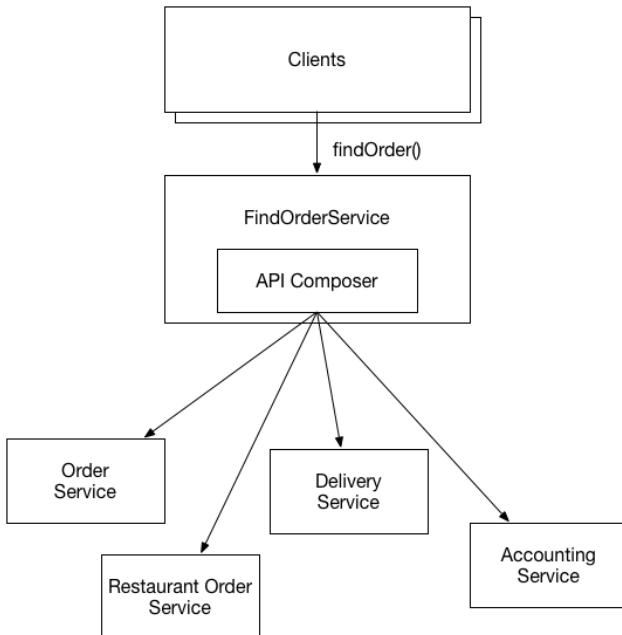
Figure 7.5. Implementing API composition in the API Gateway. The API queries the provider services to retrieve the data, combines the results, and returns a response to the client.



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 CompletableFuture, 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 while 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 let's 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.

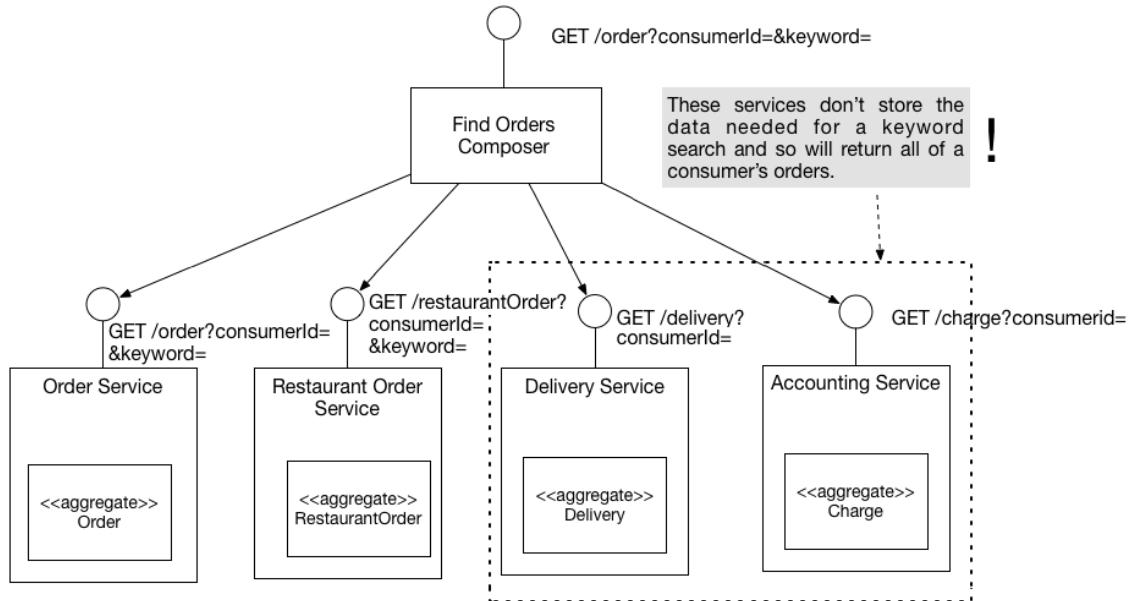
This query operation returns an `OrderHistory` object, which contains a summary of the matching orders sorted by increasing age. It is called by the module that implements the `Order History` view. This view displays a summary of each order, which includes the order number, the order status, the order 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 develops 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 [“Motivations for using CQRS”](#) highlighted three problems that are commonly encountered when implementing queries in a microservice architecture:

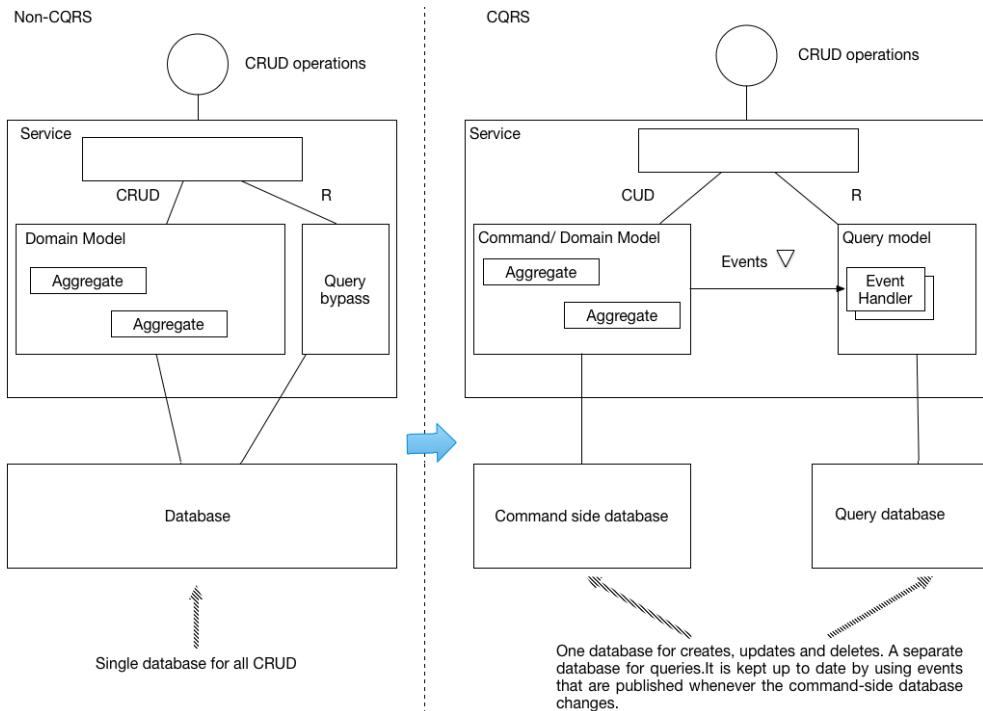
1. Using the API Composition pattern to retrieve data scattered across multiple services results in expensive, inefficient in-memory joins.
2. The service that owns the data, stores the data in a form or in a database that does not efficiently support the required query.
3. The need to separate concerns 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.

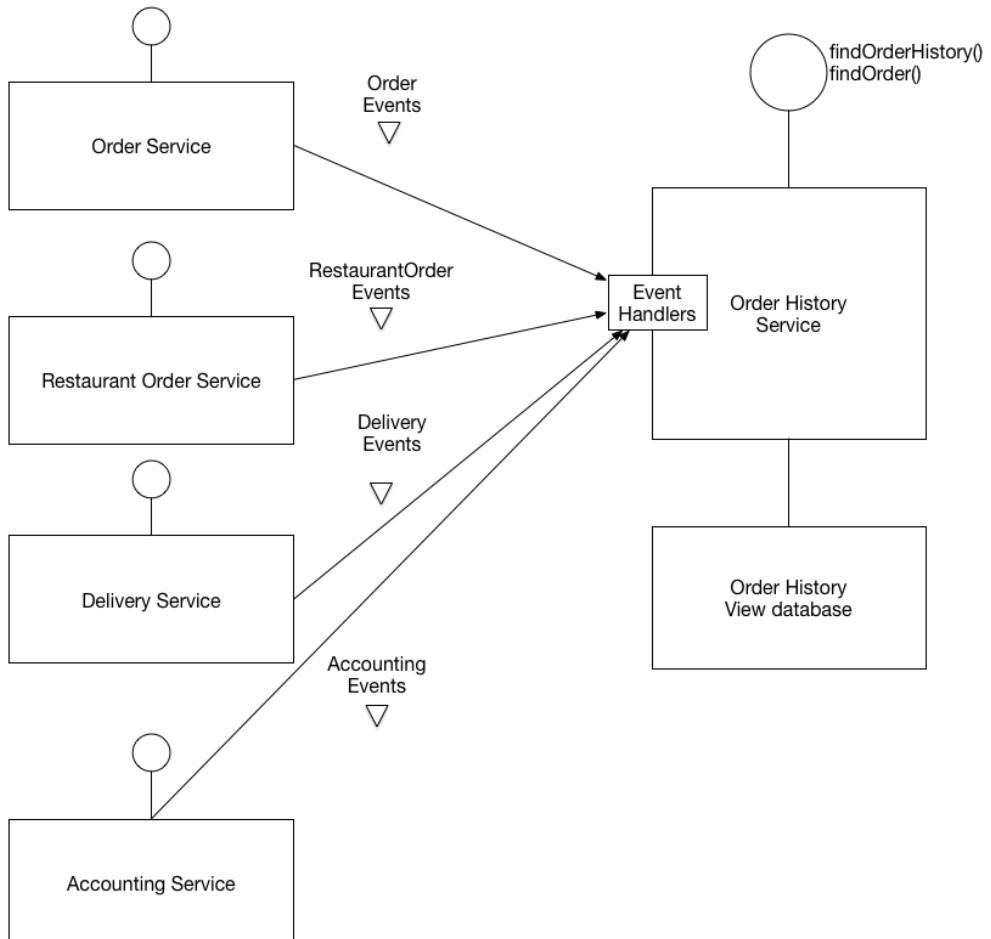
In a CQRS-based service, the command side domain model handles create, update and delete operations and its mapped to its own database. It might also handle simple queries, such as non-join, primary key-based queries. The command side publishes domain events whenever its data changes. These events might 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.

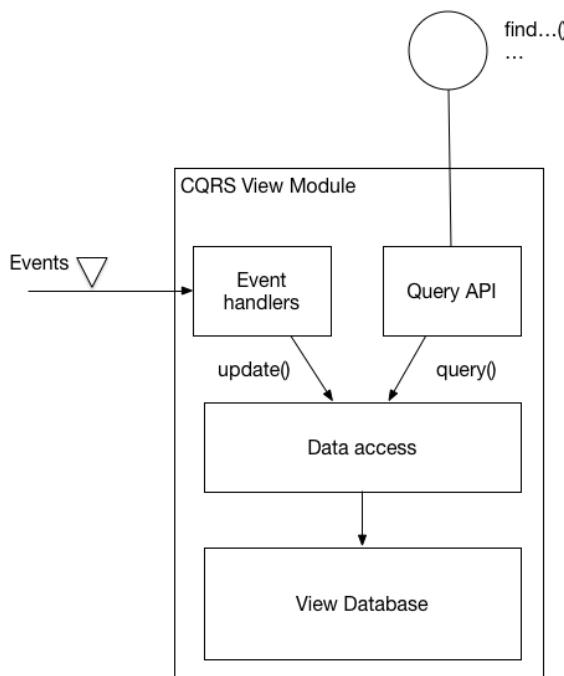
A UI application such as a native mobile application or single page JavaScript application can handle replication lag by updating its local model once the command is successful without issuing a query. It can, for example, update its model using data returned by the command. Hopefully, when a user action 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 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 or 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 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.

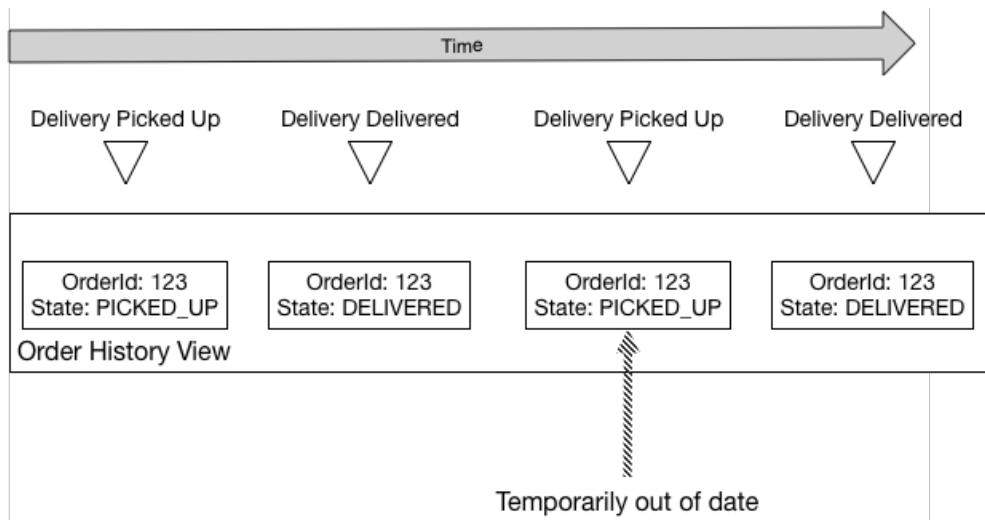
For example, an event handler for an `Order*` event might be invoked at the same time as an event handler for a `Delivery*` event for same order. Both events handlers then simultaneously invoke the DAO to update the database record for that order. A DAO must be written in a way that ensures that this 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 Historyview` 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 `Order History` 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. A 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

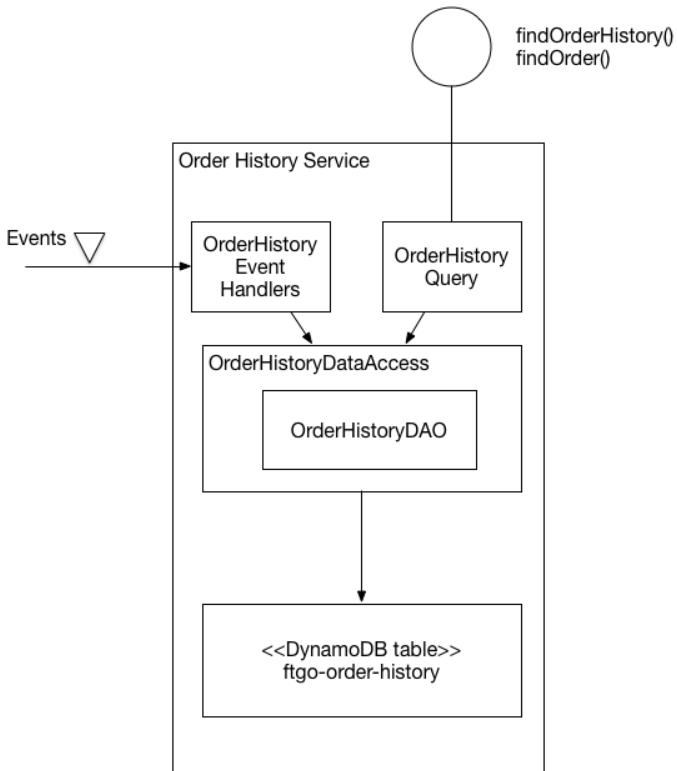
Another problem with view creation is that the time and resources required to process all events keeps growing over time. Eventually, view creating will become too slow and expensive. The solution is to use a two-step incremental algorithm. The first step, periodically computes a snapshot of each aggregate 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.

ftgo-order-history table						
Primary key		orderId	consumerId	orderCreationTime	status	lineItems
...	xyz-abc	22939283232	CREATED	[{...}, {...}, {...}]
***	***	***	***	***	***	***

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.

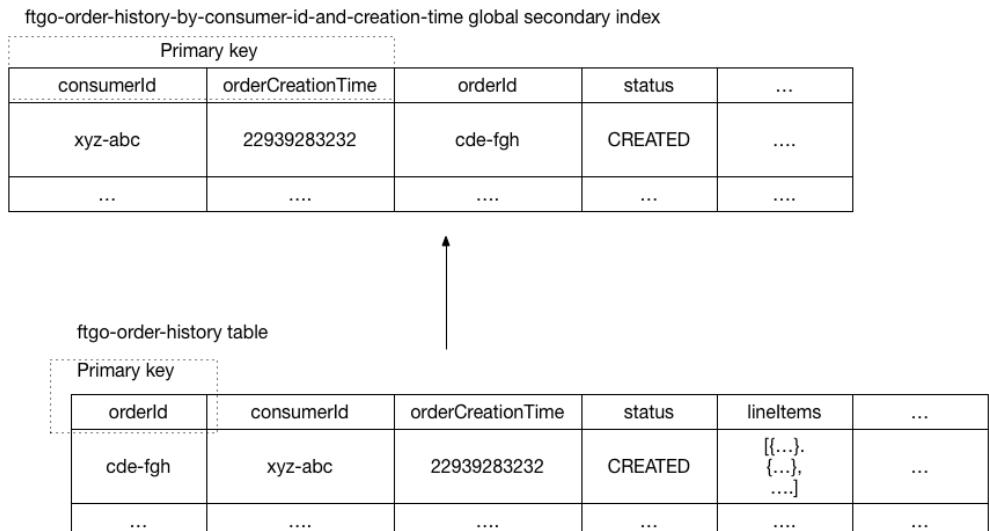
The DynamoDB `query()` operation requires a primary key to consist of two scalar attributes. The first attribute is a partition key. The partition key is so called because it is hashed and used to select item's storage partition. The second attribute, which is optional is the sort key. A `query()` operation 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 specified 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 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 ensure that the order had not been changed since it was loaded. On the one hand, this is fairly easy to implement since DynamoDB 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. Let's 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 exist 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`. This 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 `OrderCreatedevent`. 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())  
            .withUpdateExpression("SET orderStatus = :orderStatus, " +  
                "creationDate = :cd, consumerId = :consumerId, lineItems = " +  
                " :lineItems, keywords = :keywords, restaurantName = " +  
                ":restaurantName")  
            .withValueMap(new Maps()  
                .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);  
    }
```

- ➊ the primary key of the Order item to update.
- ➋ the update expression that updates the attributes
- ➌ 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)  
            .withHashKey("consumerId", consumerId)  
            .withRangeKeyCondition(new RangeKeyCondition("creationDate")  
                .gt(filter.getSince().getMillis()));  
  
        filter.getStartKeyToken().ifPresent(token ->  
            spec.withExclusiveStartKey(toStartingPrimaryKey(token)));  
  
        Map<String, Object> valuesMap = new HashMap<>();  
  
        String filterExpression = Expressions.and(  
            keywordFilterExpression(valuesMap, filter.getKeywords()),  
            statusFilterExpression(valuesMap, filter.getStatus()));  
  
        if (!valuesMap.isEmpty())  
            spec.withValueMap(valuesMap);  
  
        if (StringUtils.isNotBlank(filterExpression)) {  
            spec.withFilterExpression(filterExpression);  
        }  
  
        filter.getPageSize().ifPresent(spec::withMaxResultSize);  
  
        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

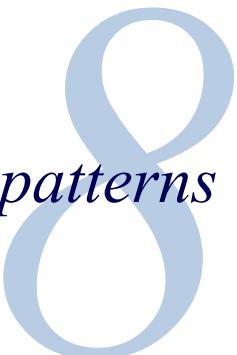
After building a `QuerySpec`, this method then executes a query and builds an `OrderHistory` from the returned items.

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



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

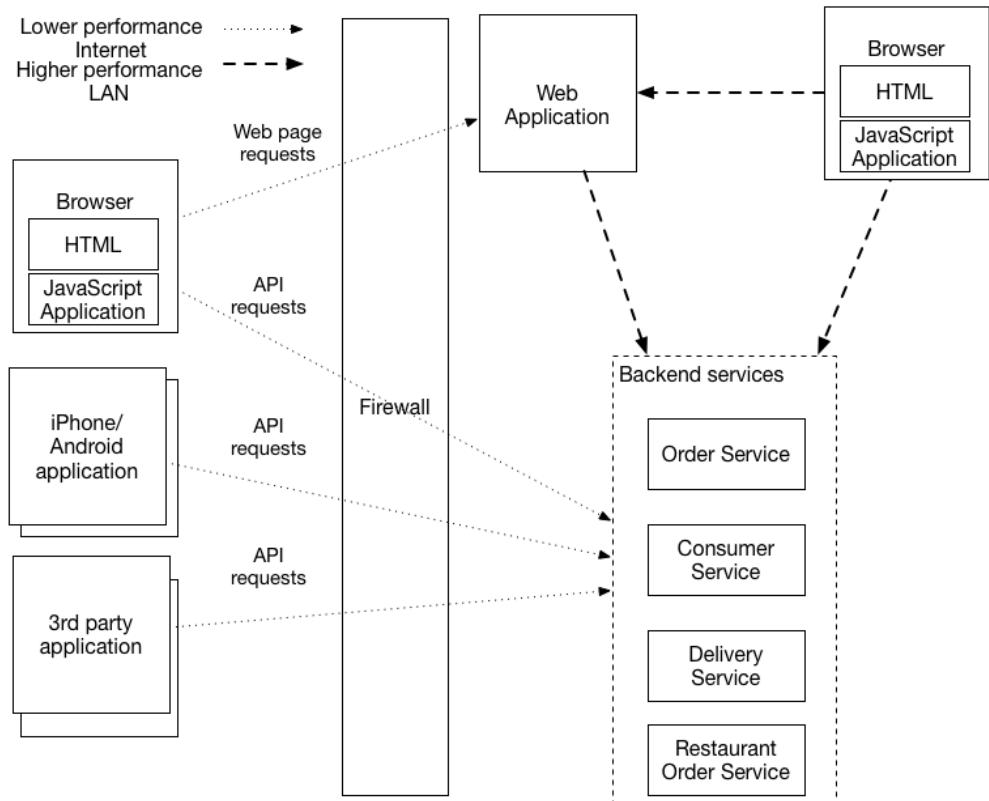
have a single, one-size fits all API.

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' APIs:

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

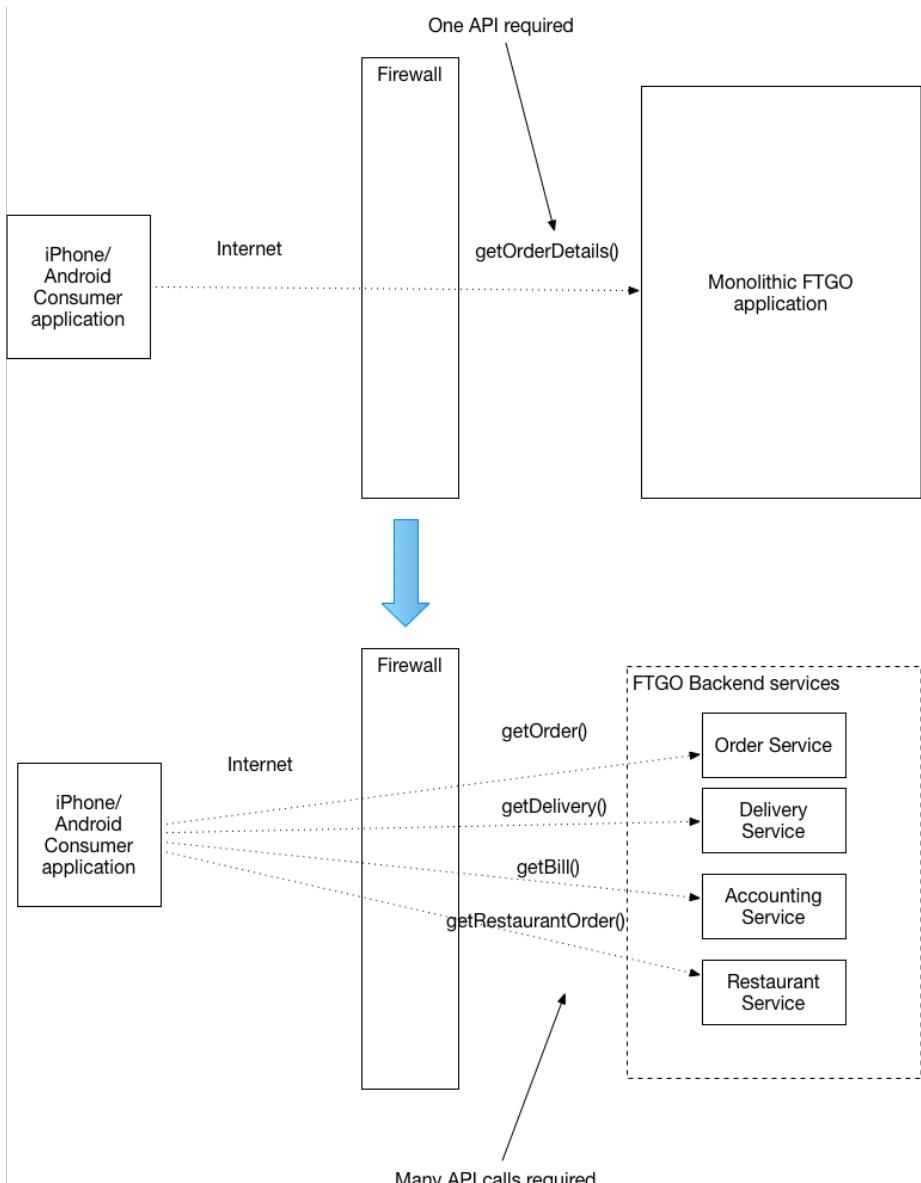
Consumers use the FTGO mobile client to place and manage their orders. Let's imagine that you are developing the mobile client's View Order view, which displays an order. As I described in chapter 7, the information displayed by this view includes basic order information including its status; its payment 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 Consumerweb 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

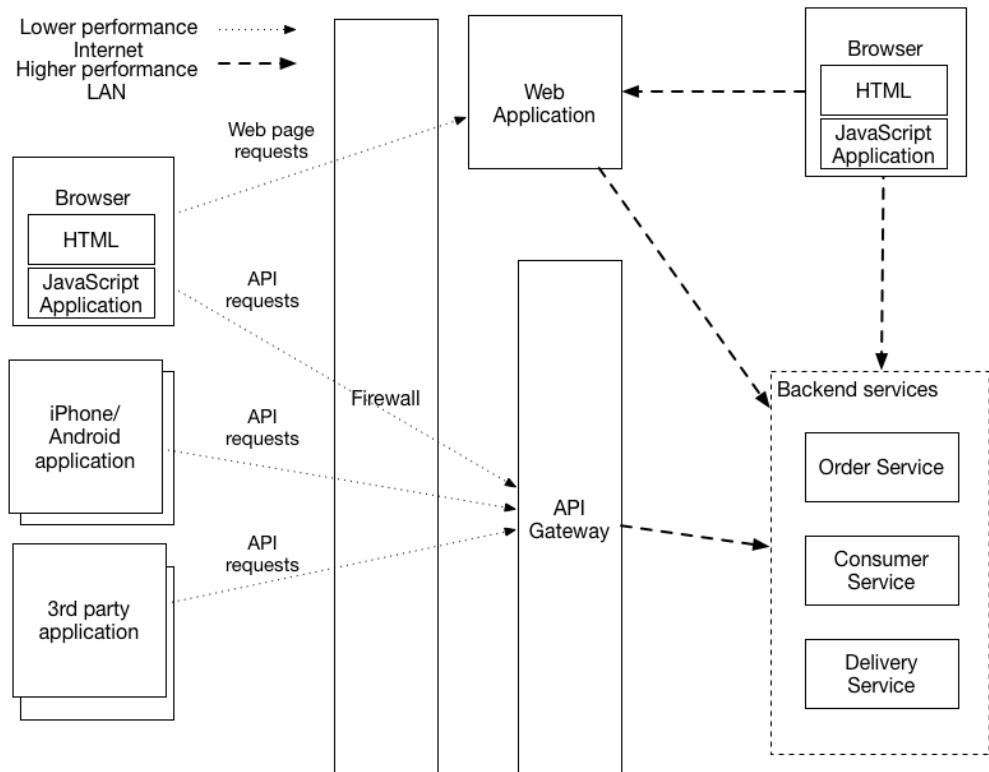
As you have just seen, there are numerous drawbacks with services accessing services directly. It is often not practical for a client to perform API composition over the Internet. The lack of encapsulation makes it difficult for developers to change service decomposition and APIs. Services sometimes 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 [“API design issues for the FTGO mobile client”](#), 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 for API requests 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 its clients. It might also have other responsibilities such as 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 for API calls from outside the firewall



The API gateway is responsible for request routing, API composition and protocol translation. All API 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

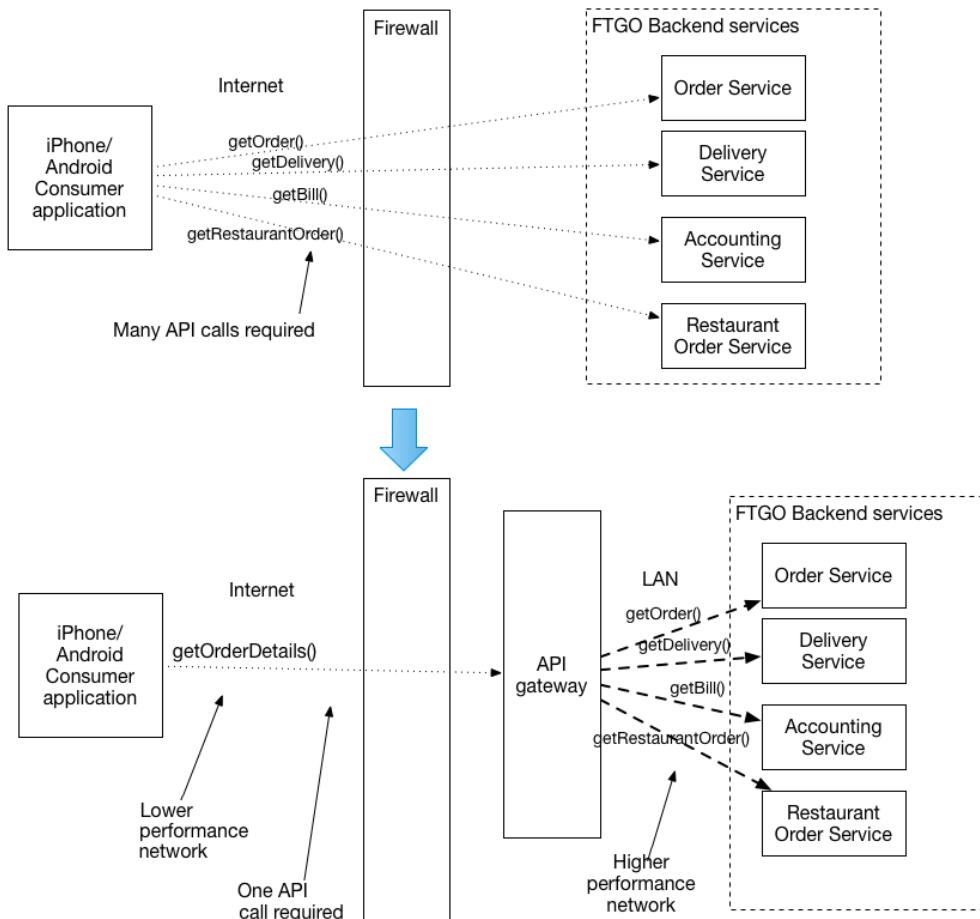
One of the key functions of an API gateway is request routing. An API gateway implements some API operations by simply routing requests to the corresponding service. When it receives a request, the API gateway consults a routing map that specifies which service to route the request to. A routing map 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 (e.g., 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.

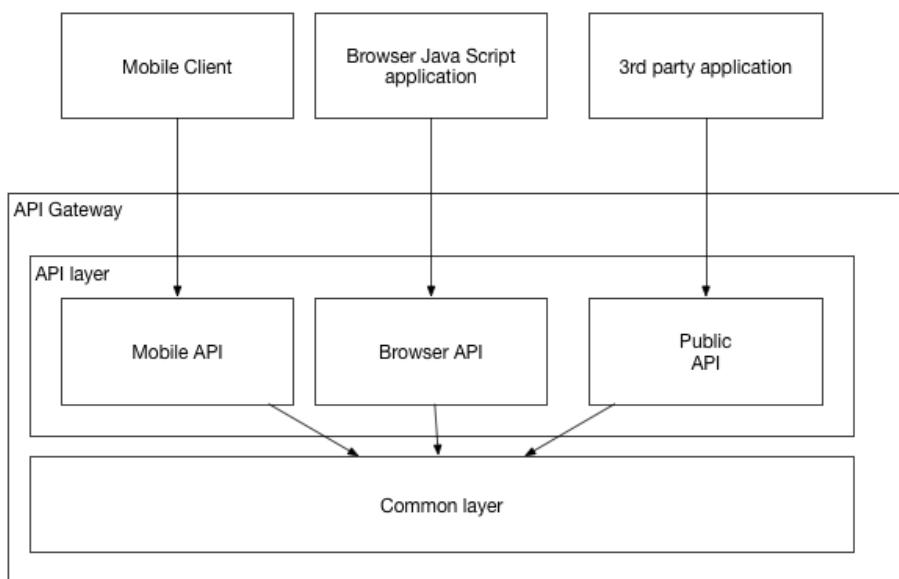
An important benefit of using a dedicated edge service is that it separates concerns. The API gateway focusses on API routing and composition. Another benefit is that it centralizes responsibility for critical edge functions such as authentication. This is particularly valuable when, as I describe below, an application has multiple API gateways that are possibly written using a variety of languages and frameworks. The drawback of this approach is that it increases network latency because of the extra hop. It also adds to the complexity of the application.

As a result, it's 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.

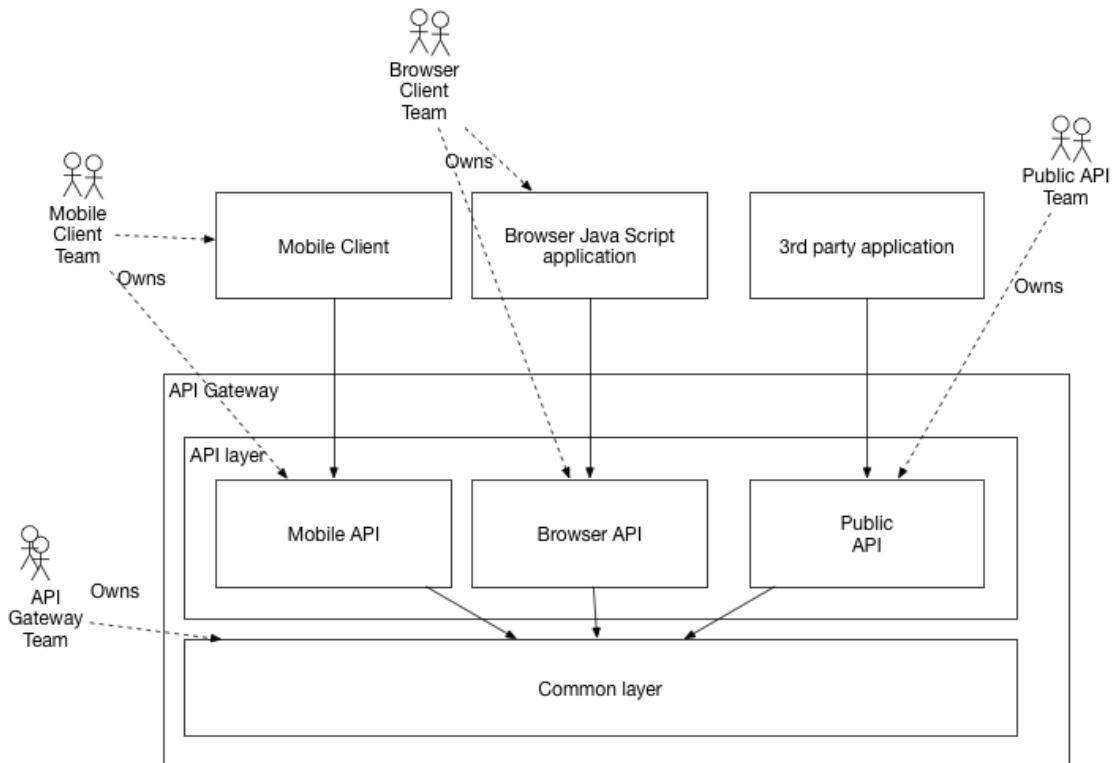
An API module implements other, more complex API operations using API composition. The implementation of this API operation consist of custom code. Each API operation implementation handles requests by invoking multiple services and combining the results.

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.



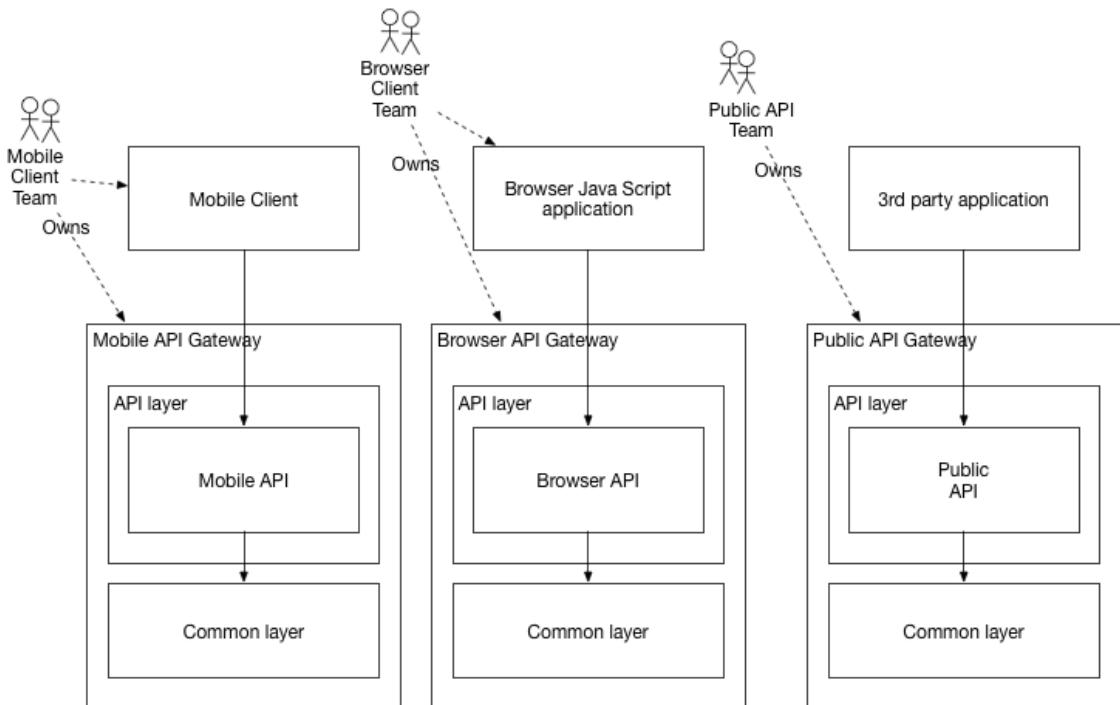
When a team needs to change their API, they simply check-in the changes to the source repository for the API gateway. Of course, in order to work well the API gateway's deployment pipeline must be fully automated. Otherwise, the client teams will often be blocked waiting for the API gateway team to deploy 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-all³⁹ 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 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

³⁹ www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15

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 results⁴⁰ 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.

⁴⁰ medium.com/netflix-techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

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
            .findRestaurantOrderById(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.

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 `CompletableFuture`s
- 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 as a thread and ultimately result in the API gateway being unable to handle any other requests. The solution, 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 developer 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 Balancer⁴¹, 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 your own API gateway.

8.3.2 Developing your own API gateway

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 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 an open source project from Pivotal.

⁴¹ aws.amazon.com/blogs/aws/new-aws-application-load-balancer/

Using Netflix Zuul

Netflix developed the Zuul⁴² 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 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 chapter 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 project⁴³, 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 provides the Mono abstraction that I use below

Spring Cloud Gateway provides a simple yet comprehensive way to

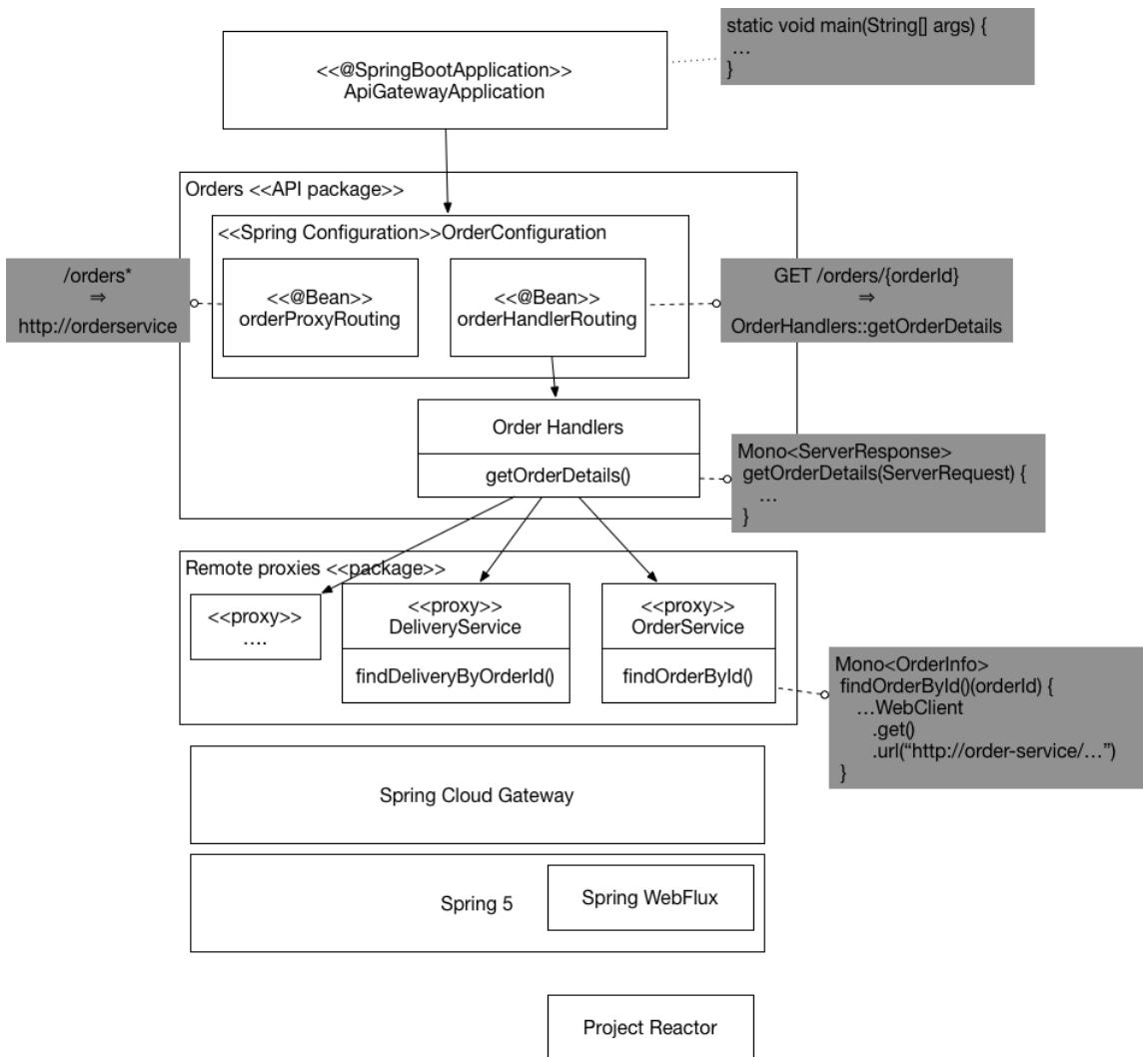
1. route requests to backend services
2. implement requests handlers that perform API composition
3. handle edge functions such as authentication.

Figure 8.8 shows the key parts of an API gateway that is built using this framework.

⁴² github.com/Netflix/zuul

⁴³ cloud.spring.io/spring-cloud-gateway/

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/*")))
            .and()
            ...
            .build();
    }

    @Bean
    public RouterFunction<ServerResponse>
        orderHandlerRouting(OrderHandlers orderHandlers) { ②
        return RouterFunctions.route(GET("/orders/{orderId}"),
            orderHandlers::getOrderDetails);
    }

    @Bean
    public OrderHandlers orderHandlers(OrderService orderService,
        RestaurantOrderService restaurantOrderService,
        DeliveryService deliveryService,
        ...
    )
}
```

```

        AccountingService accountingService) {
    return new OrderHandlers(orderService, restaurantOrderService,
                           deliveryService, accountingService);
}
}

```

- ① By default, route all requests whose path begins with /orders to the URL orderDestinations.orderServiceUrl
- ② Route a GET /orders/{orderId} to orderHandlers::getOrderDetails
- ③ 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)
            .onErrorReturn(Optional.empty()); 1  

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>,
          Optional<DeliveryInfo>, Optional<BillInfo>>> combined =
        Mono.when(orderInfo, restaurantOrderInfo, deliveryInfo, billInfo); 3  

4

    Mono<OrderDetails> orderDetails =
        combined.map(OrderDetails::makeOrderDetails); 5

    return orderDetails.flatMap(person -> ServerResponse.ok()
        .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 `Mono`s 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 `Mono`s. 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 `Mono`s 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 5.17, 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();
        return response.flatMap(resp -> resp.bodyToMono(OrderInfo.class)); ① 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

- Your application's external clients usually access the application's services via an API gateway. An API gateway provides each client with a custom API. It 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 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 can route 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.



Testing microservices

This chapter covers:

- Effective testing strategies for microservices
- Using the test pyramid to determine where to focus testing efforts
- Techniques for testing services in isolation
- Using consumer-driven contract testing to write tests that quickly yet reliably verify inter-service communication
- When and how to do end-to-end testing of applications

FTGO, like many organizations, had adopted a very traditional approach to testing. Testing is primarily an activity that happens after the development. The FTGO developers throw their code over a wall to the QA team, who verify that the software works as expected. What's more, most of their testing is done manually.

Sadly, this approach to testing is broken for two reasons. The first reason is manual testing is extremely inefficient. You should never ask a human to do what a machine can do better. Compared to machines, humans are slow and can't work 24x7. You won't be able to deliver software rapidly and safely if you rely on manual testing. It's essential that you write automated tests.

The second problem with the traditional approach to testing is that testing is done far too late in the delivery process. There certainly is a role for tests that critique an application after it's been written. But experience has shown us that those tests are insufficient. A much better approach is for developers to write automated tests as part of development. It improves their productivity since, for example, they'll have tests that provide immediate feedback while editing code.

Sadly, FTGO, like many organizations, has not fully embraced the idea of developers writing automated tests. The Sauce Labs state of testing report⁴⁴ describes how the majority of organizations are still manually testing their applications. Only 23% of organizations were mostly automated, and a minuscule 3% of organizations were fully automated!

The reliance on manual testing isn't because of a lack of tooling and frameworks. For example, JUnit, which is a popular Java testing framework, was first released in 1998. The reason for lack of automated tests is mostly cultural: "testing is QA's job", "it's not the best use of a developer's time", etc. It also doesn't help that developing a fast running, yet effective, maintainable test suite is challenging. Also, a typical large monolithic application is extremely difficult to test.

One key motivation for using the microservice architecture is, as I described in chapter 2, improving testability. Yet at the same time, the complexity of the microservice architecture demands that you use write automated tests. Furthermore, some aspects of testing microservices is challenging. That's because we need to verify that services can interact correctly while minimizing the number of slow, complex and unreliable end-to-end-tests that launch many services.

I begin this chapter by describing effective testing strategies for a microservices-based application that enable you to be confident that your software works while minimizing test complexity and execution time. After that I walk through examples of the different types of tests - unit, integration, component, and end-to-end - that you will need to write for your applications. This chapter is long but it covers testing ideas and techniques that are essential to modern software development in general, and the microservice architecture in particular. You will see comprehensive examples of the different kinds of tests that you will need to write for your services. Let's start by taking a look at testing strategies for microservices.

9.1 Testing strategies for microservice architectures

Let's imagine that you have made a change to the FTGO application's Order Service. Naturally, the next step is for you to run your code and verify that the change works correctly. One option is to test the change manually. First, you run the Order Service and all of its dependencies, which include infrastructure services, such as a database, but also other application services. Then you 'test' the service by either invoking its API or using the FTGO application's UI. The downside of this approach is that it's a slow, manual way to test your code.

A much better option is to have automated tests that you can run during development. Your development workflow should simply be edit code, run tests (ideally with a single keystroke), repeat. The fast running tests quickly tell you whether your changes work within a few seconds. But how do you write fast running tests? And, are they sufficient or do you need more comprehensive tests. These are the kind of questions that I answer in this and other sections in this chapter.

⁴⁴ saucelabs.com/news/sauce-labs-releases-third-annual-state-of-testing-survey-results

I start this section with an overview of important automated testing concepts. We will look at the purpose of testing, and the structure of a typical test. I cover the different types of tests that you will need to write. I also describe the test pyramid, which provides valuable guidance about where you should focus your testing efforts. After covering testing concepts, I then discuss strategies for testing microservices. We'll look at the distinct challenges of testing applications that have a microservice architecture. I describe techniques that you can use to write simpler and faster yet still effective tests for your microservices. Let's take a look at testing concepts.

9.1.1 Overview of testing

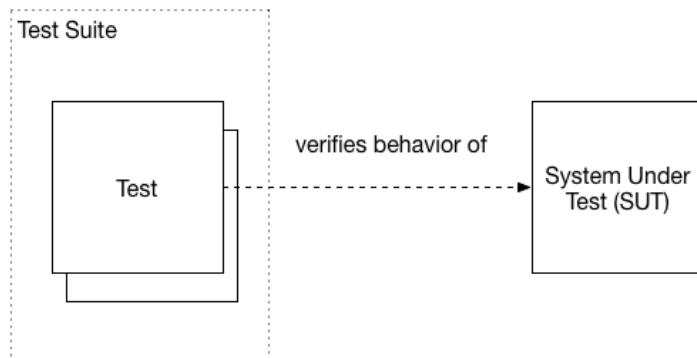
In this chapter, my focus is on automated testing and I use the term test as a shorthand for automated test. Wikipedia defines a test case, a.k.a. test, as follows:

A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

– https://en.wikipedia.org/wiki/Test_case

In other words, the purpose of a test is, as figure 9.1 shows, to verify the behavior of the system under test (SUT). In this definition, *system* is a fancy term that means the thing being tested. It might be something as small as a class, as large as the entire application or something in between, such as a cluster of classes or an individual service. A collection of related tests form a test suite.

Figure 9.1. The goal of a test is to verify the behavior of the System Under Test. An SUT might be as small as class or as large as an entire application.



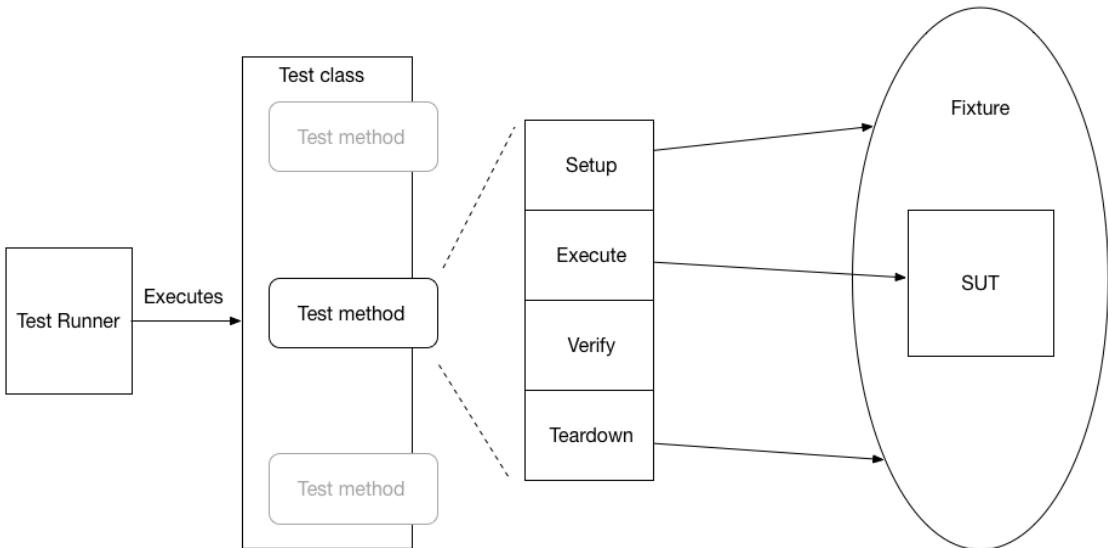
Let's first look at the concept of an automated test. I then discuss the different kinds of tests that you will need to write. After that, I discuss the test pyramid, which describes the relative proportions of the different types of tests that you should write.

Writing automated tests

Automated tests are usually written using a testing framework. JUnit, for example, is a popular Java testing framework. Figure 9.2 shows the structure of an automated test.

Each test is implemented by a test method, which belongs to a test class.

Figure 9.2. Each automated test is implemented by a test method, which belongs to a test class. A test consists of four phases: setup, which initializes the test fixture, which is everything required to run the test; execute, which invokes the SUT; verify, which verifies the outcome of the test; and teardown, which cleans up the test fixture.



An automated test typically consists of four phases⁴⁵:

1. setup - initialize the test fixture, which consists of the SUT and its dependencies, to the desired initial state. For example, create the class under test and initialize it to the state required for it to exhibit the desired behavior.
2. exercise - Invoke the SUT, e.g. invoke a method on the class under test
3. verify - Make assertions about the invocation's outcome and the state of the SUT. For example, verify the method's return value and the new state of the class under test.
4. teardown - clean up the test fixture, if necessary. Many tests omit this phase but some types of database test will, for example, roll back a transaction initiated by the setup phase.

In order to reduce code duplication and simplify the tests, a test class might have setup methods that are run before a test method, and teardown methods that are run afterwards. A test suite is a set of test classes. The tests are executed by a test runner.

Testing using mocks and stubs

An SUT often has dependencies. The trouble with dependencies is that they can complicate and slow down tests. For example, the OrderController class invokes

⁴⁵ xunitpatterns.com/Four%20Phase%20Test.html

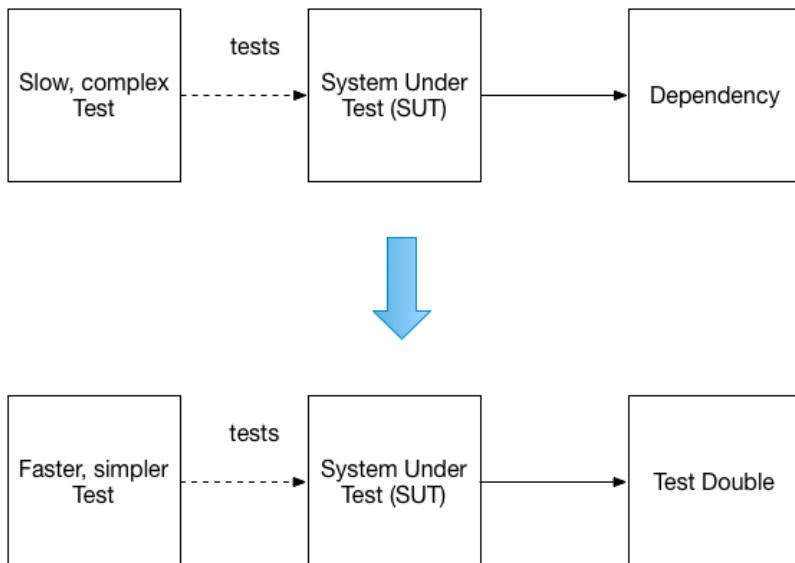
©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

the `OrderService`, which ultimately depends on numerous other application services and infrastructure services. It wouldn't be practical to test the `OrderController` class by running a large portion of the system. We need a way to test an SUT in isolation.

The solution is, as figure 9.3 shows, to replace the SUT's dependencies with test doubles. A test double is an object that simulates the behavior of the dependency.

Figure 9.3. Replacing a dependency with a test double enables the SUT to be tested in isolation. The test is simpler and faster.



There are two types of test doubles, stubs and mocks. The terms stubs and mocks are often used interchangeably although they have slightly different behavior. A stub is a test double that returns values to the SUT. A mock is a test double that a test uses to verify that the SUT correctly invokes a dependency. Also, a mock is often a stub.

Later on in this chapter, you will see examples of test doubles in action. For example, in section [“Developing unit tests for controllers”](#), I show how to test the `OrderController` class in isolation by using a test double for the `OrderService` class. In that example, the `OrderService` test double is implemented using Mockito, which is a popular mock object framework for Java. And, in section [“Developing component tests”](#), I show how to test the Order Service using test doubles for the other services that it invokes. Those test doubles respond to command messages sent by the Order Service. Let's now look at the different types of tests.

The different types of tests

There are many different types of tests. Some tests, such as performance tests and usability tests, verify that the application satisfies its quality of service requirements. In this chapter, I focus on automated tests that verify the functional aspects of the

application or service. I describe how to write four different types of tests:

- Unit tests - tests a small part of a service, such as a class
- Integration tests - verifies that a service can interact with infrastructure services, such as databases, and other application services
- Component tests - acceptance tests for an individual service
- end-to-end tests - acceptance tests for the entire application

They primarily differ in scope. At one end of the spectrum are unit tests, which verify behavior of the smallest meaningful program element. For an object-oriented language, such as Java, that is a class. At the other end of the spectrum are end-to-end tests which verify the behavior of an entire application. In the middle are component tests, which test individual services. Integration tests, as you will see below in section “[Writing integration tests](#)”, have a relatively small scope but they are more complex than pure unit tests. Scope is only one way of characterizing tests. Another way is to use the test quadrant.

Compile-time unit tests

Testing is an integral part of development. The modern development workflow consists of edit code - run tests. Moreover, if you are a Test-Driven Development (TDD) practitioner you develop a new feature or fix a bug by first writing a failing test and then writing the code to make it pass. And, even if you are not a TDD adherent, an excellent way to fix a bug is to write a test which reproduces the bug, and then write the code that fixes it.

The tests that you run as part of this workflow are known as compile-time tests. In a modern IDE, such as IntelliJ IDEA or Eclipse, you typically don't compile your code as a separate steps. Instead, you use a single keystroke to compile the code and run the tests. In order to stay in the flow, these tests need to execute quickly; ideally no more than a few seconds.

Using the test quadrant to categorize tests

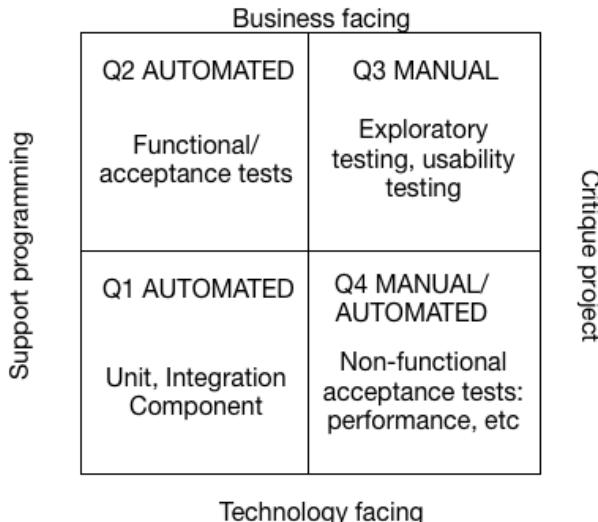
A good way to categorize tests is Brian Marick's test quadrant⁴⁶. The test quadrant, which is shown in figure 9.4, categorizes tests along two dimensions:

- the first dimension is whether the test is business facing or technology facing - a business facing test is described using the terminology of a domain expert whereas a technology facing test is described using the terminology of developers and the implementation.
- the second dimension is whether the goal of the test is to support programming or critique the application - developers use tests that supports programming as part of their daily work. Tests that critique the application aim to identify areas that need improvement.

Figure 9.4. The test quadrant categorizes tests along two dimension. The first dimension is whether a test is business facing or technology facing. The second dimension is whether the

⁴⁶ www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1

purpose of the test is to support programming or critique the application.



The test quadrant defines four different categories of tests:

- Q1 - support programming/technology facing - unit and integration tests
- Q2 - Support programming/business facing - component and end-to-end test
- Q3 - Critique application/business facing - usability and exploratory testing
- Q4 - Critique application/technology facing - non-functional acceptance tests, such as performance tests

The test quadrant is not the only way of organizing tests. There is also the test pyramid, which provides guidance on how many tests of each type to write.

Using the test pyramid as guide to where to focus your testing efforts

We must write different kinds of tests in order to be confident that our application works. The challenge, however, is that the execution time, and complexity of a test increases with its scope. Also, the larger the scope of a test and the more moving parts that it has, the less reliable it becomes. Unreliable tests are almost as bad as no tests since if you can't trust a test you are likely to ignore failures.

On one end of the spectrum are unit tests for individual classes. They are fast to execute, easy to write and reliable. At the other end of the spectrum are end-to-end tests for the entire application. These tend to be slow, difficult to write and often unreliable because of their complexity. Since we don't have unlimited budget for development and testing, we want to focus on writing tests that have small scope without compromising the effectiveness of the test suite.

The test pyramid⁴⁷, which is shown in figure 9.5, is a good guide. At the base of the

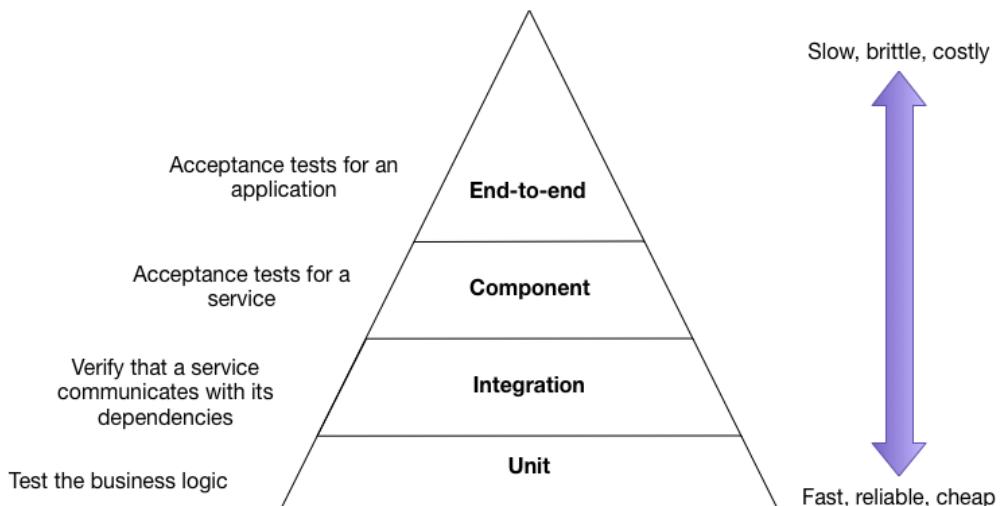
⁴⁷ martinfowler.com/bliki/TestPyramid.html

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

pyramid are the fast, simple and reliable unit tests. At the top of the pyramid are the slow, complex and brittle end-to-end tests. Like the USDA food pyramid although more useful and less controversial⁴⁸, the test pyramid describes the relative proportions of each type of test.

Figure 9.5. The test pyramid describes the relative proportions of each type of test that you need to write. As you move up the pyramid you should write fewer and fewer tests.



The key idea of the test pyramid is that as we move up the pyramid we should write fewer and fewer tests. We should write lots of unit tests and very few end-to-end tests. As you will see in this chapter, I describe strategy that emphasizes testing the pieces of a service. It even minimizes the number of component test, which test an entire service. But before looking at the details of the testing strategy, let's take a look at the deployment pipeline, which runs the automated tests.

It's clear how to test individual microservices, such as the `Consumer Service`, which don't depend on any other services. But what about services, such as `Order Service`, that depend on numerous other services? And, how can we be confident that the application as a whole works? This is the key challenge of testing applications that have a microservice architecture. The complexity of testing has moved from the individual services to the interactions between them. Let's look at how to tackle this problem.

9.1.2 The challenge of testing microservices

Inter-process communication plays a much more important role in a microservices-

⁴⁸ en.wikipedia.org/wiki/History_of_USDA_nutrition_guides

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

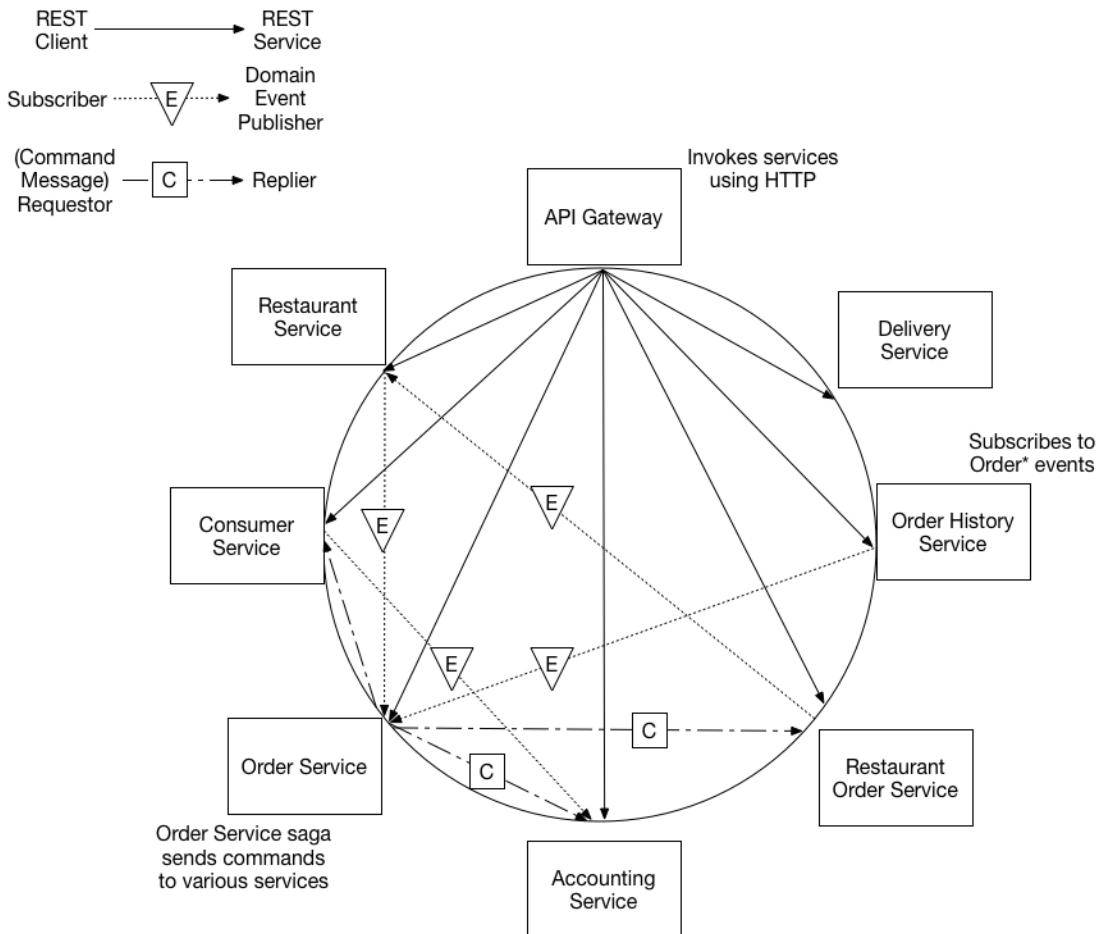
<https://forums.manning.com/forums/microservices-patterns>

based application than in a monolithic application. A monolithic application might communicate with a few external clients and services. For example, the Monolithic version of the FTGO application uses a few third party web services, such as Stripe for payments, Twilio for messaging, and Amazon SES for email, which have stable APIs. Any interaction between the modules of the application is through programming language-based APIs. Inter-process communication is very much on the edge of the application.

In contrast, inter-process communication is central to microservice architecture. A microservices-based application is a distributed system. Teams are constantly developing their services and evolving their APIs. It's essential that developers of a service write tests that verify that their service interacts with its dependencies and clients.

As I described in chapter 3, services communicate with each other using a variety of interaction styles and IPC mechanisms. Some services use request/reply-style interaction that's implemented using a synchronous protocol, such as REST or gRPC. Other services interact through request/asynchronous reply or publish/subscribe using asynchronous messaging. For instance, figure 9.6 shows how some of the services in the FTGO application communicate. Each arrow points from a consumer service to a producer service.

Figure 9.6. Some of the inter-service communication in the FTGO application. Each arrow points from a consumer service to a producer service.



The arrow points in the direction of the dependency from the consumer of the API to the provider of the API. The assumptions that a consumer makes about an API depends on the nature of the interaction:

- REST client → Service - the API gateway routes requests to services and implements API composition.
- Domain event consumer → publisher - the Order History Service consumes events published by the Order Service.
- Command message requestor → replier - the Order Service sends command messages to various services and consumes the replies.

Each interaction between a pair of services represents an agreement or contract between the two services. The Order History Service and the Order Service must, for example, agree on the event message structure and the channel that they are

published to. Similarly, the API gateway and the services must agree on the REST API endpoints. And, the Order Service and each service that it invokes using request/async reply, must agree on the command channel and the format of the command and reply messages.

As a developer of a service, you need to be confident that the services that you consume have stable APIs. Similarly, you don't want to unintentionally make breaking changes to your service's API. For example, if you are working on the Order Service you want to be sure that the developers of your service's dependencies, such as Consumer Service and Restaurant Order Service, don't change their APIs in ways that incompatible with your service. Similarly, you must ensure that you don't you don't change the Order Services's API in a way that breaks the API Gateway or Order History Service.

One way to verify that two services can interact is to run both services, invoke an API that triggers the communication, and verify that it has the expected outcome. This will certainly catch integration problems but it's basically an end-to-end. The test could very likely need to run numerous other transitive dependencies of those services. A test might also need to invoke complex, high-level functionality, such as business logic, even its goal is to test relatively low-level IPC. It's best to avoid writing end-to-end tests like these. Somehow, we need to write faster, simpler, and more reliable tests that ideally test services in isolation. The solution is to use what is known as consumer-driven contract testing.

Consumer-driven contract testing

Let's imagine that you are member of the team developing the API Gateway, which I described in chapter 8. The API Gateway's OrderServiceProxy invokes various REST endpoints including, such the GET /orders/{orderId} endpoint. It's essential that we write tests that verify that the API Gateway and the Order Service agree on an API. In the terminology of consumer contract testing, the two services participate in a consumer-provider relationship. The API Gateway is a consumer and the Order Service is a provider. A consumer contract test is an integration test for a provider, such as the Order Service, that verifies that its API matches the expectations of a consumer, such as the API Gateway.

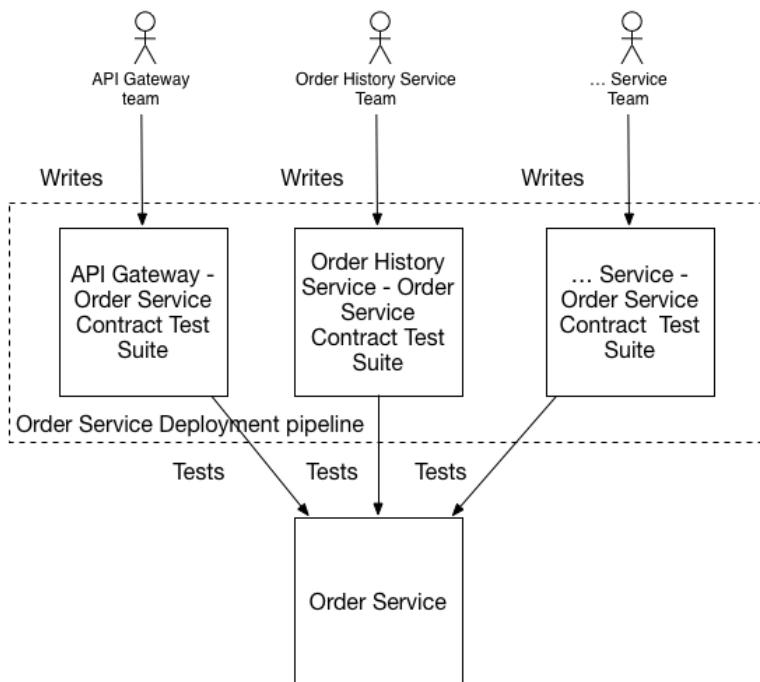
A consumer contract test focusses on verifying that the "shape" of provider's API meets the consumer's expectations. For a REST endpoint, a contract test verifies that the provider implements an endpoint that

- has the expected HTTP method and path
- accepts the expected headers, if any
- accepts a request body, if any
- returns a response with the expected status code, headers and body

It is important to remember that contract tests do not thoroughly test the provider's business logic. That's the job of unit tests. Later on, for example, you will see that consumer contract tests for a REST API are in fact mock controller tests.

The team that develops the consumer writes a contract test suite and adds it (e.g. via a pull request) to the provider's test suite. The developers of other services that invoke the Order Service also contributes a test suite, as shown in figure 9.7. Each test suite will test those aspects of the Order Service's API that are relevant to each consumer. The test suite for the Order History Service, for example, verifies that the Order Service publishes the expected events.

Figure 9.7. Each team that develops a service that consumes the Order Service's API contributes a contract test suite. The test suite verifies that the API matches the consumer's expectations. This test suite, along with those contributed by other teams, is run by the Order Service's deployment pipeline.



These test suites are executed by the deployment pipeline for the Order Service. If a consumer contract test fails then that tells the producer team that they have made a breaking change to the API. They must either fix the API or talk to the consumer team.

Consumer-driven contract tests typically use testing by example. The interaction between a consumer and provider is defined by a set of examples, which are also known as contracts. Each contract consists of example messages that are exchanged during one interaction. For instance, a contract for a REST API consists of an example HTTP request and response. On the surface, it might seem better to define the interaction using schemas written using, for example, OpenAPI or JSON schema. However, it turns out that schemas aren't that useful when writing tests. A test can validate the response using the schema but it still needs to invoke the provider with an

example request. What's more, consumer tests also need example responses.

That's because even though the focus of consumer-driven contract testing is to test a provider, contracts are also used to verify that the consumer conforms to the contract. For instance, a consumer-side contract test for a REST client uses the contract to configure an HTTP stub service that verifies that the HTTP request matches the contract's request and sends back's the contract's HTTP response. Testing both sides of interaction ensures that the consumer and provider really do agree on the API. Later on we will look at examples of how to this kind of testing. But first, let's look at how to write consumer contract tests using Spring Cloud Contract.

Testing services using Spring Cloud Contract

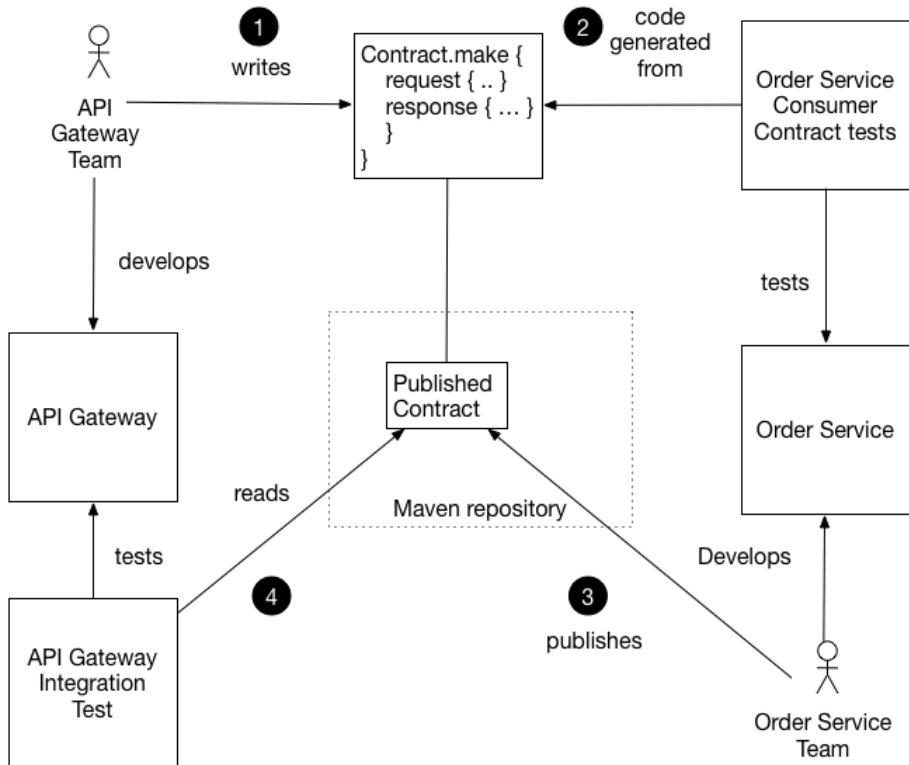
Two popular contract testing frameworks are Spring Cloud Contract ⁴⁹, which is a consumer contract testing framework for Spring applications, and the Pact family of frameworks⁵⁰, which support a variety of languages. The FTGO application is a Spring framework-based application so in this chapter I am going to describe how to use Spring Cloud Contract. It provides a Groovy Domain Specific Language (DSL) for writing contracts. Each contract is a concrete example of an interaction between a consumer and a provider, such as an HTTP request and response. Spring Cloud Contract code generates contract tests for the provider. It also configures mocks, such as a mock HTTP server, for consumer integration tests.

Let's imagine, for example, that you are working on the API Gateway and want to write a consumer contract test for the Order Service. Figure 9.8 shows the process, which requires you to collaborate with the Order Service teams. You write contracts that define how the API Gateway interacts with the Order Service. The Order Service uses these contracts to test the Order Service and you use them to test the API Gateway.

⁴⁹ cloud.spring.io/spring-cloud-contract/

⁵⁰ github.com/pact-foundation

Figure 9.8. The API Gateway team writes the contracts. The Order Service team uses those contracts to test the Order Service and publishes them to a repository. The API Gateway team uses the published contracts to test the API Gateway.



The sequence of steps is as follows:

1. You write one or contracts, such as the one shown in listing 9.1. Each contract consists of an HTTP request that the API Gateway might send to the Order Service and an expected HTTP response. You give the contracts, perhaps via a Git pull request, to the Order Service team.
2. The Order Service team tests the Order Service using consumer contract tests, which Spring Cloud Contract code generates from contracts
3. The Order Service team publishes the contracts that tested the Order Service to a Maven repository.
4. You use the published contracts to write tests for the API Gateway

Because you test the API Gateway using the published contracts, you can be confident that it works with the deployed Order Service.

The contracts are the key part of this testing strategy. Listing 9.1 shows an example contract. It consists of an HTTP request and an HTTP response.

Listing 9.1. A Spring Cloud Contract contract, which describes how the API Gateway invokes the Order Service

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/orders/1223232'
    }
    response {
        status 200
        headers {
            header('Content-Type': 'application/json;charset=UTF-8')
        }
        body("{...}")
    }
}
```

- ① The HTTP request's method and path
- ② The HTTP response's status code, headers and body

The request element is an HTTP request for the REST endpoint `GET /orders/{orderId}`. The response element is an HTTP response, which describes an Order, that is expected by the API Gateway. The Groovy contracts are part of the provider's code base. Each consumer team writes contracts, which describe how their service interacts with the provider, and gives them, perhaps via a Git pull request, to the provider team. The provider team is responsible for packaging the contracts as a JAR and publishing them to a Maven repository. The consumer-side tests download the JAR from the repository.

Each contract's request and response play dual roles of test data and the specification of expected behavior. In a consumer-side test, the contract is used to configure a stub, which is similar to a Mockito mock object and simulates the behavior of the Order Service. It enables the API Gateway to be tested running the Order Service. In the provider-side test, the generated test class invokes the provider with contract's request and verifies that it returns a response that matches contract's response. Later on in section “[Writing integration tests](#)”, I describe the details of how to use Spring Cloud Contract, but first let's look at how to use consumer contract testing for messaging APIs.

Consumer contract tests for messaging APIs

A REST client isn't the only kind of consumer that has expectations of a provider's API. Services that subscribe to domain events and use command/reply-based messaging are also consumers. They consume some other service's messaging API and so make assumptions about the nature of that API. We must also write consumer contract tests for these services.

Spring Cloud Contract also provides support for testing messaging-based interactions. The structure of a contract and how it used by the tests depends on the type of interaction. A contract for domain event publishing consists of an example domain

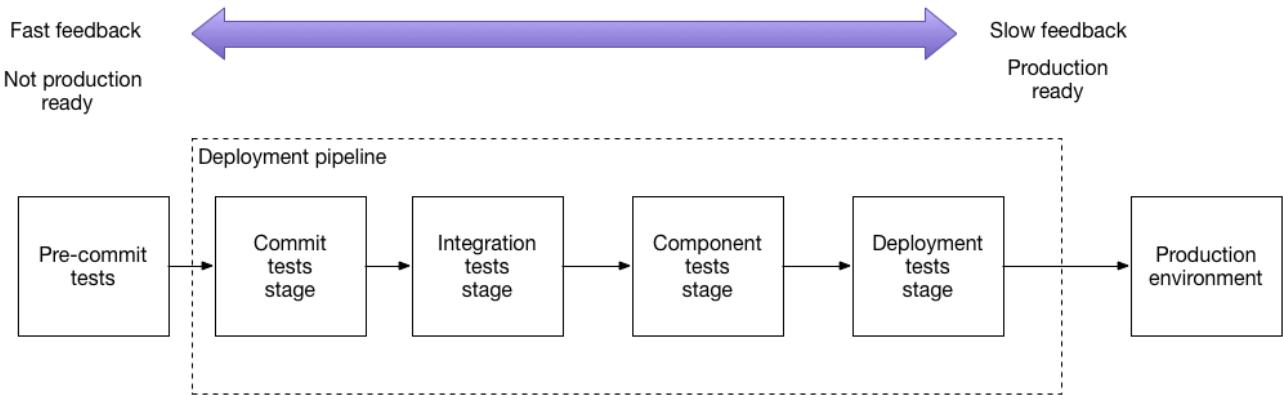
event. A provider test causes the provider to emit an event and verifies that the matches the contract's event. A consumer test verifies that the consumer can handle that event. In section “[Writing integration tests](#)”, I describe an example test.

A contract for a Command/reply messaging is similar to an HTTP contract. It consists of a request message and a reply message. A provider test invokes the API with the contract's command message and verifies that the reply matches the contract's reply. A consumer test uses the contract to configure a stub subscriber, which listens for the contract's command message and replies with. In section “[Writing integration tests](#)”, I describe an example test. But first take a look at the deployment pipeline, which runs these and other tests.

9.1.3 The deployment pipeline

Every service has a deployment pipeline⁵¹. A deployment pipeline is the process for getting code from the developer's desktop into production. As figure 9.9 shows, it consists of a series of stages that execute test suites, followed by a stage that releases or deploys the service. Ideally, it is fully automated but it might contain manual steps. A deployment pipeline is often implemented using a Continuous Integration (CI) server, such as Jenkins.

Figure 9.9. An example deployment pipeline for the Order Service. It consists of a series of stages. The pre-commit tests are run by the developer prior to committing their code. The remaining stages are executed by an automated tool, such as the Jenkins CI server.



As code flows through the pipeline, the test suites subject it to increasingly more thorough testing in environments that are more production like. At the same time, the execution time of each test suite typically grows. The idea is to provide feedback about test failures as rapidly as possible.

The example deployment pipeline shown in figure 9.9 consists of the following stages:

- Pre-commit tests stage - runs the unit tests. Executed by the developer before

⁵¹ Jez Humble, Continuous delivery book

committing their changes.

- Commit tests stage - compiles the service, runs the unit tests, and performs static code analysis
- Integration tests stage - runs the integration tests
- Component tests stage - runs the component tests for the service
- Deploy stage - deploys the service into production

The CI server runs the commit stage when a developer commits a change. It executes extremely quickly and so provides rapid feedback about the commit. The later stages take longer to run and so provide less immediate feedback. If all of the tests pass, the final stage in this pipeline deploys it into production.

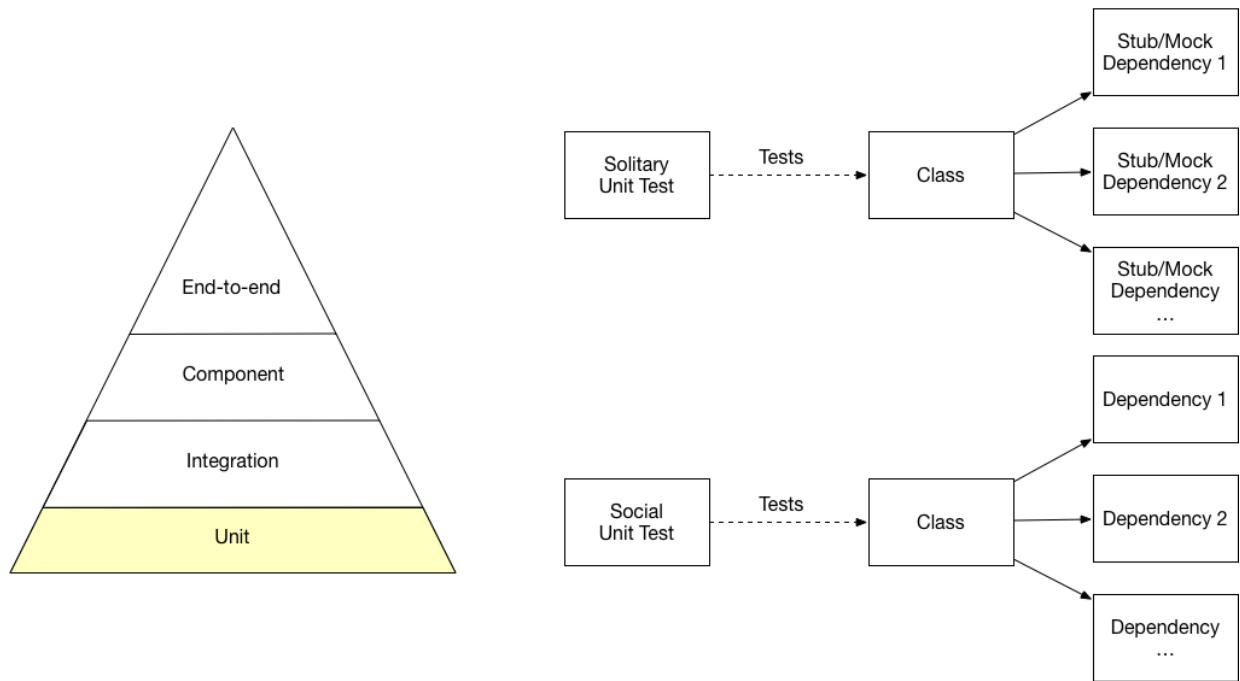
In this example, the deployment pipeline is fully automated all the way from commit to deployment. There are, however, situations that require manual steps. For example, you might need a manual testing stage, such as a staging environment. In such a scenario, the code progresses to the next stage when a tester clicks a button to indicate that was successful. Alternatively, a deployment pipeline for an on-premise product would simply release the new version of the service. Later on, the released services would be packaged into a product release and shipped to customers. Now that we have looked at the organization of the deployment pipeline and when it executes the different types of tests, let's head to the bottom of the test pyramid and look at how to write unit tests for a service.

9.2 Writing unit tests for a service

Let's imagine that you want to write a test that verifies that the FTGO application's `Order Service` correctly calculates the subtotal of an `Order`. You could write tests that run the `Order Service`, invoke its REST API to create an `Order` and check that HTTP response contains the expected values. The drawback of this approach is that not only is the test complex but it is also slow. If these tests were the compile-time tests for the `Order` class then you would waste a lot of time waiting for it to finish. A much more productive approach is to write unit tests for the `Order` class.

As figure 9.10 shows, unit tests are the lowest level of the test pyramid. They are technology facing tests that support development. A unit test verifies that a unit, which is a very small part of a service, works correctly. A unit is typically a class, and so the goal of unit testing is to verify that it behaves as expected.

Figure 9.10. Unit tests are the base of the pyramid. They are fast running, easy to write and reliable. A solitary unit test tests a class in isolation using mocks or stubs for its dependencies. A social unit test tests a class and its dependencies.

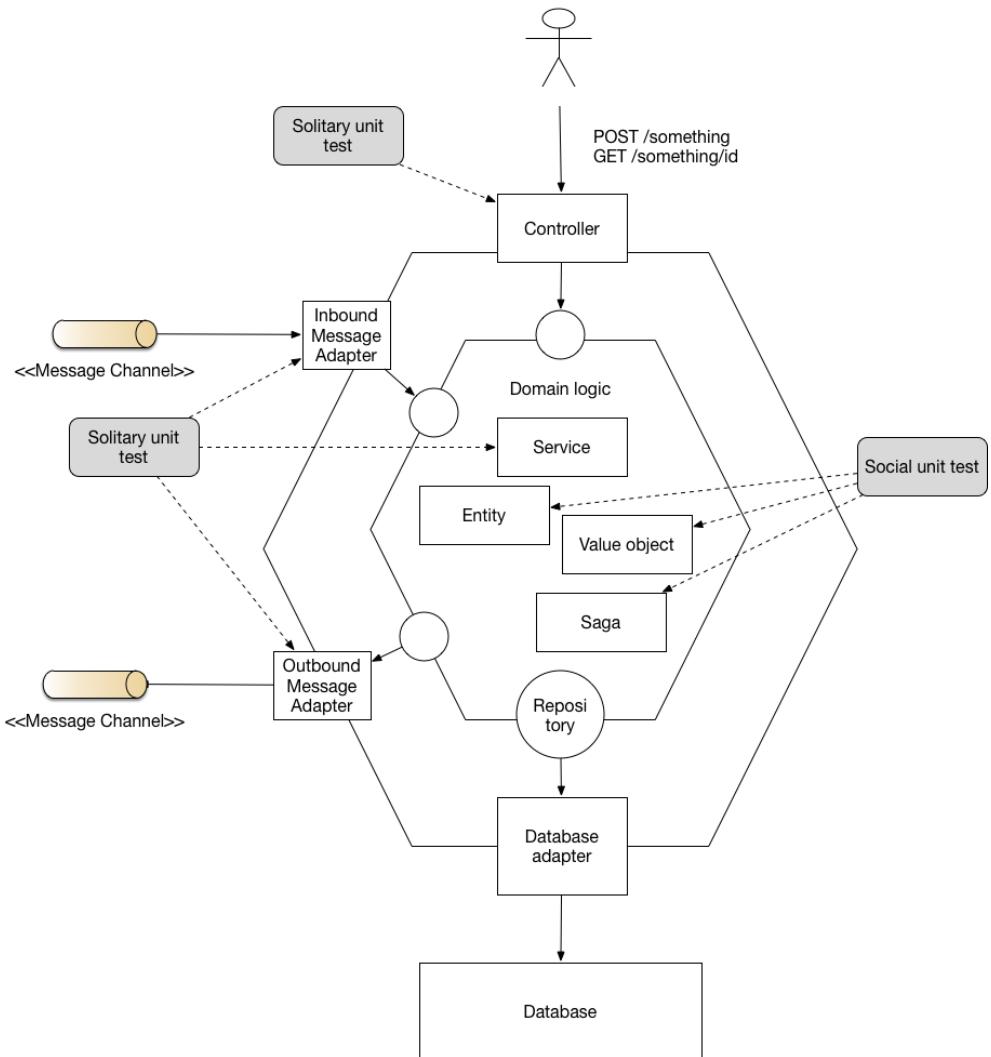


There are two types of unit tests:

- solitary unit test - tests a class in isolation using mock objects for the class's dependencies
- social unit test - tests a class and its dependencies.

The responsibilities of the class determine which type of test to use. Figure 9.11 shows the type of unit test that you will typically use to test the classes that comprise a typical service. Controller and service classes are often tested using solitary unit tests. Domain objects, such as entities and value objects, are typically tested using social unit tests.

Figure 9.11. The responsibilities of a class determine whether to use a solitary or social unit test.



The typical testing strategy for each class is as follows:

- Entities, such as `Order`, which as I described in chapter 5 are objects with persistent identity, are tested using social unit tests.
- Value objects, such as `Money`, which as I described in chapter 5 are objects that are collections of values, are tested using social unit tests.
- Sagas, such as `CreateOrderSaga`, which as I described in chapter 4 maintain data consistency across services, are tested using social unit tests.
- Domain services, such as `OrderService`, which as I described in chapter 5 are

classes that implement business logic which doesn't belong in entities or value objects, are tested using solitary unit tests

- Controllers, such as `OrderController`, which handle HTTP requests, are tested using solitary unit tests
- Inbound and outbound messaging gateways are tested using solitary unit tests

Let's begin by looking at how to test entities.

9.2.1 Developing unit tests for entities

Listing 9.2 shows the `shouldCalculateTotal()` test for the `Order` entity. It has an `@Before` `setUp()` method that creates an `Order` before running each test. Its `@Test` methods might further initialize the `Order`, invoke one of its methods, and then make assertions about the return value and the state of the `Order`.

Listing 9.2. A simple, fast running, unit test for the Order entity. It is a social unit test for the Order class and its dependencies.

```
public class OrderTest {

    private ResultWithEvents<Order> createResult;
    private Order order;

    @Before
    public void setUp() throws Exception {
        createResult = Order.createOrder(CONSUMER_ID, AJANTA_ID,
CHICKEN_VINDALOO_LINE_ITEMS);
        order = createResult.result;
    }

    @Test
    public void shouldCalculateTotal() {
        assertEquals(CHICKEN_VINDALOO_PRICE.multiply(CHICKEN_VINDALOO_QUANTITY),
order.getOrderTotal());
    }

    ...
}
```

The `@Test` `shouldCalculateTotal()` method verifies that `Order.getOrderTotal()` returns the expected value. Unit tests thoroughly tests the business logic. You can use them as compile-time tests since they execute extremely quickly. The `Order` class relies on the `Money` value object and so it's important to test that class as well. Let's look at how to do that.

9.2.2 Writing unit tests for value objects

Value objects are immutable and so they tend to be easy to test. You don't have to worry about side-effects. A test for value object typically creates a value object in a particular state, invokes one of its methods and make assertions about the return value. Listing 9.3 shows the tests for the `Money` value object, which is a simple class that

represents a money value. These tests verify the behavior of the Money class's methods including add(), which adds two Money objects and multiply(), which multiplies a Money by an integer.

Listing 9.3. A simple, fast running test for the Money value object. It's a solitary test since the Money class does not depend on any other application classes.

```
public class MoneyTest {

    private final int M1_AMOUNT = 10;
    private final int M2_AMOUNT = 15;

    private Money m1 = new Money(M1_AMOUNT);
    private Money m2 = new Money(M2_AMOUNT);

    @Test
    public void shouldAdd() {①
        assertEquals(new Money(M1_AMOUNT + M2_AMOUNT), m1.add(m2));
    }

    @Test
    public void shouldMultiply() {②
        int multiplier = 12;
        assertEquals(new Money(M2_AMOUNT * multiplier), m2.multiply(multiplier));
    }

    ...
}
```

- ① Verify that two Money objects can be added together
- ② Verify that a Money object can be multiplied by an integer

Entities and value objects are the building blocks a service's business logic. Some business logic, however, also resides in the service's sagas and services. Let's look at how to test those.

9.2.3 Developing unit tests for sagas

A saga, such as the CreateOrderSaga class, implements important business logic and so needs to be tested. It's a persistent object that sends command messages to saga participants and processes their replies. As I described in chapter 4, the CreateOrderSaga exchanges Command/Reply messages with several services, such as the Consumer Service and the Restaurant Order Service. A test for this class creates a saga and verifies that it sends the expected sequence of messages to the saga participants. One test that you need to write is for the happy path. You must also write tests for the various scenarios where the saga rolls back because a saga participant sent back a failure message.

One approach would be to write tests that use a real database and message broker along with stubs simulate the various saga participants. For example, a stub for the Consumer Service would subscribe to the consumerService command channel and send back the

desired reply message. However, tests written using this approach would be quite slow. A much more effective approach is write tests that mock those classes that interact with the database and message broker. That way, we can just focus on testing the saga's core responsibility.

Listing 9.4 shows a test for the `CreateOrderSaga`. It's written using the Eventuate Tram Saga testing framework⁵². This framework provides an easy to use DSL that abstracts away the details of interacting with sagas. With this DSL, you can create a saga and verify that it sends the correct command messages. Under the covers, the Saga testing framework configures the saga framework with mocks for the database and messaging infrastructure.

Listing 9.4. A simple, fast running unit test for the `CreateOrderSaga`. It is a social unit test that tests the saga class and its dependencies.

```
public class CreateOrderSagaTest {

    @Test
    public void shouldCreateOrder() {
        given()
            .saga(new CreateOrderSaga(restaurantOrderServiceProxy),
                  new CreateOrderSagaData(ORDER_ID,
                                         CHICKEN_VINDALOO_ORDER_DETAILS)).①
        expect()
            .command(new ValidateOrderByConsumer(CONSUMER_ID, ORDER_ID,
                                                 CHICKEN_VINDALOO_ORDER_TOTAL)).
            to(ConsumerServiceChannels.consumerServiceChannel).
        andGiven()
            .successReply().②
        expect()
            .command(new CreateRestaurantOrder(AJANTA_ID, ORDER_ID, null)).
            to(RestaurantOrderServiceChannels.restaurantOrderServiceChannel);
    }③

    @Test
    public void shouldRejectOrderDueToConsumerVerificationFailed() {
        given()
            .saga(new CreateOrderSaga(restaurantOrderServiceProxy),
                  new CreateOrderSagaData(ORDER_ID,
                                         CHICKEN_VINDALOO_ORDER_DETAILS)).④
        expect()
            .command(new ValidateOrderByConsumer(CONSUMER_ID, ORDER_ID,
                                                 CHICKEN_VINDALOO_ORDER_TOTAL)).
            to(ConsumerServiceChannels.consumerServiceChannel).
        andGiven()
            .failureReply().⑤
        expect()
            .command(new RejectOrderCommand(ORDER_ID)).
            to(OrderServiceChannels.orderServiceChannel);⑥
    }
}
```

⁵² github.com/eventuate-tram/eventuate-tram-sagas

- 1 Create the saga
- 2 Verify that it sends a `ValidateOrderByConsumer` message to the Consumer Service
- 3 Send a Success reply to that message
- 4 Verify that it sends a `CreateRestaurantOrder` message to the Restaurant Order Service
- 5 Send a failure reply indicating that the Consumer Service rejected the Order
- 6 Verify that the saga sends a `RejectOrderCommand` message to the Order Service

The `@Test shouldCreateOrder()` method tests the happy path. The `@Test shouldRejectOrderDueToConsumerVerificationFailed()` method tests the scenario where the Consumer Service rejects the order. It verifies that the `CreateOrderSaga` sends a `RejectOrderCommand` to compensate for the consumer being rejected. The `CreateOrderSagaTest` class has methods that test other failure scenarios. Let's now look at how to test domain services.

9.2.4 Writing unit tests for domain services

The majority of a service's business logic is implemented by the entities, value objects and sagas. Domain service classes, such as the `OrderService` class, implement the remainder. This class is a typical domain service class. Its methods invoke entities and repositories. They also publish domain events. An effective way to test this kind of class is to use a mostly solitary unit test, which mocks dependencies such as repositories and messaging classes.

Listing 9.5 shows the `OrderServiceTest` class, which tests the `OrderService`. It uses the Mockito framework to create mocks for the service's dependencies. Each test implements the test phases as follows:

1. setup - configures the mock objects for the service's dependencies
2. execute - invokes a service method
3. verify - verifies that the value returned by the is correct and that the dependencies have been invoked correctly

Listing 9.5. A simple, fast running unit test for the `OrderService` class. It's a solitary unit test, which uses Mockito mocks for the service's dependencies.

```
public class OrderServiceTest {

    private OrderService orderService;
    private OrderRepository orderRepository;
    private DomainEventPublisher eventPublisher;
    private RestaurantRepository restaurantRepository;
    private SagaManager<CreateOrderSagaData> createOrderSagaManager;
    private SagaManager<CancelOrderSagaData> cancelOrderSagaManager;
    private SagaManager<ReviseOrderSagaData> reviseOrderSagaManager;

    @Before
    public void setup() {
        orderRepository = mock(OrderRepository.class);
        eventPublisher = mock(DomainEventPublisher.class);
        restaurantRepository = mock(RestaurantRepository.class);
    }
}
```

1

```

createOrderSagaManager = mock(SagaManager.class);
cancelOrderSagaManager = mock(SagaManager.class);
reviseOrderSagaManager = mock(SagaManager.class);
orderService = new OrderService(orderRepository, eventPublisher,
        restaurantRepository, createOrderSagaManager,
        cancelOrderSagaManager, reviseOrderSagaManager); ②
}

@Test
public void shouldCreateOrder() {
    when(restaurantRepository
        .findOne(AJANTA_ID)).thenReturn(AJANTA_RESTAURANT); ③
    when(orderRepository.save(any(Order.class))).then(invocation -> { ④
        Order order = (Order) invocation.getArguments()[0];
        order.setId(ORDER_ID);
        return order;
    });

    Order order = orderService.createOrder(CONSUMER_ID, ⑤
        AJANTA_ID, CHICKEN_VINDALOO_MENU_ITEMS_AND_QUANTITIES);

    verify(orderRepository).save(same(order)); ⑥
    verify(eventPublisher).publish(Order.class, ORDER_ID, ⑦
        singletonList(
            new OrderCreatedEvent(CHICKEN_VINDALOO_ORDER_DETAILS)));

    verify(createOrderSagaManager) ⑧
        .create(new CreateOrderSagaData(ORDER_ID,
            CHICKEN_VINDALOO_ORDER_DETAILS),
            Order.class, ORDER_ID);
}
}

```

- ① Create Mockito mocks for the OrderService's dependencies
- ② Create an OrderService injected with mock dependencies
- ③ Configure RestaurantRepository.findOne() to return the Ajanta restaurant
- ④ Configure OrderRepository.save() to set the Order's id
- ⑤ Invoke OrderService.create()
- ⑥ Verify that the OrderService saved the newly created Order in the database
- ⑦ Verify that the OrderService published an OrderCreatedEvent
- ⑧ Verify that the OrderService created a CreateOrderSaga

The `setUp()` method creates an `OrderService` injected with mock dependencies. The `@Test` `shouldCreateOrder()` method verifies that the `OrderService.createOrder()` invokes the `OrderRepository` to save the newly created `Order`, publishes an `OrderCreatedEvent`, and creates a `CreateOrderSaga`. Now that we have looked at how to unit test the domain logic classes, let's look at how to unit test the adapters that interact with external systems.

9.2.5 Developing unit tests for controllers

Services, such as the Order Service, typically have one or more controllers, which handle HTTP requests from other services and the API gateway. A controller class consists of a set of request handler methods. Each method implements an REST API endpoint. A method's parameters represent values from the HTTP request, such as path variables. It typically invokes a domain service or a repository and returns a response object. The OrderController, for instance, invokes the OrderService and the OrderRepository. An effective testing strategy for controllers are solitary unit tests that mock the services and repositories.

You could write a test class similar to the OrderServiceTest class: simply instantiate a controller class and invoke its methods. However, this approach doesn't test some important functionality, such request routing. It's much more effective to use a mock MVC testing framework, such as Spring Mock Mvc, which is part of the Spring framework, or Rest Assured Mock MVC, which builds on Spring Mock Mvc. Tests written using one of these frameworks make what appear to be HTTP requests and make assertions about HTTP responses. These frameworks enable you to test HTTP request routing and conversion of Java objects to and from JSON without having to make actual network calls. Under the covers, Spring Mock Mvc instantiates just enough of the Spring MVC classes to make this possible.

Are these really unit tests?

Since these tests use the Spring framework you might argue that they are not unit tests. They are certainly more heavy weight than the unit tests I've described so far. The Spring Mock Mvc documentation refers to these as out-of-servlet-container⁵³ integration tests. However, Rest Assured Mock MVC footnote[github.com/rest-assured/rest-assured/wiki/Usage#spring-mock-mvc-module] describes these tests as unit tests. Despite this debate over terminology, these are important tests to write.

Listing 9.6 shows the OrderControllerTest class, which tests the Order Service's OrderController. It is written using RestAssured Mock MVC, which provides a simple DSL that abstracts away the details of interacting with controllers. RestAssured makes it easy to send a mock HTTP request to a controller and verify the response. The OrderControllerTest creates a controller that is injected with Mockito mocks for the OrderService and OrderRepository. Each test configures the mocks, makes an HTTP request, verifies that the response is correct, and possibly verifies that controller invoked the mocks.

⁵³ docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#spring-mvc-test-vs-end-to-end-integration-tests

Listing 9.6. A simple, fast running, unit test for the OrderController class. It's a solitary unit test, which uses mocks for the OrderController's dependencies.

```
public class OrderControllerTest {

    private OrderService orderService;
    private OrderRepository orderRepository;

    @Before
    public void setUp() throws Exception {
        orderService = mock(OrderService.class);
        orderRepository = mock(OrderRepository.class);
        orderController = new OrderController(orderService, orderRepository);
    } ➊

    @Test
    public void shouldFindOrder() {
        when(orderRepository.findOne(1L)).thenReturn(CHICKEN_VINDALOO_ORDER); ➋

        given().➌
            standaloneSetup(configureControllers(
                new OrderController(orderService, orderRepository)));
        when().➍
            get("/orders/1");
        then().➎
            statusCode(200).
            body("orderId", ➏
                equalTo(new Long(OrderDetailsMother.ORDER_ID).intValue()).
            body("state",
                equalTo(OrderDetailsMother.CHICKEN_VINDALOO_ORDER_STATE.name()).
            body("orderTotal",
                equalTo(CHICKEN_VINDALOO_ORDER_TOTAL.asString()))
        ;
    }

    @Test
    public void shouldFindNotOrder() { ... }

    private StandaloneMockMvcBuilder controllers(Object... controllers) { ... }
}
```

- ➊ Create mocks for the OrderController's dependencies
- ➋ Configure the mock OrderRepository to return an Order
- ➌ Configure the OrderController
- ➍ Make an HTTP request
- ➎ Verify the response status code
- ➏ Verify elements of the JSON response body

The `shouldFindOrder()` test method first configures the `OrderRepository` mock to return an `Order`. It then makes an HTTP request to retrieve the order. Finally, it checks that the request was successful and that the response body contains the expected data.

Controllers aren't the only adapters that handle requests from external systems. There are also event/message handlers so let's look at how to unit test those.

9.2.6 Writing unit tests for event and message handlers

Services often process messages sent by external systems. The Order Service, for example, has `OrderEventConsumer`, which is a message adapter that handles domain events published by other services. Like controllers, message adapters tend to be simple classes that invoke domain services. Each of a message adapter's methods typically invokes a service method with data from the message or event.

We can unit test message adapters using an approach that similar to the one that we used for unit testing controllers. Each test instances the message adapter, sends a message to a channel and verifies that the service mock was invoked correctly. Behind the scenes, however, the messaging infrastructure is stubbed so no message broker is involved. Let's look at how to test the `OrderEventConsumer` class.

[Listing 9.7](#) shows part of the `OrderEventConsumerTest` class, which tests `OrderEventConsumer`. It verifies that `OrderEventConsumer` routes each event to the appropriate handler method and correctly invokes the `OrderService`. The tests uses the Eventuate Tram Mock Messaging framework, which provides an easy to use DSL for writing mock messaging tests that uses the same given-when-then format as RestAssured. Each test instantiates `OrderEventConsumer` injected with a mock `OrderService`, publishes a domain event, and verifies that the `OrderEventConsumer` correctly invokes the service mock.

Listing 9.7. A fast running unit test for the `OrderEventConsumer` class, which is a message adapter that handles domain events published by other services. It's a solitary test, which uses mocks for the `OrderEventConsumer`'s dependencies.

```
public class OrderEventConsumerTest {

    private OrderService orderService;
    private OrderEventConsumer orderEventConsumer;

    @Before
    public void setUp() throws Exception {
        orderService = mock(OrderService.class);
        orderEventConsumer = new OrderEventConsumer(orderService); ①
    }

    @Test
    public void shouldCreateMenu() {

        given(). ②
            eventHandlers(orderEventConsumer.domainEventHandlers());
        when().
            aggregate("net.chrisrichardson.ftgo.restaurantservice.domain.Restaurant",
                      AJANTA_ID).
            publishes(new RestaurantCreated(AJANTA_RESTAURANT_NAME,
                                           RestaurantMother.AJANTA_RESTAURANT_MENU)) ③
    }
}
```

```

        then().
            verify(() -> {
                verify(orderService)
                    .createMenu(AJANTA_ID,
                        new RestaurantMenu(RestaurantMother.AJANTA_RESTAURANT_MENU_ITEMS));
            })
        ;
    }
}

```

4

- ➊ Instantiate OrderEventConsumer with mocked dependencies
- ➋ Configure OrderEventConsumer domain handlers
- ➌ Publish a RestaurantCreated event
- ➍ Verify that OrderEventConsumer invoked OrderService.createMenu()

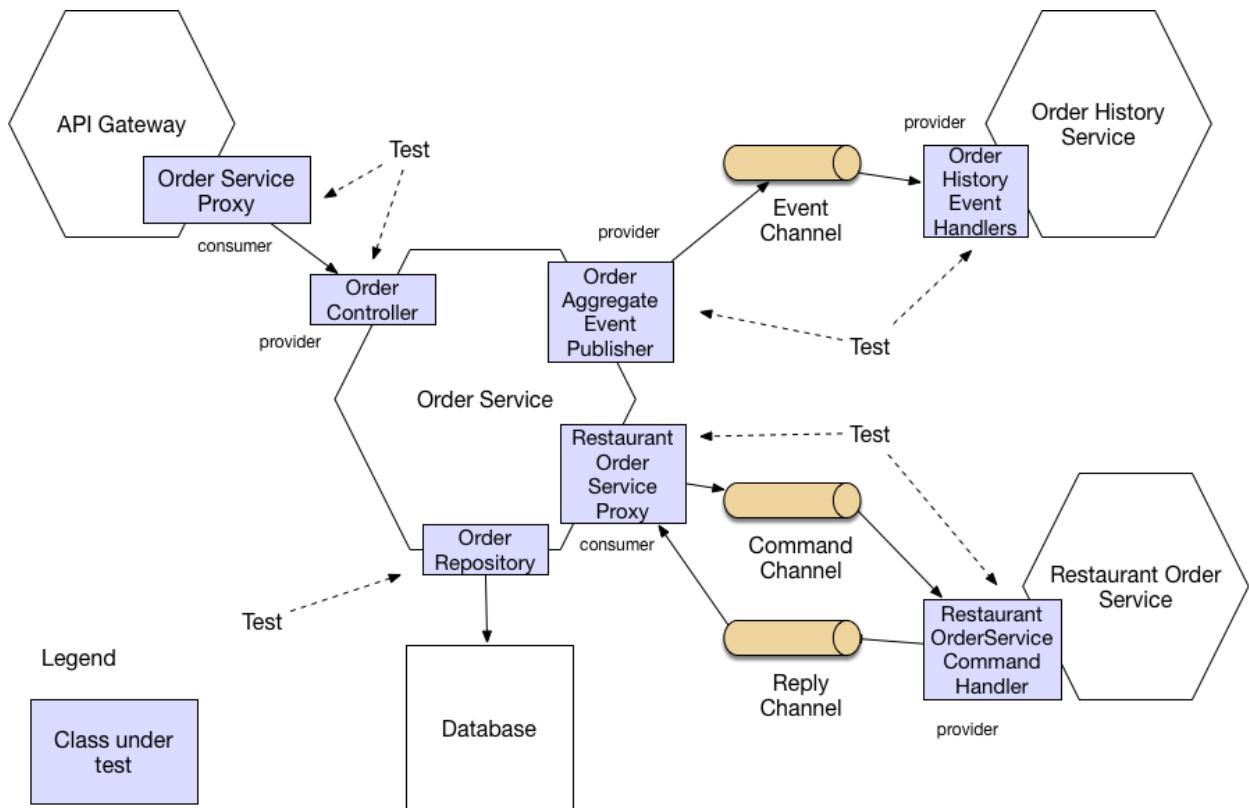
The `setUp()` method creates an `OrderEventConsumer` injected with a mock `OrderService`. The `shouldCreateMenu()` method publishes a `RestaurantCreated` event and verifies that the `OrderEventConsumer` invoked `OrderService.createMenu()`. The `OrderEventConsumerTest` class along with the other unit test classes execute extremely quickly. The unit tests run in just a few seconds.

The unit tests, however, don't verify that a service, such as the `Order Service`, properly interacts with other services. For example, the unit tests do not verify that the `Order` class be persisted in MySQL. Nor do they verify that the `CreateOrderSaga` sends command messages in the right format to the right message channel. And they don't verify that the `RestaurantCreated` processed by `OrderEventConsumer` has the same structure as the event published by the `Restaurant Service`. In order to verify that a service properly interacts with other services we must write integration tests. Let's look at how to do that.

9.3 Writing integration tests

Services typically interact other services. For example, the `Order Service`, as figure 9.12 shows, interacts with several services. Its REST API is consumed by the `API Gateway` and its domain events are consumed by services including the `Order History Service`. The `Order Service` uses several other services. It persists `Orders` in MySQL; It also sends commands to and consumes replies from several other services, such as the `Restaurant Order Service`

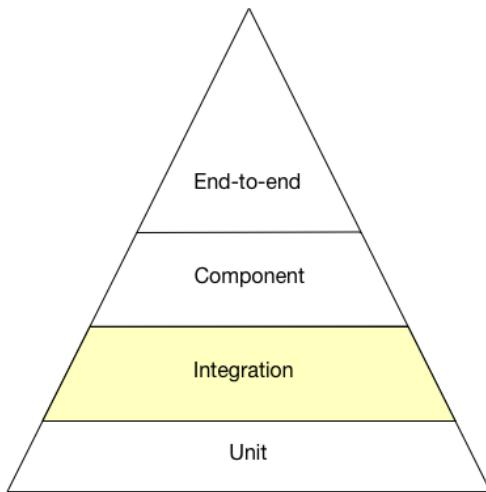
Figure 9.12. Integration tests must verify that a service can communicate with its clients and dependencies. But instead of testing whole services, the strategy is to test the individual adapter classes that implement the communication.



In order to be confident that a service, such as the Order Service, works as expected, we must write tests that verify that a service can properly interact with infrastructure services and other application services. One approach is to launch all the services, and test them through their APIs. This, however, is what is known as end-to-end testing, which is slow, brittle and costly. As I describe below in section “[Writing end-to-end tests](#)”, there is sometimes a role for end-to-end testing but it’s at the top of the test pyramid and so you want to minimize the number of end-to-end tests.

A much more effective strategy is to write what are known as integration tests. As figure 9.13 shows, integration are the layer above unit tests in the testing pyramid. They verify that a service can properly interact with infrastructure services and other services. However, unlike end-to-end tests, they don’t launch services. Instead, we use a couple of strategies that significantly simplifies the tests without impacting their effectiveness.

Figure 9.13. Integration tests are the layer above unit tests. They verify that a service can communicate with its dependencies, which includes infrastructure services, such as the database, and application services.



The first strategy is to test the adapter and perhaps its supporting classes instead of testing entire services. For example, in section “[Persistence integration tests](#)” you will see a JPA persistence test, which verifies that Orders are persisted correctly. Rather than testing this through the Order Service’s API, it directly tests the OrderRepository class. Similarly, in section “[Integration testing publish/subscribe-style interactions](#)” you will see a test for the OrderDomainEventPublisher that verifies that the Order Service publishes correctly structured domain events by testing. The benefit of testing only a small number of classes is that the tests are significantly simpler and faster.

The second strategy for simplifying integration tests that verify interactions between application services is to use contracts, which I described earlier in section “[Consumer-driven contract testing](#)”. A contract is a concrete example of an interaction between a pair of services. As table 9.1 shows, the structure of a contract depends on the type of interaction between the services.

Table 9.1. The structure of the contract depends on the type of interaction between the services.

Interaction style	Consumer	Provider	Contract
REST-based, request/reply	API Gateway	Order Service	HTTP request and response
Publish/subscribe	Order History Service	Order Service	Domain event
Request/async reply	Order Service	Restaurant Order Service	Command message and reply message

A contract consists of either one message, in the case of publish/subscribe style interactions, or two messages, in the case of request/reply and request/async reply style interactions.

The contracts are used to test both the consumer and the provider, which ensures that they agree on the API. They are used in slightly different ways depending on whether you are testing the consumer or the provider:

- consumer-side tests - these are tests for the consumer's adapter. They use the contracts to configure stubs that simulate the provider. They enable you write integration tests for a consumer that don't require a running provider.
- provider-side tests - these are tests for the provider's adapter. They use the contracts to test the adapters using mocks for the adapters's dependencies.

Later on this section, I describe examples of these types of tests but let's first look at how to write persistence tests.

9.3.1 Persistence integration tests

Services typically store data in a database. For instance, the `Order Service` persists aggregates, such as `Order`, in MySQL using JPA. Similarly, the `Order History Service` maintains a CQRS view in AWS DynamoDB. The unit tests that we wrote earlier only test in-memory objects. In order to be confident that a service works correctly, we must write persistence integration tests, which verify that a service's database access logic works as expected. In the case of the `Order Service`, this means testing the JPA repositories, such as the `OrderRepository`.

Each phase of a persistence integration test behaves as follows:

- Setup - set up the database by creating the database schema and initializing it to a known state. It might also begin a database transaction
- Execute - perform a database operation
- Verify - make assertions about the state of the database and objects retrieved from the database
- Teardown - an optional phase that might undo the changes made to the database by, for example, rolling back the transaction that was started by the setup phase.

Listing 9.8 shows a persistent integration test for the `Order` aggregate and `OrderRepository`. Apart from relying on JPA to create the database schema, the persistence integration tests don't make any assumption about the state of the database. Consequently, tests don't need to roll back the changes that they make to the database, which avoids problems with the ORM caching data changes in memory.

Listing 9.8. An integration test for the Order JPA entity and its repository. It verifies that an Order can be persisted in the database.

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = OrderJpaTestConfiguration.class)
public class OrderJpaTest {
```

```

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private TransactionTemplate transactionTemplate;

    @Test
    public void shouldSaveAndLoadOrder() {

        Long orderId = transactionTemplate.execute((ts) -> {
            Order order = new Order(CONSUMER_ID, AJANTA_ID, CHICKEN_VINDALOO_LINE_ITEMS);
            orderRepository.save(order);
            return order.getId();
        });

        transactionTemplate.execute((ts) -> {
            Order order = orderRepository.findOne(orderId);

            assertNotNull(order);
            assertEquals(OrderState.CREATE_PENDING, order.getState());
            assertEquals(AJANTA_ID, order.getRestaurantId());
            assertEquals(CONSUMER_ID, order.getConsumerId().longValue());
            assertEquals(CHICKEN_VINDALOO_LINE_ITEMS, order.getLineItems());
            return null;
        });
    }
}

```

The `shouldSaveAndLoadOrder()` test method executes two transactions. The first saves a newly created `Order` in the database. The second transaction load the `Order` and verifies that its fields are properly initialized.

One problem that you need to solve is how to provision the database that is used persistence integration tests. An effective solution to run an instance of the database during testing is to use Docker. Later in section “[Developing component tests](#)” I describe how to use Docker Compose Gradle plugin automatically run services during component testing. You can use a similar approach to run MySQL, for example, during persistence integration testing.

The database is only one of the external services with which a service interacts. Let’s now look at how to write integration tests for inter-service communication between application services starting with REST.

9.3.2 **Integration testing REST-based request/reply style interactions**

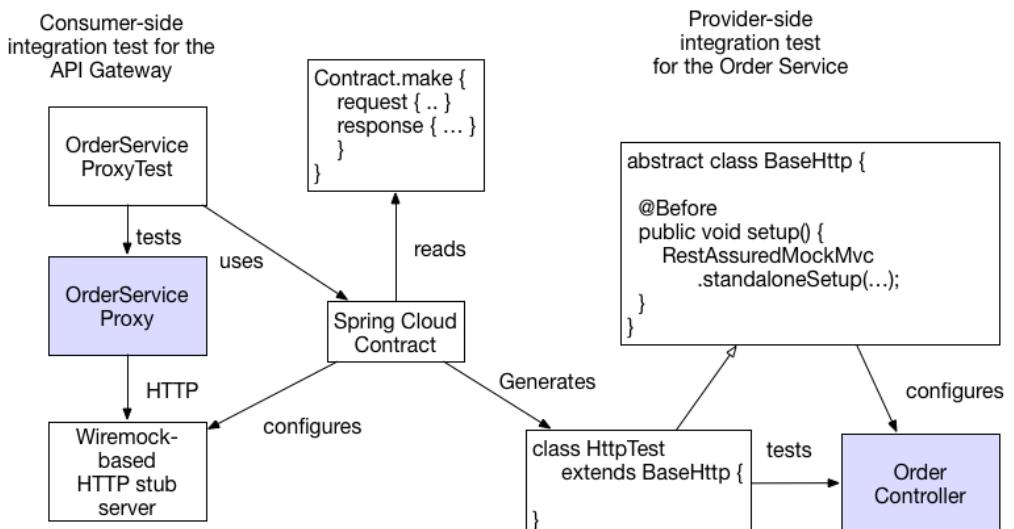
REST is a widely used inter-service communication mechanism. The REST client and REST service must agree on the REST API, which includes the REST endpoints and the structure of the request and response bodies. The client must send an HTTP request to the correct endpoint and the service must send back the response that the client expects.

For example, in chapter 8 I described how the FTGO application's API Gateway makes REST API calls to numerous services including The OrderService's GET /orders/{orderId} endpoint is one of the endpoints invoked by the API Gateway. In order to be confident that the API Gateway and Order Service can communicate without using an end-to-end tests we need to write integration tests

As I described earlier in section "[Consumer-driven contract testing](#)", a good integration testing strategy is to use consumer-driven contract tests. The interaction between the API Gateway and GET /orders/{orderId} can be described using a set of HTTP-based contracts. Each contract consists of an HTTP request and an HTTP reply. The contracts are used to test the API Gateway and the Order Service.

Figure 9.14 shows how to use Spring Cloud Contract to test REST-based interactions. The consumer-side API Gateway integration tests use the contracts to configure an HTTP stub server that simulates the behavior of the Order Service. A contract's request specifies an HTTP request from the API gateway and contract's response specifies the response that the stub sends back to the API Gateway. Spring Cloud Contract uses the contracts to code generate the provider-side Order Service integration tests, which test the controllers using Spring Mock MVC or RestAssuredMockMvc. The contract's request specifies the HTTP request to make to the controller and the contract's response specifies the controller's expected response.

Figure 9.14. The contracts are used to verify that the adapter classes on both sides of the REST-based communication between the API Gateway and the Order Service conform to the contract. The consumer-side tests verify that the OrderServiceProxy, which is the REST client and the provider-side tests verify that the OrderController, which implements the REST API endpoints.



The consumer-side OrderServiceProxyTest invokes the OrderServiceProxy, which

has been configured to make HTTP requests to WireMock. Wiremock is a tool for efficiently mocking HTTP servers and in this test it simulates the Order Service. Spring Cloud Contract manages WireMock and configure it to respond to the HTTP requests defined by the contracts.

On the provider-side, Spring Cloud Contract generates a test class called `HttpTest`, which use `RestAssuredMockMvc` to test the Order Service's controllers. Test classes, such as `HttpTest`, must extend a handwritten base class. In this example, the base class is called `BaseHttp`, which instantiates the `OrderController` injected with mock dependencies and calls `RestAssuredMockMvc.standaloneSetup()` to configure Spring MVC. Let's take a closer look at how this works starting with an example contract.

An example contract for a REST API

A REST contract, such as the one shown in listing 9.9, specifies an HTTP request, which is sent by the REST client, and the HTTP response, which the client expects to get back from REST server. A contract's request specifies the HTTP method, the path and optional headers. A contract's response specifies the HTTP status code, optional headers and when appropriate the expected body.

Listing 9.9. A Spring Cloud Contract contract that describes an HTTP-based request/reply style interaction.

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/orders/1223232'
    }
    response {
        status 200
        headers {
            header('Content-Type': 'application/json;charset=UTF-8')
        }
        body('''{"orderId" : "1223232", "state" : "CREATE_PENDING"}''')
    }
}
```

This particular contract describes a successful attempt by the API Gateway to retrieve an Order from the Order Service. Let's now look at how to use this contract to write integration tests, starting with the tests for the Order Service.

Consumer-driven contract integration tests for the Order Service

The consumer-driven contract integration tests for the Order Service verifies that its API meets its clients' expectations. Listing 9.10 shows the `HttpBase`, which is the base class for the test class code generated by Spring Cloud Contract. It's responsible for the setup phase of the test. It creates the controllers injected with mock dependencies and configures those mocks to return values cause the controller to generate the expected response.

Listing 9.10. The abstract base class for the tests that are code generated by Spring Cloud Contract. It configures just enough of Spring MVC to route requests to the OrderController, which it injects with mocks.

```
public abstract class HttpBase {

    private StandaloneMockMvcBuilder controllers(Object... controllers) {
        ...
        return MockMvcBuilders.standaloneSetup(controllers)
            .setMessageConverters(...);
    }

    @Before
    public void setup() {
        OrderService orderService = mock(OrderService.class);
①        OrderRepository orderRepository = mock(OrderRepository.class);
        OrderController orderController =
            new OrderController(orderService, orderRepository);

        when(orderRepository.findOne(1223232L))
            .thenReturn(OrderDetailsMother.CHICKEN_VINDALOO_ORDER);
②        ...
        RestAssuredMockMvc.standaloneSetup(controllers(orderController));
    }
}
```

- ① Create the OrderRepository injected with mocks
- ② Configure the OrderResponse to return an Order when findOne() is invoked with the orderId specified in the contract.

The argument 1223232L that is passed to the mock OrderRepository's findOne() method matches the orderId specified in the contract shown in listing 9.10. This test verifies that the Order Service has an GET /orders/{orderId} endpoint that matches let's its client's expectations. Let's take a look at the corresponding client test.

Consumer-side integration test for the API Gateway's OrderServiceProxy

The API Gateway's OrderServiceProxy invokes the GET /orders/{orderId} endpoint. Listing 9.11 shows the OrderServiceProxyIntegrationTest test class, which verifies that it conforms to the contracts. This class is annotated with @AutoConfigureStubRunner, which is provided by Spring Cloud Contract. It tells Spring Cloud Contract to run the WireMock server on a random port and configure it using the specified contracts. OrderServiceProxyIntegrationTest configures the OrderServiceProxy to make requests to the WireMock port.

Listing 9.11. A consumer-side integration test for the API Gateway's OrderServiceProxy. It uses Spring Cloud Contract to configure a WireMock-based stub HTTP server, which uses the contracts to simulate the Order Service.

```

@RunWith(SpringRunner.class)
@SpringBootTest(classes=TestConfiguration.class,
    webEnvironment= SpringBootTest.WebEnvironment.NONE)
@AutoConfigureStubRunner(ids =
    {"net.chrisrichardson.ftgo.contracts:ftgo-order-service-contracts"}, ①
    workOffline = false)
@DirtiesContext
public class OrderServiceProxyIntegrationTest {

    @Value("${stubrunner.runningstubs.ftgo-order-service-contracts.port}") ②
    private int port;
    private OrderDestinations orderDestinations;
    private OrderServiceProxy orderService;

    @Before
    public void setUp() throws Exception {
        orderDestinations = new OrderDestinations();
        String orderServiceUrl = "http://localhost:" + port;
        orderDestinations.setOrderServiceUrl(orderServiceUrl);
        orderService = new OrderServiceProxy(orderDestinations,
            WebClient.create()); ③
    }

    @Test
    public void shouldVerifyExistingCustomer() {
        OrderInfo result = orderService.findOrderById("1223232").block();
        assertEquals("1223232", result.getOrderId());
        assertEquals("CREATE_PENDING", result.getState());
    }

    @Test(expected = OrderNotFoundException.class)
    public void shouldFailToFindMissingOrder() {
        orderService.findOrderById("555").block();
    }
}

```

- ① Tell Spring Cloud Contract to configure WireMock with the Order Service's contracts
- ② Obtain the randomly assigned port that WireMock is running on
- ③ Create an OrderServiceProxy configured to make requests to WireMock

Each test method invokes the OrderServiceProxy and verifies that either it returns the correct values or throws the expected exception. The shouldVerifyExistingCustomer() test method verifies that findOrderById() returns values equal to those specified in the contract's response. The shouldFailToFindMissingOrder() attempts to retrieve a non-existent Order and verifies that the OrderServiceProxy throws an OrderNotFoundException exception. Testing both the REST client and the REST service using the same contracts ensures that they agree on the API. Let's now look at how to do the same kind of

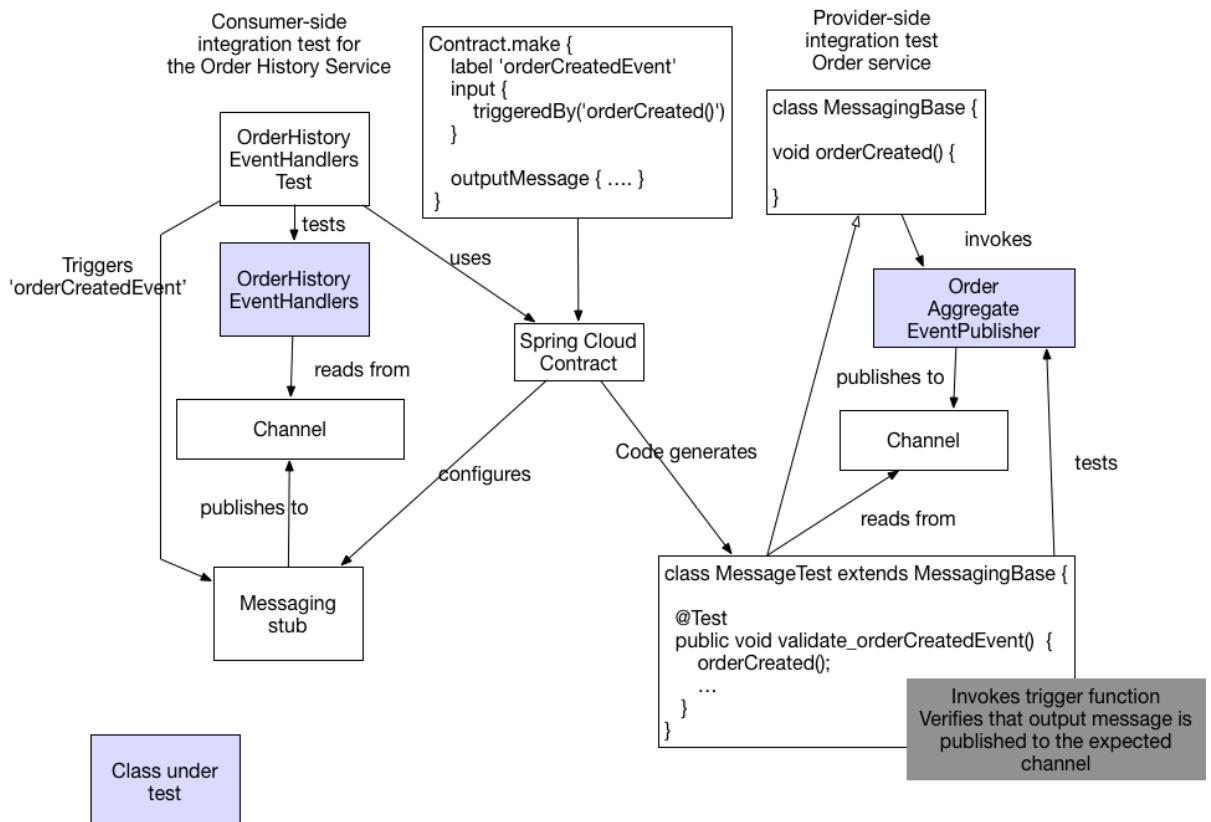
testing for services that interact using messaging.

9.3.3 Integration testing publish/subscribe-style interactions

Services often publish domain events, which are consumed by one or more other services. Integration testing must verify that the publisher and its consumers agree on the message channel, and the structure of the domain events. The Order Service, for example, publishes Order* events whenever it creates or updates an Order aggregate. The Order History Service is one of the consumers of those events. We must, therefore, write tests that verify that these services can interact.

Figure 9.15 shows the approach to integration testing publish/subscribe interactions. It is quite similar to the approach used for testing REST interactions. As before, the interactions are defined by a set of contracts. What's different is that each contract specifies a domain event.

Figure 9.15. The contracts are used to test both sides of the publish/subscribe interaction. The provider-side tests verify that the OrderDomainEventPublisher publishes events that confirm to the contract. The consumer-side tests verify that the OrderHistoryEventHandlers consume the example events from the contract.



Each consumer-side test publishes the event specified by the contract and verifies that `OrderHistoryEventHandlers` invokes its mocked dependencies correctly. On the provider-side, Spring Cloud Contract code generates test classes that extend `MessagingBase`, which is hand-written abstract superclass. Each test method invokes a superclass method, which is expected to trigger the publication of an event by the service.

In this example, each hook method invokes `OrderDomainEventPublisher`, which is responsible for publishing `Order` aggregate events. The test method then verifies that the `OrderDomainEventPublisher` published the expected event. Let's look at the details of how these tests work starting with the contract.

The contract for publishing an `OrderCreated` event

Listing 9.12 shows the contract for a `OrderCreated` event. It specifies the event's channel along with the expected body and message headers.

Listing 9.12. A contract for a publish/subscribe interaction style. The contract's output message is an example of an `OrderCreated` domain event.

```
package contracts;

org.springframework.cloud.contract.spec.Contract.make {
    label 'orderCreatedEvent'
    input {
        triggeredBy('orderCreated()')
    }

    outputMessage {
        sentTo('net.chrisrichardson.ftgo.orderservice.domain.Order')

body(''{"orderDetails": {"lineItems": [{"quantity": 5, "menuItemId": "1", "name": "Chicken Vindaloo", "price": "12.34", "total": "61.70"}], "orderTotal": "61.70", "restaurantId": 1, "consumerId": 1511300065921}, "orderState": "CREATE_PENDING"}''')
        headers {
            header('event-aggregate-type', 'net.chrisrichardson.ftgo.orderservice.domain.Order')
            header('event-aggregate-id', '1')
        }
    }
}
```

①

②

- ① Used by the consumer test to trigger the event to be published
- ② Invoked by the code generated provider test

The contract also has two other important elements:

- **label** - is used by a consumer test to trigger publication of the event by Spring Contact
- **triggeredBy** - the name of the superclass method invoked by the generated test method to trigger the publishing of the event

Let's look at how the contract is used starting with the provider-side test for the OrderService

Consumer-driven contract tests for the Order Service

The provider-side test for the Order Service is another consumer-driven contract integration test. It verifies that the OrderDomainEventPublisher, which is responsible for publishing Order aggregate domain events, publishes events that match its clients' expectations. Listing 9.13 shows the MessagingBase, which is the base class for the test class code generated by Spring Cloud Contract. It's responsible for configuring the OrderDomainEventPublisher class to use in-memory messaging stubs. It also defines the methods, such as orderCreated(), which are invoked by the generated tests to trigger the publishing of the event.

Listing 9.13. The abstract base class for the provider-side tests generated by Spring Cloud Contract from the messaging contracts. It defines methods, such as orderCreated(), which are invoked by the code generated tests to trigger the publication of events.

```

@RunWith(SpringRunner.class)
@SpringBootTest(classes = MessagingBase.TestConfiguration.class, webEnvironment =
SpringBootTest.WebEnvironment.NONE)
@AutoConfigureMessageVerifier
public abstract class MessagingBase {

    @Configuration
    @EnableAutoConfiguration
    @Import({EventuateContractVerifierConfiguration.class,
        TramEventsPublisherConfiguration.class,
        TramInMemoryConfiguration.class})
    public static class TestConfiguration {

        @Bean
        public OrderDomainEventPublisher
            OrderDomainEventPublisher(DomainEventPublisher eventPublisher) {
            return new OrderDomainEventPublisher(eventPublisher);
        }
    }

    @Autowired
    private OrderDomainEventPublisher OrderDomainEventPublisher;

    protected void orderCreated() {
        OrderDomainEventPublisher.publish(CHICKEN_VINDALOO_ORDER,
            singletonList(new OrderCreatedEvent(CHICKEN_VINDALOO_ORDER_DETAILS)));
    }
}

```

① orderCreated() is invoked by code generated test subclass to publish the event.

This test class configures the `OrderDomainEventPublisher` with in-memory messaging stubs. The `orderCreated()` is invoked by the test method generated from the contract shown earlier in listing 9.12. It invokes the `OrderDomainEventPublisher` to publish an `OrderCreated` event. The test method attempts to receive this event and then verifies that it matches the event specified in the contract. Let's now look at the corresponding consumer-side tests.

Consumer-side contract test for the Order History Service

The Order History Service consumes events published by the Order Service. As I described in chapter 7, the class that handles these events is the `OrderHistoryEventHandlers` class. Its event handlers invoke the `OrderHistoryDao` to update the CQRS view. Listing 9.14 shows the consumer-side integration test. It creates an `OrderHistoryEventHandlers` injected with a mock `OrderHistoryDao`. Each test method first invokes Spring Cloud Cloud to publish the event defined in the contract and then verifies that `OrderHistoryEventHandlers` invokes the `OrderHistoryDao` correctly.

Listing 9.14. The consumer-side integration test for the `OrderHistoryEventHandlers` class, which it injects with a mock `OrderHistoryDao`. Each test calls a Spring Cloud API to publish a message specified by a contract and verifies that `OrderHistoryEventHandlers` correctly invokes the `OrderHistoryDao`.

```

@RunWith(SpringRunner.class)
@SpringBootTest(classes= OrderHistoryEventHandlersTest.TestConfiguration.class,
    webEnvironment= SpringBootTest.WebEnvironment.NONE)
@AutoConfigureStubRunner(ids =
    {"net.chrisrichardson.ftgo.contracts:ftgo-order-service-contracts"}, 
    workOffline = false)
@DirtiesContext
public class OrderHistoryEventHandlersTest {

    @Configuration
    @EnableAutoConfiguration
    @Import({OrderHistoryServiceMessagingConfiguration.class,
        TramCommandProducerConfiguration.class,
        TramInMemoryConfiguration.class,
        EventuateContractVerifierConfiguration.class})
    public static class TestConfiguration {

        @Bean
        public OrderHistoryDao orderHistoryDao() {
            return mock(OrderHistoryDao.class);
        }
    }

    @Test
    public void shouldHandleOrderCreatedEvent() throws ... {
        stubFinder.trigger("orderCreatedEvent");
        eventually(() -> {
            verify(orderHistoryDao).addOrder(any(Order.class), any(Optional.class));
        });
    }
}

```

1

2

3

```
}
```

- ① Create a mock OrderHistoryDao to inject into OrderHistoryEventHandlers
- ② Trigger the orderCreatedEvent stub, which emits an OrderCreated event
- ③ Verify that OrderHistoryEventHandlers invoked orderHistoryDao.addOrder()

The `shouldHandleOrderCreatedEvent()` test method tells Spring Cloud Contract to publish the `OrderCreated` event. It then verifies that that `OrderHistoryEventHandlers` invoked `orderHistoryDao.addOrder()`. Testing both domain event's publisher and consume using the same contracts ensures that they agree on the API. Let's now look at how to do integration test services that interact using request/async reply.

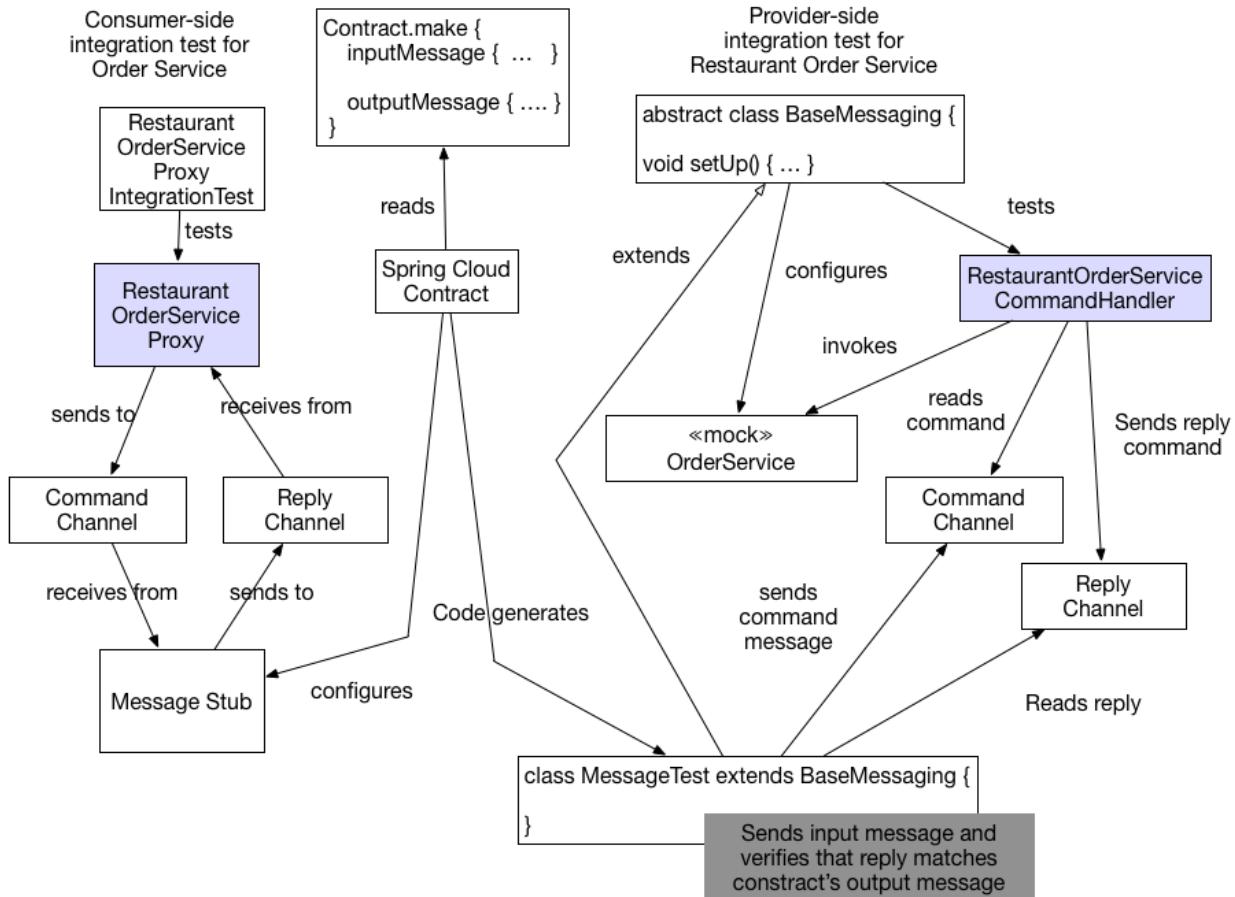
9.3.4 Integration contract tests for Request/async reply interactions

Publish/subscribe isn't the only kind of messaging-based interaction style. Services also interact using request/async reply. For example, in chapter 4 we saw that the `Order Service` implements sagas that send command messages to various services, such as the `Restaurant Order Service`, and process the reply messages.

The two parties in a request/async reply interaction are the requestor, which is the service that sends the command, and the replier, which is the service that processes the command and sends back a reply. The must must agree on the name of command message channel and the structure of the command and reply messages. Let's look at how to write integration tests for request/async reply interactions.

Figure 9.16 shows how to test the interaction between the `Order Service` and the `Restaurant Order Service`. The approach to integration testing request/async reply interactions is quite similar to the approach used for testing REST interactions. The interactions between the services are defined by a set of contracts. What's different is that a contract specifies an input message and an output message instead of an HTTP request and reply.

Figure 9.16. The contracts are used to test the adapter classes that implement each side of the request/async reply interaction. The provider-side tests verify that the RestaurantOrderServiceCommandHandler, which handles commands and sends back replies. The consumer-side tests verify the RestaurantOrderServiceProxy sends commands that conform to the contract, and that it handles the example replies from the contract.



The consumer-side test verifies that the command message proxy class sends correctly structured command message and correctly processes reply messages. In this example, the `RestaurantOrderServiceProxyTest` tests the `RestaurantOrderServiceProxy`. It uses Spring Cloud Contract to configure messaging stubs that verify that the command message matches a contract's input message and reply with the corresponding output message.

The provider-side tests are code generated by Spring Cloud Contract. Each test method corresponds to contract. It sends the contract's input message as a command message and verifies that the reply message matches the contract's output message. Let's look at the details starting with the contract.

Example request/async reply contract

Listing 9.15 shows the contract for one interaction. It consists of an input message and output message. Both messages specify a message channel, a message body and message headers. The naming convention is from the provider's perspective. The input message's `messageFrom` element specifies the channel that the message is read from. Similarly, the output message's `sentTo` element specifies the channel that the reply should be sent to.

Listing 9.15. A Spring Cloud Contract contract, which describes how the Order Service invokes the Restaurant Order Service using request/async reply

```
package contracts;

org.springframework.cloud.contract.spec.Contract.make {
    label 'createRestaurantOrder'
    input {
        messageFrom('restaurantOrderService')

        messageBody('''{ "orderId":1,"restaurantId":1,"restaurantOrderDetails":{} }''')
        messageHeaders {

            header('command_type','net.chrisrichardson.ftgo.restaurantorderservice.api.CreateRe
staurantOrder')

            header('command_saga_type','net.chrisrichardson.ftgo.orderservice.sagas.createorder
.CreateOrderSaga')
                header('command_saga_id',$(consumer(regex('[0-9a-f]{16}-[0-9a-
f]{16}'))))
                    header('command_reply_to',
'net.chrisrichardson.ftgo.orderservice.sagas.createorder.CreateOrderSaga-reply')
                }
            }
        outputMessage {

        sentTo('net.chrisrichardson.ftgo.orderservice.sagas.createorder.CreateOrderSaga-
reply')
            body([
                restaurantOrderId: 1
            ])
            headers {
                header('reply_type',
'net.chrisrichardson.ftgo.restaurantorderservice.api.CreateRestaurantOrderReply')
                    header('reply_outcome-type', 'SUCCESS')
            }
        }
    }
}
```

In this example contract, the input message is a `CreateRestaurantOrder` command that is sent to the `restaurantOrderService` channel. The output message is a successful reply that is sent to the `CreateOrderSaga`'s reply channel. Let's look at how to use this contract in tests starting with the consumer-side tests for the `Order Service`.

Consumer-side contract integration test for a request/async reply interaction

The strategy for writing a consumer-side integration test for a request/async reply interaction is similar to testing a REST client. The test invokes the service's messaging proxy and verifies two aspects of its behavior. First, it verifies that the messaging proxy sends a command message that conforms to the contract. Second, it verifies that the proxy properly handles the reply message.

Listing 9.16 shows the consumer-side integration test for the RestaurantOrderServiceProxy, which is the messaging proxy used by the Order Service to invoke the Restaurant Order Service. Each test sends a command message using RestaurantOrderServiceProxy and verifies that it returns the expected result. It uses Spring Cloud Contract to configure messaging stubs that find the contract whose input message matches the command message and sends its output message as the reply. The tests use in-memory messaging for simplicity and speed.

Listing 9.16. The consumer-side contract integration test for the Order Service. Each test method invokes the RestaurantOrderServiceProxy and verifies that it returns the expected result.

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes=
RestaurantOrderServiceProxyIntegrationTest.TestConfiguration.class,
webEnvironment= SpringBootTest.WebEnvironment.NONE)
@AutoConfigureStubRunner(ids =
{"net.chrisrichardson.ftgo.contracts:ftgo-restaurant-order-service-
contracts"}, 
workOffline = false)
@DirtiesContext
public class RestaurantOrderServiceProxyIntegrationTest {

    @Configuration
    @EnableAutoConfiguration
    @Import({TramCommandProducerConfiguration.class,
    TramInMemoryConfiguration.class,
EventuateContractVerifierConfiguration.class})
    public static class TestConfiguration { ... }

    @Autowired
    private SagaMessagingTestHelper sagaMessagingTestHelper;

    @Autowired
    private RestaurantOrderServiceProxy restaurantOrderServiceProxy;

    @Test
    public void shouldSuccessfullyCreateRestaurantOrder() {
        CreateRestaurantOrder command = new CreateRestaurantOrder(AJANTA_ID,
OrderDetailsMother.ORDER_ID,
        new RestaurantOrderDetails(Collections.singletonList(new
RestaurantOrderLineItem(CHICKEN_VINDALOO_MENU_ITEM_ID, CHICKEN_VINDALOO,
CHICKEN_VINDALOO_QUANTITY))));

        String sagaType = CreateOrderSaga.class.getName();
    }
}
```

```

        CreateRestaurantOrderReply reply =
sagaMessagingTestHelper.sendAndReceiveCommand(restaurantOrderServiceProxy.create,
command, CreateRestaurantOrderReply.class, sagaType);

        assertEquals(new CreateRestaurantOrderReply(OrderDetailsMother.ORDER_ID),
reply);

    }

}

```

The `shouldSuccessfullyCreateRestaurantOrder()` test method sends a `CreateRestaurantOrder` command message and verifies that the reply contains the expected data. It uses `SagaMessagingTestHelper`, which is a test helper class, that synchronously sends and receives messages. Let's now look at how to write provider-side integration tests.

Writing provider-side consumer-driven contract test for request/async reply interactions

A provider-side integration test must verify that the provider handles a command message by sending the correct reply. Spring Cloud Contract generates test classes that have a test method for each contract. Each test method sends the contract's input message and verifies that the reply matches the contract's output message. The provider-side integration tests for the Restaurant Order Service test the `RestaurantOrderServiceCommandHandler`. Listing 9.17 shows the `AbstractRestaurantOrderConsumerContractTest` class, which is the base class for the Spring Cloud Contract generated tests. It creates a `RestaurantOrderServiceCommandHandler` injected with a mock `RestaurantOrderService`.

Listing 9.17. The abstract superclass of the provider-side consumer-driven contract tests for the Restaurant Order Service, which are code generated by Spring Cloud Contract. It configures just enough message routing to invoke `RestaurantMessageHandlers`, which it injects with a mock `RestaurantOrderService`.

```

@RunWith(SpringRunner.class)
@SpringBootTest(classes =
AbstractRestaurantOrderConsumerContractTest.TestConfiguration.class, webEnvironment =
SpringBootTest.WebEnvironment.NONE)
@AutoConfigureMessageVerifier
public abstract class AbstractRestaurantOrderConsumerContractTest {

    @Configuration
    @Import(RestaurantMessageHandlersConfiguration.class)
    public static class TestConfiguration {
        ...
        @Bean
        public RestaurantOrderService restaurantOrderService() {
            return mock(RestaurantOrderService.class);
        }
    }
}

```

1

```

    }
}

@Autowired
private RestaurantOrderService restaurantOrderService;

@Before
public void setup() {
    reset(restaurantOrderService);
    when(restaurantOrderService
        .createRestaurantOrder(eq(1L), eq(1L),
            any(RestaurantOrderDetails.class)))
        .thenReturn(new RestaurantOrder(1L, 1L,
            new RestaurantOrderDetails(Collections.emptyList())));
}
}

```

- ➊ Overrides the definition of the `restaurantOrderService` @Bean with a mock
- ➋ Configures the mock to return the values that match a contract's output message

The `RestaurantOrderServiceCommandHandler` invokes the `RestaurantOrderService` with arguments that are derived from a contract's input message and creates a reply message that is derived from the return value. The test class's `setup()` method configures the mock `RestaurantOrderService` to return the values that match the contract's output message

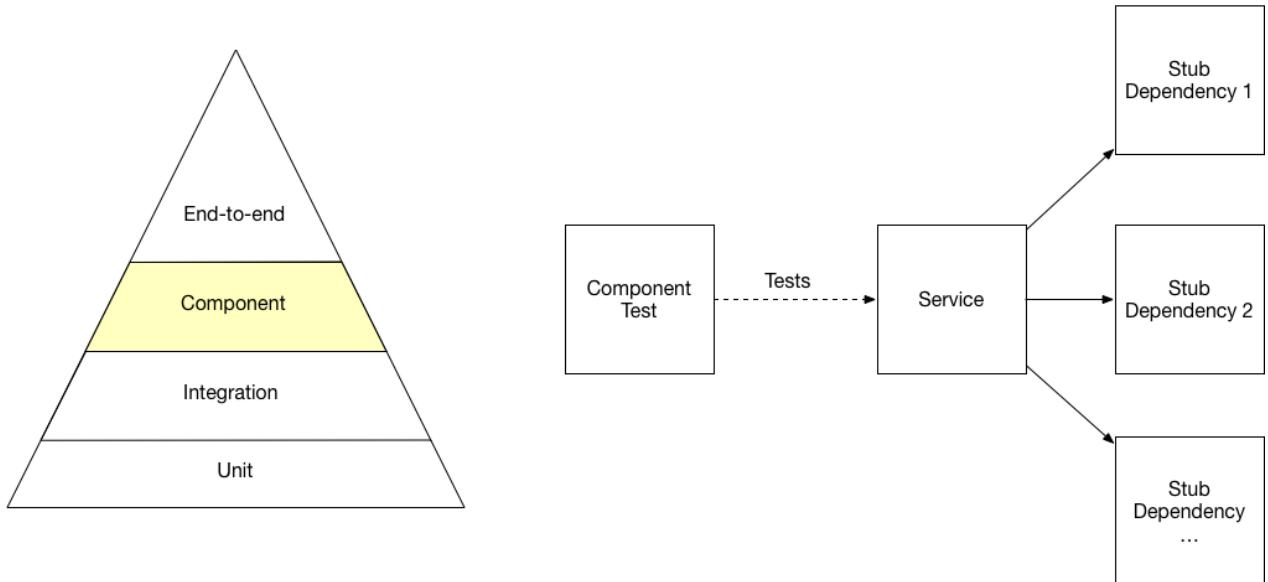
Integration tests and unit tests verify the behavior of individual parts of a service. The integration tests verify that services can communicate with its clients and dependencies. The unit tests verify that a service's logic is correct. Neither type of test actually run the entire service. In order to verify that a service as a whole works, let's move up the pyramid and look at how to write component tests.

9.4 Developing component tests

So far we have looked at how to test individual classes and clusters of classes. But let's imagine that we now want to verify that the Order Service works as expected. In other words, we want to write the service's acceptance tests, which treat it as a black box and verify its behavior through its API. One approach is to write what are essentially end-to-end tests and deploy the Order Service and all of its transitive dependencies. This is a slow, brittle, and expensive way to test a service.

A much better way to write acceptance tests for a service is to use what is known as component testing. As figure 9.17 shows, component tests are sandwiched between integration tests and end to end tests. Component testing verifies the behavior of a service in isolation. It replaces a service's dependencies with stubs that simulate their behavior. It might even use in-memory versions of infrastructure services, such as databases. As a result, component tests are much easier to write and faster to run.

Figure 9.17. A component test tests a service in isolation. It typically uses stubs for the service's dependencies.



I begin this section by briefly describing how to use a testing DSL called Gherkin to write acceptance tests for services, such as the Order Service. After that I describe various component testing design issues. I then show how to write acceptance tests for the Order Service. Let's look at writing acceptance tests using Gherkin.

9.4.1 Defining acceptance tests

Acceptance tests are business facing tests for a software component. In other words, they describe the desired externally visible behavior from the perspective of the component's clients rather than in terms of the internal implementation. They are tests that are derived from user stories or use cases. For example, one of the key stories for the Order Service is the *Place Order* story:

As a consumer of the Order Service

I should be able to place an order

We can expand this story into scenarios such as:

```
Given a valid consumer
Given using a valid credit card
Given the restaurant is accepting orders
When I place an order for Chicken Vindaloo at Ajanta
Then the order should be AUTHORIZED
And an OrderAuthorized event should be published
```

This scenario describes the desired behavior of the Order Service in terms its API.

Each scenario defines an acceptance test. The *Givens* correspond to the test's setup phase, the *when* maps to the execute phase and the *then* and the *ands* to the verification phase. Later on, you see a test for this scenario that

1. Creates an Order by invoking the POST /orders endpoint
2. Verifies the state of the Order by invoking the GET /orders/{orderId} endpoint
3. Verifies that the Order Service published an OrderAuthorized event by subscribing to the appropriate message channel

We could just translate each scenario into Java code. An easier option, however, is write the acceptance tests using a DSL such as Gherkin.

9.4.2 Writing acceptance tests using Gherkin

Writing acceptance tests in Java is challenging. There is a risk that that the scenarios and the Java tests diverge. There is also a disconnect between the high-level scenarios and the Java tests, which consist of low-level implementation details. Also, there is a risk that a scenario lack precisions or ambiguous and so cannot be translated into Java code. A much better approach is to eliminate the manual translation step and to write executable scenarios.

Gherkin is a DSL for writing executable specifications. When using Gherkin, you define your acceptance tests using english-like scenarios, such as the one shown earlier. You then execute the specifications using Cucumber, which is a test automation framework for Gherkin. Gherkin + Cucumber eliminate the need to manually translate scenarios into runnable code.

The Gherkin specification for a service, such as the Order Service, consists of a set of features. Each feature is described by a set of scenarios such as the one that you saw earlier. A scenario has the Given-When-Then structure. The *givens* are the preconditions, then *when* is the action or event that occurs, and the *then/ands* are the expected outcome.

For example, the desired behavior of the Order Service is defined by several features including Place order, Cancel order, and Revise order. Listing 9.18 is an excerpt of the Place Order feature. The feature consists of the several elements:

- Name - for this feature is Place Order
- Specification brief - describes why this feature exists. For this feature, the specification brief is the user story.
- Scenarios - Order Authorized and Order rejected due to expired credit card.

Listing 9.18. The Gherkin definition of the Place Order feature. and some of its scenarios

Feature: Place order

As a consumer of the Order Service

```
I should be able to place an order

Scenario: Order authorized
  Given A valid consumer
  Given using a valid credit card
  Given the restaurant is accepting orders
  When I place an order for Chicken Vindaloo at Ajanta
  Then the order should be AUTHORIZED
  And an OrderAuthorized event should be published

Scenario: Order rejected due to expired credit card
  Given A valid consumer
  Given using an expired credit card
  Given the restaurant is accepting orders
  When I place an order for Chicken Vindaloo at Ajanta
  Then the order should be REJECTED
  And an OrderRejected event should be published

...
```

In both scenarios, a consumer attempts to place an order. In the first scenario, they succeed. In the second scenario, the order is rejected because their credit card has expired. For more information about Gherkin, please see the book *Writing Great Specifications: Using Specification by Example and Gherkin* by Kamil Nicieja⁵⁴.

Executing Gherkin specifications using Cucumber

Cucumber is an automated testing framework that executes tests written in Gherkin. It's available in a variety of languages, including Java. When using Cucumber for Java, you write a step definition class, such as one shown in listing 9.19. A step definition class consists of methods that define the meaning of each Given-Then-When step. Each step definition method is annotated with either @Given, @When, @Then or @And. Each of these annotations has a value element that's a regular expression, which Cucumber matches against the steps.

Listing 9.19. The Java step definitions class, which makes the Gherkin scenarios executable. Each method has Cucumber annotation, which has a regular expression parameter that matches the text of a step. The method's body performs the action implied by that step.

```
public class StepDefinitions ... {
  ...
  @Given("A valid consumer")
  public void useConsumer() { ... }

  @Given("using a(.*?) (.*) credit card")
  public void useCreditCard(String ignore, String creditCard) { ... }
```

⁵⁴ www.manning.com/books/writing-great-specifications

```

@When("I place an order for Chicken Vindaloo at Ajanta")
public void placeOrder() { ... }

@Then("the order should be (.*)")
public void theOrderShouldBe(String desiredOrderState) { ... }

@And("an (.*) event should be published")
public void verifyEventPublished(String expectedEventClass) { ... }

}

```

Each type of method is part for a particular phase of the test:

- @Given - the setup phase
- @When - the execute phase
- @Then and @And - the verification phase

Later on in section [“The OrderServiceComponentTestStepDefinitions class”](#), when I describe this class in more detail, you will see that many of these methods make REST calls to the Order Service. For example, the `placeOrder()` method creates the Order by invoking the `POST /orders` REST endpoint. The `theOrderShouldBe()` method verifies the status of the order by invoking `GET /orders/{orderId}`. But before getting into the details of how to write step classes, let’s explore some design issues with component tests.

9.4.3 Designing component tests

Let’s imagine that you are implementing the component tests for the Order Service. In sections [“Writing acceptance tests using Gherkin”](#) and [“Executing Gherkin specifications using Cucumber”](#), I showed how to specify the desired behavior using Gherkin and execute them using Cucumber. But before a component test can execute the Gherkin scenarios it must first run the Order Service and set up the service’s dependencies. You need to test the Order Service in isolation so the component test must configure stubs for several services including the Restaurant Order Service. It also needs to set up a database and the messaging infrastructure. There are a few different options that tradeoff realism with speed and simplicity.

In-process component tests

One option is to write in-process component tests. An in-process component tests runs the service with in-memory stubs and mocks for its dependencies. For example, you can write a component test for a Spring Boot based-service using the Spring Boot testing framework. A test class, which is annotated with `@SpringBootTest`, runs the service in the same JVM as the test. It uses dependency injection to configure the service to use mocks and stubs. For instance, a test for the Order Service would configure it to use an in-memory JDBC database, such as H2, HSQLDB, or Derby, and in-memory stubs for Eventuate Tram. In-process tests are simpler to write and faster but have the downside of not testing the deployable service.

Out-of-process component testing

A more realistic approach is to package the service in a production-ready format and run it as a separate process. For example, in chapter 10 I describe that it's increasingly common to package services as Docker container images. An out-of-process component test uses real infrastructure services, such as databases and message brokers, but uses stubs for any dependencies that are application services. For example, an out-of-process component test for the FTGO Order Service would use MySQL and an Apache Kafka and stubs for services including Consumer Service and Accounting Service. Since the Order Service interacts with those services using messaging those stubs would consume messages from Apache Kafka and send back reply messages.

A key benefit of out-of-process component testing is that improves test coverage since what's being tested is much closer to what is being deployed. The drawback is that this type of test is more complex to write, slower to execute and potentially more brittle than an in-process component test. You also have to figure out how to stub the application services. Let's look at how to do that.

How to stub services in out-of-process component tests?

The service under test often invokes dependencies using interaction styles that involve sending back a response. The Order Service, for example, uses request/async response and sends command messages to various services. The API Gateway uses HTTP, which is a request/response interaction style. An out-of-process test must configures stub for these kinds of dependencies, which handle requests and send back replies.

One option is to use Spring Cloud Contract, which we looked at earlier in section "[Writing integration tests](#)" when discussing integration tests. We could write contracts that configures stubs for component tests. One thing to consider, however, is that it's likely that these contracts, unlike those used for integration, would only be used by the component tests.

Also, another drawback of using Spring Cloud Contract for component testing is that because its focus is consumer contract testing it takes a somewhat heavyweight approach. The JAR files containing the contracts must be deployed in a Maven Repository rather than just being on the classpath. It's also challenging to handle interactions involving dynamically generated values. Consequently, a simpler option is to configure stubs from within the test itself.

A test can, for example, configure an HTTP stub using the WireMock stubbing DSL. Similarly, a test for a service that uses Eventuate Tram messaging can configure messaging stubs. Later in section I show an easy to use Java library that does this. Now that we have looked at how to design component tests, let's look at how to write component tests for the FTGO Order Service.

9.4.4 Writing component tests for the FTGO Order Service

As you have just seen earlier in this section, there are few different ways to implement

component tests. In this section, I describe the component tests for the Order Service, which use the out-of-process strategy to test the service running as a Docker container. You will see how the tests use a Gradle plugin to start and stop the Docker container. I describe how to use Cucumber to execute the Gherkin-based scenarios, which define the desired behavior for the Order Service.

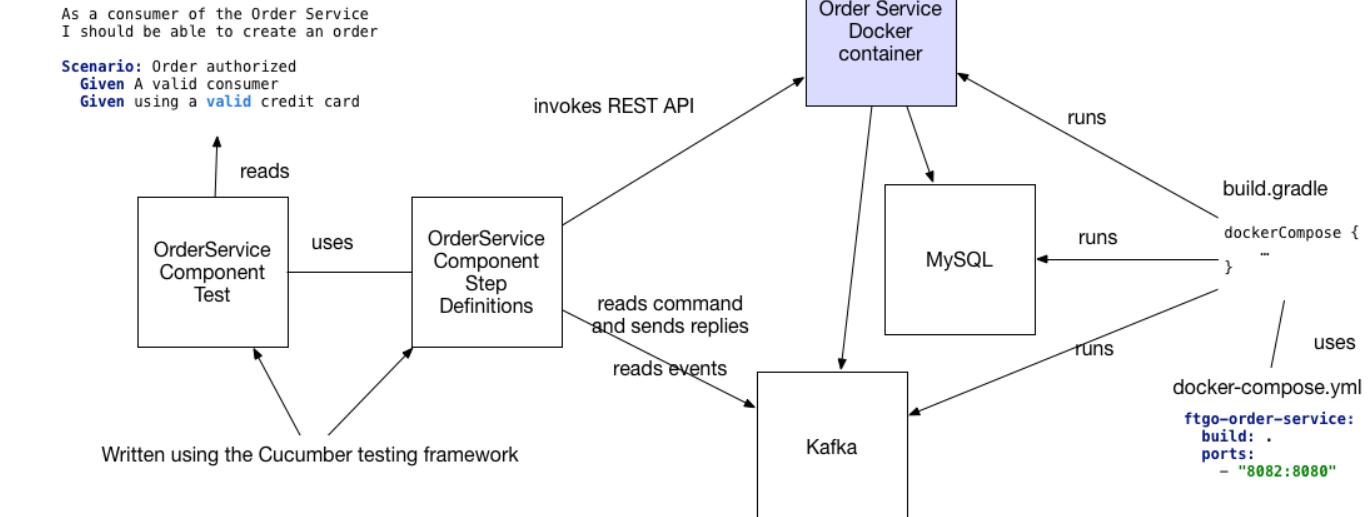
Figure 9.18 shows the design of the component tests for the Order Service. `OrderServiceComponentTest` is the test class that runs Cucumber:

```
@RunWith(Cucumber.class)
@CucumberOptions(features = "src/component-test/resources/features")
public class OrderServiceComponentTest {
}
```

It has an `@CucumberOptions` annotation, which specifies where to find the Gherkin feature files. It is also annotated with `@RunWith(Cucumber.class)`, which tells JUNIT to use the Cucumber test runner. But unlike a typical JUNIT-based test class it doesn't have any test methods. Instead, it defines the tests by reading the Gherkin features and uses the `OrderServiceComponentTestStepDefinitions` class to make them executable.

Figure 9.18. The component tests for the Order Service use the Cucumber testing framework to execute tests scenarios written using Gherkin acceptance testing DSL. The tests use Docker to run the Order Service along with its infrastructure services, such as Apache Kafka and MySQL.

`src/component-test/resources/createorder.feature`



Using Cucumber with the Spring Boot testing framework requires slightly unusual structure. Despite not being a test class, `OrderServiceComponentTestStepDefinitions` still is annotated with `@ContextConfiguration`, which is part of the Spring Testing

framework. It creates Spring ApplicationContext, which defines the various Spring components, including messaging stubs. Let's look at the details of the step definitions.

The OrderServiceComponentTestStepDefinitions class

The OrderServiceComponentTestStepDefinitions class is the heart of the tests. This class defines the meaning of each step in the Order Service's component tests. Listing 9.20 shows the `useCreditCard()` method, which defines the meaning of the Given using ... credit card step.

Listing 9.20. The @Given useCreditCard() method defines the meaning of the Given using ... credit card step and configures the Accounting Service stub to either send back a successful or a failure response

```
@ContextConfiguration(classes =
OrderServiceComponentTestStepDefinitions.TestConfiguration.class)
public class OrderServiceComponentTestStepDefinitions {

    ...

    @Autowired
    protected SagaParticipantStubManager sagaParticipantStubManager;

    @Given("using a(?) (.*) credit card")
    public void useCreditCard(String ignore, String creditCard) {
        if (creditCard.equals("valid"))
            sagaParticipantStubManager
                .forChannel("accountingService")
                .when(AuthorizeCommand.class).replyWithSuccess();
        else if (creditCard.equals("invalid"))
            sagaParticipantStubManager
                .forChannel("accountingService")
                .when(AuthorizeCommand.class).replyWithFailure();
        else
            fail("Don't know what to do with this credit card");
    }
}
```

This method uses the `SagaParticipantStubManager` class, which is a test helper class that configures stubs for saga participants. This `useCreditCard()` method uses it to configure the Accounting Servicestub to reply with either a success or a failure message depending on the specified credit card.

Listing 9.21 shows the `placeOrder()` method, which defines the When I place an order for Chicken Vindaloo at Ajanta step. It invokes the Order Service REST API to create the Order and saves the response for validation in a later step.

Listing 9.21. The placeOrder() method defines the When I place an order for Chicken Vindaloo at Ajanta step. It invokes the Order Service REST API to create the Order and saves the response for validation in a later step.

```
@ContextConfiguration(classes =
OrderServiceComponentTestStepDefinitions.TestConfiguration.class)
public class OrderServiceComponentTestStepDefinitions {

    private int port = 8082;
    private String host = System.getenv("DOCKER_HOST_IP");

    protected String baseUrl(String path) {
        return String.format("http://%s:%s%s", host, port, path);
    }

    private Response response;

    @When("I place an order for Chicken Vindaloo at Ajanta")
    public void placeOrder() {

        response = given().
            body(new CreateOrderRequest(consumerId,
                RestaurantMother.AJANTA_ID, Collections.singletonList(new
CreateOrderRequest.LineItem(RestaurantMother.CHICKEN_VINDALOO_MENU_ITEM_ID,
                OrderDetailsMother.CHICKEN_VINDALOO_QUANTITY))).
            contentType("application/json").
        when().
            post(baseUrl("/orders"));
    }
}
```

The baseUrl() help method returns the URL of the order service.

Listing 9.22 shows the theOrderShouldBe() method, which defines the meaning of the Then the order should be ... step. It verifies that Order was successfully created and that it's in the expected state.

Listing 9.22. The @Then theOrderShouldBe() method verifies that the HTTP request was successful and that the order is in the the expected state

```
@ContextConfiguration(classes =
OrderServiceComponentTestStepDefinitions.TestConfiguration.class)
public class OrderServiceComponentTestStepDefinitions {

    @Then("the order should be (.*)")
    public void theOrderShouldBe(String desiredOrderState) {

        Integer orderId =
            this.response. then(). statusCode(200).
                extract(). path("orderId");

        assertNotNull(orderId);

        eventually(() -> {

```

```

        String state = given().
            when().
            get(baseUrl("/orders/" + orderId)).
            then().
            statusCode(200)
            .extract().
            path("state");
        assertEquals(desiredOrderState, state);
    });

}

```

The assertion of the expected state is wrapped in a call to `eventually()`, which repeatedly executes the assertion.

Listing 9.23 shows the `verifyEventPublished()` method, which defines the `And an ... event should be published` step. It verifies that the expected domain event was published was published.

Listing 9.23. The Cucumber step definitions class for the Order Service component tests

```

@ContextConfiguration(classes =
OrderServiceComponentTestStepDefinitions.TestConfiguration.class)
public class OrderServiceComponentTestStepDefinitions {

    @Autowired
    protected MessageTracker messageTracker;

    @And("an (.*) event should be published")
    public void verifyEventPublished(String expectedEventClass) throws
ClassNotFoundException {

    messageTracker.assertDomainEventPublished("net.chrisrichardson.ftgo.orderservice.do
main.Order",

    (Class<DomainEvent>)Class.forName("net.chrisrichardson.ftgo.orderservice.domain." +
expectedEventClass));
    }
    ....
}

```

The `verifyEventPublished()` method uses the `MessageTracker` class, which is a test helper class that records the events that have been published during the test. This class along with `SagaParticipantStubManager` are instantiated by the `TestConfiguration @Configuration` class. Now that we have looked at the step definitions, let's look at how to run the component tests.

Running the component tests

Since these tests are relatively slow, we don't want to run them as part of `./gradlew test`. Instead, we will put the test code in a separate `src/component-test/java` directory and run them using `./gradlew componentTest`. Please take a look at the

`ftgo-order-service/build.gradle` file to see the Gradle configuration.

The tests use Docker to run the `Order Service` and its dependencies. As I describe in chapter 10, Docker containers are a lightweight, operating system virtualization mechanism that lets you deploy a service instance in an isolated sandbox. Docker Compose is an extremely useful tool that lets you define a set of containers and start and stop them as a unit. The FTGO application has a `docker-composefile` in the root directory that defines containers for all of the services and infrastructure service.

We can use the Gradle Docker Compose plugin to run the containers before executing the tests and stop the containers once the tests complete.

```
apply plugin: 'docker-compose'

dockerCompose.isRequiredBy(componentTest)
componentTest.dependsOn(assemble)

dockerCompose {
    startedServices = [ 'ftgo-order-service' ]
}
```

This snippet of Gradle configuration does two things. First, it configures the Gradle Docker Compose to run before the component tests and start the `Order Service` along with the infrastructure services that it is configured to depend on. Second, it configures the `componentTest` to depend on `assemble` so that the JAR file required by the Docker image is built first. With this in place, we can run these component tests with the following commands:

```
./gradlew :ftgo-order-service:componentTest
```

These commands, which take a couple of minutes, perform the following actions:

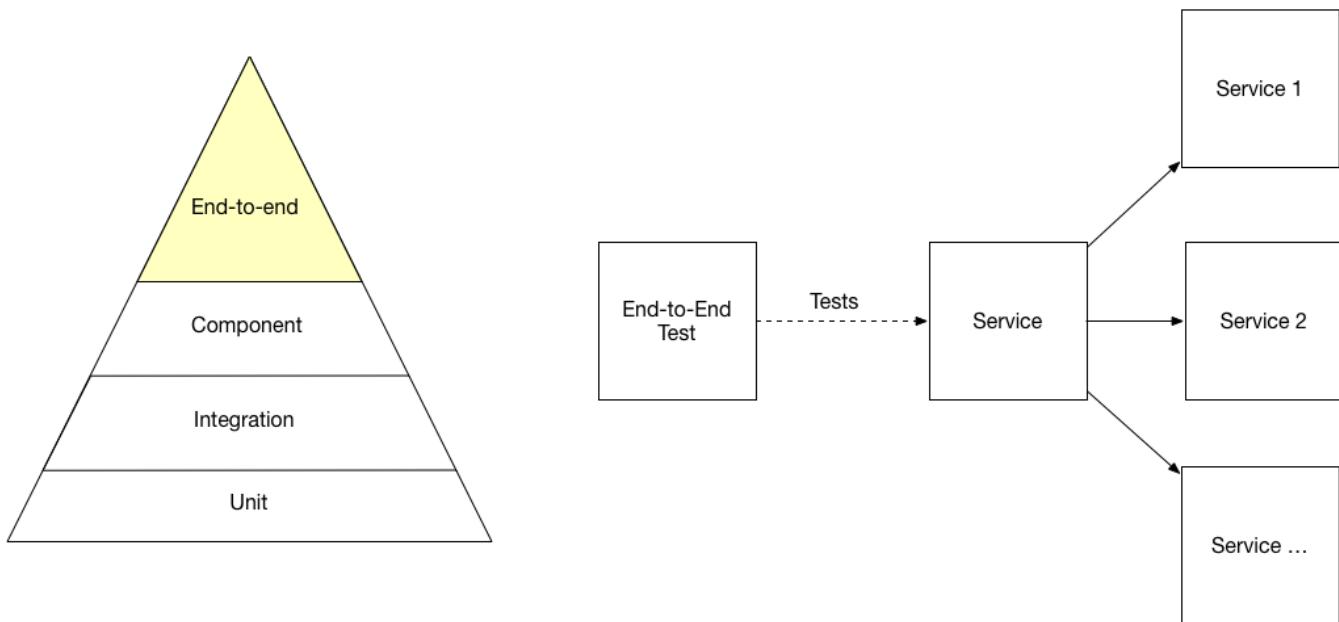
1. build the `Order Service`
2. run the service and its infrastructure services
3. run the tests
4. stop the running services

Now that we have looked at how to test a service in isolation, let's take a look at how to test the entire application.

9.5 Writing end-to-end tests

Component testing tests each service separately. End-to-end testing, however, tests the entire application. As figure 9.19 shows, end-to-end testing is the top of the test pyramid. That is because these kinds of tests are slow, brittle and time consuming to develop.

Figure 9.19. End-to-end tests are at the top of the test pyramid. They are slow, brittle and time consuming to develop. You should minimize the number of end-to-end tests.



End-to-end tests have a large number of moving parts. You must deploy multiple services and their supporting infrastructure services. As a result, end-to-end tests are slow. Also, if your test needs to deploy a large number of services there is a good chance that one of them will fail to deploy, which makes the tests unreliable. Consequently, you should minimize the number of end-to-end tests.

9.5.1 Designing end-to-end tests

Since these tests are slow and costly to develop and maintain its best to write as few as possible. A good strategy is to write user journey tests. A user journey test corresponds to a user's journey through the system. For example, rather than testing create order, revise order and cancel order separately, you can write a single test that does all three. This approach significantly reduces the number of tests that you must write and shortens the test execution time.

9.5.2 Writing end-to-end tests

End-to-end tests are, like the acceptance tests we looked at in section “[Developing component tests](#)”, business facing tests. It makes sense to write them in a high-level DSL that is understood by the business people. You can, for example, write the end-to-end tests using Gherkin and execute them using Cucumber. Listing 9.24 shows an example of such a test. Its similar to the acceptance tests we looked at earlier. The main difference is that rather a single Then, this test has multiple actions.

Listing 9.24. A Gherkin-based specification of a user journey. It describes how the consumer creates, revises and cancels an Order

```
Feature: Create revise and cancel
```

```
As a consumer of the Order Service
I should be able to create, revise and cancel an order
```

```
Scenario: Order created, revised and cancelled
Given A valid consumer
Given using a valid credit card
Given the restaurant is accepting orders
When I place an order for Chicken Vindaloo at Ajanta
Then the order should be AUTHORIZED
Then the order total should be 16.33
And when I revise the order by adding 2 vegetable samosas
Then the order total should be 20.97
And when I cancel the order
Then the order should be CANCELLED
```

This scenario places an order, revises it and then cancels it. Let's look at how to run it.

9.5.3 Running the end-to-end tests

End-to-end tests must run the entire application including any required infrastructure services. As you saw in earlier in section [“Developing component tests”](#), the Gradle Docker Compose plugin provides a convenient way to do this. Instead of running a single application service, however, the Docker Compose file runs all of the application's services. Now that we have looked at different aspects of designing and writing end-to-end tests, let's look at an example end-to-end test.

The `ftgo-end-to-end-test` module implements the end-to-end tests for the FTGO application. The implementation of the end-to-end test is quite similar to the implementation of the component tests we saw earlier in section [“Developing component tests”](#). These tests are written using Gherkin and executed using Cucumber. The Gradle Docker Compose plugin runs the containers before the tests run. It takes about XYZ minutes.

This might not seem like a long time. However, this a relatively simple application with just a handful of containers and tests. Imagine if there were hundreds of containers and many more tests. The tests could take a really long. Consequently, it's best to focus on writing tests that are lower down the pyramid.

9.6 Summary

- Automated testing is the key foundation of rapid, yet safe delivery of software. What's more, because of its inherent complexity, to fully benefit from the microservice architecture you must automate your tests.
- The purpose of a test is to verify the behavior of the system under test (SUT). In this definition, *system* is a fancy term that means the thing being tested. It might be something as small as a class, as large as the entire application or something in

between, such as a cluster of classes or an individual service. A collection of related tests form a test suite.

- A good way to simplify and speed up a test is to use test doubles. A test double is an object that simulates the behavior of a SUT's dependency. There are two types of test doubles, stubs and mocks. A stub is a test double that returns values to the SUT. A mock is a test double that a test uses to verify that the SUT correctly invokes a dependency.
- Use the test pyramid to determine where to focus your testing efforts for your services. The majority of your tests should be fast, reliable and easy to write unit tests. You must minimize the number of end-to-end tests since they are slow, brittle and time consuming to write.
- Use contracts, which are example messages, to drive the testing of interactions between services. Rather than write slow running tests that run both services and their transitive dependencies, simply write tests that verify the adapters of both services conform to the contracts.
- Write component tests to verify the behavior of a service via its API. You should simplify and speed up component tests by testing a service in isolation using stubs for its dependencies.
- Write user journey tests to minimize the number of end-to-end tests, which are slow, brittle and time consuming to write. A user journey test simulates a user's journey through the application and verifies high-level behavior of a relatively large slice of the application's functionality. Since there are few tests the amount of per-test overhead, such as test set up, is minimized, which speeds up the tests.

10

Developing production ready services

This chapter covers:

- Writing secure services
- Developing configurable services
- Designing observable services
- Developing services using a microservice chassis framework

So far, the focus on this book has been on service decomposition, inter-service communication, transaction management, querying and business logic design, and testing. These are all important topics since there is no point in developing an application that doesn't meet its functional requirements. However, in order for a service to be ready to be deployed into production we must ensure that it also satisfies three critically important quality attributes: security, configurability and observability.

The first quality attribute is application security. It's essential to develop secure applications unless you want your company to be in the headlines for a data breach. Fortunately, most aspects of security in a microservice architecture are not any different than in a monolithic application. However, the microservice architecture forces you to implement some aspects of application-level security differently. For example, you need to implement a mechanism to pass the identity of the user from one service to another.

The second quality attribute that you must address is service configurability. A service typically uses one or more external services, such as message brokers and databases. The network location and credentials of each external service often depends on the environment that the service is running. You can't hardwire the configuration

properties into the service. Instead, you must use an externalized configuration mechanism that provides a service with configuration properties at runtime.

The third quality attribute is observability. A microservice architecture is a distributed system. Every request is handled by the API gateway and at least one service. Let's imagine, for example, that you are trying to determine which of six services is causing a latency issue. Or, imagine trying to understand how a request is handled when the log entries are scattered across five different services. In order to make it easier to understand the behavior of your application and troubleshoot problems, you must implement several observability patterns.

I begin this chapter by describing how to implement security in a microservice architecture. Next, I describe how to design services that are configurable. I cover a couple of different service configuration mechanisms. After that I describe how to make your services easier to understand and troubleshoot by using the observability patterns. I end this chapter by showing how to simplify the implementation of these and other concerns by developing your services on top of a microservice chassis framework. Let's first look at security.

10.1 Developing secure services

Almost every day there are headlines about how hackers have stolen a company's data. Cybersecurity has become a critical issue for every organization. In order to develop secure software and avoid making the headlines, an organization needs to tackle diverse range of security issues including physical security of the hardware, encryption of data in transit and at rest, authentication and authorization, and policies for patching software vulnerabilities. Almost all of these topics are outside the scope of this book. What's more they are the same regardless of whether you are using a monolithic or microservice architecture. In this section, I am going to focus on one topic: how the microservice architecture impacts security at the application level.

An application developer is primarily responsible for implementing three different aspects of security:

- Authentication - verifying the identity of the application or human (a.k.a. the principal) that's attempting to access the application. For example, an application typically verifies a principal's credentials, such as a user id and password or an application's API key and secret.
- Authorization - verifying that the principal is allowed to perform the requested operation on the specified data. Applications often use a combination of role-based security and access control lists (ACLs). Role-based security assigns each user one or more roles that grant them permission to invoke particular operations. ACLs that grant users or roles permission to perform an operation on a particular business object, a.k.a. aggregate.
- Auditing - tracking the operations that a principal performs in order to detect security issues and help customer support and enforce compliance.

I'll describe auditing in more detail in section "[Designing observable services](#)". It this

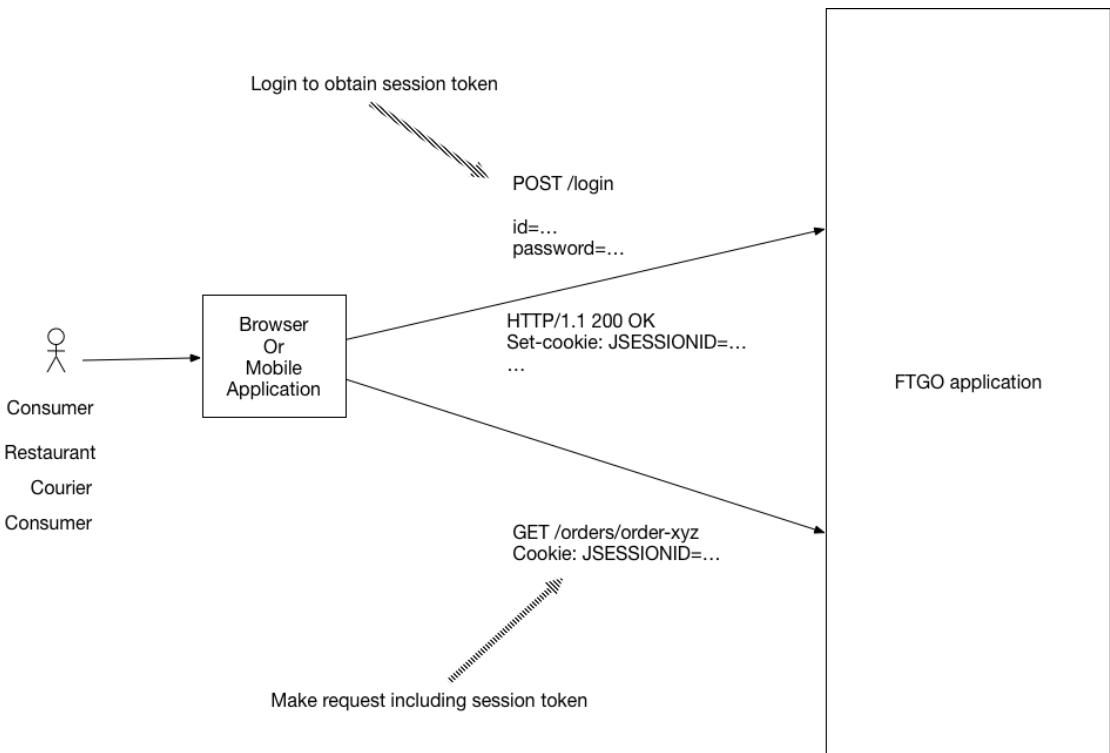
section I describe how to implement authentication and authorization. Correctly implementing authentication and authorization is challenging. It's best to use a proven framework. Which framework to use depends upon your application's technology stack. For example, Spring Security is a popular framework for Java applications. It's a sophisticated framework that handles authentication and authorization. Passport is a popular framework for NodeJS applications that is focussed on authentication.

I begin this section by first describing how security is implemented in the FTGO monolith application. I then describe the challenges with implementing security in a microservice architecture and how techniques that work well in a monolithic architecture can't be used in a microservice architecture. After that I describe how to implement security in a microservice architecture. Let's start by reviewing how the monolithic FTGO application handles security.

10.1.1 Overview of security in a traditional monolithic application

The FTGO application has several kinds of human users including consumers, couriers, and restaurant staff. They access the application using browser-based web applications and mobile applications. All FTGO users must login in order to access the application. Figure 10.1 shows how the clients of the monolithic FTGO application authenticate and make requests.

Figure 10.1. A client of the FTGO application first logs in to obtain a session token. It includes the session token in each subsequent request it makes to the client.



When a user logs in with their user id and password, the client makes a POST request containing the users credentials to the FTGO application. The FTGO application verifies the credentials and returns a session token to the client. The client includes the session token in each subsequent request to the FTGO application.

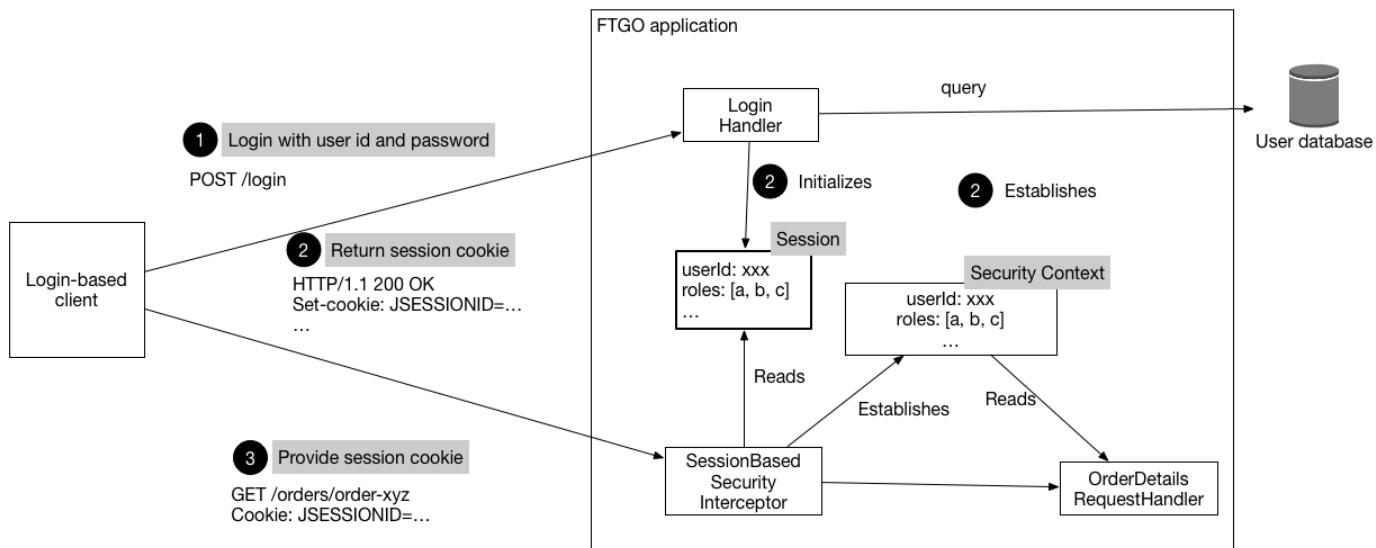
Figure 10.2 shows a high-level view of how the FTGO application implements security. The FTGO application is written in Java and so uses the Spring Security framework. However, I'll describe the design using generic terms that are applicable to other frameworks, such as Passport for NodeJS. One key part of the security architecture is the session, which stores the principal's id and roles. The FTGO application is a traditional Java EE application and so the session is an `HttpSession` in-memory session. A session is identified by a session token, which the client includes in each request. It is usually an opaque token such as a cryptographically strong random number. The FTGO application's session token is an HTTP cookie called `JSESSIONID`.

The other key part of the security implementation is the security context, which stores information about the user making the current request. The Spring Security framework uses the standard Java EE approach of storing the security context in a static thread-local variable, which is readily accessible to any code that's invoked to handle the request.

A request handler can call

`SecurityContextHolder.getContext().getAuthentication()` to obtain information about the current user, such as their identity and roles. In contrast the Passport framework stores the security context as the `user` attribute of the request.

Figure 10.2. A client of the FTGO application first logs in to obtain a session token. It includes the session token in each request it makes to the client.



The client makes two types of requests. The first is a login request, which a client uses to obtain a session token. A login request is handled by the `LoginHandler`. The `LoginHandler` verifies the credentials, initializes the session, stores information about the principal in the session, and returns a session token to client.

The second type of request is one that invokes an operation. This kind of request is first processed by the `SessionBasedSecurityInterceptor` interceptor. The interceptor authenticates the request by verifying the session token and establishes a security context. The security context describes the principal and its roles. It is used by the request handlers to determine whether to allow a user to perform the requested operation.

The FTGO application uses role-based authorization. It defines several roles corresponding to the different kinds of users including CONSUMER, RESTAURANT, COURIER and ADMIN. It uses Spring Security's declarative security mechanism to restrict access to URLs and service methods to specific roles. Roles are also interwoven into the business logic. For example, a consumer can only access their orders where as an administrator can access all orders.

The security design used by the monolithic FTGO application is only one possible way to implement security. For example, one drawback of using an in-memory session is that it requires all requests for a particular session to be routed to the same application

instance. This requirement complicates load balancing and operations. You must, for example, implement a session draining mechanism that waits for all sessions to expire before shutting down an application instance. An alternative approach, which avoids these problems, is to store the session in a database.

You can sometimes eliminate the server-side session entirely. For example, many applications have API clients that provide their credentials, such as an API key and secret, in every request. As a result, there is no need to maintain a server-side session. Alternatively, the application can store session state in the session token. Later in this section, I'll describe one way to accomplish that. But first let's begin with looking at the challenges of implementing security in a microservice architecture.

10.1.2 *Implementing security in a microservice architecture*

A microservice architecture is a distributed architecture. Each external request is handled by the API gateway and at least one service. Consider, for example, the `getOrderDetails()` query, which I described in chapter 8. The API gateway handles this query by invoking several services including the Order Service, Restaurant Order Service and the Accounting Service. Each service must implement some aspects of security. For instance, the Order Service must only allow a consumer to see their orders, which requires a combination of authentication and authorization. In order to implement security in a microservices architecture we need to determine who is responsible for authenticating the user and who is responsible for authorization.

One challenge with implementing security in a microservices application is that we can't just copy the design from a monolithic application. That's because there are two aspects of the monolithic application's security architecture that are non-starters for a microservice architecture. The first is using an in-memory security context to pass around user identity. Services can't share memory and so in a microservices architecture, we need a different mechanism for passing user identity from one service to another. The second problematic aspect of a monolithic architecture is the session. Just as an in-memory security context doesn't make sense, neither does an in-memory session. In theory, multiple services could access an database-based session except that it would violate the principle of loose coupling. We need a different session mechanism in a microservice architecture. Let's begin our exploration of security in a microservice architecture by looking at how to handle authentication.

API Gateway handle authentication

There are a couple of different ways to handle authentication. One option is for the individual services to authenticate the user. The problem with this approach is that it permits unauthenticated requests to enter the internal network. It relies on every development team correctly implementing security in all of their services. As a result, there is a significant risk of an application containing security vulnerabilities.

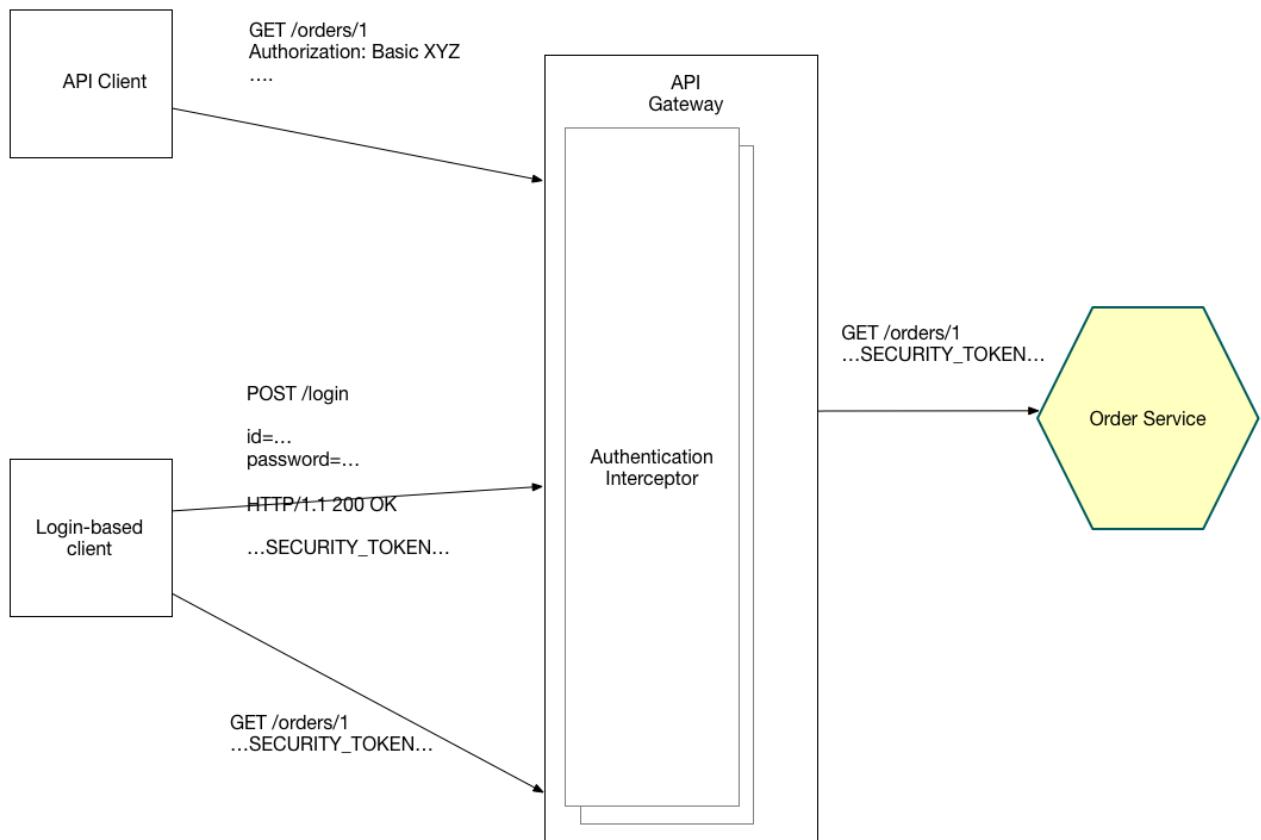
Another problem with implementing authentication in the services is that different clients authenticate in different ways. Pure API clients supply credentials with each request using, for example, Basic authentication. Other clients might first login and

then supply a session token with each request. We want to avoid requiring services to handle a diverse set of authentication mechanism.

A better approach is for the API gateway to authenticate a request before forwarding it to the services. Centralizing API authentication in the API gateway has the advantage that there is only one place to get right. As a result, there is a lot less chance of a security vulnerability. Another benefit is that only the API gateway has to deal with the various different authentication mechanisms. It hides this complexity from the services.

Figure 10.3 shows this approach works. Clients authenticate with the API gateway. API clients include credentials in each request. Login-based clients POST the user's credentials to the API gateway's authentication and receive a session token. Once the API Gateway has authenticated a request, it invokes one or more services.

Figure 10.3. The API Gateway authenticates requests from clients and includes a security token in requests to services. The services use the token to obtain information about the principal. The API Gateway might also use the security token as a session token.



A service invoked by the API Gateway needs to know the principal making the request.

It must also verify that the request has actually been authenticated. The solution is for the API gateway to include a token in each service request. The service uses the token to validate the request and obtain information about the principal. The API gateway might also give the same token to session-oriented clients to use as the session token. A little later, I describe how to implement tokens but let's first look at the other main aspect of security: authorization.

Handling authorization

Simply authenticating a client's credentials is insufficient. An application must also implement an authorization mechanism. For example, in the FTGO application, the `getOrderDetails()` query can only be invoked by the consumer who placed the Order (an example of an instance-based security) and a customer service agent who is helping the consumer.

One place to implement authorization is the API gateway. It can, for example, restrict access to the `GET /orders/{orderId}` to only users who are consumers and customer service agents. If a user is not allowed to access a particular path the API Gateway can reject the request before forwarding it on to the service. As with authentication, centralizing authorization within the API gateway, reduces the risk of security vulnerabilities. You can implement authorization in the API gateway using a security framework, such as Spring Security.

One drawback, however, of implementing authorization in the API gateway is that it risks coupling the API gateway to the services requiring them to be updated in lock step. What's more, the API gateway can typically only implement role-based access to URL paths. It's generally not practical for it to implement ACLs that control access to individual domain objects since that requires detailed knowledge of a service's domain logic.

The other place to implement authorization is in the services. A service can implement role-based authorization for URLs and for service methods. It can also implement ACLs for managing access to aggregates. The Order Service can, for example, implement the role-based and ACL-based authorization mechanism for controlling access to orders. Other services implement the FTGO application implement similar authorization logic.

Using JWTs to pass user identity and roles

When implementing security in a microservice architecture, you need to decide which type of token an API Gateway should use to pass user information to the services. There are two types of tokens to choose from. One option is to use opaque tokens, which are typically UUIDs. The downside of opaque tokens is they reduce performance and increase latency. That's because the recipient of such a token must make an RPC call to a security service to validate the token and retrieve the user information.

An alternative approach, which eliminates the call to the security service, is to use a transparent token containing information about the user. One such popular standard for transparent tokens is JSON Web Tokens (JWT). JWT is a standard way to securely

represent claims, such as user identity and roles, between two parties. A JWT has a payload, which is a JSON object that contains information about the user, such as their identity and roles, and other metadata, such as an expiration date. It is signed with secret that is only known to the creator of the JWT, such as the API gateway and the recipient of the JWT, such a service. The secret ensures that a malicious third party cannot forge a JWT or tamper with a JWT.

One issue with JWT is that since a token is self-contained, it is irrevocable. By design, a service will perform the request operation after verifying the JWT's signature and expiration date. As a result, there is no practical way to revoke an individual JWT that has fallen into the hands of a malicious third party. The solution is to issue JWTs with a short expiration time since that limits what a malicious party could do. However, a drawback of short-lived JWTs is that the application must somehow continually reissue JWTs to keep the session active. Fortunately, this is one of the many protocols that are solved by a security standard calling OAuth 2.0. Let's look at how that works.

Using OAuth 2.0 in a microservice architecture

Let's imagine that you want to implement a `User Service` for the FTGO application that manages a user database containing user information, such as credentials and roles. The API Gateway calls the `User Service` to authenticate a client request and obtain a JWT. You could design a `User Service API` and implement it using your favorite web framework. However, this is generic functionality that isn't specific to the FTGO application. Developing such a service would not be an efficient use of development resources.

Fortunately, you don't need to develop this kind of security infrastructure. Instead, you can use an off-the-shelf service or framework that implements a standard called OAuth 2.0. OAuth 2.0 is an authorization protocol that was originally designed to enable a user of a public cloud service, such as Github or Google, to grant a third party application access to their information without revealing their password. For example, it's the mechanism that enables you to securely grant a third party cloud-based Continuous Integration service access to your github repository.

While the original focus of OAuth 2.0 is authorizing access to public cloud services, you can also use it for authentication and authorization in your application. The details of OAuth 2.0 are outside of the scope of this book and for more information you should see this online book⁵⁵. However, let's take a quick look at how a microservice architecture might use OAuth 2.0.

The key concepts in OAuth 2.0 are the following:

- **Authorization Server** - provides an API for authenticating users and obtaining an access token and a refresh token. Spring OAuth is a great example of framework for building an OAuth 2.0 authorization server.
- **Access Token** - a token that grants access to a `Resource Server`. The format of

⁵⁵ www.oauth.com/

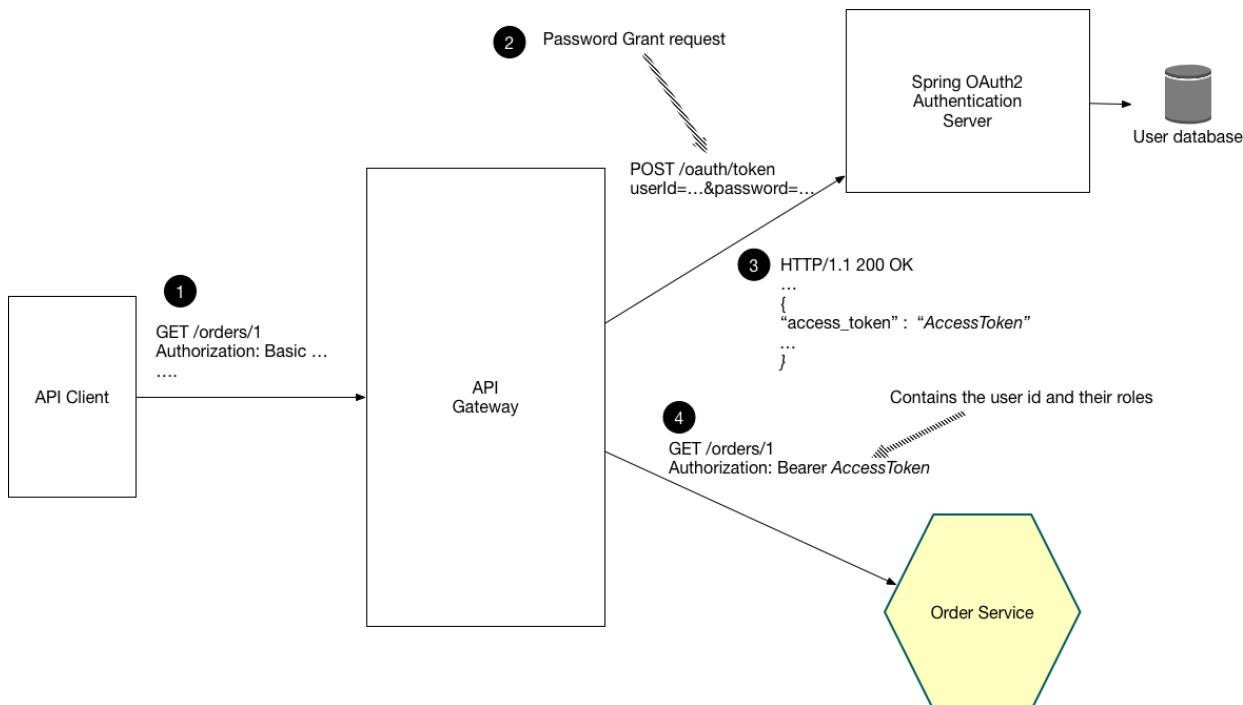
the access token is implementation dependent. However, some implementations, such as Spring OAuth, use JWTs.

- Refresh Token - a long-lived, yet revocable token that a Client uses to obtain a new AccessToken.
- Resource Server - a service that uses an access token to authorize access. In a microservice architecture, the services are resource servers.
- Client - a client that wants to access a Resource Server. In a microservice architecture, the API Gateway is the OAuth 2.0 client.

Later, I describe how to support login-based clients. But first, I describe how to authenticate API clients.

Figure 10.4 shows how the API Gateway authenticates a request from an API client. The API Gateway authenticates the API client by making a request to the OAuth 2.0 Authorization server, which returns an access token. The API Gateway then makes one or requests containing the access token to the services.

Figure 10.4. An API Gateway authenticates an API client by making a Password Grant request to the OAuth 2.0 Authentication Server. The server returns an access token, which the API gateway passes to the services. A service verifies the token's signature and extracts information the user including their identity and roles.

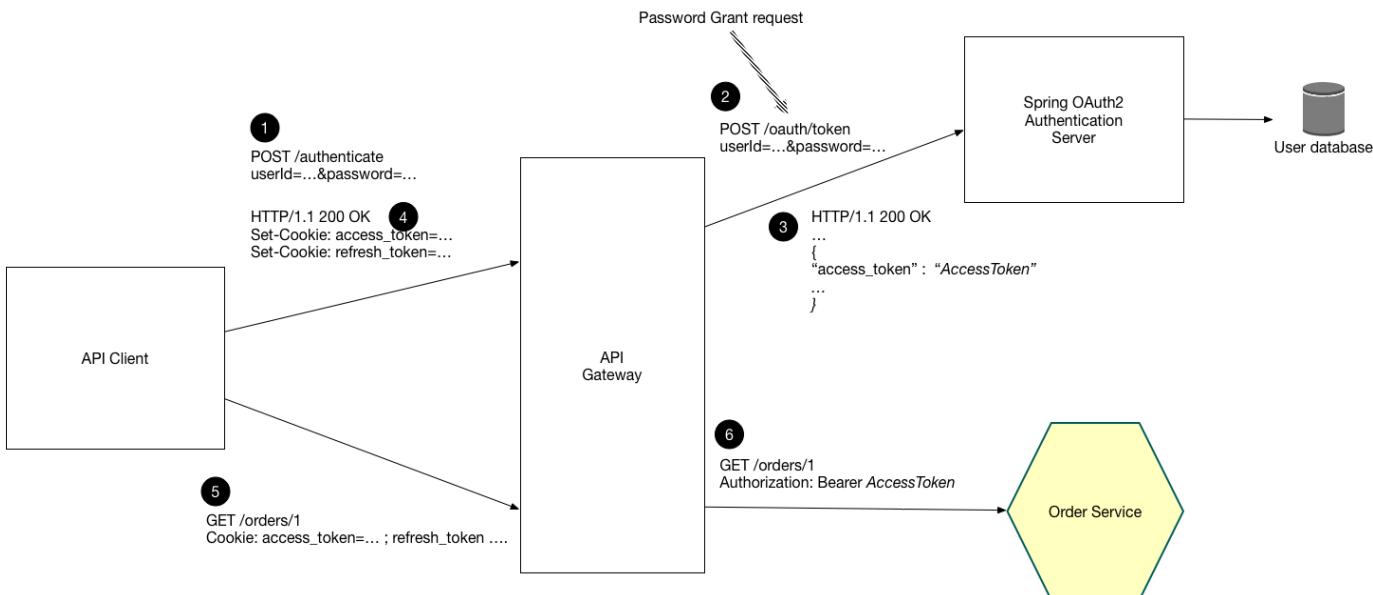


The sequence of events is as follows:

1. Client makes a request supplying its credentials using basic authentication
2. The API Gateway makes an OAuth 2.0 Password Grant request⁵⁶ request to the OAuth 2.0 Authentication Server
3. The Authentication Server validates the API clients credentials and returns an access token and a refresh token
4. The API gateway includes the access token in the requests that it makes to the services
5. A service validates the access token and uses it to authorize the request

An OAuth 2.0-based API gateway can authenticate session-oriented clients by using an OAuth 2.0 access token as a session token. What's more when the access token expires it can obtain a new access token using the refresh token. Figure 10.5 shows how an API gateway can use OAuth 2.0 to handle session-oriented clients. An API client initiates a session by posting its credentials to the API gateway's authentication endpoint. The API gateway returns an access token and a refresh token to the client. The API client then supplies both tokens when it makes requests to the API gateway.

Figure 10.5. A client logs in by POST its credentials to the API Gateway. The API Gateway authenticates the credentials using the OAuth 2.0 Authentication server and returns the access token and refresh token as cookies. A client includes the tokens in the requests that it makes to the API gateway.



The sequence of events is as follows:

1. The login-based client POSTs their credentials to the API gateway

⁵⁶ www.oauth.com/oauth2-servers/access-tokens/password-grant/

2. The API Gateway makes an OAuth 2.0 Password Grant request⁵⁷ request to the OAuth 2.0 Authentication Server
3. The Authentication Server validates the client's credentials and returns an access token and a refresh token
4. The API Gateway returns the access and refresh tokens to the client, e.g. as cookies.
5. The client includes the access and refresh tokens in requests that it makes to the API gateway
6. The API gateway validates the access token and includes in requests that it makes to the services

If the access token has expired or is about to expire, the API Gateway obtains a new access token by making an OAuth 2.0 Refresh Grant request⁵⁸, which contains the refresh token, to the Authorization server. If the refresh token has not expired or has been revoked then the Authorization Server returns a new access token. The API Gateway passes the new access token to the services and returns it to the client.

An important benefit of using OAuth 2.0 is that it's a proven security standard. Using an off-the-shelf OAuth 2.0 Authentication Server means that you don't have to waste time reinventing the wheel or risk developing an insecure design. However, OAuth 2.0 is not the only way to implement security in a microservice architecture. Regardless of which approach you use, the three key ideas are:

1. The API Gateway is responsible for authenticating clients
2. The API gateway and the services use a transparent token, such as a JWT, to pass around information about the principal
3. A service uses the token to obtain the principal's identity and roles

Now that we have looked at how to make services secure, let's look at how to make them configurable.

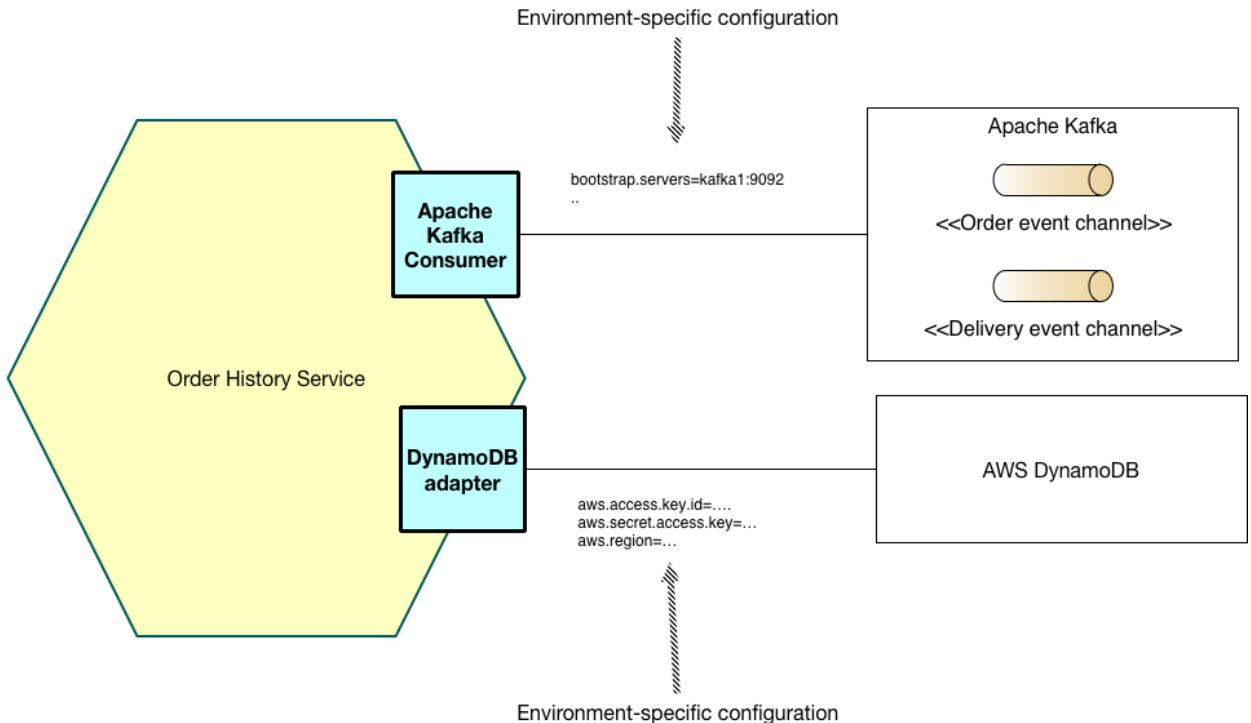
10.2 Designing configurable services

Let's imagine that you are responsible for the Order History Service service. As figure 10.6 shows, it consumes events from Apache Kafka and reads and writes AWS DynamoDB table items. In order for this service to run, it needs various configuration properties including the network location of Apache Kafka and the credentials and network location for AWS DynamoDB.

⁵⁷ www.oauth.com/oauth2-servers/access-tokens/password-grant/

⁵⁸ www.oauth.com/oauth2-servers/access-tokens/refreshing-access-tokens/

Figure 10.6. The Order History Service uses Apache Kafka and AWS DynamoDB. It needs to be configuration with each service's network location, credentials, etc.



The values of these configuration properties depend on which environment it is running it. For example, the developer and production environments will use different Apache Kafka brokers and different AWS credentials. It doesn't make sense to hardwire a particular environment's configuration property values into the deployable service since that would require it to be rebuilt for each environment. Instead, a service should be built once by the deployment pipeline and deployed into multiple environments.

Nor does it make sense to hardwire different sets of configuration properties into the source code and use, for example, the Spring frameworks profile mechanism to select the appropriate set at runtime. That's because doing so would introduce a security vulnerability and limit where it can be deployed. Instead, you should supply the appropriate configuration properties to the service at runtime using an *externalized configuration* mechanism.

A externalized configuration mechanism provides the configuration property values to a service instance at runtime. There are two main approaches:

- Push model - the runtime infrastructure passes the configuration properties to the service instance using, for example, operating system environment variables or a configuration file

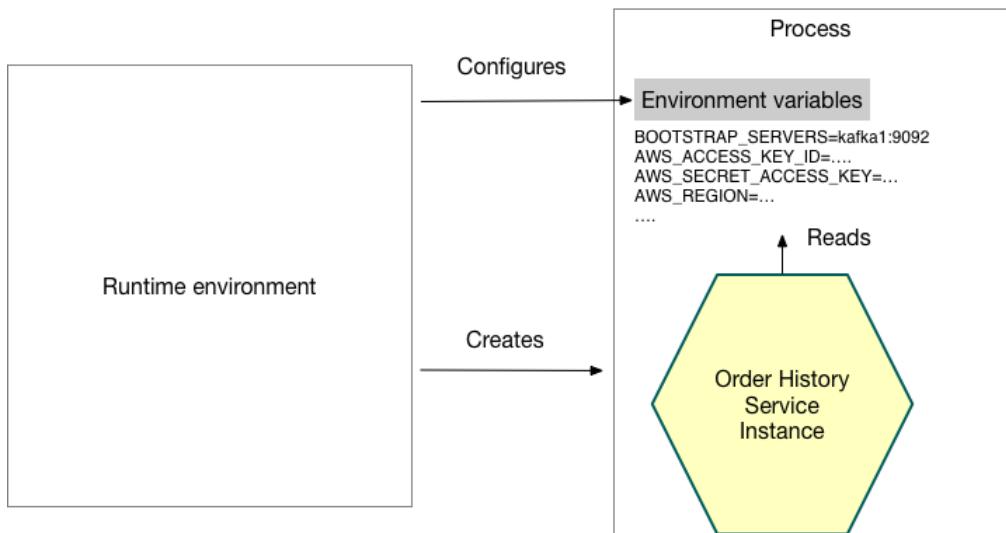
- Pull model - the service instance reads its configuration properties from a configuration server

Let's look at each approach starting with the push model.

10.2.1 Using push-based externalized configuration

The push model relies on the collaboration of the deployment environment and the service. The deployment environment supplies the configuration properties when it creates a service instance. It might, as figure 10.7 shows, pass the configuration properties as environment variables. Alternatively, the deployment environment might supply the configuration properties using a configuration file. The service instance then reads the configuration properties when it starts up.

Figure 10.7. When the runtime environment creates an instance of the Order History Service, it sets the environment variables containing the externalized configuration. The Order History Service reads those environment variables.



Obviously, the deployment environment and the service must agree on how the configuration properties are supplied. The precise mechanism depends on the specific deployment environment. For example, later in chapter 10 I describe how you can specify the environment variables of a Docker container.

Let's imagine that you have decided to supply externalized configuration property values using environment variables. Your application could simply call `System.getenv()` to obtain their values. However, if you are a Java developer then it's likely that you are using a framework that provides a more convenient mechanism. The FTGO services are built using Spring Boot, which has an extremely flexible

externalized configuration mechanism⁵⁹ that retrieves configuration properties from a variety of sources with well defined precedence rules. Let's look at how it works.

The Spring Boot framework reads properties from a variety of sources. The sources that I find useful in a microservice architecture are the following:

1. Command-line arguments
2. SPRING_APPLICATION_JSON, which is an operating system environment variable or JVM system property that contains JSON
3. JVM System properties
4. Operating system environment variables
5. A configuration file in the current directory

A particular property value from a source earlier in this list overrides the same property from a source later in this list. For example, operating system environment variables override properties read from a configuration file.

Spring Boot makes these properties available to the Spring framework's ApplicationContext. A service can, for example, obtain the value of a property using the @Value annotation:

```
public class OrderHistoryDynamoDBConfiguration {  
  
    @Value("${aws.region}")  
    private String awsRegion;
```

The Spring framework initializes the awsRegion field to the value of the aws.region property. This property is read from one of the sources listed above, such as a configuration file or from the AWS_REGION environment variable.

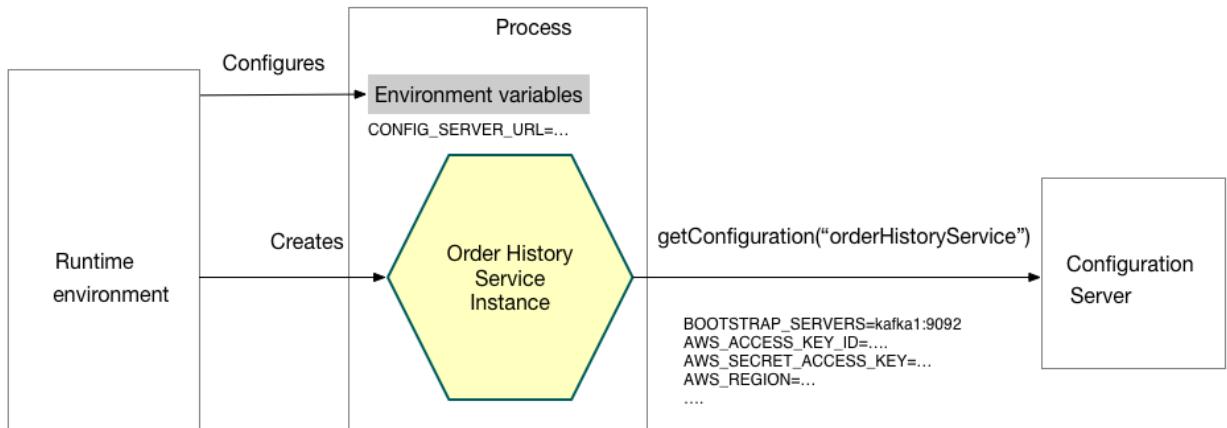
The push model is an effective and widely used mechanism for configuring a service. One limitation, however, is that reconfiguring a running service might be challenging, if not impossible. The deployment infrastructure might not allow you to change the externalized configuration of a running service without restarting it. You cannot, for example, change the environment variables of a running process. Another limitation is that there is a risk of the configuration property values being scattered throughout the definition of numerous services. As a result, you might want to consider using a pull-based model. Let's look at how it works.

10.2.2 Using pull-based externalized configuration

In the pull model, a service instance reads its configuration properties from a configuration server. Figure 10.8 shows how it works. On startup, a service instance queries the configuration service for its configuration. The configuration properties for accessing the configuration server, such as its network location, are provided to the service instance via a push-based configuration mechanism, such as environment variables.

⁵⁹ docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html

Figure 10.8. On startup, a service instance retrieves its configuration properties from a configuration server. The runtime environment provides the configuration properties for accessing the configuration server.



There are a variety of ways to implement a configuration server including:

- Version control system such as Git
- SQL and NoSQL databases
- Specialized configuration servers, such as Spring Cloud Config Server, Hashicorp Vault, and AWS Parameter Store

The Spring Cloud Config project is a good example of configuration server-based framework. It consists of a server and a client. The server supports a variety of backends for storing configuration properties including version control systems, databases and Hashicorp Vault. The client retrieves configuration properties from the server and injects them the Spring ApplicationContext

Using a configuration server has several benefits including:

- Centralized configuration - all of the configuration properties are stored in one place, which makes them easier to manage. What's more, in order to eliminate duplicate configuration properties, some implementations let you define global defaults, which can be overridden on a per-service basis.
- Transparent decryption of sensitive data - encrypting sensitive data such as database credentials is a security best practice. One challenge of using encryption, however, is that usually the service instance needs to decrypt them and so needs the encryption keys. Some configuration server implementations automatically decrypt properties before returning them to the service.
- Dynamic reconfiguration - a service could potentially detect updated property values by, for example, polling, and reconfigure itself.

The primary drawback of using a configuration server is that unless it's provided by the infrastructure it is yet another piece of infrastructure that needs to be setup and

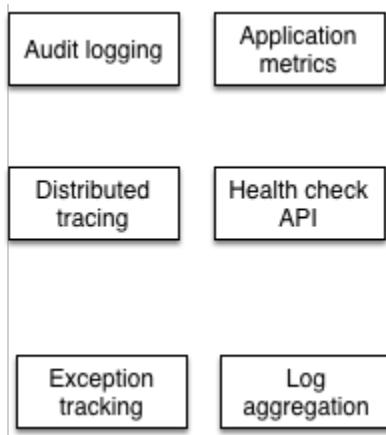
maintained. Fortunately, there are various open-source frameworks, such as Spring Cloud Config, which make it easier to run a configuration server. Now that we have looked at how to design configurable services, let's look at how to design observable services.

10.3 Designing observable services

Let's imagine that you have deployed the FTGO application into production. You probably want to know what the application is doing: requests per second, resource utilization, etc. You also need to be alerted if there is a problem, such as a failed service instance or a disk filling up, ideally before it impacts a user. And, if there is a problem you need to be able to troubleshoot and identify the root cause.

There are many aspects of managing an application in production that are outside the scope of the developer, such as monitoring hardware availability and utilization. These are clearly the responsibility of operations. There are, however, several patterns that you, as a service developer, must implement to make your service easier to manage and troubleshoot. These patterns expose a service instance's behavior and health. They enable a monitoring system to track and visualize the state of a service and generate alerts when there is a problem. These patterns also make troubleshooting problems easier.

Figure 10.9. The observability patterns enable developers and operations to understand the behavior of an application and troubleshoot problems. Operations are responsible for the infrastructure and developers are responsible for ensuring that their services are observable.



There are the following patterns:

- Health check API - expose an endpoint that returns the health of the service
- Log aggregation - log service activity and write logs into a centralized logging server, which provides searching and alerting
- Distributed tracing - assign each external request a unique id and trace requests as

they flow between services

- Exception tracking - report exceptions to an exception tracking service, which deduplicates exceptions, alerts developers, and tracks the resolution of each exception
- Application metrics - service maintain metrics, such as counters and gauges, and expose them to a metrics server
- Audit logging - log user actions

A distinctive feature of most of these patterns is that each pattern has a developer component and an operations component. Consider, for example, the Health check API pattern. The developer is responsible for ensuring that their service implements a health check endpoint. However, operations is responsible for the monitoring system that periodically invokes the health check API. Similarly, a developer is responsible for ensuring that their services logs useful information, while operations is responsible for log aggregation. Let's take a look at each of these patterns starting with the Health Check API pattern.

10.3.1 Health check API

Sometimes a service might be running yet its not able to handle requests. For instance, a newly started service instance might not be ready accept requests. The FTGO Consumer Service, for example, takes around ten seconds to initialize the messaging and database adapters. It would be pointless for the deployment infrastructure to route HTTP requests to a service instance until it's actually ready to process them.

Also, a service instance can fail without terminating. For example, a bug might cause an instance of the Consumer Service to run out of database connections and not be able to access the database. The deployment infrastructure should not route requests a service instance that have failed yet are still running. And, if the service instance does not recover, the deployment infrastructure must terminate it and create a new instance.

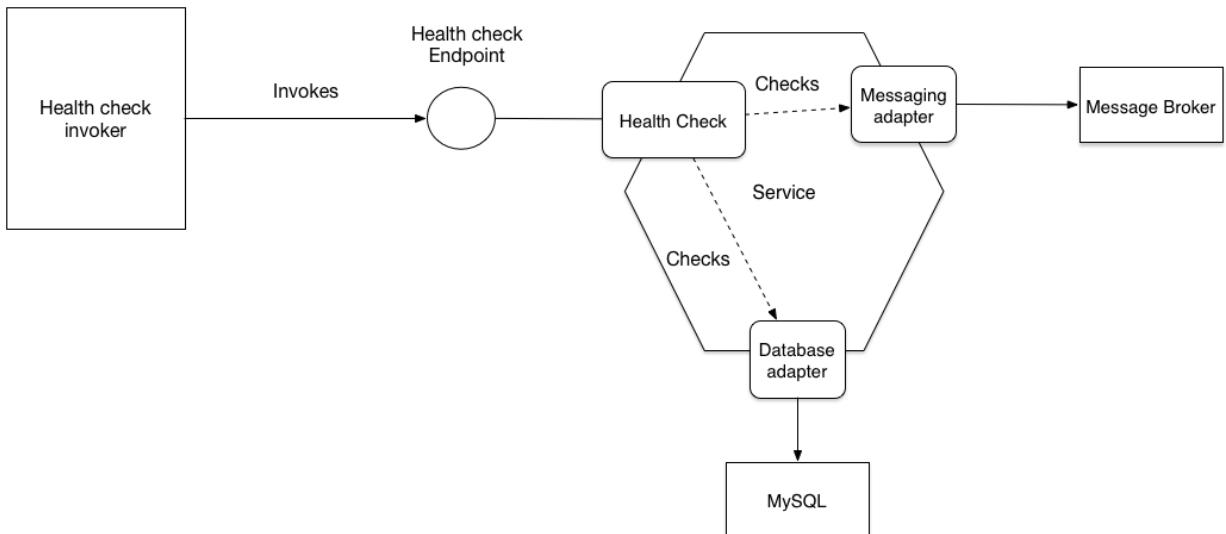
A service instance needs to be able to tell the runtime infrastructure whether or not it's able to handle requests. A good solution is for a service to implement a health check endpoint, which is shown in figure 10.10. The Spring Boot actuator Java library, for example, implements a `GET /health` endpoint, which returns `200` if and only if the service is healthy, and `503` otherwise. Similarly, the `HealthChecks .NET` library implements a `GET /hc` endpoint⁶⁰. The deployment infrastructure periodically invokes this endpoint to determine the health of the service instance and take the appropriate action if its unhealthy.

⁶⁰ docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/implement-resilient-applications/monitor-app-health

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

Figure 10.10. A service implements a health check endpoint, which is periodically invoked by the runtime infrastructure to determine the health of the service instance



A *Health Check Request Handler* typically tests the service instance's connections to external services. It might, for example, execute a test query against a database. If all of the tests succeed then the *Health Check Request Handler* returns a health response, such as an HTTP 200 status code. If any of them fail then it returns an unhealthy response, such as an HTTP 500 status code.

The *Health Check Request Handler* might simply return an empty HTTP response with the appropriate status code. Alternatively, it might return a detailed description of the health of each of the adapters. The detailed information is useful for troubleshooting. However, since it might contain sensitive information, some frameworks, such as Spring Boot Actuator, let you configure the level of detail in the health endpoint response response.

There are two issues you need to consider when using health checks. The first is the implementation of the endpoint, which must report back on the health of the service instance. The second issue how to configure the deployment infrastructure to invoke the health check endpoint. Let's first look at how to implement the endpoint.

Implementing the health check endpoint

The code that implements the health check endpoint must somehow determine the health of the service instance. One simple approach is to verify that the service instance can access its external infrastructure services. How to do this depends on the infrastructure service. The health check code can, for example, verify that it is connected to an RDBMS by obtaining a database connection and executing a test query. A more elaborate approach is to execute a synthetic transaction, which simulates the invocation the service's API. This kind of health check is more thorough but it is

likely to be more time consuming to implement and take longer to execute.

A great example of a health check library is Spring Boot Actuator. As I mentioned earlier, it implements a `/health` endpoint. The code that implements this endpoint, returns the result of executing a set of health checks. By using convention over configuration, Spring Boot Actuator implements a sensible set of health checks based on the infrastructure services used by the service. If, for example, a service uses a JDBC `DataSource` then Spring Boot Actuator configures a health check that executes a test query. Similarly, if the service uses the RabbitMQ message broker, it automatically configures a health check that verifies that the RabbitMQ server is up.

You can also customize this behavior by implementing additional health checks for your service. You implement a custom health check by defining a class that implements the `HealthIndicator` interface. This interface defines a `health()` method, which is called by the implementation of the `/health` endpoint. It returns outcome of the health check.

Invoking the health check endpoint

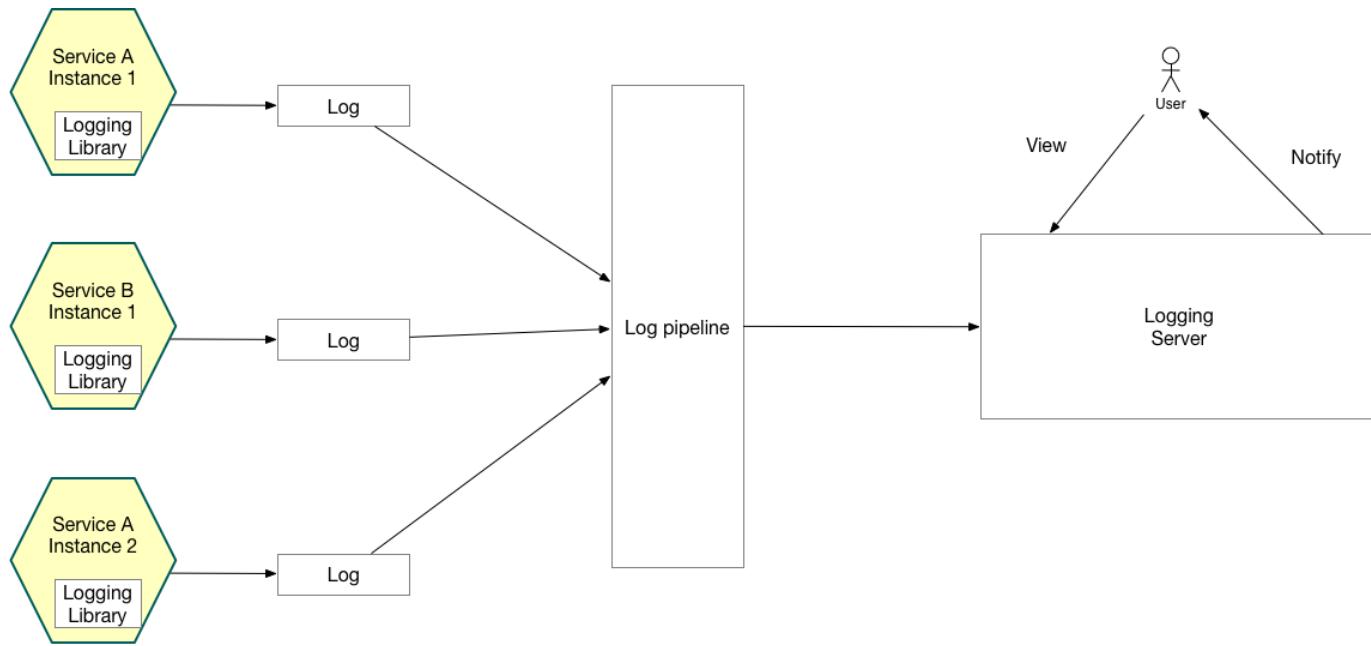
A health check endpoint isn't much use if nobody calls it. When you deploy your service, you must configure the deployment infrastructure to invoke the endpoint. How you do that depends on the specific details of your deployment infrastructure. For example, as I described in 3 you can configure some service registries, such as Netflix Eureka, to invoke the health check endpoint in order to determine whether traffic should be routed to the service instance. In chapter 10, I describe how to configure Docker and Kubernetes to invoke a health check endpoint.

10.3.2 Log aggregation

Logs are a valuable troubleshooting tool. If you want to know what's wrong with your application a good place to start are the log files. However, it's challenging to use logs in a microservices architecture. For example, let's imagine that you are debugging a problem with the `getOrderDetails()` query. As I described in chapter 8, the FTGO application implements this query using API composition. As a result, the log entries you need are scattered across log files of the API Gateway and several services including the Order Service and the Restaurant Order Service.

The solution is to use log aggregation. As figure 10.11 shows, the log aggregation pipeline sends the logs of all of the service instances to a centralized logging server. Once the logs are stored by the logging server, you can view, search and analyze them. You can also configure alerts that are triggered when certain messages appear in the logs.

Figure 10.11. The log aggregation infrastructure ships the logs of each service instance to a centralized logging server. Users can view and search the logs. They can also setup alerts, which are triggered when a log entries match search criteria.



The logging pipeline and server are usually the responsibility of operations. But service developers are responsible for writing services generate useful logs. Let's first look at how a service generates a log.

Service logging

As a service developer, there are couple of issues you need to consider. First, you need to decide which logging library to use. The second issue is where to write the log entries. Let's first look at the logging library.

Most programming language have one or more logging libraries that make it easy to generate correctly structured log entries. For example, three popular Java logging libraries are Logback, log4j and JUL (java.util.logging). There is also SLF4J, which is a logging facade API for the various logging frameworks. Similarly, Log4JS is a popular logging framework for NodeJS. One reasonable way to use logging is to simply sprinkle calls to one of these logging library to your service's code. However, if you have strict logging requirements that can't be enforced by the logging library then you might need to define your own logging API that wraps a logging library.

You also need to decide where to log. Traditionally, you would configure the logging framework to write to a log file in a well-known location in the filesystem. However, with the more modern deployment technologies, such as containers and serverless, which I describe in chapter 10, this is often not the best approach. In some

environments, such as AWS Lambda, there isn't even a 'permanent' filesystem to write the logs to! Instead, your service should log to `stdout`. The deployment infrastructure will then decide what to do with the output of your service.

Logging infrastructure

The logging infrastructure is responsible for aggregating the logs, storing them, and enabling the user to search them. One popular logging infrastructure is the ELK stack. ELK consists of three open source products:

- Elasticsearch - a text search oriented NoSQL database that is used as the logging server
- Logstash - a log pipeline that aggregates the service logs and writes them to Elasticsearch
- Kibana - a visualization tool for Elasticsearch

Other example open-source log pipelines include Fluentd and Apache Flume. Examples of logging servers including cloud services, such as AWS CloudWatch Logs, as well as numerous commercial offerings. Log aggregation is a useful debugging tool in a microservice architecture. Let's now look at distributed tracing, which is another way of understanding the behavior of a microservices-based application.

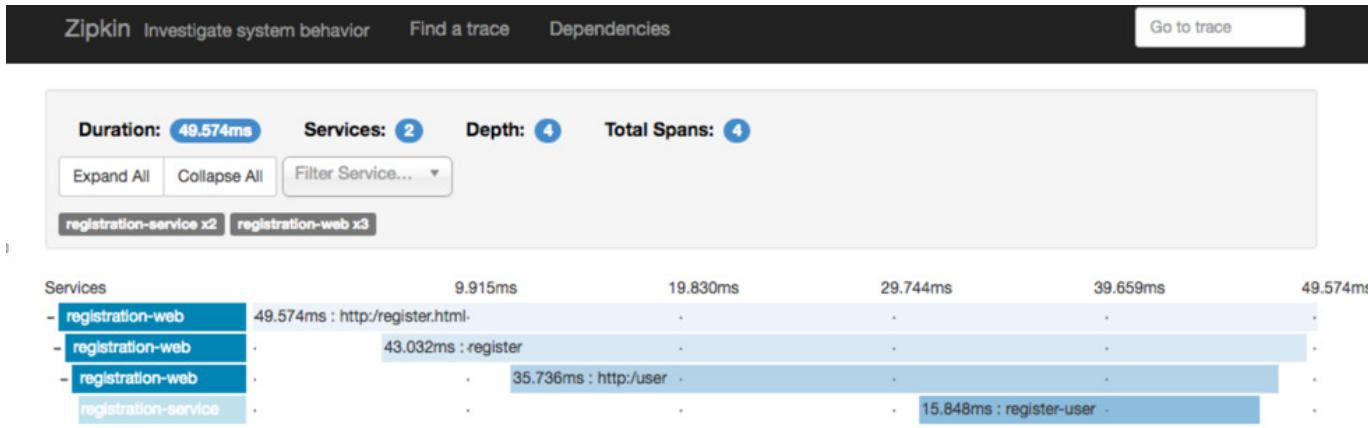
10.3.3 Distributed tracing

Let's imagine that you are an FTGO developer and are investigating why the `getOrderDetails()` query has slowed down. You have ruled out the problem being an external networking issue. The increased latency must be caused by either the API gateway or one of the services that it has invoked. One option is to look at each service's average response time. The trouble this option is that it is an average across requests rather than the timing breakdown for an individual request. What's more, more complex scenarios might involve many nested service invocations. You might not even be familiar with all of the services. As a result, it can be challenging to troubleshoot and diagnose these kinds of performance problems in a microservice architecture.

A good way to get insight into what your application is doing is to use distributed tracing. Distributed tracing is analogous to a performance profiler in a monolithic application. It records information (e.g. start time, end time) about the tree of service calls that are made when handling a request. You can then see how the services interact during the handling of external requests including a breakdown of where the time is spent.

Figure 10.12 shows an example of how a distributed tracing server displays what happens when the API Gateway handles a request. It shows the inbound request to the API gateway and the request that the gateway makes to the Order Service. For each request, the distributed tracing server shows the operation that is performed and the timing the request.

Figure 10.12. The Zipkin server shows how the FTGO application handles a request that is routed by the API gateway to the Order Service



TODO replace with proper diagram

Figure 10.12 shows what in distributed tracing terminology is called a trace. A trace represents an external request and consists of one or more spans. A span represents an operation and its key attributes are an operation name, start timestamp and end time. A span can have one or more child spans, which represents nested operations. For example, a top-level span might represent the invocation of the API gateway, as is the case in figure 10.12. Its child spans represent the invocations of services by the API gateway.

A valuable side-effect of distributed tracing is that assigns a unique id to each external request. A service can include the request id in its log entries. When combined with log aggregation, the request id enables you to easily find all log entries for a particular external request. For example, here is an example log entry from the Order Service:

```
2018-03-04 17:38:12.032 DEBUG [ftgo-order-service,8d8fdc37be104cc6,8d8fdc37be104cc6,false] 7 --- [nio-8080-exec-6]
org.hibernate.SQL : select order0_.id as id1_3_0_,
order0_.consumer_id as consumer2_3_0_, order0_.city as city3_3_0_,
order0_.delivery_state as delivery4_3_0_, order0_.street1 as street5_3_0_,
order0_.street2 as street6_3_0_, order0_.zip as zip7_3_0_, order0_.delivery_time as delivery8_3_0_, order0_._

The [ftgo-order-service,8d8fdc37be104cc6,8d8fdc37be104cc6,false] part of the log entry (the SLF4J Mapped Diagnostic Context61) contains information from the distributed tracing infrastructure. It consists of four values:
```

- ftgo-order-service - the name of the application

⁶¹ www.slf4j.org/manual.html

- `8d8fdc37be104cc6` - the `traceId`
- `spanId` - the `spanId`
- `false` - indicates that this span was not exported to the distributed tracing server.

If you search the logs for `8d8fdc37be104cc6` then you will find all log entries for that request.

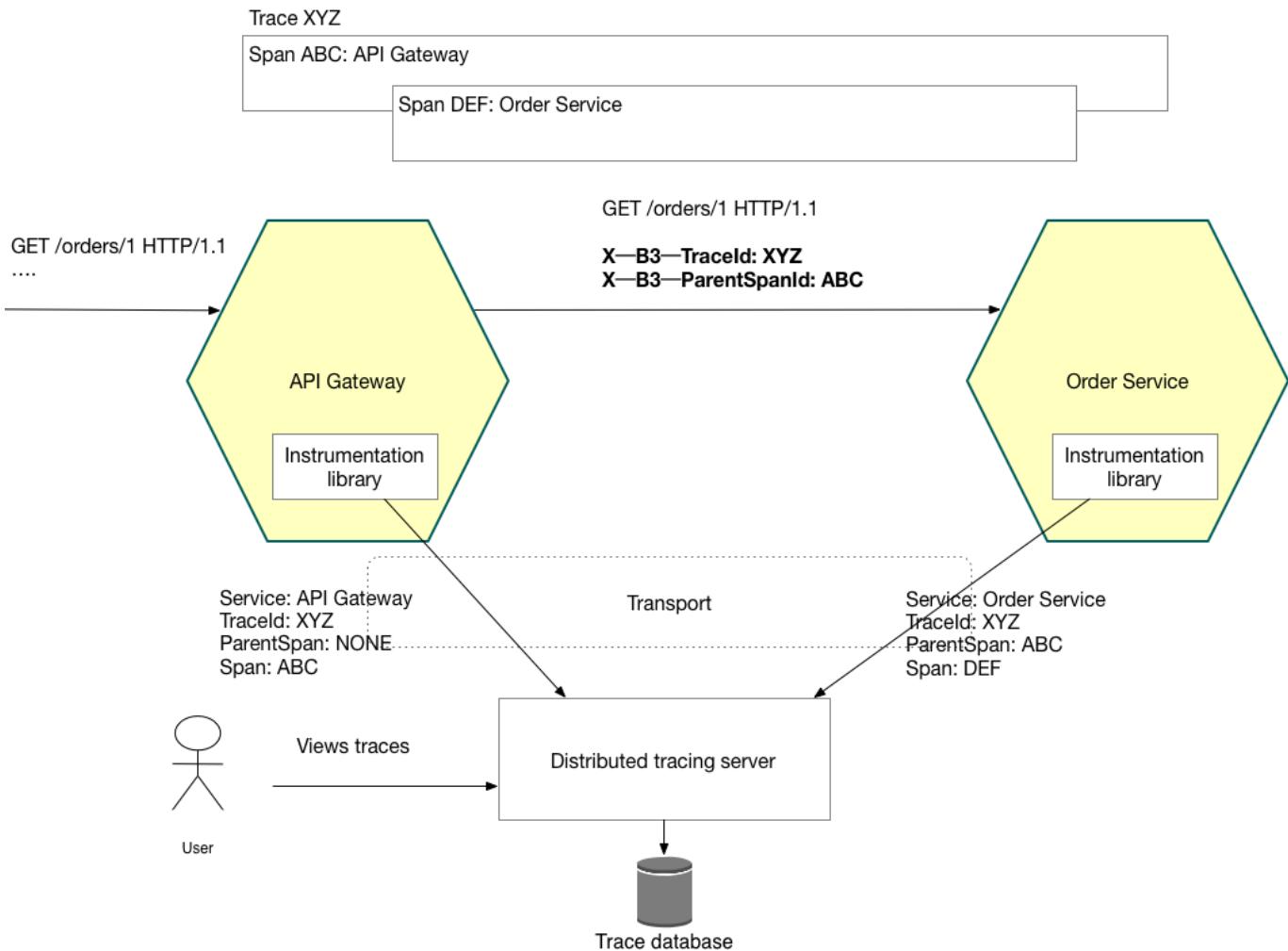
Figure 10.13 shows how distributed tracing works. There are two parts to distributed tracing, an instrumentation library, which is used by each service, and a distributed tracing server. The instrumentation library manages the traces and spans. It also adds tracing information, such as the current trace id and the the parent span id, to outbound requests. For example, one common standard for propagating trace information is the B3 standard⁶², which uses headers such as `X-B3-TraceId` and `X-B3-ParentSpanId`. The instrumentation library also reports traces to the distributed tracing server. The distributed tracing server stores the traces and provides a UI for visualizing them.

⁶² github.com/openzipkin/b3-propagation

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

Figure 10.13. Each service (including the API gateway) uses an instrumentation library. The instrumentation library assigns an id to each external request, propagates tracing state between services, and reports spans to the distributed tracing server.



Let's take a look at the instrumentation library and the distribution tracing server beginning with the library.

Using an instrumentation library

The instrumentation library builds the tree of spans and sends them to the distributed tracing server. The service code could call the instrumentation library directly. However, this means that instrumentation logic is intertwined with business and other logic. A cleaner approach is to use interceptors or Aspect-Oriented Programming (AOP).

A great example of an AOP-based framework is Spring Cloud Sleuth. It uses the Spring framework's AOP mechanism to automagically integrate distributed tracing into the service. As a result, you simply have to add Spring Cloud Sleuth as a project dependency. Your service does not need to call a distributed tracing API except in those cases that are not handled by Spring Cloud Sleuth.

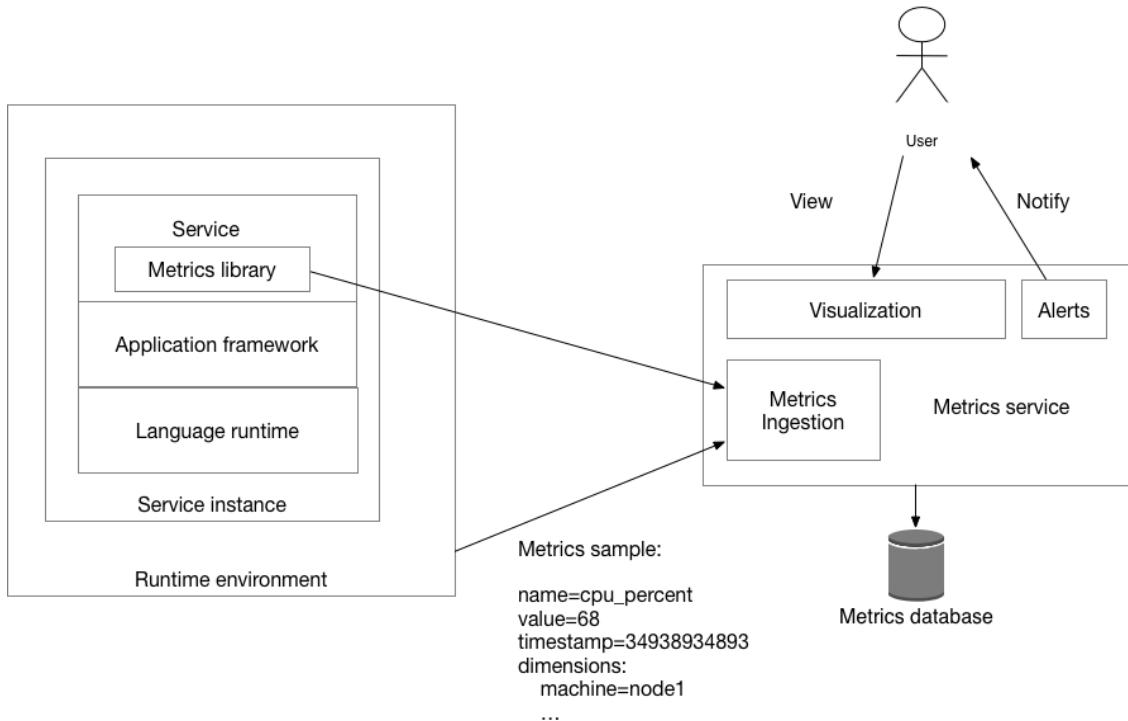
Distributed tracing server

The instrumentation library sends the spans to a distributed tracing server. The distributed tracing server stitches the spans together to form complete traces and stores them in a database. One popular distributed tracing server is Open Zipkin. Zipkin was originally developed by Twitter. Services can deliver spans to Zipkin using either HTTP or message broker. Zipkin stores the traces in a storage backend, which is either a SQL or NoSQL database. It has a UI that displays traces as is shown earlier in figure 10.12. Other examples of AWS X-ray is another example of a distributed tracing server.

10.3.4 Application metrics

A key part of the production environment is monitoring and alerting. As figure 10.14 shows, the monitoring system gathers metrics, which provide critical information about the health of an application, from every part of the technology stack. Metrics range from infrastructure-level metrics, such as CPU, memory and disk utilization, to application-level metrics, such as service request latency and number of requests executed. The Order Service, for example, gathers metrics about the number of placed, approved and rejected orders. The metrics are collected by a metrics service, which provides visualization and alerting.

Figure 10.14. Metrics at every level of the stack are collected and stored in a metrics service, which provides visualization and alerting.



Metrics are sampled periodically. A metric sample has the following three properties:

- name - the name of the metric, e.g. `jvm_memory_max_bytes` or `placed_orders`
- value - a numeric value
- timestamp - the time of the sample

In addition, some monitoring systems support the concept of dimensions, which are arbitrary name-value pairs. For example, `jvm_memory_max_bytes` is reported with dimensions, such as `area="heap",id="PS Eden Space"` and `area="heap",id="PS Old Gen"`. Dimensions are often used to provide additional information, such as the machine name or service name, or service instance identifier. A monitoring system typically aggregates (e.g. sums or averages) metric samples along one or more dimensions.

Many aspects of monitoring are the responsibility of operations. However, a service developer is responsible for two aspects of metrics. First, they must instrument their service so that collects metrics about its behavior. Second, they must expose those service metrics along with metrics from the JVM and the application framework to the metrics server. Let's first look at how a service collects metrics.

Collecting service-level metrics

How much work you need to do to collect metrics depends on the frameworks that your application uses and the metrics you want to collect. A Spring Boot-based service can, for example, gather (and expose) basic metrics, such as JVM metrics, simply by including the Micrometer Metrics library as a dependency and using a few lines of configuration. Spring Boot's auto-configuration takes care of configuring the metrics library and exposing the metrics. A service only needs to use the Micrometer Metrics API directly if it gathers application-specific metrics.

Listing 10.1 shows how the `OrderService` can collect metrics about the number of orders placed, approved and rejected. It uses the `MeterRegistry`, which is the interface provided by Micrometer Metrics to gather custom metrics. Each method invokes an appropriately named counter.

Listing 10.1. The OrderService tracks the number of orders placed, approved and rejected using the Micrometer Metrics library. Each method increments the appropriate counter.

```
public class OrderService {

    @Autowired
    private MeterRegistry meterRegistry; ①

    public Order createOrder(...) {
        ...
        meterRegistry.counter("placed_orders").increment(); ②
        return order;
    }

    public void approveOrder(long orderId) {
        ...
        meterRegistry.counter("approved_orders").increment(); ③
    }

    public void rejectOrder(long orderId) {
        ...
        meterRegistry.counter("rejected_orders").increment(); ④
    }
}
```

- ① The Micrometer Metrics library API for managing application-specific meters
- ② Increment the `placedOrders` counter when an order has successfully been placed
- ③ Increment the `approvedOrders` counter when an order has approved
- ④ Increment the `rejectedOrders` counter when an order has rejected

Delivering metrics to the metrics service

A service delivers metrics to the metrics service in one of two ways: push or pull. With the push model, a service instance sends the metrics to the metrics service by invoking an API. AWS Cloudwatch metrics, for example, implements the push model. With the pull model, the metrics service (or its agent running locally) invokes an service API to

retrieve the metrics from the service instance. Prometheus, which is a popular open-source monitoring and alerting system, uses the pull model.

The FTGO application's Order Service is configured to integrate with Prometheus by including the `micrometer-registry-prometheus` library. Because this library is on the classpath, Spring Boot exposes an `GET /actuator/prometheus` endpoint, which returns metrics in the format that Prometheus expects. The custom metrics from the `OrderService` are reported as follows:

```
$ curl -v http://localhost:8080/actuator/prometheus | grep _orders
# HELP placed_orders_total
# TYPE placed_orders_total counter
placed_orders_total{service="ftgo-order-service",} 1.0
# HELP approved_orders_total
# TYPE approved_orders_total counter
approved_orders_total{service="ftgo-order-service",} 1.0
```

The `placed_orders` counter is, for example, reported as a metric of type `counter`.

The Prometheus server periodically polls this URL to retrieve metrics. Once the metrics are in Prometheus, you can view them using Grafana⁶³, which is a data visualization tool. You can also set up alerts for these metrics, such as when the rate of change for `placed_orders_total` falls below some threshold.

10.3.5 Exception tracking

A service should rarely log an exception and when it does it's important to identify the root cause. The exception might be a symptom of a failure or a programming bug. The traditional way to view exceptions is to look in the logs. You might even configure the logging server to alert you if an exception appears in the log file.

There are, however, several problems with this approach:

1. Log files are oriented around single line log entries, whereas exceptions consist of multiple lines
2. There is no mechanism to track the resolution of exceptions that occur in log files. You would have to manually copy/paste the exception into an issue tracker.
3. There are likely to be duplicate exceptions but there is not automatic mechanism to treat them as one

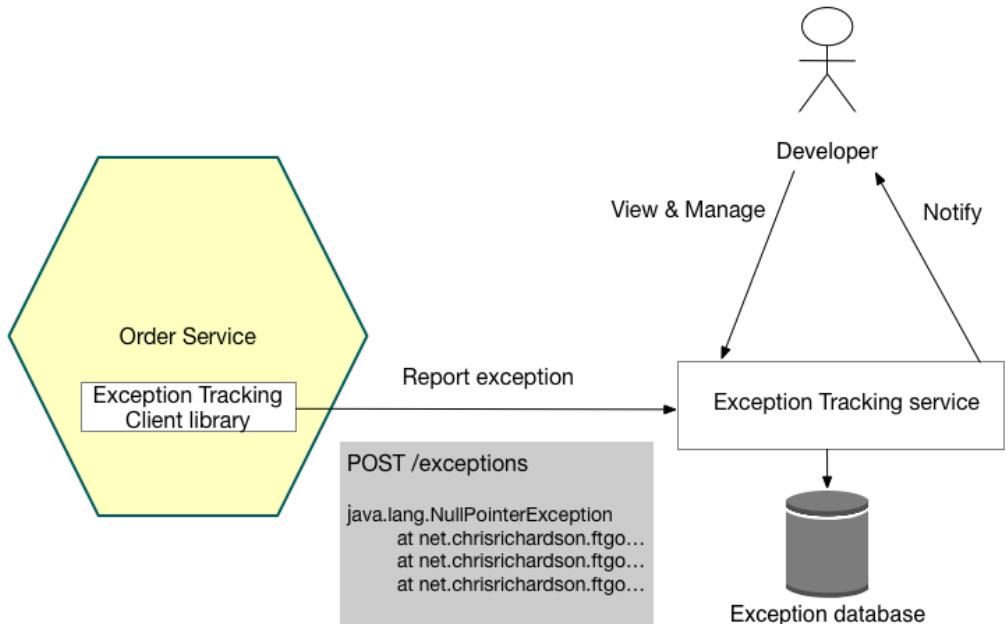
A better approach is to use an exception tracking service. As figure 10.15 shows, you configure your service to report exceptions to an exception tracking service via, for example, a REST API. The exception tracking service de-duplicates identical exceptions, generates alerts and manages the resolution of exceptions.

⁶³ grafana.com/

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

Figure 10.15. A service reports exceptions to an exception tracking service, which deduplicates exceptions and alerts developers. It has a UI for viewing and managing exceptions.



There are several exception tracking services. Some, such as Honey Badger, are purely cloud-based. Others, such as Sentry.io, are cloud-based and open-source. There are usually several ways to integrate the exception tracking client with your application. Your service could simply invoke the Exception Tracking Service's API directly. A better approach is to use a client library. For example, HoneyBadger's client library provides a variety of integration mechanisms including a Servlet Filter that catches a reports exceptions.

10.3.6 Audit logging

The purpose of audit logging is to record each user's actions. An audit log is typically used to help customer support, ensure compliance and to detect suspicious behavior. Each audit log entry records the identity of the user, the action they performed and the business object(s). An application usually stores the audit log in a database table.

There are a few different ways to implement audit logging:

- Add audit logging code to the business logic
- Use aspect-oriented programming (AOP)
- Use event sourcing

Let's look at each option.

Add audit logging code to the business logic

The first option, which is the most straightforward, is to sprinkle audit logging code throughout your service's business logic. Each service method, for example, can simply create an audit log entry and save it in the database. The drawback with this approach is that it intertwines auditing logging code and business logic, which reduces maintainability. The other drawback is that its potentially error-prone since it relies on the developer writing audit logging code.

Use aspect-oriented programming (AOP)

The second option is to use aspect-oriented programming (AOP). You can use an AOP framework, such as Spring AOP, to define advice that automatically intercepts each service method call and persists an audit log entry. This is a much more reliable approach since it automatically records every service method invocation. The main drawback of using AOP is that the advice only has access to the method name and its arguments and so it might be challenging to determine the business object that is being acted upon and generate a business-oriented audit log entry.

Use event sourcing

The third and final option is to implement your business logic use event sourcing. As I mentioned in chapter 6, event sourcing automatically provides an audit log for create and update operations. You simply need to record the identity of the user in each event. One limitation with using event sourcing, however, is that it doesn't record queries. If your service must create log entries for queries then you will have to use one of the other options as well.

10.4 Developing services using a microservice chassis

In this chapter I've described numerous concerns that a service must implement including metrics, reporting exceptions to exception tracker, logging and health checks, externalized configuration, and security. Moreover, as I've described in chapter 3, a service might also need to handle service discovery and implement circuit breakers. This is not something that you would want to setup from scratch each time you implement a new service. If you did, it would potentially be days, if not weeks, before you wrote your first line of business logic. Instead, you want to build your services upon a microservices chassis.

10.4.1 Using a microservice chassis

A microservices chassis is a framework or set of frameworks that handle these numerous concerns. They significant reduce the amount of code that you need to write. You might not even need to write any code and instead simply configure the microservice chassis framework to fit your requirements. As a result, you are able to focus on developing your services business logic.

The FTGO application uses Spring Boot and Spring Cloud as the microservice chassis. Spring Boot provides functions such as externalized configuration. Spring Cloud

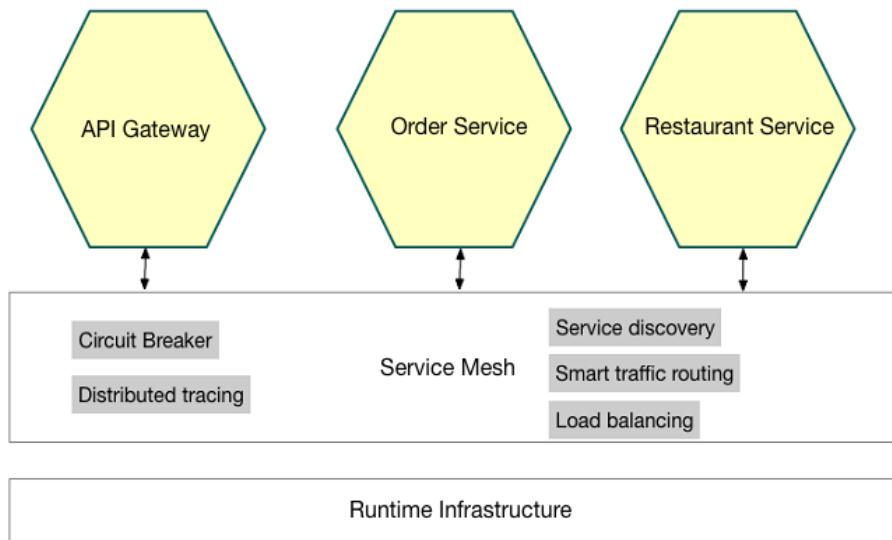
provides functions, such as circuit breakers. It also implements client-side service discovery although the FTGO application relies on the infrastructure for service discovery. Spring Boot and Spring Cloud are not the only microservices chassis frameworks. If, for example, you writing services in GoLang then you could use either Go Kit⁶⁴ or Micro⁶⁵.

One drawback of using a microservice chassis is that you need one for every language/platform combination that you use to develop services. Fortunately, it's likely that many of the functions implemented by a microservice chassis will instead be implemented by the infrastructure. For example, as I described in chapter 3, many deployment environments handle service discovery. What's more, many of the network-related functions of a microservice chassis will be handled by what is known as a service mesh.

10.4.2 From microservice chassis to service mesh

A service mesh is low-level infrastructure that mediates the communication between a service and other services and external applications. As figure 10.16 shows, all network traffic in and out of a service goes through the service mesh. It implements various functions including circuit breakers, distributed tracing, service discovery, load balancing and rule-based traffic routing. As a result, you no longer need to implement these functions in the services.

Figure 10.16. All network traffic in and out of a service flows through the service mesh. The service mesh implements various functions including circuit breakers, distributed tracing, service discovery, and load balancing.



⁶⁴ github.com/go-kit/kit

⁶⁵ github.com/micro/micro

There are various service mesh implementations including Istio, Linkerd and Conduit. As of the time of writing, Linkerd is the most mature with Istio and Conduit still under active development. For more information about this exciting new technology please take a look at each product's documentation.

10.5 Summary

- It's essential that a service implements its functional requirements but it must also be secure, configurable and observable.
- Many aspects of security in a microservice architecture are no different than in a monolithic architecture. However, there are some aspects of application security that are necessarily different including how user identity is passed between the API gateway the services and who is responsible for authentication and authorization. A commonly used approach is for the API Gateway to authenticate clients. The API gateway includes a transparent token, such as a JWT, in each request to a service. The token contains the identity of the principal and their roles. The services use the information in the token to authorize access to resources. OAuth 2.0 is a good foundation for security in a microservice architecture.
- A service typically uses one or more external services, such as message brokers and databases. The network location and credentials of each external service often depends on the environment that the service is running. You must use an externalized configuration mechanism that provides a service with configuration properties at runtime. One commonly used approach is for the deployment infrastructure to supply those properties via operating system environment variables or a properties file when it creates a service instance. Another option is for a service instance to retrieve its configuration from a configuration properties server.
- Operations and developers share responsibility for observability. Operations are responsible for the observability infrastructure, such as servers handle log aggregation, metrics, exception tracking, and distributed tracing. Developers are responsible for ensuring that their services are observable. Services must have health check API endpoint, generates log entries, collect and expose metrics, report exceptions to an exception tracking service, and implement distributed tracing.
- In order to simplify and accelerate development, you should develop services on top of a microservices chassis framework. A microservices chassis is a framework, which handles various cross-cutting concerns including those that I've described in this chapter. However, over time it's likely that many of the networking-related functions of a microservice chassis will migrate into a service mesh, which is a layer of infrastructure software through which all of a service's network traffic flows.

11

Deploying microservices

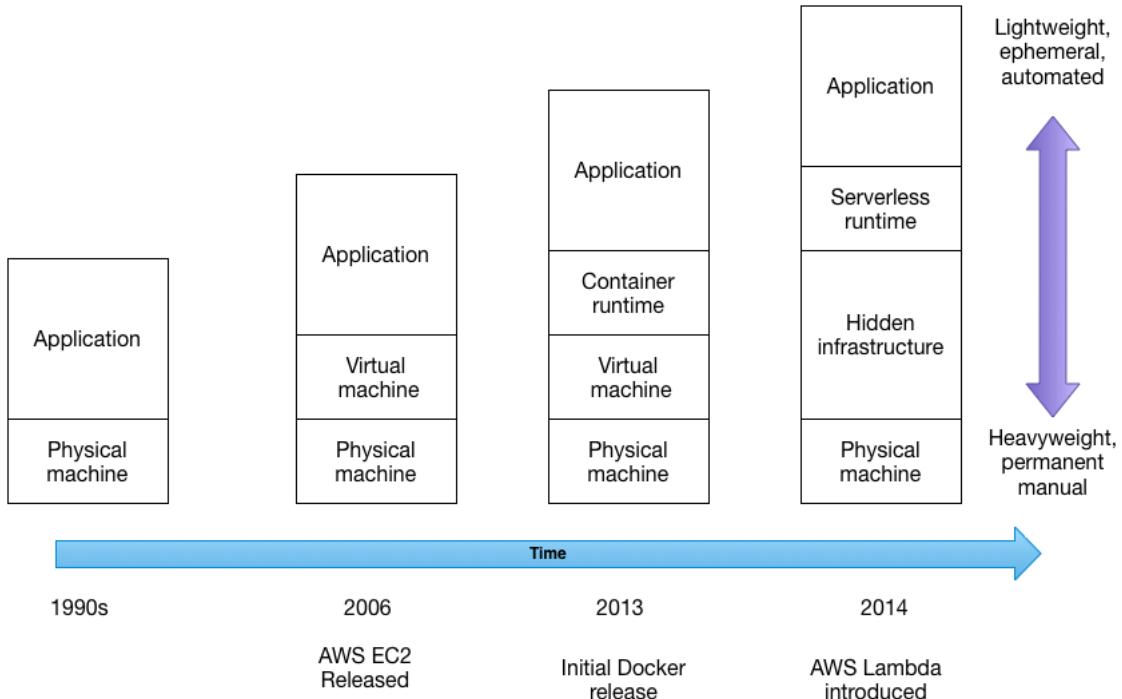
This chapter covers:

- The four key deployment patterns: how they work, and their benefits and drawbacks
- Deploying services with Docker and Kubernetes
- Deploying services with AWS Lambda
- How to pick a deployment pattern

Let's imagine that you and your team have developed a service. It works well on your laptops and the Jenkins server. But obviously, this is not good enough. Software has no value until it is running in production and available to its users. You need to deploy your service into production.

Deployment is a combination of two inter-related concepts: process and architecture. The deployment process consists of the steps that must be performed by people - developers and operations - in order to get software into production. The deployment architecture defines the structure of the environment in which that software runs. Both aspects of deployment have changed radically since I first started developing Enterprise Java applications in the late 1990s. The manual process of developers throwing code over the wall to production has become highly automated. Physical production environments have, as figure 11.1 shows, been replaced by increasingly lightweight and ephemeral computing infrastructure.

Figure 11.1. Heavyweight, and long-lived physical machines have been abstracted away by increasingly lightweight and ephemeral technologies



Back in the 1990s, if you wanted to deploy an application into production the first step was to throw your application along with a set of operating instructions over the wall to operations. You might, for example, simply file a trouble ticket asking operations to deploy the application. Whatever happened next was entirely the responsibility of operations unless they encountered a problem that they needed your help to fix. Typically, operations bought expensive servers and installed expensive and heavyweight application server such as WebLogic or WebSphere. After that they would log in to the application server console and deploy your applications. They would lovingly care for those machines, as if they were pets, installing patches and updating the software.

In the mid 2000s, the expensive application servers were replaced with open-source, lightweight web containers such as Apache Tomcat and Jetty. You could still run multiple applications on each web container but it became feasible to have one application per web container. Also, virtual machines started to replace physical machines. But machines were still treated as beloved pets and deployment was still a fundamentally manual process.

Today, the deployment process is radically different. Instead of handing off code to a separate production team, the adoption of DevOps means that the development team is also responsible for deploying their application or services. In some organizations,

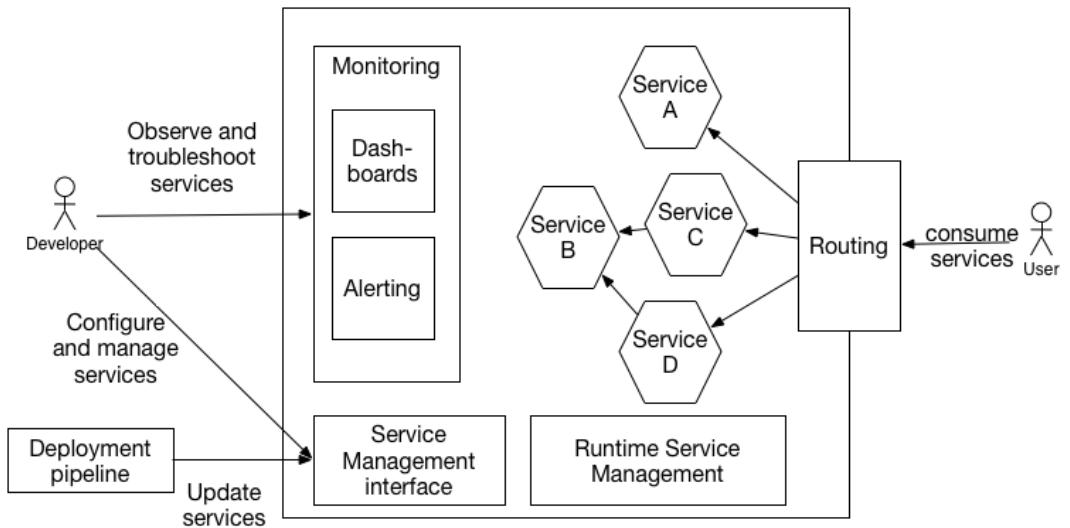
operations provides developers with a console for the deploying their code. Or better yet, once the tests pass, the deployment pipeline automatically deploys the code into production.

The computing resources used in a production environment have also changed radically with physical machines being abstracted away. Virtual machines running on a highly automated cloud, such as AWS, have replaced the long-lived, pet-like physical and virtual machines. Today's virtual machines are immutable. They are treated as disposable cattle instead of pets and are discarded and recreated rather than being reconfigured. Containers, which are an even more lightweight abstraction layer of top of virtual machines, are an increasingly popular way of deploying applications. You can also use an even more lightweight serverless deployment platform, such as AWS Lambda, for many use cases.

It is no coincidence that the evolution of deployment processes and architectures has coincided with the growing adoption of the microservice architecture. An application might have tens or hundreds of services written in a variety of languages and frameworks. Since each service is a small application, you essentially have tens or hundreds as many applications in production. It is no longer practical, for example, for system administrators to hand configure servers and services. If you want to deploy microservices at scale, you need a highly automated deployment process and infrastructure.

Figure 11.2 shows a high-level view of a production environment. The production environment enables developers to configure and manage their services, the deployment pipeline to deploy new versions of services, and users to access functionality implemented by those services.

Figure 11.2. A simplified view of the production environment. It provides four main capabilities: service management, which enables developers to deploy and manage their services; runtime management, which ensures that the services are running; monitoring, which visualizes service behavior and generates alerts; and request routing, which routes requests from users to the services.



A production environment must implement four key capabilities:

1. Service management interface - enables developers to create, update, and configure services. Ideally, this interface is a REST API, which is invoked by command line and GUI deployment tools
2. Runtime service management - attempts to ensure that the desired number of service instances are running at all times. If a service instance crashes or is somehow unable to handle requests, the production environment must restart it. If a machine crashes, then the production environment must restart those service instances on a different machine.
3. Monitoring - provides developers with insight into what their services are doing including log files, and metrics. If there are problems, the production environment must alert the developers. I describe monitoring, a.k.a. observability, in chapter {chapter-prod-ready}.
4. Request routing - routes requests from users to the services

In this chapter I describe the four main deployment options.

- Deploying services as language-specific packages, such as Java JAR or WAR files. It's worthwhile exploring this option because even though I recommend using one of the more options, its drawbacks motivate the other options.
- Deploying services as virtual machines, which simplifies deployment by packaging a service as a virtual machine image that encapsulates the service's technology stack.

- Deploying services as containers, which are more lightweight than virtual machines. I show how to deploy the FTGO application's `Restaurant service` using Kubernetes, which is a popular Docker orchestration framework.
- Deploying services using serverless deployment, which is even more modern than containers. We will look at how to deploy the `Restaurant service` using AWS Lambda, which is a popular serverless platform.

Let's first look at how to deploy services as language-specific packages.

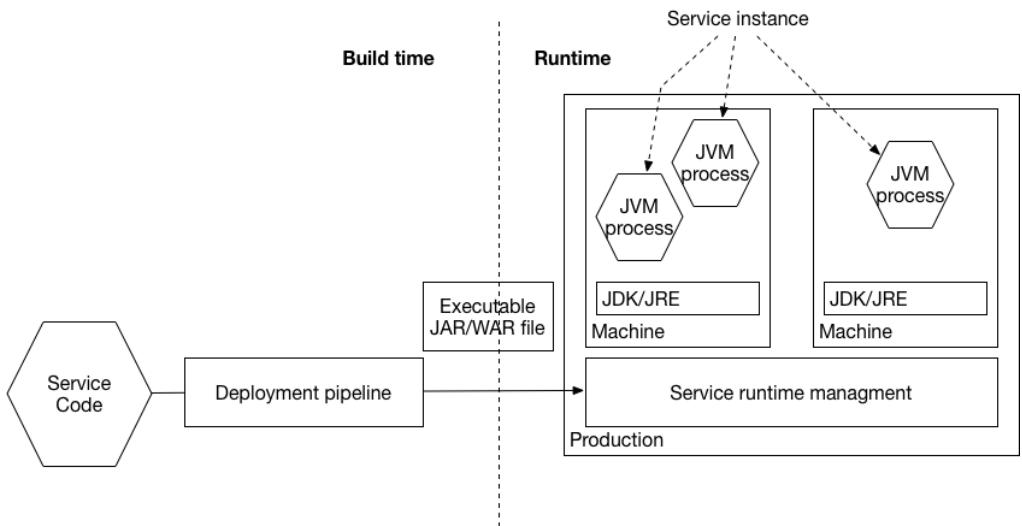
11.1 Deploying services as *language-specific packages*

Let's imagine that you want to deploy the FTGO application's `Restaurant service`, which is a Spring Boot-based Java application. One way to deploy this service is using the *Service as a language-specific package* pattern. When using this pattern what is deployed in production and what is managed by the service runtime is a service in its language-specific package. In the case of the `Restaurant service` that's either the executable JAR file or a WAR file. For other languages, such as NodeJS, a service is a directory of source code and modules. For some languages, such as GoLang, a service is an operating system-specific executable.

To deploy the `Restaurant service` service on a machine, you would first install the necessary runtime, which in this case is the JDK. If it's a WAR file, then you would also need to install a web container such as Apache Tomcat. Once you have configured the machine, you then copy the package to the machine and start the service. Each service instance runs as a JVM process.

Ideally, you have set up your deployment pipeline to automatically deploy the service in to production as shown in figure 11.3. The deployment pipeline builds an executable JAR file or WAR file. It then invokes the production environment's service management interface to deploy the new version.

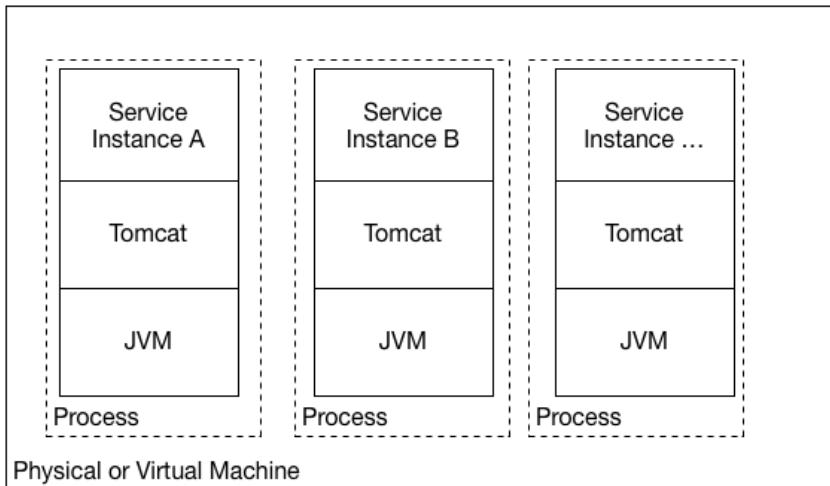
Figure 11.3. The deployment pipeline builds an executable JAR file and deploys it into production. In production, each service instance is a JVM running on the a machine that has the JDK or JRE installed.



A service instance is typically a single process but sometimes might be a group of processes. A Java service instance is, for example, a process running the JVM. A NodeJS service might spawn multiple worker processes in order to process requests concurrently. Some languages support deploying multiple service instances within the same process.

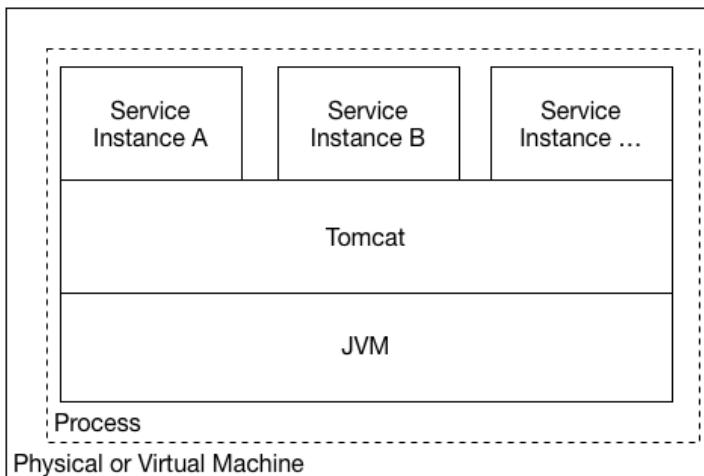
Sometimes, you might deploy a single service instance on a machine but you also have the option to deploy multiple service instances on the same machine. For example, as figure 11.4 shows, you could run multiple JVMs on a single machine. Each JVM runs a single service instance.

Figure 11.4. Deploying multiple service instances on the same machine. They might be instances of the same service or they might be instances of different services. The overhead of the OS is shared amongst the service instance. Each service instance is a separate process and so there is some isolation between them.



Some languages also let you run multiple services instances in a single process. For example, as figure 11.5, shows you can run multiple Java services on a single Apache Tomcat.

Figure 11.5. Deploying multiple services instances on the same web container or application server. They might be instances of the same service or they might be instances of different services. The overhead of the OS and runtime is shared amongst all of the service instances. However, since the service instances are in the same process there is no isolation between them.



This approach is commonly used when deploying applications on traditional, expensive and heavyweight application servers, such as WebLogic and Websphere. You can also package services as OSGI bundles and run multiple service instances in each OSGI container.

The Service as a language-specific package pattern has both benefits and drawbacks. Let's first look at the benefits.

11.1.1 Benefits of the Service as a language-specific package pattern

The Service as a language-specific package pattern has a few benefits:

- Fast deployment
- Efficient resource utilization, especially when running multiple instances in same machine or within the same process.

Let's look at each one.

Fast deployment

One major benefit of this pattern is that deploying a service instance is relatively fast. You simply copy the service to a host and start it. If the service is written in Java then you copy a JAR or WAR file. For other languages, such as NodeJS or Ruby you copy the source code. In either case, the number of bytes that are copied over the network is relatively small.

Also, starting a service is rarely time consuming. If the service is its own process, you simply start it. Otherwise, if the service is one of several instances running in the same container process then you either dynamically deploy it into the container or restart the container. Because of the lack of overhead, starting a service is usually very fast.

Efficient resource utilization

Another major benefit is that uses resources relatively efficiently. Multiple service instances share the machine and its operating system. It is even more efficient if a multiple service instances run within the same process. For example, multiple web applications could share the same Apache Tomcat server and JVM.

11.1.2 Drawbacks of the Service as a language-specific package pattern

Despite its appeal, the Service as a language-specific package pattern has several significant drawbacks:

- Lack of encapsulation of the technology stack
- No ability to constrain the resources consumed by a service instance
- Lack of isolation when running multiple service instances on the same machine
- Automatically determining where to place service instances is challenging

Let's look at each drawback.

Lack of encapsulation of the technology stack

One major drawback is that the operation team must know the specific details of how to deploy each and every service. Each service needs a particular version of the runtime. A Java web application, for example, needs particular versions of Apache Tomcat and the JDK. Operations must install the correct version of each required software package.

To make matters worse, services can be written in a variety of languages and frameworks. They might also be written in multiple versions of those languages and frameworks. Consequently, the development team must share lots of details with operations. This complexity increases the risk of errors during deployment. A machine might, for example, have the wrong version of the language runtime.

No ability to constrain the resources consumed by a service instance

Another drawback is that you can't constrain the resources consumed by a service instance. A process can potentially consume all of a machine's CPU or memory and so starve other service instances and operating system of resources. This might happen, for example, because of a bug.

Lack of isolation when running multiple service instances on the same machine

The problem is even worse when running multiple instances on the same machine. The lack of isolation means that a misbehaving service instance can impact other service instances. As a result, there is the risk of the application being unreliable especially when running multiple service instances on the same machine.

Automatically determining where to place service instances is challenging

Another challenge with running multiple service instances on the same machine is determining the placement of service instances. Each machine has a fixed set of resources, CPU, memory, etc, and each service instance needs amount of resources. It is important to assign service instances to machines in a way that uses the machines efficiently without overloading them. As I describe below, VM-based clouds and container orchestration frameworks handle this automatically. When deploying services natively, it's likely that you will need to manually decide the placement.

As you can see, despite its familiarity, the Service as a language-specific package pattern has some significant drawbacks. You should rarely use this approach, except perhaps when efficiency outweighs all other concerns. Let's now look at modern ways of deploying services that avoid these problems.

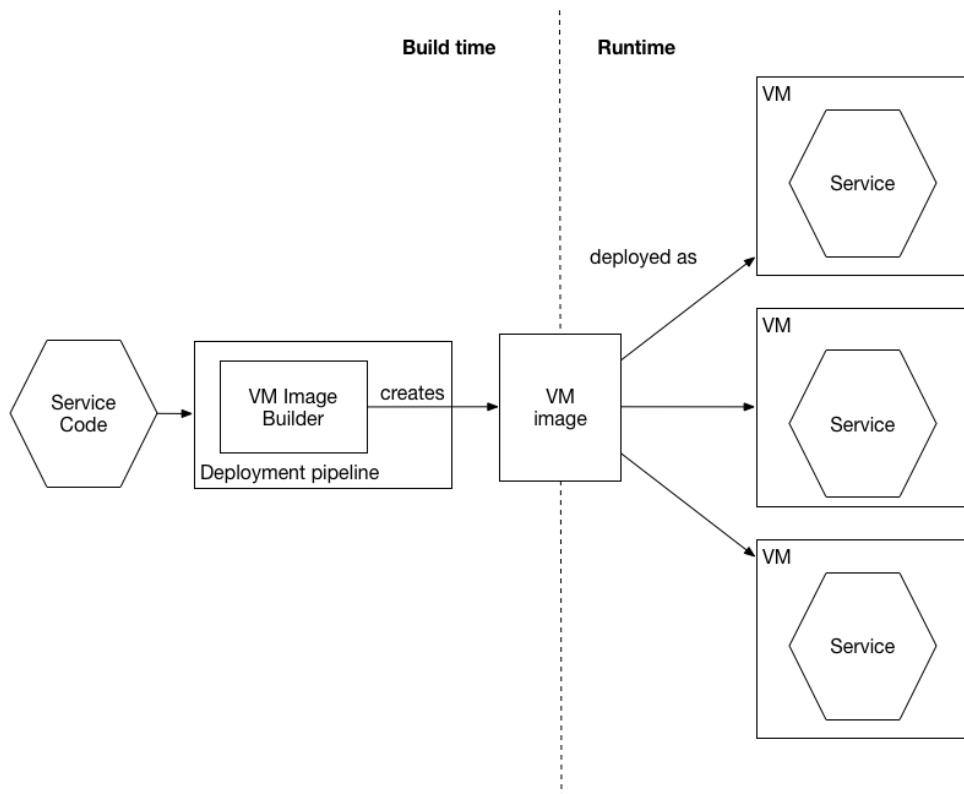
11.2 Deploying services as virtual machines

Once again, let's imagine that you want to deploy the FTGO Restaurant service on AWS EC2. Rather than using deploying the service as an executable or WAR file, a more modern approach is to package it as an Amazon Machine Image (AMI). The build process for the AMI would install the JDK and the service's executable JAR. It would also configure the AMI to run the service as an operating system service using

Linux's init system, such as upstart. In production, the service would be a set of EC2 instances that were instantiated from the AMI. The EC2 instances would typically be managed by an AWS Auto Scaling group, which attempts to ensure that the desired number of healthy instances are always running.

Ideally, the Virtual Machine image is built by the service's deployment pipeline. The deployment pipeline, as 11.6 shows, runs a VM image builder to create a VM image that contains the service's code and whatever software is required to run it. The VM image builder configures the VM image machine to run the application when the VM boots. At runtime, each service instance is a VM that is instantiated from the VM image.

Figure 11.6. The deployment pipeline packages a service as a virtual machine image containing everything required to run the service including the language runtime. At runtime, each service instance is a VM instantiated from that image.



There are a variety of tools that your deployment pipeline can use to build VM images. One early tool for creating EC2 AMIs is Aminator⁶⁶, which was created by Netflix who used it to deploy their video streaming service on AWS. A more modern VM

⁶⁶ github.com/Netflix/aminator

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

image builder is Packer⁶⁷, which unlike Aminator supports a variety of virtualization technologies including EC2, Digital Ocean, Virtual Box and VMware. To use Packer to create an AMI, you write a configuration file that specifies the base image, and a set of provisioners that install software and configure the AMI.

Let's look at the benefits and drawbacks of using this approach.

11.2.1 The benefits of deploying services as VMs

The Service per Virtual Machine pattern has a number of benefits:

- The VM image encapsulates the technology stack
- Isolated service instances
- Leverages mature cloud infrastructure

Let's look at each one.

The VM image encapsulates the technology stack

An important benefit of this pattern is that the VM image contains the service and all of its dependencies. It eliminates the error-prone requirement to correctly install and setup the software that a service needs in order to run. Once a service has been packaged as a virtual machine it becomes a black box that it encapsulates your service's technology stack. The VM image can be deployed anywhere without modification. The API for deploying the service becomes the VM management API. Deployment becomes much simpler and more reliable.

Service instances are isolated

A major benefit of virtual machines is that each service instance runs in complete isolation. That, after all, is one of the main goals of virtual machine technology. Each virtual machine has a fixed amount of CPU and memory and can't steal resources from other services.

Use mature cloud infrastructure

Another benefit of deploying your microservices as virtual machines is that you can leverage mature, highly automated cloud infrastructure. Public clouds, such as AWS, attempt to schedule VMs on physical machines in a way to avoid overloading the machine. They also provide valuable features such as load balancing of traffic across VMs and auto-scaling.

11.2.2 The drawbacks of deploying services as VMs

The Service per VM pattern also has some drawbacks.

- Less efficient resource utilization
- Relatively slow deployments
- System administration overhead

⁶⁷ www.packer.io/

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

Let's look at each drawback in turn.

Less efficient resource utilization

One drawback is less efficient resource utilization. Each service instance has the overhead of an entire virtual machine including its operating system. Moreover, a typical public IaaS virtual machines offers a limited set of VM sizes and so it's likely that the VM will be underutilized. This is less likely to be a problem for Java-based services since they are relatively heavyweight. However, this pattern might be an inefficient way of deploying lightweight NodeJS and GoLang services.

Relatively slow deployments

Another downside of this pattern is that deploying a new version of a service is a relatively slow process. Building a VM image typically takes some number of minutes because of the size of the VM. There are lots of bits to be moved over the network. Also, instantiating a VM from a VM image is time consuming because, once again, of the amount of data that must be moved over the network. The operating system running inside the VM also takes some time to boot. Of course, slow is a relative term. This process, which perhaps just takes minutes, is much faster than the traditional deployment process. However, it's much slower than the more lightweight deployment patterns I'll describe later.

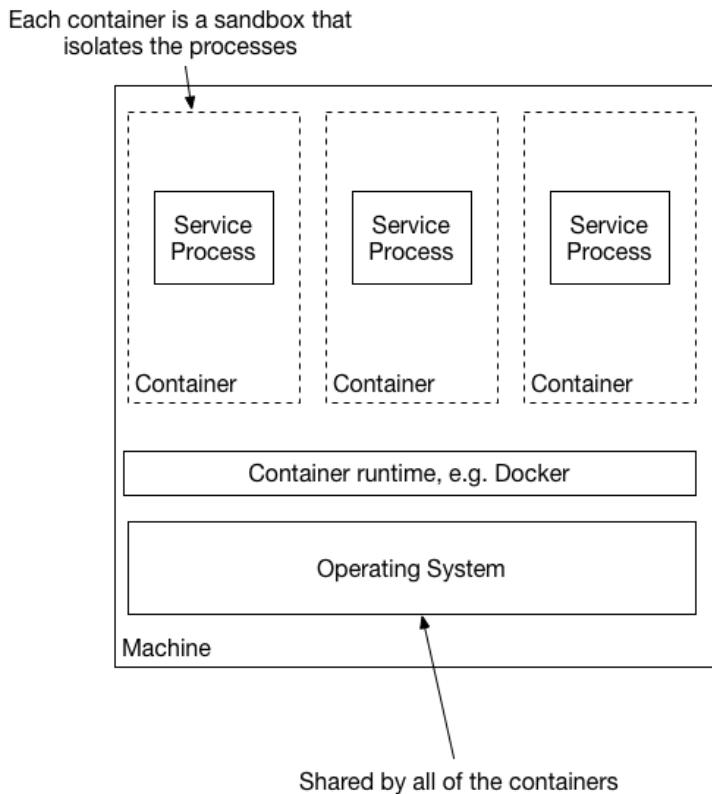
System administration overhead

Another drawback of this approach is that system administration overhead of maintaining the VM images. You are responsible for patching the operation system and runtime. System administration might seem inevitable when deployment software. But later on in section "[Deploying services using serverless deployment](#)" I'll describe serverless deployment, which eliminates this kind of system administration. Let's now look at an alternative way to deploy microservices that is more lightweight yet still has many of the benefits of virtual machines.

11.3 Deploying services as containers

Containers are a more modern and lightweight deployment mechanism. They are an operating-system-level virtualization mechanism. A container, as figure 11.7 shows, consists of usually one, but sometimes multiple processes running in a sandbox, which isolates it from other containers. A container running a Java service, for example, would typically consist of simply the JVM process.

Figure 11.7. A container consists of one or processes running in an isolated sandbox. Multiple containers usually run on a single machine. The containers share the operating system.



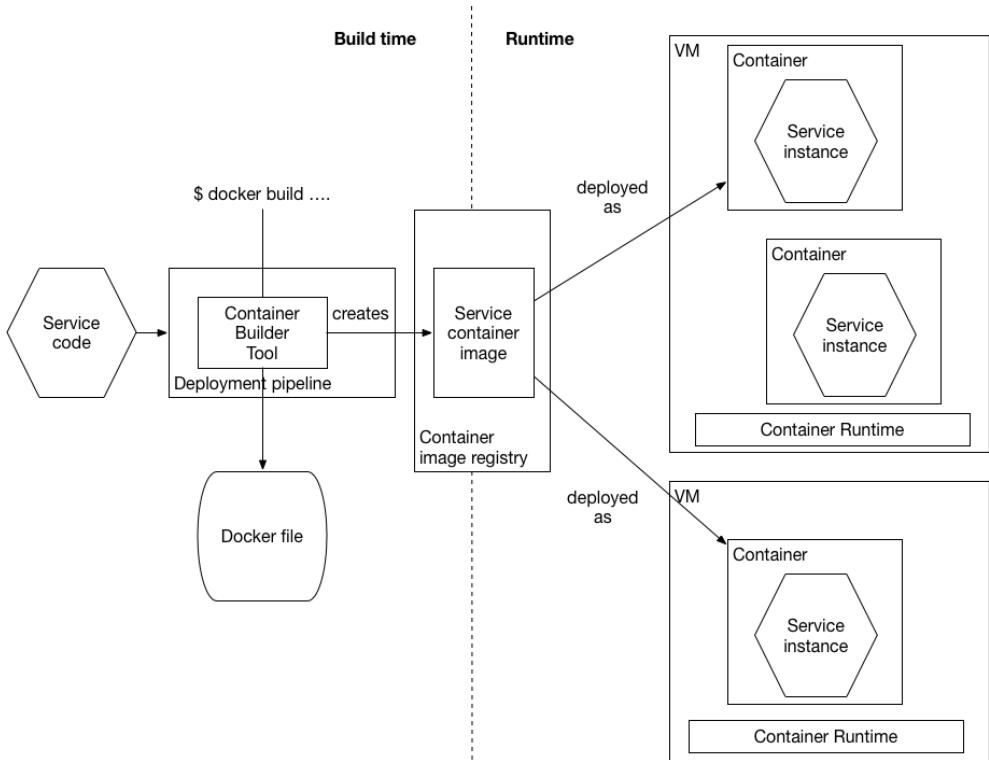
From the perspective of a process running in a container, it is as if it's running on its own machine. It typically has its IP address, which eliminates port conflicts. All Java processes can, for example, listen on port 8080. Each container also has its own root filesystem. The container runtime uses operating system mechanisms to isolate the containers from each other. The most popular example of a container runtime is Docker although there are others such as Solaris Zones.

When you create a container, you can specify its CPU, memory resources, and, depending on the container implementation, perhaps the I/O resources. The container runtime enforces these limits and prevents a container from hogging the resources of its machine. When using a Docker orchestration framework, such as Kubernetes, it's especially important to specify a container's resources. That's because the orchestration framework uses a container's requested resources to select the machine to run the container and thereby ensure that machine are not overloaded.

Figure 11.8 shows the process of deploying a service as a container. At build-time, the deployment pipeline uses a container image building tool, which reads the service's code and a description of the image, to create the container image and stores it in a

registry. At runtime, the container image is pulled from the registry and used to create containers.

Figure 11.8. A service is packaged as a container image, which is stored in a registry. At runtime the service consists of multiple containers instantiated from that image. Containers typically run on virtual machines. A single VM will usually run multiple containers.



Let's take a look at build-time and runtime steps in more detail.

11.3.1 Deploying services using Docker

To deploy a service as a container, you must package it as a container image. A container image is a filesystem image consisting of the application and any software required to run the service. It's often a complete Linux root filesystem, although more lightweight images are also used. For example, to deploy a Spring Boot-based service, you build a container image containing the service's executable JAR and the correct version of the JDK. Similarly, to deploy a Java web application, you would build a container image containing the WAR file, Apache Tomcat and the JDK.

Building a Docker image

The first step of building an image is to create a Dockerfile. A Dockerfile describes how to build a Docker container image. It specifies the base container image; a series

of instructions for installing software and configuring the container; and the shell command to run when the container is created. Listing 11.1 shows the Dockerfile used to build an image for the Restaurant service. It builds a container image containing the service’s executable JAR file. It configures the container to run the `java -jar` command on startup.

Listing 11.1. The Dockerfile used to build the Restaurant service.

```
FROM java:openjdk-8u111-alpine          1
RUN apk --no-cache add curl             2
CMD java ${JAVA_OPTS} -jar ftgo-restaurant-service.jar    3
HEALTHCHECK --start-period=30s --interval=5s CMD curl http://localhost:8080/health
|| exit 1 4
COPY build/libs/ftgo-restaurant-service.jar .           5
```

- ➊ The base image
- ➋ Install curl for use by the health check
- ➌ Configure Docker to run `java -jar ..` when the container is started
- ➍ Configure Docker to invoke the health check endpoint
- ➎ Copies the JAR in Gradle’s build directory into the image

The base image `java:openjdk-8u111-alpine` is a minimal footprint Linux image containing the JDK. The Dockerfile simply copies the service’s JAR into the image and configures the image to execute the JAR on startup. It also configures Docker to periodically invoke the the health check endpoint, which I described in chapter TODO. The `HEALTHCHECK` directive says to invoke the health check endpoint API, which I described in chapter {chapter-prod-ready}, every five seconds after an initial thirty second delay, which gives the service time to start.

Once you have written the Dockerfile, you can then build the image. Listing 11.2 shows the shell commands to build the image for the Restaurant service. The script builds the service’s JAR file and executes the `docker build` command to create the image.

Listing 11.2. The shell commands used to build the container image for the Restaurant service.

```
cd ftgo-restaurant-service
../gradlew assemble
docker build -t ftgo-restaurant-service .
```

- ➊ change to the service’s directory
- ➋ Build the service’s JAR
- ➌ Build the image

The `docker build` command has two arguments. The `-t` argument specifies the name to of the image. The `.` specifies what Docker calls the *context*. The context, which in this example is the current directory, consists of the Dockerfile and the files used to

build the image. The `docker build` uploads the context to the Docker daemon, which builds the image.

Pushing a Docker image to a registry

The final step of the build process is to push the newly built Docker image to what is known as a registry. A Docker registry is the equivalent of a Java Maven repository for Java libraries, or a NodeJS npm registry for node packages. Docker hub is an example of a public Docker registry and is equivalent to Maven Central or NpmJS.org. But for your applications you will probably want to use a private registry provided by services, such as Docker Cloud registry or AWS EC2 Container Registry.

You must use two Docker commands to push an image to a registry. First, you use the `docker tag` command to give the image a name that is prefixed with the hostname and optional port of the registry. The image name is also suffixed with the version, which will be important when you make a new release of the service. For example, if the hostname of the registry is `registry.acme.com` then you would use this command to tag the image:

```
docker tag ftgo-restaurant-service registry.acme.com/ftgo-restaurant-
service:1.0.0.RELEASE
```

Next, you use the `docker push` command to upload that tagged image to the registry.

```
docker push registry.acme.com/ftgo-restaurant-service:1.0.0.RELEASE
```

This command often takes much less time than you might expect. That's because a Docker image has what is known as a layered file system, which enables Docker to only transfer part of the image over the network. An image's operating system, Java runtime and the application are in separate layers. Docker only needs to transfer those layers that don't exist in the destination. As a result, transferring an image over a network is quite fast when Docker only has to move the application's layers, which are a small fraction of the image. Now that we have pushed the image to a registry let's look at how to create a container.

Running a Docker container

Once you have packaged your service as a container image, you can then create one or more containers. The container infrastructure will pull the image from the registry onto a production server. It will then create one or more containers from that image. Each container is an instance of your service.

As you might expect, Docker provides `docker run` command which creates and starts a container. Listing 11.3 shows how to use this command to run the Restaurant service. The `docker run` command has several arguments including container image and a specification of environment variables to set in the runtime container. These are used to pass externalized configuration, such as the database's network location, etc.

Listing 11.3. Using docker run to run a containerized service

```
docker run \
  -d \
  --name ftgo-restaurant-service \
  -p 8082:8080 \
  -e SPRING_DATASOURCE_URL=... -e SPRING_DATASOURCE_USERNAME=... -e
  SPRING_DATASOURCE_PASSWORD=... \
  registry.acme.com/ftgo-restaurant-service:1.0.0.RELEASE
```

- ➊ Run it as a background daemon
- ➋ The name of the container
- ➌ Binds port 8080 of the container to port 8082 of the host machine
- ➍ Environment variables
- ➎ Image to run

The `docker run` command pulls the image from the registry if necessary. It then creates and starts the container, which runs the `java -jar` command specified in the `Dockerfile`.

Using the `docker run` command might seem simple but there are a couple of problems. One problem is that `docker run` is not a reliable way to deploy a service since it creates a container running on a single machine. The Docker engine provides some basic management features such as automatically restarting containers if they crash or the machine is rebooted. However, it doesn't handle machine crashes.

Another problem is that services typically don't exist in isolation. They depend on other services such as databases and message brokers. It would be nice to deploy or undeploy a service and its dependencies as a unit.

A better approach that's especially useful during development is to use Docker Compose. Docker compose is a tool that lets you declaratively define a set of containers using a YAML file and then start and stop those containers as a group. What's more the YAML file is a convenient way to specify numerous externalized configuration properties. To learn more about Docker Compose, I recommend reading *Docker in Action* by Jeff Nickoloff and looking at the `docker-compose.yml` file in the example code.

The problem with Docker Compose, however, is that it is limited to single machine. To deploy services reliably, you must use a Docker orchestration framework, such as Kubernetes, which turns a set of machines into a pool of resources. I'll describe how to use Kubernetes later in section "["Deploying the FTGO application with Kubernetes"](#)". But first, let's review the benefits and drawbacks of using containers.

11.3.2 Benefits of deploying services as containers

Deploying services as containers has several benefits. First, containers have many of the benefits of virtual machines:

- Encapsulation of the technology stack - the API for managing your services

becomes the container API.

- Service instances are isolated
- Service instances's resources are constrained

However, unlike virtual machines, containers are a lightweight technology. Container images are typically very fast to build. For example, on my laptop it takes as little as 5 seconds to package a Spring Boot application as a container image. Moving a container image over the network, such as to and from the container registry, is also relatively fast, primarily because only a subset of an image's layers need to be transferred. Containers also start very quickly since there is no lengthy OS boot process. When a container starts, all that runs is the service.

11.3.3 Drawbacks of deploying services as containers

One significant drawback of containers is that you are responsible for the undifferentiated heavy lifting of administering the container images. You must patch the operating system and runtime. Also, unless you are using a hosted container solution such as Google Container Engine or AWS ECS, then you must administer the container infrastructure and possibly the VM infrastructure that it runs on.

11.4 Deploying the FTGO application with Kubernetes

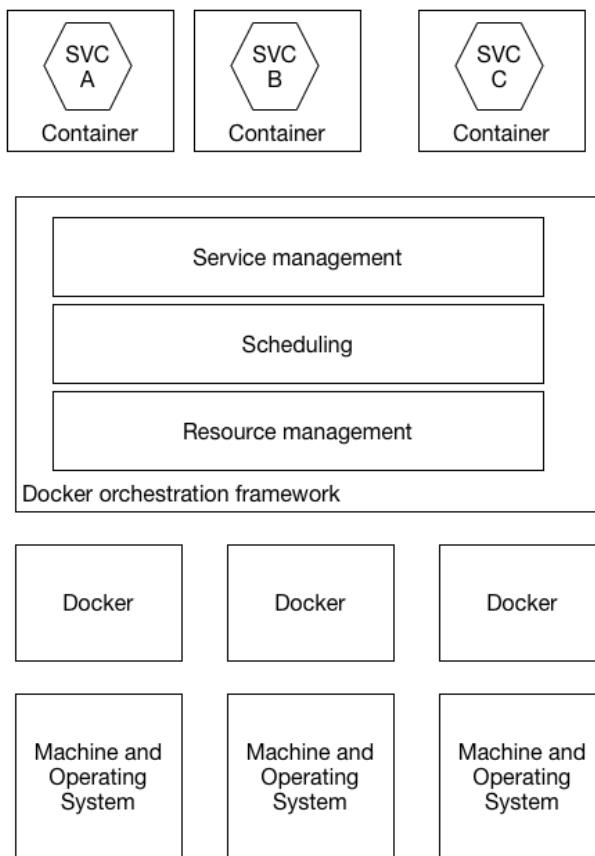
Now we have looked at containers and their tradeoffs, let's look at how to deploy the FTGO application's Restaurant service using Kubernetes. Docker Compose, which I described in section "[Running a Docker container](#)", is great for development and testing. However, to reliably run containerized services in production, you need to use a much more sophisticated container runtime, such as Kubernetes. Kubernetes is a Docker orchestration framework, which is a layer of software on top of Docker that turns a set of machines into a single pool for resources for running services. It endeavors to keep the desired number of instances of each service running at all times even when service instances or machines crash. The agility of containers combined with the sophistication of Kubernetes are a compelling way to deploying services.

In this section, I first give an overview of Kubernetes, its functionality and its architecture. After that, I show how to deploy a service using Kubernetes. Kubernetes is a very complex topic and its out of the scope of this book to cover it exhaustively. I am going to show how to use Kubernetes from the perspective of a developer. For more information, you will need to read one of those books that are dedicated to this topic such as *Kubernetes in Action*.

11.4.1 Overview of Kubernetes

Kubernetes is a Docker orchestration framework. A Docker orchestration framework treats a set of machines running Docker as a pool of resources. You simply tell the Docker orchestration framework to run N instances of your service and it handles the rest. Figure 11.9 shows the architecture of a Docker orchestration framework.

Figure 11.9. A Docker orchestration framework turns a set of machines running Docker into cluster of resources. It assigns containers to machines. And attempts to keep the desired number of healthy containers running at all times.



A Docker orchestration framework has three main functions: resource management, scheduling, and service management.

- Resource management - treats a cluster of machines as a pool of CPU, memory, and storage volumes. It essentially turns the collection of machines into a one single machine.
- Scheduling - selects the machine to run your container. By default, scheduling considers the resource requirements of the container and each node's available resources. It might also implement affinity, which collocates containers on the same node, and anti-affinity, which places containers on different nodes.
- Service management - implements the concept of named, and versioned services, which map directly to services in the microservice architecture. The orchestration framework ensures that the desired number of healthy instances is running at all times. It loads balances requests across them. The orchestration framework

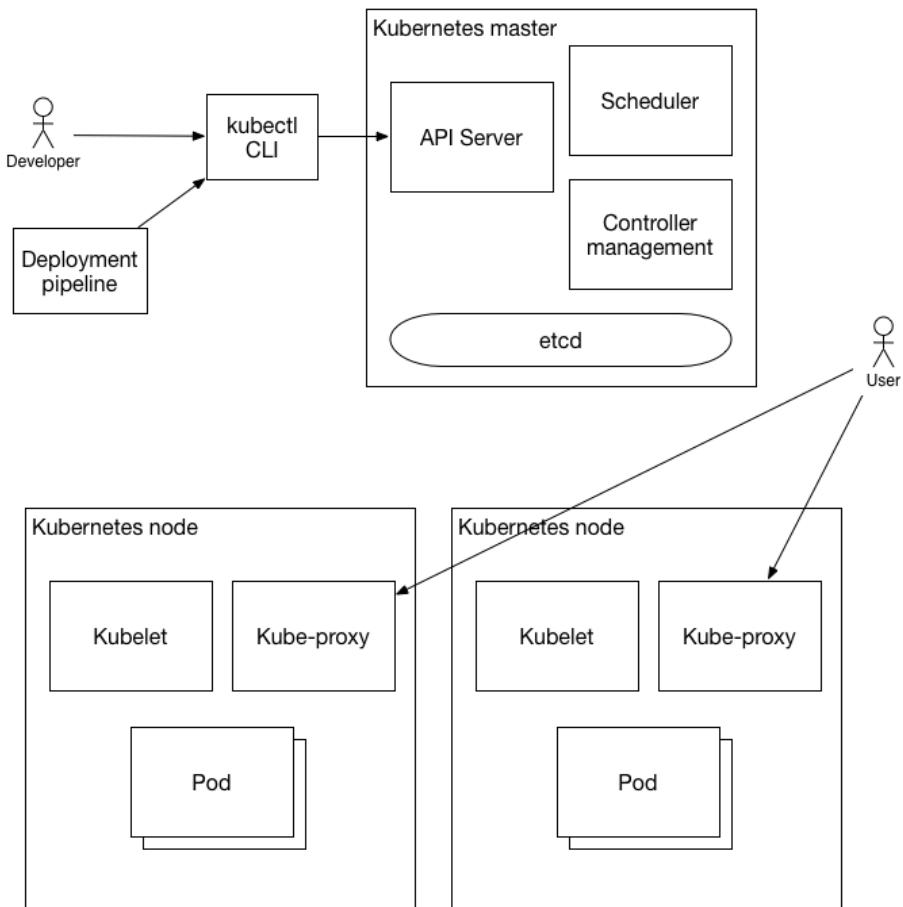
performs rolling upgrades of services and let's you rollback to an old version.

Docker orchestration frameworks are an increasingly popular way to deploy applications. Docker Swarm is part of the Docker engine and so is easy to set up and use. Kubernetes is much more complex to setup and administer but it's much more sophisticated. As of the time of writing, Kubernetes has tremendous momentum with a massive open-source community. Let's take a closer look at Kubernetes.

Kubernetes architecture

Kubernetes runs on a cluster of machines. Figure 11.10 shows the architecture of a Kubernetes cluster. Each machine in a Kubernetes cluster is either a master or a node. A typical cluster has a small number of masters - perhaps just one - and many nodes. A master machine is responsible for managing the cluster. A node is a worker than runs one or more pods. A pod is Kubernetes's unit of a deployment and consists of a set of containers.

Figure 11.10. A Kubernetes cluster consists of a master, which manages the cluster, and nodes, which run the services



A master runs several components including:

- API server - the REST API for deploying and managing services, which is, for example, used by the `kubectl` command-line interface
- etcd - A key-value NoSQL database that stores the cluster data
- scheduler - selects a node to run a pods
- controller manager - runs the controllers, which ensure that the actual state of the cluster matches the intended state. For example, one type of controller known as a replication controller ensures that desired number of instances of a service are running by starting and terminating instances.

A node runs several components including:

- kubelet - creates and manages the pods running on the node
- kube-proxy - manages networking including load balancing across pods

- pods - the application services

Let's now look at key Kubernetes concepts that you will need to master to deploy services on Kubernetes.

Key Kubernetes concepts

As I mentioned in the introduction to this section, Kubernetes is quite complex. However, it's possible to use Kubernetes productivity once you have mastered a few key concepts, a.k.a. objects. Kubernetes defines many types of objects. From a developer's perspective, the most important of which are the following:

- Pod - a pod is the basic unit of deployment in Kubernetes. It consists of one or more containers that share an IP address and storage volumes. The pod for a service instance often consists of a single container, such as a container running the JVM. However, in some scenarios a pod contains one or more sidecar containers, which implement supporting functions. For example, an NGINX server could have a sidecar that periodically does a `git pull` to download the latest version of the website. A pod is ephemeral since either the pod's containers or the node it's running on might crash.
- Deployment - a declarative specification of a pod. A deployment is a controller that ensures that the desired number of instances of the pod (i.e. service instances) are running at all times. It supports versioning with rolling upgrades and rollbacks. Later on in section "[Deploying the Restaurant service on Kubernetes](#)", you will see that each service in a microservice architecture is a Kubernetes deployment.
- Service - provides clients of application service with a static/stable network location. It is a form of infrastructure-provided service discovery, which I described in chapter 3. A service has an IP address and a DNS name, which resolves to that IP address, and load balances TCP and UDP traffic across one or more pods. The IP address and a DNS name are only accessible within the Kubernetes. Later on, I'll describe how to configure services that are accessible from outside the cluster.
- ConfigMap - a ConfigMap is a named collection of name-value pairs that defines the externalized configuration (see chapter {chapter-prod-ready} for an overview of externalized configuration) for one or more application services. The definition of a pod's container can reference a ConfigMap to define the containers environment variables. It can also use a ConfigMap to create configuration files inside the container. You can store sensitive information, such as passwords, in a form of ConfigMap, called a Secret.

Now that we have reviewed the key Kubernetes concepts let's see them in action by looking at how to deploy an application service on Kubernetes.

11.4.2 Deploying the Restaurant service on Kubernetes

As I mentioned earlier, to deploy a service on Kubernetes, you need to define a deployment. The easiest way to create a Kubernetes object, such as a deployment, is by writing a YAML file. Listing 11.4 is a YAML file defining a deployment for

the Restaurant service. This deployment specifies to run two replicas of a pod. The pod has just one container. The container definition specifies the Docker image to run along with other attributes such as the values of environment variables.

Listing 11.4. The YAML definition of the Kubernetes Deployment for the ftgo-restaurant-service

```

apiVersion: extensions/v1beta1
kind: Deployment
 ①
metadata:
  name: ftgo-restaurant-service
 ②
spec:
  replicas: 2
 ③
  template:
    metadata:
      labels:
        svc: ftgo-restaurant-service
 ④
    spec:
      containers:
        - name: ftgo-restaurant-service
          image: ftgo-restaurant-service:1.0.0.RELEASE
          ports:
            - containerPort: 8080
 ⑥
          env:
            - name: SPRING_DATASOURCE_USERNAME
 ⑦
              valueFrom:
                secretKeyRef:
                  name: ftgo-db-secret
                  key: username
            - name: SPRING_DATASOURCE_PASSWORD
 ⑧
              valueFrom:
                secretKeyRef:
                  name: ftgo-db-secret
                  key: password
            - name: SPRING_DATASOURCE_URL
              value: jdbc:mysql://mysql/eventuate
...
 ⑨
      livenessProbe:
        httpGet:
          path: /health
          port: 8080
        initialDelaySeconds: 30
        periodSeconds: 20
      readinessProbe:
        httpGet:
          path: /health
          port: 8080
        initialDelaySeconds: 30
        periodSeconds: 20

```

- ① Specifies that this is an object of type Deployment
- ② The name of the deployment
- ③ Number of pod replicas
- ④ Gives each pod a label called `svc` whose value is `ftgo-restaurant-service`
- ⑤ The specification of the pod, which defines just one container

- ⑥ The container's port
- ⑦ The container's environment variables, which are the service's externalized configuration. They are read by Spring Boot and made available as properties in the application context.
- ⑧ Sensitive values that are retrieved from the Kubernetes Secret called `ftgo-db-secret`
- ⑨ Configure Kubernetes to invoke the health check endpoint

This deployment definition configures Kubernetes to invoke the Restaurant Service's health check endpoint. As I described in chapter {chapter-prod-ready}, a health check endpoint enables the Kubernetes to determine the health of the service instance. Kubernetes implements two different checks. The first check is the `readinessProbe`, which it uses to determine whether it should route traffic to a service instance. In this example, Kubernetes invokes the `/health` HTTP endpoint every twenty seconds after an initialize thirty seconds delay, which gives it a chance to initialize. If some number (default 1) of consecutive `readinessProbes` succeeds then Kubernetes considers the service to be ready, whereas if some number (default 3) of consecutive `readinessProbes` fail in a row it is considered not to be ready. Kubernetes will only route traffic to the service instance when the `readinessProbe` indicates that it is ready.

The second health check is the `livenessProbe`. It is configured the same way as the `readinessProbe`. However, rather than determining whether traffic should be routed to a service instance, the `livenessProbe` determines whether Kubernetes should terminate and restart the service instance. If some number (default 3) of consecutive `livenessProbes` fail in a row then Kubernetes will terminate and restart the service.

Once you have written the YAML file, you can create or update the deployment by using the `kubectl apply` command:

```
kubectl apply -f ftgo-restaurant-service/src/deployment/kubernetes/ftgo-restaurant-service.yml
```

This command makes a request to the Kubernetes API server that results in the creation of the deployment and the pods.

Of course, in order to create this deployment, you must first create the Kubernetes Secret called `ftgo-db-secret`. One quick and insecure way to do this is as follows:

```
echo -n mysqluser > ./deployment/kubernetes/dbuser.txt
echo -n mysqlpw > ./deployment/kubernetes/dbpassword.txt
kubectl create secret generic ftgo-db-secret \
--from-file=username=./deployment/kubernetes/dbuser.txt \
--from-file=password=./deployment/kubernetes/dbpassword.txt
```

This script creates files containing the database user id and password and then creates a secret from those. Please see the Kubernetes documentation⁶⁸ for more secure ways to create secrets.

⁶⁸ kubernetes.io/docs/concepts/configuration/secret/#creating-your-own-secrets

Creating a Kubernetes service

At this point the pods are running and the Kubernetes deployment will do its best to keep them running. The problem is, however, that the pods have dynamically assigned IP addresses and as such, aren't that useful to a client that wants to make an HTTP request. As I described in chapter 3, the solution to use a service discovery mechanism. One approach is to use a client-side discovery mechanism and install a service registry, such as Netflix OSS Eureka. Fortunately, we can avoid doing this by using the service discovery mechanism that is built into Kubernetes and define a Kubernetes service.

A service is a Kubernetes object that provides the client's of one or more pods with a stable endpoint. It has an IP address and a DNS name that resolves that IP address. The service load balances traffic to that IP address across the pods. Listing 11.5 shows the Kubernetes Service for the `Restaurant` service. This service routes traffic to [ftgo-restaurant-service:8080](#) to the pods defined by the deployment shown in listing 11.5.

Listing 11.5. The YAML definition of the Kubernetes service for the ftgo-restaurant-service

```
apiVersion: v1
kind: Service
metadata:
  name: ftgo-restaurant-service
spec:
  ports:
    - port: 8080
      targetPort: 8080
    selector:
      svc: ftgo-restaurant-service
---

```

- ① The name of the service, which is also the DNS name
- ② The exposed port
- ③ The container port to route traffic to
- ④ selects the containers to route traffic to

The key part of the service definition is the `selector`, which selects the target pods. It selects those pods that have a label named `svc` with the value `ftgo-restaurant-service`. If you look closely, you will see that the container defined in listing 11.4 has such a label.

Once you have written the YAML file you can create the service using this command:

```
kubectl apply -f ftgo-restaurant-service-service.yml
```

Now that we have created the Kubernetes service, any clients of the `Restaurant` service that are running inside the Kubernetes cluster can access its REST API via [ftgo-restaurant-service:8080](#).

11.4.3 Zero-downtime deployments

Let's imagine that you have updated the `Restaurant` service and you want to deploy those changes into production. Updating a running service is a simple three step process when using Kubernetes:

1. Build a new container image and push it to the registry using the same process I described earlier. The only difference is that the image will be tagged with a different version tag, e.g. `ftgo-restaurant-service:1.1.0.RELEASE`
2. Edit the YAML file for the service's deployment so that it references the new image
3. Update the deployment using the `kubectl apply -f` command

Kubernetes will then perform a rolling upgrade of the pods. It will incrementally create pods running version `1.1.0.RELEASE` and terminate the pods running version `1.0.0.RELEASE`. What's great about how Kubernetes does this, is that it doesn't terminate old pods until their replacements are ready to handle requests. Later on in section XYZ, I describe the health check mechanism, which determines whether a pod is ready. As a result, there will always be pods available to handle requests. Eventually, assuming that the new pods start successfully, all of the deployment's pods will be running the new version.

But what if there is a problem and the version `1.1.0.RELEASE` pods don't start? Perhaps there is a bug, such as a mispelt container name or a missing environment variable for a new configuration property. If the pods fail to start then the deployment will become stuck. At this point, you have two options. One option is to fix the YAML file and rerun `kubectl apply -f` to update the deployment. The other option is to rollback the deployment.

A deployment maintains the history of what are termed 'rollouts'. Each time you update the deployment, it creates a new rollout. As a result, you can easily rollback a deployment to a previous version by executing the following command:

```
kubectl rollout undo deployment ftgo-restaurant-service
```

Kubernetes will then replace the pods running version `1.1.0.RELEASE` version with pods running the older version `1.0.0.RELEASE`. Now that we have looked at how to deploy and upgrade applications services, let's take a look at how to make them accessible from outside of the Kubernetes cluster.

11.4.4 Deploying the API gateway

The Kubernetes service for the `Restaurant` service, which is shown in listing 11.5, is only accessible from within the cluster. That is not a problem for the `Restaurant Service` but what about the `API Gateway`? Its role is to route traffic from the outside world to the service. It therefore needs to be accessible from outside the cluster. Fortunately, a Kubernetes service supports this use case as well. The service that we looked at earlier is a `ClusterIP` service, which is the default. There are, however, two other types of services: `NodePort` and `LoadBalancer`. Let's take a look at how they

work.

A NodePort service is accessible via cluster-wide port on all of the nodes in the cluster. Any traffic to that port on any cluster node is load balanced to the backend pods. You must select an available port in the range 30000-32767. For example, listing 11.6 shows a service that routes traffic to port 30000 to the Consumer Service.

Listing 11.6. The YAML definition of a NodePort service that routes traffic to port 8082 to the Consumer service

```
apiVersion: v1
kind: Service
metadata:
  name: ftgo-api-gateway
spec:
  type: NodePort          ①
  ports:
    - nodePort: 30000      ②
      port: 80
      targetPort: 8080
  selector:
    svc: ftgo-api-gateway
---
```

- ① Specify a type of NodePort
- ② The cluster-wide port

The API Gateway is within the cluster using the URL [ftgo-api-gateway](#) and outside the URL [`<node-ip-address>:3000/`](#), where `node-ip-address` is the IP address of one of the nodes. After configuring a NodePort service, you can, for example, configure an AWS Elastic Load Balancer (ELB) to load balance requests from the Internet across the nodes. A key benefit of this approach is that the ELB is entirely under your control. You have complete flexibility when configuring it.

A NodePort type service is not the only option, however. You can also use a LoadBalancer service, which automatically configures a cloud-specific load balancer. The load balancer will be an ELB, if Kubernetes is running on AWS. One benefit of this type of service is that you no longer have to configure your own load balancer. The drawback, however, is that while Kubernetes does give a few options for configuring the ELB, such as the SSL certificate, you have a lot less control over its configuration.

11.5 Deploying services using serverless deployment

The Language-specific packaging (section “[Deploying services as language-specific packages](#)”), Service per VM (section “[Deploying services as virtual machines](#)”), and Service per container (section “[Deploying services as containers](#)”) patterns are all quite different. However, they share some common characteristics. The first is that with all three patterns you must pre-provision some computing resources: either physical machines, virtual machines or containers. Some deployment platforms implement auto-

scaling, which dynamically adjusts the number of VMs or containers based on the load. However, you will always need to pay for some VMs or containers, even if they are idle.

Another common characteristic is that you are responsible for system administration. If you are running any kind of machine, then you must patch the operating system. In the case of physical machines, this also includes racking and stacking. You are also responsible for administering the language runtime. This is an example of what Amazon called 'undifferentiated heavy lifting'. Since the early days of computing, system administration has just been one of those things that you need to do. It turns out, however, that there is a solution: serverless.

11.5.1 Overview of serverless deployment with AWS Lambda

At AWS Re:Invent 2014, Werner Vogels, the CTO of Amazon introduced AWS Lambda with the amazing phrase "magic happens at the intersection of functions, events, and data". As this phrase suggests, AWS Lambda was initially for deploying event-driven services. It is "magic" because, as I describe below, AWS Lambda is an example of server-less deployment technology.

AWS Lambda supports Java, NodeJS, C# and Python. A lambda function is a stateless service. It typically handles requests by invoking AWS services. For example, a lambda function that is invoked when an image is uploaded to an S3 bucket could insert an item into a DynamoDB IMAGES table and publish a message to Kinesis to trigger image processing. A lambda function can also invoke third party web services.

To deploy microservice you package your application as a ZIP file or JAR file, and upload it to AWS Lambda and specify the name of the function to invoke to handle a request (a.k.a an event). AWS Lambda automatically runs enough instances of your microservice to handle incoming requests. You are simply billed for each request based on the time taken and the memory consumed. Of course, the devil is in the details and later you will see that AWS Lambda has limitations. But the notion that neither you as a developer nor anyone in your organization need worry about any aspect of servers, virtual machines, or containers is incredibly powerful.

11.5.2 Developing a lambda function

Unlike when using the other three patterns, you must use a different programming model for your lambda functions. A lambda function's code and the packaging depends on the programming language. A Java lambda function is a class that implements the generic interface `RequestHandler`, which is defined by the AWS Lambda Java core library and shown listing 11.7. This interface takes two type parameters, `I`, which is the input type, and `O`, which is the output type. The type of `I` and `O` depend on the specific kind of request that the lambda handles.

Listing 11.7. A Java lambda function is a class that implements the RequestHandler interface.

```
public interface RequestHandler<I, O> {
    public O handleRequest(I input, Context context);
}
```

The RequestHandler interface defines a single handleRequest() method. This method has two parameters, an input object and a context, which provides access to the lambda execution environment, such as the request id. The handleRequest() method returns an output object. For lambda functions that handle HTTP requests that are proxied by an AWS API Gateway, I and O are APIGatewayProxyRequestEvent and APIGatewayProxyResponseEvent respectively. As you will see below, the handler functions are quite similar to old-style Java EE servlets.

A Java lambda is packaged as either a ZIP file or a JAR file. A JAR file is a uber JAR (a.k.a. fat JAR) created by, for example, the Maven shade plugin. A ZIP file has the classes in the root directory and JAR dependencies in the lib directory. Later on I'll show how a Gradle project can create a ZIP file. But first, let's look at the different ways of invoking lambda function.

11.5.3 Invoking lambda functions

There are four ways to invoke a lambda function.

- HTTP requests
- Events generated by AWS services
- Scheduled invocations
- Directly using an API call

Let's look at each one.

Handling HTTP requests

One way to invoke a lambda function is to configure an AWS API Gateway to route HTTP requests to your lambda. The API Gateway exposes your lambda function as an HTTPS endpoint. It functions as an HTTP proxy and invokes the lambda function with an HTTP request object and expects the lambda function to return an HTTP response object. By using the API Gateway with AWS Lambda you can, for example, deploy RESTful services as lambda functions.

Handling events generated by AWS services

The second way to invoke a lambda function, is to configure your lambda function to handle events generated by an AWS service. Examples of events that can trigger a lambda function include the following:

- an object being created in a S3 bucket
- an item is created, updated or deleted in a DynamoDB table

- a message is available to read from a Kinesis stream
- an email being received via the Simple email service.

Because of this integration with other AWS services, AWS lambda is useful for a wide range of tasks.

Defining scheduled lambda functions

Another way to invoke a lambda function is to use a Linux cron-like schedule. You can configure your lambda function to be invoked periodically, e.g. every minute, 3 hours or 7 days. Alternatively, you can use a cron expression to specify when AWS should invoke your lambda. Cron expressions give you tremendous flexibility. You can, for example, configure a lambda to be invoked at 2.15pm Monday through Friday.

Invoking a lambda function using a web service request

The fourth way to invoke a lambda function is for your application to invoke it using a web service request. The web service request specifies the name of the lambda function and the input event data. Your application can invoke a lambda function synchronously or asynchronously. If your application invokes the lambda function synchronously, the web service's HTTP response contains the response of the lambda function. Otherwise, if it invokes the lambda function asynchronously, the web service response simply indicates whether the execution of the lambda was successfully initiated.

11.5.4 Benefits of using lambda functions

Deploying services using lambda functions has several benefits:

- eliminates many system administration tasks - you are no longer responsible for low-level system administration. There are no operating systems or runtimes to patch. As a result, you can focus on developing your application.
- elasticity - AWS Lambda runs as many instances of your application as are needed to handle the load. You don't have the challenge of predicting needed capacity and the risk of under-provisioning or over-provisioning VMs or containers.
- usage-based pricing - unlike a typical IaaS cloud, which charges by the minute or hour for a VM or Container even when it's idle, AWS Lambda only charges you for the resources that are consumed while processing each request.

11.5.5 Drawbacks of using lambda functions

As you can see, AWS Lambda is an extremely convenient way to deploy services but there are some significant drawbacks and limitations:

- long-tail latency - because AWS Lambda dynamically runs your code, some requests have high latency because of the time it takes for AWS to provision an instance of your application and for the application to start. This is particularly challenging when running Java-based services because they typically take at least several seconds to start. Consequently, AWS Lambda might not be suited for latency sensitive services.

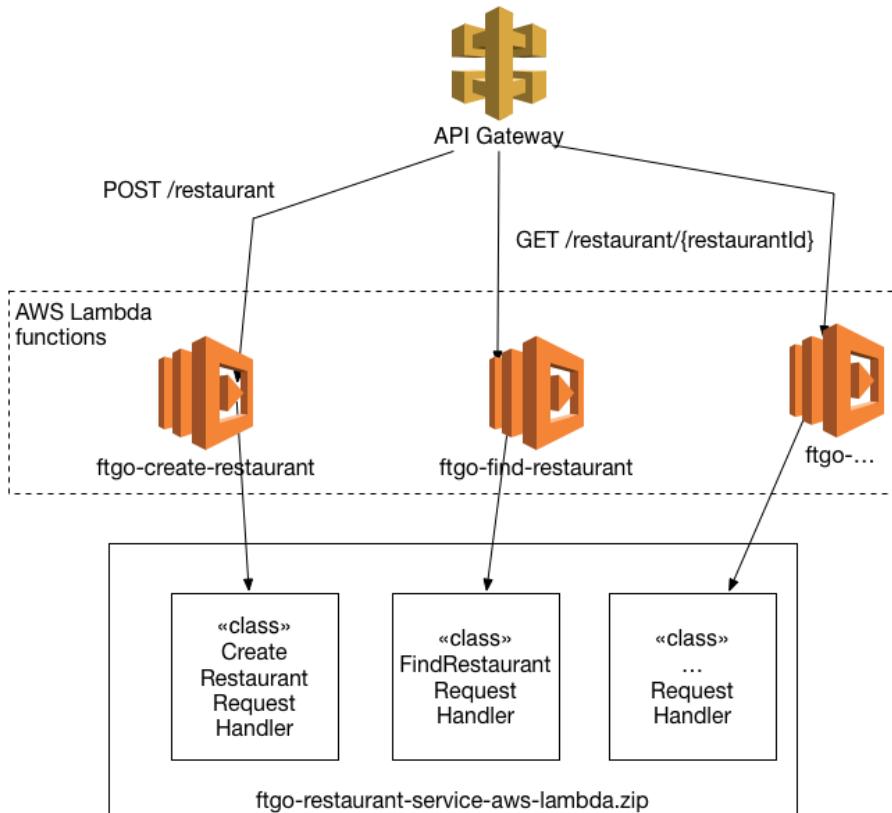
- limited event/request-based programming model - AWS lambda is not intended to be used to deploy long running services, such as a service that consumes messages from a third party message broker

Because of these drawbacks and limitations, AWS Lambda is not a good fit for all services. However, when choosing a deployment pattern, I recommend first evaluating whether serverless deployment supports your service's requirements before considering alternatives.

11.6 Deploying a *RESTful* service using AWS Lambda and AWS Gateway

Let's take a look at how to deploy the Restaurant Service using AWS Lambda. It's a service that has a REST API for creating and managing restaurants. It doesn't have long-lived connections to Apache Kafka, for example, and so it's a good fit for AWS lambda. Figure 11.12 shows the deployment architecture for this service. The service consists of several lambda functions, one for each REST endpoint. An AWS API Gateway is responsible for routing HTTP requests to the lambda functions.

Figure 11.11. Deploying the Restaurant service as AWS Lambda functions. The AWS API Gateway routes HTTP requests to the AWS lambda functions, which are implemented by request handler classes defined by the Restaurant Service



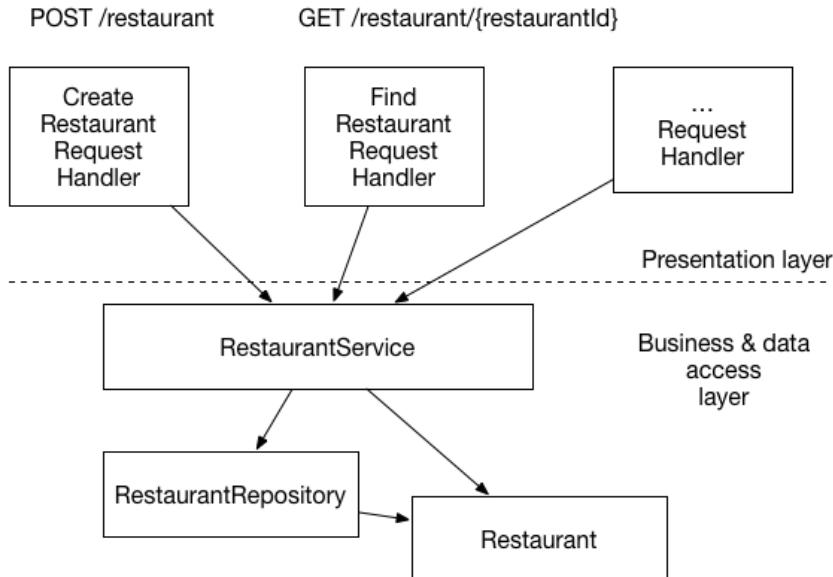
Each lambda function has a request handler class. The `ftgo-create-restaurant` lambda function invokes the `CreateRestaurantRequestHandler` class, and the `ftgo-find-restaurant` lambda function invokes `FindRestaurantRequestHandler`. Since these request handler classes implement closely related aspects of the same service they are packaged together in the same ZIP file called `restaurant-service-aws-lambda.zip`. Let's look at the design of the service including those handler classes.

11.6.1 *The design of the AWS Lambda version of the Restaurant Service*

The architecture of the service, which is shown in figure 11.12, is quite similar to that of a 'traditional' service. The main difference is that Spring MVC controllers have been replaced by AWS Lambda request handler classes. The rest of the business logic is unchanged.

Figure 11.12. The design of the AWS Lambda-based Restaurant Service. The presentation layer consists of request handler classes, which implement the lambda functions. They invoke

the business tier, which is written in a traditional style consisting of a service class, an entity and a repository.

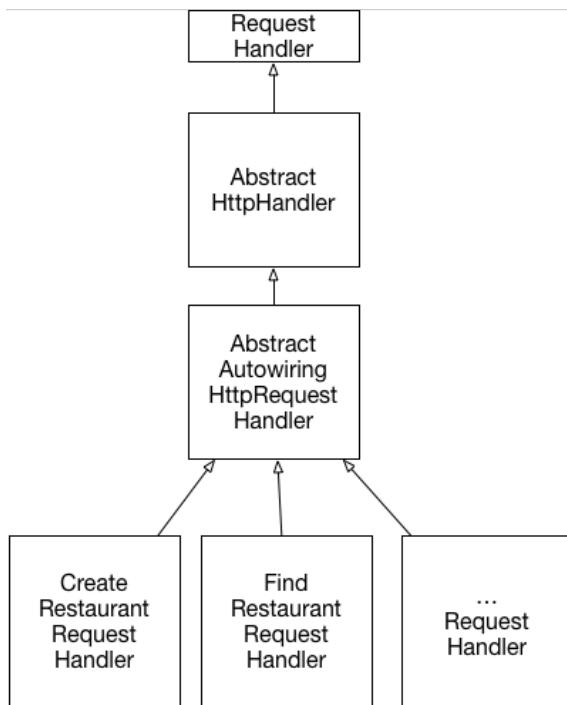


The service consists of a presentation tier consisting of the request handlers, which are invoked by AWS Lambda to handle the HTTP requests, and a traditional business tier. The business tier consists of the `RestaurantService`, `Restaurant` JPA entity, and `RestaurantRepository`, which encapsulates the database. Let's take a look at the `FindRestaurantRequestHandler` class.

The FindRestaurantRequestHandler class

The `FindRestaurantRequestHandler` class implements the `GET /restaurant/{restaurantId}` endpoint. This class along with the other request handler classes, are the leaves of the class hierarchy shown in figure 11.13. The root of the hierarchy is the `RequestHandler`, which is part of the AWS SDK. Its abstract subclasses handle errors and inject dependencies.

Figure 11.13. The design of the request handler classes. The abstract superclasses implement dependency injection and error handling.



The `AbstractHttpHandler` class is the abstract base class for HTTP request handlers. It catches unhandled exceptions thrown during request handling and returns a `500 - internal server error` response. The `AbstractAutowiringHttpRequestHandler` class implements dependency injection for request handlers. I'll describe these abstract superclasses shortly, but first let's look at the code for `FindRestaurantRequestHandler`.

Listing 11.8 shows the code the `FindRestaurantRequestHandler` class. The `FindRestaurantRequestHandler` class has a `handleHttpRequest()` method, which takes an `APIGatewayProxyRequestEvent` representing an HTTP request as parameter. It invokes the `RestaurantService` to find the restaurant and returns an `APIGatewayProxyResponseEvent` describing the HTTP response.

Listing 11.8. The `FindRestaurantRequestHandler` class, which implements the GET `/restaurant/{restaurantId}` REST endpoint

```

public class FindRestaurantRequestHandler extends
AbstractAutowiringHttpRequestHandler {

    @Autowired
    private RestaurantService restaurantService;
  
```

1

```

@Override
protected Class<?> getApplicationContextClass() {
    return CreateRestaurantRequestHandler.class;
}

@Override
protected APIGatewayProxyResponseEvent
handleHttpRequest(APIGatewayProxyRequestEvent request, Context context) {
    long restaurantId;
    try {
        restaurantId =
Long.parseLong(request.getPathParameters().get("restaurantId"));
    } catch (NumberFormatException e) {
        return makeBadRequestResponse(context);
    }

    Optional<Restaurant> possibleRestaurant =
restaurantService.findById(restaurantId);

    return possibleRestaurant
        .map(this::makeGetRestaurantResponse)
        .orElseGet(() -> makeRestaurantNotFoundResponse(context,
restaurantId));
}

private APIGatewayProxyResponseEvent makeBadRequestResponse(Context context) {
... }

private APIGatewayProxyResponseEvent makeRestaurantNotFoundResponse(Context
context, long restaurantId) { ... }

private APIGatewayProxyResponseEvent makeGetRestaurantResponse(Restaurant
restaurant) { ... }
}

```

- ➊ The Spring Java configuration class to use for the application context
- ➋ Return a 400 - bad request response if the restaurantId is missing or invalid
- ➌ Return either the restaurant or a 404 - not found response

As you can see, its quite similar to a servlet, except that instead of a `service()` method, which takes a `HttpServletRequest` and returns `HttpServletResponse`, it has a `handleRequest()`, which takes a `APIGatewayProxyRequestEvent` and returns `APIGatewayProxyResponseEvent`. Let's now take a look at its superclass, which implements dependency injection.

Dependency injection using the `AbstractAutowiringHttpRequestHandler` class

An AWS Lambda function is neither a web application nor an application with a `main()` method. But it would be shame to not be able to use the features of Spring Boot that we have been accustomed to. The `AbstractAutowiringHttpRequestHandler` class, which is shown in listing 11.9, implements dependency injection for request handlers. It creates an `ApplicationContext` using `SpringApplication.run()` and autowires dependencies, prior to handling the first request. Subclasses such as

`FindRestaurantRequestHandler` must implement the `getApplicationContextClass()` method.

Listing 11.9. The `AbstractAutowiringHttpRequestHandler`, which is an abstract RequestHandler that implements dependency injection.

```
public abstract class AbstractAutowiringHttpRequestHandler extends AbstractHttpHandler {

    private static ConfigurableApplicationContext ctx;
    private ReentrantReadWriteLock ctxLock = new ReentrantReadWriteLock();
    private boolean autowired = false;

    protected synchronized ApplicationContext getAppCtx() { (1)
        ctxLock.writeLock().lock();
        try {
            if (ctx == null) {
                ctx = SpringApplication.run(getApplicationContextClass());
            }
            return ctx;
        } finally {
            ctxLock.writeLock().unlock();
        }
    }

    @Override
    protected void beforeHandling(APIGatewayProxyRequestEvent request, Context context) { (2)
        super.beforeHandling(request, context);
        if (!autowired) {
            getAppCtx().getAutowireCapableBeanFactory().autowireBean(this);
            autowired = true;
        }
    }

    protected abstract Class<?> getApplicationContextClass();
}
```

- 1 Create the Spring Boot application context just once
- 2 Inject dependencies into the request handler using autowiring before handling the first request
- 3 Returns the `@Configuration` class used to create the `ApplicationContext`

This class overrides the `beforeHandling()` method defined by `AbstractHttpHandler`. Its `beforeHandling()` method injects dependencies using autowiring before handling the first request.

The `AbstractHttpHandler` class

The request handlers for the Restaurant Service ultimately extend `AbstractHttpHandler`, which is shown in listing 11.10. This class implements `RequestHandler<APIGatewayProxyRequestEvent and APIGatewayProxyResponseEvent>`. Its key responsibility is to catch exceptions thrown when handling a request and throw a 500 error code.

Listing 11.10. The AbstractHttpHandler class, which is an abstract RequestHandler that catches uncaught exceptions and returns a 500 HTTP response

```
public abstract class AbstractHttpHandler implements
RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    private Logger log = LoggerFactory.getLogger(this.getClass());

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
input, Context context) {
        log.debug("Got request: {}", input);
        try {
            beforeHandling(input, context);
            return handleHttpRequest(input, context);
        } catch (Exception e) {
            log.error("Error handling request id: {}", context.getAwsRequestId(), e);
            return buildErrorResponse(new AwsLambdaError(
                "Internal Server Error",
                "500",
                context.getAwsRequestId(),
                "Error handling request: " + context.getAwsRequestId() + " " +
input.toString()));
        }
    }

    protected void beforeHandling(APIGatewayProxyRequestEvent request, Context
context) {
        // do nothing
    }

    protected abstract APIGatewayProxyResponseEvent
handleRequest(APIGatewayProxyRequestEvent request, Context context);
}
```

11.6.2 Packaging the service as ZIP file

Before the service can be deployed, we must package it as a ZIP file. We can easily build the ZIP file using the following Gradle task:

```
task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtime
    }
}
```

This task builds a ZIP with the classes and resources at the top-level and the JAR dependencies in the lib directory. Now that we have built the ZIP file, let's look at how to deploy the lambda function.

11.6.3 Deploying lambda functions using the Serverless framework

Using the tools provided by AWS to deploy lambda functions and configure the API

gateway is quite tedious. Fortunately, the Serverless open-source project makes using lambda functions a lot easier. When using serverless, you write a simple `serverless.yml` file that defines your lambda functions and their RESTful endpoints. Serverless then deploys the lambda functions, and creates and configures an API Gateway that routes requests to them.

Listing 11.11 is an excerpt of the `serverless.yml` that deploys the Restaurant Service as a lambda.

Listing 11.11. The `serverless.yml`, which deploys the Restaurant service

```
service: ftgo-application-lambda

provider:
  name: aws
  runtime: java8
  timeout: 35
  region: ${env:AWS_REGION}
  stage: dev
  environment:
    SPRING_DATASOURCE_DRIVER_CLASS_NAME: com.mysql.jdbc.Driver
    SPRING_DATASOURCE_URL: ...
    SPRING_DATASOURCE_USERNAME: ...
    SPRING_DATASOURCE_PASSWORD: ...

package:
  artifact: ftgo-restaurant-service-aws-lambda/build/distributions/ftgo-restaurant-
service-aws-lambda.zip

functions:
  create-restaurant:
    handler:
      net.chrisrichardson.ftgo.restaurantservice.lambda.CreateRestaurantRequestHandler
      events:
        - http:
            path: restaurants
            method: post
  find-restaurant:
    handler:
      net.chrisrichardson.ftgo.restaurantservice.lambda.FindRestaurantRequestHandler
      events:
        - http:
            path: restaurants/{restaurantId}
            method: get
```

- ➊ tell serverless to deploy on AWS
- ➋ Supply the service's externalized configuration via environment variables
- ➌ The ZIP file containing the lambda functions
- ➍ Lambda function definitions consisting of the handler function and HTTP endpoint

You can then use the `serverless deploy` command, which reads the `serverless.yml` file, deploys the lambda functions and configures the AWS API

Gateway. After a short wait, your service will be accessible via the API gateway's endpoint URL. AWS Lambda will provision as many instances of each Restaurant Service lambda function that are needed to support the load. If you change the code, you can easily update the lambda by rebuilding the ZIP file and rerunning `serverless deploy`. No servers involved!

The evolution of infrastructure is remarkable. Not that long ago, we manually deployed applications on physical machines. Today, highly, automated public clouds provide a range of virtual deployment options. One option is to deploy services as virtual machines. Or better yet, we can package services as containers and deploy them using sophisticated Docker orchestration frameworks such as Kubernetes. Sometimes, we even avoid thinking about infrastructure entirely and deploy services as lightweight, ephemeral lambda functions.

11.7 Summary

- You should choose the most lightweight deployment pattern that supports your service's requirements. Evaluate the options in the following order: serverless, containers, virtual machines, and language-specific packages
- A serverless deployment is not a good fit for every service, because of long tail latencies and the requirement to use an event/request-based programming model. When it is a good fit, however, serverless deployment is an extremely compelling option since it eliminates the need to administer operating systems and runtimes, and provides automated elastic provisioning, and request-based pricing.
- Docker containers, which are a lightweight, OS-level virtualization technology, are more flexible than serverless deployment and have more predictable latency. It's best to use a Docker orchestration framework, such as Kubernetes, which manages containers on a cluster of machines. The drawback of using containers that you must administer the operating systems, and runtimes and most likely the Docker orchestration framework and the VMs that it runs on.
- The third deployment option is to package your service as a virtual machine. On the one hand, virtual machines are a heavyweight deployment option and so deployment is slower and it will most likely use more resources than the option. But on the other hand, modern clouds such as Amazon EC2 are highly automated and provide a rich set of features. Consequently, it might sometimes be easier to deploy a small, simple application using virtual machines than to setup a Docker orchestration framework.
- Deploying your services as language-specific packages should be considered the pattern of last resort. I recommend avoiding this option unless you only have a small number of services. For example, as I describe in chapter 11, when you starting on your journey to microservices you will probably deploy the services using the same mechanism as you use for your monolithic application, which is most likely this option. You should only consider setting up a sophisticated deployment infrastructure, such as Kubernetes, once you have developed some services.

12

Refactoring to microservices

This chapter covers:

- When to migrate a monolithic application to a microservice architecture
- Why it's essential to use an incremental approach when refactoring a monolithic application to microservices
- Implementing new features as services
- Extracting services from the monolith
- Integrating a service and the monolith

I hope that this book has given you a good understanding of the microservice architecture, its benefits and drawbacks, and when to use it. There is, however, a fairly good chance you are working on a large, complex monolithic application. Your daily experience of developing and deploying your application is slow and painful. Microservices, which appear like a good fit for your application, seem like distant nirvana. Like Mary and the rest of the FTGO development team, you are wondering how on earth you can adopt the microservice architecture?

Fortunately, there are strategies that you can use to escape from monolithic hell without having to rewrite your application from scratch. You incrementally convert your monolith into microservices by developing what's known as a strangler application. The idea of a strangler application comes from strangler vines, which grow in rain forests by enveloping and, sometimes killing, trees. A strangler application is a new application consisting of microservices that you develop by implementing new functionality as services and extracting services from the monolith. Over time, as the strangler application implements more and more functionality, it shrinks and ultimately

kills the monolith. An important benefit of developing a strangler application is that, unlike a big bang rewrite, it delivers value to the business early and often.

I begin this chapter by describing the motivations for refactoring a monolith to a microservice architecture. I then describe how to develop the strangler application by implementing new functionality as services and extracting services from the monolith. Next, I cover various design topics, including how to integrate the monolith and services, how to maintain database consistency across the monolith and services and how to handle security. I end this chapter by describing a couple of example services. One service is the `Delayed Order Service` service, which implements brand new functionality. The other service is the `Delivery Service`, which is extracted from the monolith. Let's start by taking a look at the concept of refactoring to a microservice architecture.

12.1 Overview of refactoring to microservices

Put yourself in Mary's shoes. You are responsible for the FTGO application, which is a large and old monolithic application. The business is extremely frustrated with engineering's inability to deliver features rapidly and reliably. FTGO appears to be suffering from a classic case of monolithic hell. Microservices seem, at least on the surface, to be the answer. Should you propose diverting development resources away from feature development to migrating to a microservice architecture?

I start this section by discussing why you should consider refactoring to microservices. I also discuss why it's important to be sure that your software development problems are because you are in monolithic hell rather than, for example, a poor software development process. I then describe strategies for incrementally refactoring your monolith to a microservice architecture. Next, I discuss the importance of delivering improvements earlier and often in order to maintain the support of the business. I then describe why you avoid investing in a sophisticated deployment infrastructure until you developed a few services. Finally, I describe the various strategies that you can use to introduce services into your architecture, including implementing new features as services and extracting services from the monolith.

12.1.1 Why refactor a monolith?

The microservice architecture has, as I described in chapter 1, numerous benefits. It has much better maintainability, testability, and deployability and so accelerates development. The microservice architecture is more scalable and improves fault isolation. It is also much easier to evolve your technology stack. However, refactoring a monolith to microservices is a significant undertaking. It will divert resources away from new feature development. As a result, it's likely that the business will only support the adoption of microservices if it solves a significant business problem.

If you are in monolithic hell, it is likely that eventually the business will be impacted. Examples of business problems caused by monolithic hell include:

- Slow delivery - the application is difficult to understand, maintain, and test so

developer productivity is low. As a result, the organization is unable to compete effectively and risks being overtaken by competitors.

- Buggy software releases - the lack of testability means that software releases are often buggy. This makes customers unhappy, which results in losing customers and less revenue.
- Poor scalability - scaling a monolithic application is difficult because it combines modules with very different resource requirements into one executable component. The lack of scalability means that it is either impossible or prohibitively expensive to scale the application beyond a certain point. As a result, the application cannot support the current or predicted needs of the business.

It's important to be sure that these problems are because you have outgrown your architecture. A common reason for slow delivery and buggy releases is a poor software development process. For example, if you are still relying on manual testing, then simply adopting automated testing can significantly increase development velocity. Similarly, you can sometimes solve scalability problems without changing your architecture. You should first try simpler solutions. If, and only if, you still have software delivery problems should you then migrate to the microservice architecture. Let's look at how to do that.

12.1.2 Strangling the monolith

The process of transforming a monolithic application into microservices is a form of application modernization⁶⁹. Application modernization is the process of converting a legacy application to have a modern architecture and technology stack. Developers have been modernizing applications for decades. As a result, there is wisdom accumulated through experience that we can use when refactoring an application into a microservice architecture. The most important lesson learned over the years is to not do a big bang rewrite.

A big bang rewrite is when you develop a new application - in this case a microservices-based application - from scratch. Although starting from scratch and leaving the legacy code base behind sounds appealing, it is extremely risky and will likely end in failure. You will spend months, possibly years, duplicating the existing functionality and only then can you implement the features that the business needs today! Also, you will need to develop the legacy application anyway, which diverts effort away from the rewrite and means that you have a constantly moving target. What's more, it's possible that you will waste time reimplementing features that are no longer needed. As Martin Fowler⁷⁰ reportedly said, "the only thing a Big Bang rewrite guarantees is a Big Bang!"

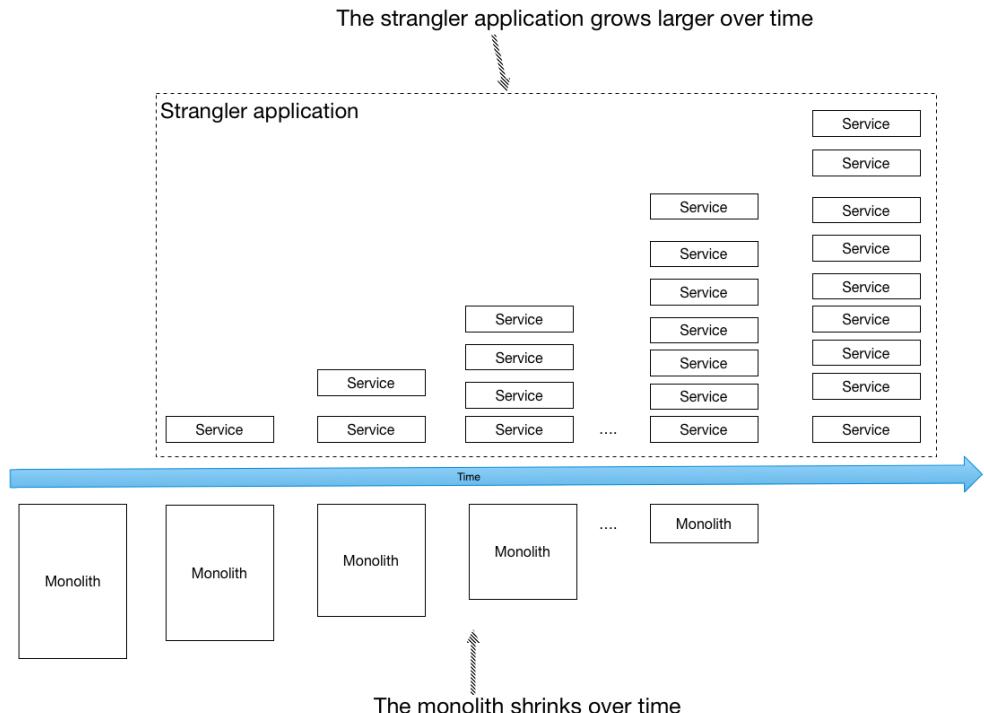
Instead of doing a big bang rewrite, you should, as figure 12.1 shows, incrementally refactor your monolithic application. You gradually build a new application consisting of microservices that runs in conjunction with your monolithic application. Over time, the amount of functionality implemented by the monolithic application shrinks until

⁶⁹ https://en.wikipedia.org/wiki/Software_modernization

⁷⁰ <http://www.randyshoup.com/evolutionary-architecture>

either it disappears entirely or it becomes just another microservice. This strategy is akin to servicing your car while driving down the highway at 70 mph. It is challenging but is far less risky than attempting a big bang rewrite.

Figure 12.1. The monolith is incrementally replaced by a stranger application comprised of services. Eventually, the monolith is replaced entirely by the strangler application or becomes s



Martin Fowler refers to this application modernization strategy as the Strangler Application pattern⁷¹. The name comes from the Strangler Vine (a.k.a. Strangler Fig)⁷² that is found in rain forests. A strangler vine grows around a tree in order to reach the sunlight above the forest canopy. Often, the tree dies, because either it's killed by the vine or, it dies of old age, leaving a tree shaped vine. Application modernization follows the same pattern. You will build a new application consisting of microservices around the legacy application, which will eventually wither away and die.

Pattern: Strangler application

Modernize an application by incrementally developing a new (strangler) application around the legacy application.

⁷¹ <http://www.martinfowler.com/bliki/StranglerApplication.html>

⁷² https://en.wikipedia.org/wiki/Strangler_fig

I say eventually because the refactoring process typically takes months, or years. For example, according to Steve Yegge⁷³ it took Amazon.com a couple of years to refactor their monolith. In the case of a very large system, you might never complete the process. You might, for example, get to point where you have tasks that are more important than breaking up the monolith, such as implementing revenue generating features. If the monolith is not an obstacle to on-going development, then you might as well leave it alone.

Demonstrate value early and often

An important benefit of incrementally refactoring to a microservice architecture is that you get an immediate return on your investment. That's very different than a big bang rewrite, which doesn't deliver any benefit until it is complete. When incrementally refactoring the monolith, you can develop each new service using a new technology stack and a modern, high velocity, DevOps-style development and delivery process. As a result, your team's delivery velocity steadily increases over time.

What's more, you can migrate the high value areas of your application to microservices first. For instance, let's imagine that you working on the FTGO application. The business might, for example, decide that the delivery scheduling algorithm is a key competitive advantage. It's likely that delivery management will be an area of constant, on-going development. By extracting delivery management into a standalone service, the delivery management team will be able to work independently of the rest of the FTGO developers and significantly increase their development velocity. They'll be able to frequently deploy new versions of the algorithm and evaluate their effectiveness.

Another benefit of being able to deliver value earlier is that it helps maintain the business's support for the migration effort. Their ongoing support is essential since the refactoring effort will mean that less time is spent on developing features. Some organizations have difficulty eliminating technical debt because past attempts were too ambitious and didn't provide much benefit. As a result, the business becomes reluctant to invest in further clean up efforts. The incremental nature of refactoring to microservices means that the development team is able to demonstrate value early and often.

Minimize changes to the monolith

A recurring theme in this chapter is that you should avoid making widespread changes to the monolith when migrating to a microservice architecture. It's inevitable that you will need to make some changes in order to support migration to services. For example, later in section "[Maintaining data consistency across a service and a monolith](#)", I describe how you often need to modify the monolith so that it can participate in sagas that maintain data consistency across the monolith and services. The problem with making widespread changes to the monolith is that it's time consuming, costly and risky. After all, that's probably why you want to migrate to microservices in the first

⁷³ <https://plus.google.com/+RipRowan/posts/eVeouesvaVX>

place.

Fortunately, there are strategies that you can use for reducing the scope of the changes that you need to make. For example, in section “[Refactoring the database](#)”, I describe the strategy of replicating data from an extracted service back to the monolith’s database. And, in section “[Maintaining data consistency across a service and a monolith](#)”, I show how you can carefully sequence the extraction of services to reduce the impact on the monolith. By applying these strategies, you can reduce the amount of work required to refactor the monolith.

Technical deployment infrastructure: you don’t need all of it yet

Throughout this book I’ve discussed a lot of shiny new technology including deployment platforms, such as Kubernetes and AWS Lambda, and service discovery mechanisms. You might be tempted to begin your migrating to microservices by selecting technologies and building out that infrastructure. You might even feel pressure from the business people and from your friendly PaaS vendor to start spending money on this kind of infrastructure.

As tempting as it seems to build out this infrastructure up front, I recommend waiting. The only thing that you cannot live without is a deployment pipeline that performs automating testing. For example, if you only have a handful of services, you don’t need a sophisticated deployment and observability infrastructure. Initially, you can even get away with just using a hard coded configuration file for service discovery. I suggest deferring any decisions about technical infrastructure until you have gained real experience with the microservice architecture. It’s only once you have a few services running that you will have the experience to pick technologies. Let’s now look at the strategies you can use for migrating to a microservice architecture.

12.2 Strategies for refactoring a monolith to microservices

There are three main strategies for incrementally migrating a monolith to a microservice architecture:

1. Implement new features as services
2. Separate the presentation tier and back end
3. Break up the monolith by extracting functionality into services

The first strategy stops the monolith from growing. It’s typically a quick way to demonstrate the value of microservices and so helps build support for the migration effort. The other two strategies break apart the monolith. When refactoring your monolith, you might sometimes use the second. But you will definitely use the third strategy since it’s how functionality is migrated from the monolith into the strangler application. Let’s take a look at each of these strategies starting with implementing new features as services.

12.2.1 Implement new features as services

The Law of Holes⁷⁴ states that "if you find yourself in a hole, stop digging". This is great advice to follow when your monolithic application has become unmanageable. In other words, if you have a large, complex monolithic application, don't implement new features by adding code to the monolith. That will make your monolith even larger and more unmanageable. Instead, you should implement new features as services.

This is a great way to begin migrating your monolithic application to a microservice architecture. It reduces the growth rate of the monolith. It accelerates the development of the new features since you are doing development in a brand new code base. It also quickly demonstrates the value of adopting the microservice architecture.

Integrating the new service with the monolith

Figure 12.2 shows the application's architecture after implementing a new feature as a service. As well as the new service and monolith, the architecture includes two other elements, which integrate the service into the application:

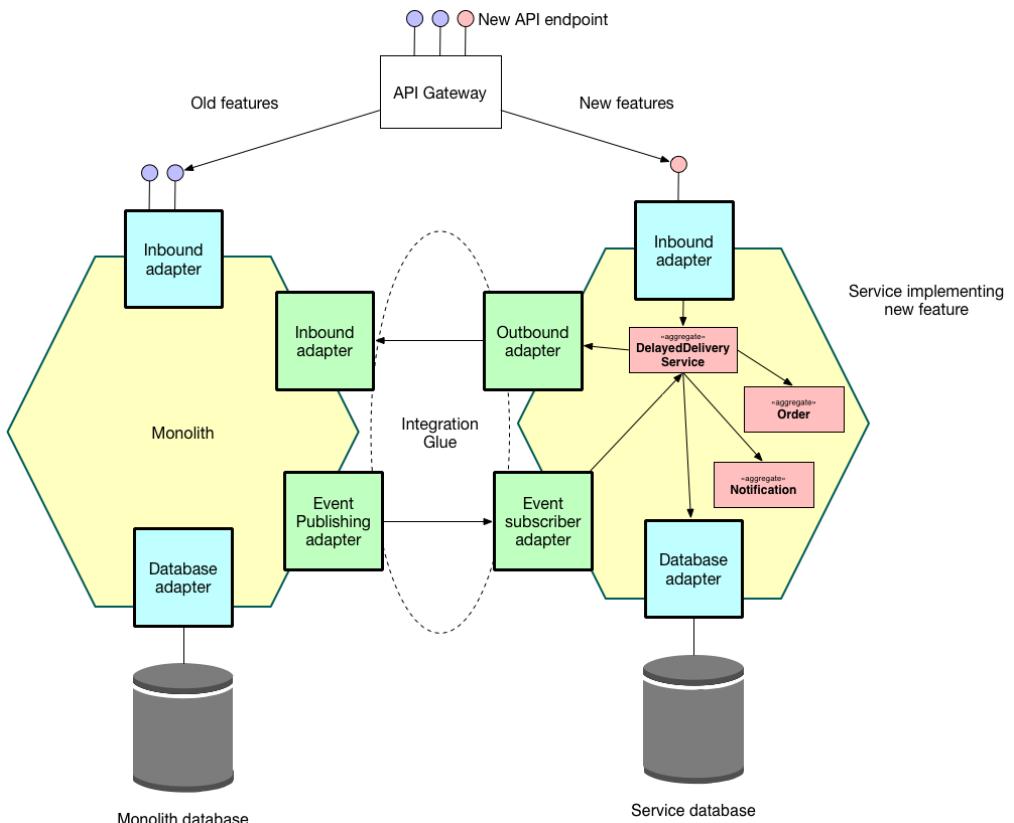
- API gateway - routes requests for new functionality to the new service and routes legacy requests to the monolith.
- Integration glue code - integrates the service with monolith. It enables the service to access data owned by the monolith and to invoke functionality implemented by the monolith.

⁷⁴ https://en.m.wikipedia.org/wiki/Law_of_holes

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

Figure 12.2. A new feature is implemented as a service that's part of the strangler application. The integration glue integrates the service with the monolith and consists of adapters that implement synchronous and asynchronous APIs. An API Gateway routes requests that invoke new functionality to the service.



The integration glue code isn't a standalone component. Instead, it consists of adapters in the monolith and the service, which use one or more inter-process communication mechanisms. For example, integration glue for the **Delayed Delivery Service**, which I describe in section "[The design of the Delayed Delivery Service](#)", uses both REST and domain events. The service retrieves customer contract information from the monolith by invoking a REST API. The monolith publishes Order domain events so that the **Delayed Delivery Service** can track the state of Orders and respond to orders that won't be delivered on time. Later on in section "[Designing the integration glue](#)", I'll describe the integration glue code in more detail.

When to implement a new feature as a service

Ideally, you should be to implement every new feature in the strangler application rather than in the monolith. You will implement a new feature as either a new service or as part of an existing service. This way you will avoid ever having to touch the

monolith code base. Unfortunately, however, not every new feature can be implemented as a service.

That's because the essence of a microservice architecture is a set of loosely coupled services that are organized around business capabilities. A feature might, for instance, simply be too small to be a meaningful service. You might, for example, just need to add a few fields and methods to an existing class. Alternatively, the new feature might be too tightly coupled to the code in the monolith. If you attempted to implement this kind of feature as service you would typically find that performance would suffer because of excessive inter-process communication. You might also have problems maintaining data consistency. If a new feature can't be implemented as a service, then the solution is often to initially implement the new feature in the monolith. Later on, you can then extract that feature along with other related features into their own service.

Implementing new features as services accelerates the development of those features. It's a good way to quickly demonstrate the value of the microservice architecture. It also reduces the monolith's growth rate. But ultimately you need to break apart the monolith using the two other strategies. You need to migrate functionality to the strangler application by extracting functionality from the monolith into services. You might also be able to improve development velocity by splitting the monolith horizontally. Let's look at how to do that.

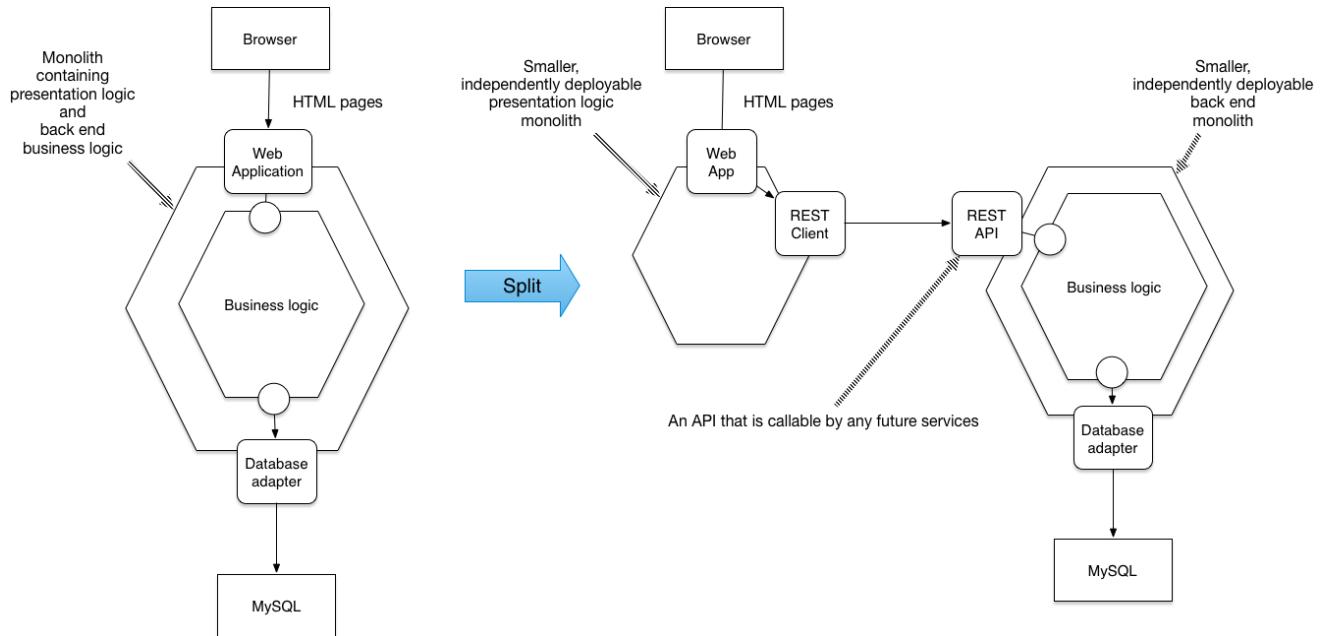
12.2.2 Separate presentation tier from the back end

One strategy for shrinking a monolithic application is to split the presentation layer from the business logic and data access layers. A typical enterprise application consists of the following layers:

- presentation logic - components that handle HTTP requests and generating HTML pages that implement a web UI. In an application that has a sophisticated user interface the presentation tier is often a substantial body of code.
- business logic - the business logic
- data access logic - components that access infrastructure components such as databases and message brokers

There is usually a clean separation between the presentation logic and the business and data access logic. The business tier has a coarse-grained API consisting of one or more facades, which encapsulate the business logic. This API is a natural seam along which you can split the monolith into two smaller applications, as shown in figure 12.3. One application contains the presentation layer. The other application contains the business and data access logic. After the split the presentation logic application makes remote calls to the business logic application.

Figure 12.3. Splitting the front-end from the back-end enables each one to be deployed independently. It also exposes an API for services to invoke.



Splitting the monolith in this way has two main benefits. It enables you to develop, deploy and scale the two applications independently of one another. In particular, it allows the presentation layer developers to rapidly iterate on the user interface and easily perform A/B testing, for example, without having to deploy the backend. Another benefit of this approach, is that it exposes a remote API that can be called by the microservices that you develop later.

This strategy, however, is only a partial solution. It is very likely that at least one or both of the resulting applications will still be an unmanageable monolith. You need to use the third strategy to replace the monolith with services.

12.2.3 Extract business capabilities into services

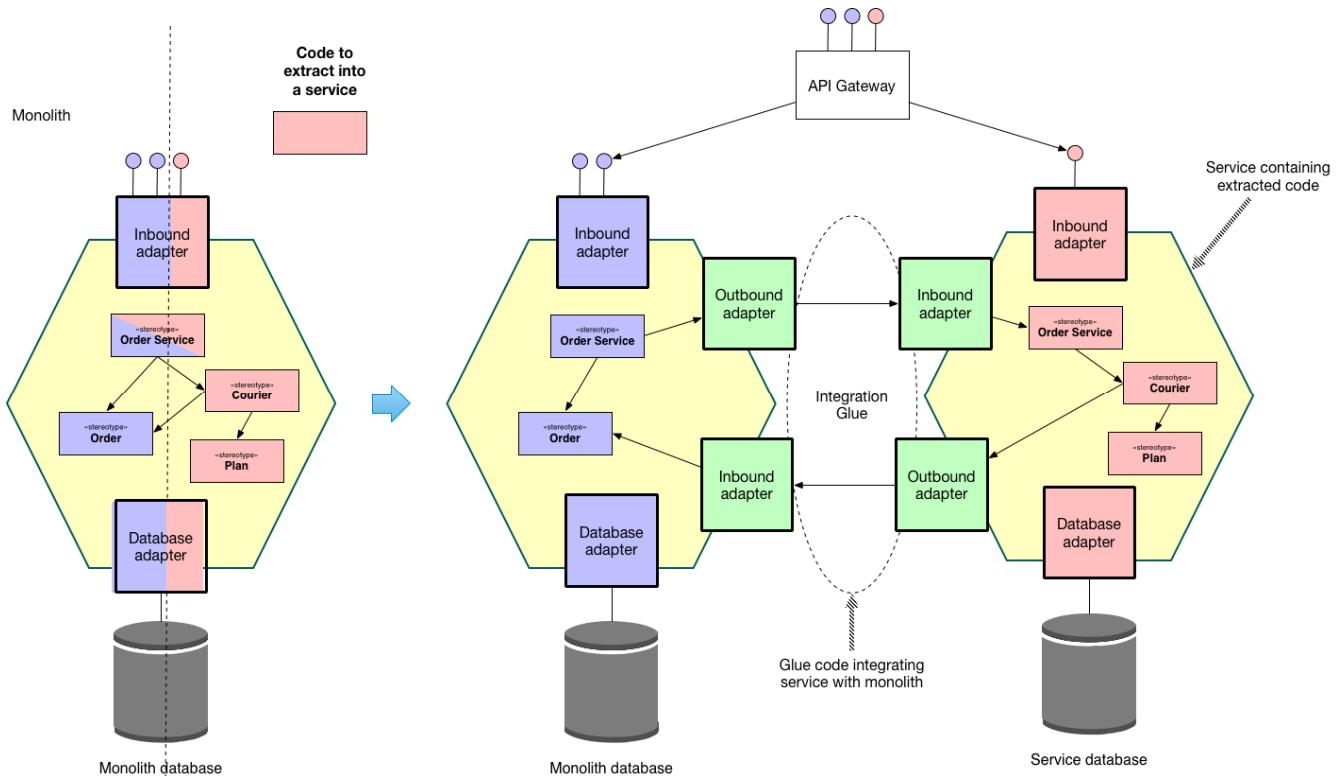
Implementing new features as services and splitting the front-end web application from the backend will only get you so far. You will still end up doing a lot of development in the monolithic code base. If you want to significantly improve your application's architecture and increase your development velocity you need to break apart the monolith by incrementally migrating business capabilities from the monolith to services. For example, later in section "[Breaking apart the monolith: extracting delivery management](#)", I describe how to extract delivery management from the FTGO monolith into a new `Delivery Service`. When you use this strategy, over time the number of business capabilities implemented by the services grows and the monolith gradually shrinks.

The functionality that you want extract into a service is a vertical slice through the monolith. The slice consists of

- inbound adapters - implement API endpoints
- domain logic
- outbound adapters - e.g. database access logic
- the monolith's database schema

As figure 12.4 shows, this code is extracted from the monolith and moved into a standalone service. An API gateway routes requests that invoke the extracted business capability to the service and routes the other requests to the monolith. The monolith and the service collaborate via the integration glue code. As I describe in section “[Designing the integration glue](#)”, the integration glue consists of adapters in the service and monolith that use one or more inter-process communication (IPC) mechanisms.

Figure 12.4. Break apart the monolith by extracting services. You identify a slice of functionality, which consists of business logic and adapters, to extract into a service. You move that code into the service. The newly extracted service and the monolith collaborate via the APIs provided by the integration glue.



Extracting services is challenging. You need to determine how to split the monolith's domain model into two separate domain models, one of which becomes the service's domain model. You need to break dependencies, such as object references. You might even need to split classes in order to move functionality into the service. You also need to refactor the database as well.

Extracting a service is often time consuming, especially since the monolith's code base is likely to be messy. Consequently, you need to carefully think about which services to extract. It's important to focus on refactoring those parts of the application that provides a lot of value. Before extracting a service, you must ask yourself what's the benefit of doing that.

For example, it's worthwhile to extract a service that implements functionality that is critical to the business and constantly evolving. It's not valuable to invest effort in extracting services when there's not much benefit from doing so. Later in this section I'll describe some strategies for determining what to extract and when. But first, let's look in more detail at some of the challenges you will face when extracting a service and how to address them.

You will encounter a couple of challenges when extracting a service:

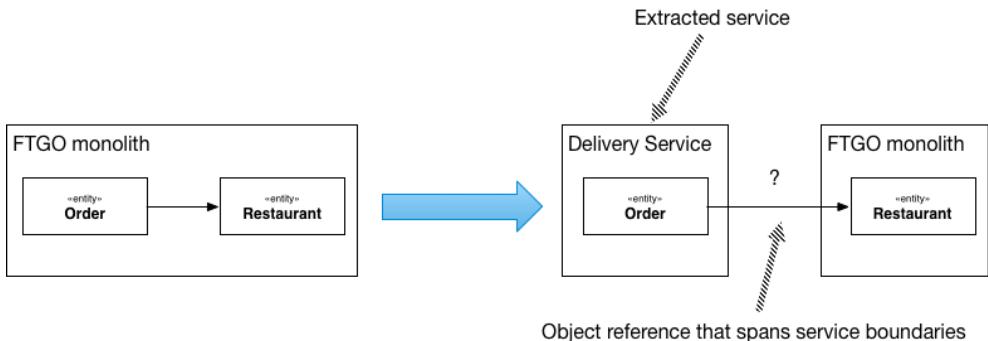
- Splitting the domain model
- Refactoring the database

Let's look at each one starting with splitting the domain model.

Splitting the domain model

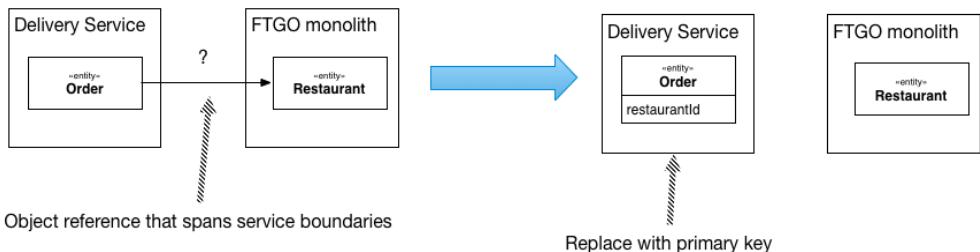
In order to extract a service you need to extract its domain model out of the monolith's domain model. You will need to perform major surgery to split the domain models. One challenge that you'll encounter is eliminating object references that otherwise span service boundaries. It's possible that classes that remain in the monolith will reference classes that have been moved to the service or vice versa. For example, let's imagine that, as figure 12.5 shows, you extract the Order Service service and as a result, its `Order` class references the monolith's `Restaurant` class. Since a service instance is typically a process, it doesn't make sense to have object references that cross service boundaries. Somehow you need to eliminate these types of object reference.

Figure 12.5. The Order domain class has a reference to a Restaurant class. If we extracted Order into a separate service, then we need to do something about its reference to Restaurant since object references between processes don't make sense.



One good way to solve this problem is to think in terms of DDD aggregates, which I described in chapter 5. Aggregates reference each other using primary keys rather than object references. You would, therefore, think of the Order and Restaurant classes as aggregates and, as figure 12.6 shows, replace the reference to Restaurant in the Order class with a `restaurantId` field that stores the primary key value.

Figure 12.6. The Order class' reference to the Restaurant is replaced with the Restaurant's primary key in order to eliminate an object that would span process boundaries.



One issue with replacing object references with primary keys is that while this is a minor change to the class, it can have a potentially large impact on the clients of the class, which expect an object reference. Later on in this section, I describe how to reduce the scope of the change by replicating data between the service and monolith. The Delivery Service, for example, could define a `Restaurant` class that's a replica of the monolith's `Restaurant` class.

Extracting a service is often much more involved than simply moving entire classes into a service. An even greater challenge with splitting a domain model is extracting functionality that's embedded in a class that has other responsibilities. This problem often occurs God classes, which I described in chapter 2 that have an excessive number of responsibilities. For example, the `Order` class is one of the god classes in the FTGO

application. It implements multiple business capabilities including order management, delivery management and so on. Later in section “[Breaking apart the monolith: extracting delivery management](#)”, I discuss how extracting the delivery management into service involves extracting a `Delivery` classes from the `Order` class. The `Delivery` entity implements the delivery management functionality that was previously bundled with other functionality in the `Order` class.

Refactoring the database

Splitting a domain model involves more than just changing code. Many classes in a domain model are persistent. Their fields are mapped to a database schema. Consequently, when you extract a service from the monolith, you are also moving data. You need to move tables from the monolith’s database to the service’s database.

Also, when you split an entity you need to split the corresponding database table and move the new table to the service. For example, when extracting delivery management into a service, you split the `Order` entity and extract a `Delivery` entity. At the database level, you split the `ORDERS` table and define a new `DELIVERY` table. You then move the `DELIVERY` table to the service.

The book Refactoring Databases⁷⁵ describes a set of refactorings for database schema. For example, it describes the *Split Table* refactoring, which splits a table into two or more tables. Many of the techniques in this book are useful when extracting services from the monolith. One such technique is the idea of replicating data in order to allow you incrementally to update clients of the database to use the new schema. We can adapt that idea to reduce the scope of the changes you must make to the monolith when extracting a service.

Replicate data to avoid widespread changes

As I described earlier in this section, extracting a service requires you to change to the monolith’s domain model. For example, you replace object references with primary keys and split classes. These types of changes can ripple through the code base and require you to make widespread changes to the monolith. For example, if you split the `Order` entity and extract a `Delivery` entity then you will have to change every place in the code that references the fields have been moved. It can be extremely time consuming to make these kinds of changes and can be a huge barrier to breaking up the monolith.

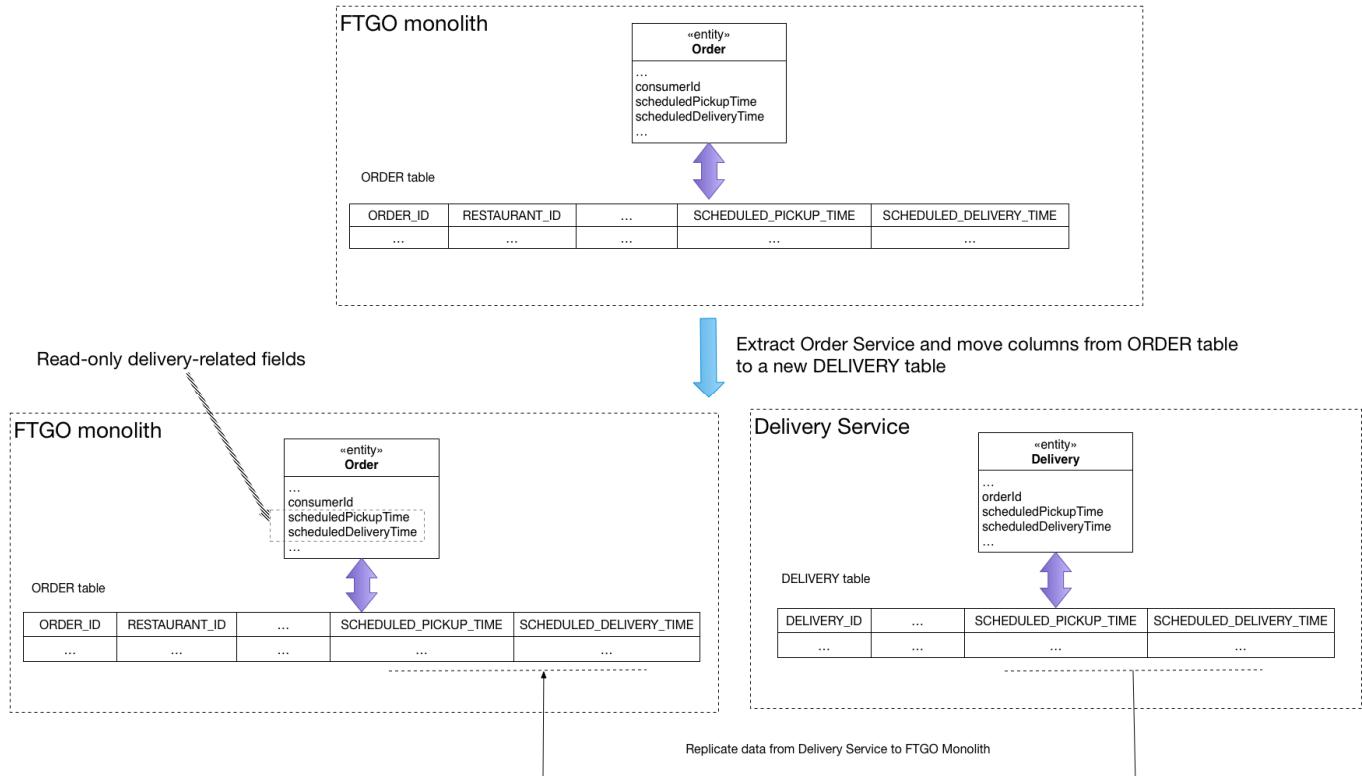
A great way to delay and possibly avoid making these kinds of expensive changes, is to use an approach that’s similar to the one described in the Refactoring Databases book. A major obstacle to refactoring a database is changing all of the clients of that database to use the new schema. The solution proposed in the book is to preserve the original schema for a transition period and using triggers to synchronize the original and new schemas. You then migrate clients from the old schema to the new schema over time.

We can use a similar approach when extracting service from the monolith. For

⁷⁵ Refactoring Databases: Evolutionary Database Design by Scott J Ambler and Pramod J. Sadalage

example, when extracting the `Delivery` entity, we leave the `Order` entity mostly unchanged for a transition period. As figure 12.7 shows, we simply make the delivery-related fields read-only and keep them up to date by replicating data from the `Delivery Service` back to the monolith. As a result, we only need to find the places in the monolith's code that update those fields and change them to invoke the new `Delivery Service`.

Figure 12.7. Minimize the scope of the changes to the FTGO monolith by replicating delivery-related data from the newly extracted Delivery Service back to the monolith's database.



Preserving the structure of the `Order` entity by replicating data from the `Delivery Service` significantly reduces the amount of work we need to do immediately. Over time, we can migrate code that uses the delivery-related `Order` entity fields or `ORDERS` table columns to the `Delivery Service`. What's more, it's possible that we never need to make that change in the monolith. If that code is subsequently extracted into a service then the service can simply access the `Delivery Service`.

What services to extract and when

Breaking apart the monolith is time consuming. It diverts effort away from implementing features. As a result, you must carefully decide the sequence in which you extract services. You need to focus on extracting services that give the largest

benefit. What's more, you want to continually demonstrate to the business that there's value in migrating to a microservice architecture.

On any journey, it's essential to know where you are going. A good way to start the migration to microservices is with a time-boxed architecture definition effort. You should spend a short amount of time, such as a couple of weeks, brainstorming your ideal architecture and defining a set of services. This gives you a destination to aim for. It's important, however, to remember that this architecture is not set in stone. As you break apart the monolith and gain experience, you should revise the architecture to take into account what you have learned.

Once you have determined the approximate destination, the next step is to start breaking apart the monolith. There are a couple of different strategies that you can use to determine the sequence in which you extract services.

One strategy is to effectively freeze development of the monolith and to extract services on demand. Instead of implementing features or fixing bugs in the monolith, you extract the necessary service or service(s) and change those. One benefit of this approach is that it forces you to break up the monolith. One drawback is that the extraction of services is driven by short-term requirements rather than long-term needs. For instance, it requires to extract service even if you are making a small change to relatively stable part of the system. As a result, you risk doing a lot of work for minimal benefit.

An alternative strategy is a more planned approach where you rank the modules of an application by the benefit you anticipate getting from extracting them. There are a few reasons why extracting a service is beneficial:

- Accelerates development - your application's roadmap suggests that a particular part of your application will undergo a lot of development over the next year then converting it to a service accelerates development
- Solves a performance, scaling or reliability problem - a particular part of your application has a performance or scalability problem or is unreliable then its valuable to convert to a service
- Enables the extraction of some other services - sometimes extracting one service simplifies the extraction of another service due to dependencies between modules

You can use these criteria to add refactoring tasks to your application's backlog ranked by expected benefit. The benefit of this approach is that it is more strategic and much more closely aligned with the needs of the business. During sprint planning, you decide whether it's more valuable to implement features or to extract services.

12.3 Designing how the service and the monolith collaborate

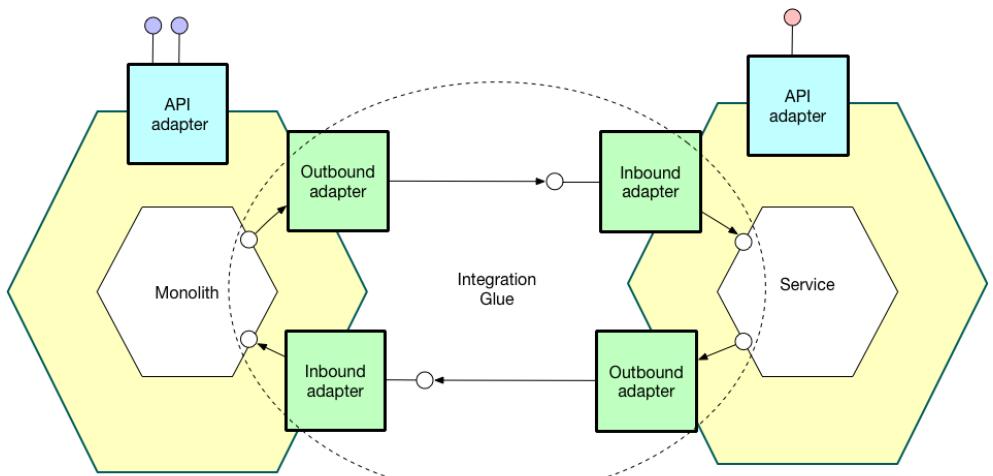
A service is rarely standalone. It usually needs to collaborate with the monolith. Sometimes a service needs to access data owned by the monolith or to invoke its operations. For example, the `Delayed Delivery Service`, which I describe in detail in section "[The design of the Delayed Delivery Service](#)", requires access to the

monolith's orders and customer contact info. The monolith might also need to access data owned by the service or to invoke its operations. For example, later in section [“Breaking apart the monolith: extracting delivery management”](#) when discussing how to extract delivery management into a service, I describe how the monolith needs to invoke the Delivery Service.

One important concern is maintaining data consistency between the service and monolith. In particular, when you extract a service from the monolith you invariably split what were originally ACID transactions. You must be careful to ensure that data consistency is still maintained. As I describe later in this section, sometimes you use sagas to maintain data consistency.

The interaction between a service and the monolith is, as I described earlier, facilitated by integration glue code. Figure 12.8 shows the structure of the integration glue. It consists of adapters in the service and monolith, which communicate using some kind of IPC mechanism. Depending on the requirements, the service and monolith might interact over REST or they might use messaging. They might even communicate using multiple IPC mechanisms.

Figure 12.8. When migrating a monolith to a microservices, the services and the monolith often need to access each other’s data. This interaction is facilitated by the integration glue, which consists of adapters that implement APIs. Some APIs are messaging-based. Other APIs are RPI-based.



For example, the `Delayed Delivery Service` uses both REST and domain events. It retrieves customer contact info from the monolith using REST. It tracks the state of Orders by subscribing to domain events published by the monolith.

In this section, I first describe the design of the integration glue. I discuss the problems that it solves and the different implementation options. After that I describe transaction management strategies including the use of sagas. I discuss how sometimes the

requirement to maintain data consistency changes the order in which you extract services. Let's first look at the design of the integration glue.

12.3.1 Designing the integration glue

When implementing a feature as a service or extracting a service from the monolith, you must develop the integration glue that enables a service to collaborate with the monolith. It consists of code in both the service and monolith that uses some kind of IPC mechanism. The structure of the integration glue depends on the type of IPC mechanism that is used. If, for example, the service invokes the monolith using REST then the integration glue consists of a REST client in the service and web controllers in the monolith. Alternatively, if the monolith subscribes to domain events published by the service, then the integration glue consists of an event publishing adapter in the service and event handlers in the monolith.

Designing the integration glue API

The first step is designing the integration glue is to decide what APIs it provides to the domain logic. There are a couple of different styles of interface to choose from depending on whether you are querying data or updating data. Let's imagine, for example, that you are working on the `Delayed Delivery Service`, which needs to retrieve customer contact info from the monolith. The service's business logic doesn't need to know the actual IPC mechanism that the integration glue uses to retrieve the information. Therefore, that mechanism should be encapsulated by an interface. Since the `Delayed Delivery Service` is querying data it makes sense to define a `CustomerContactInfoRepository`:

```
interface CustomerContactInfoRepository {
    CustomerContactInfo findCustomerContactInfo(long customerId)
}
```

The service's business logic can invoke this API without knowing how the integration glue retrieves the data.

Let's consider a different service. Imagine that you are extracting delivery management from the FTGO monolith. The monolith needs to invoke the `Delivery Service` to schedule, reschedule and cancel deliveries. Once again the details of the underlying IPC mechanism aren't important to the business logic and should be encapsulated by an interface. In this scenario, the monolith must invoke an service operation and so using a repository doesn't make sense. A better approach is to define a service interface, such as the following:

```
interface DeliveryService {
    void scheduleDelivery(...);
    void rescheduleDelivery(...);
    void cancelDelivery(...);
}
```

The monolith's business logic invokes this API without knowing how it's implemented by the integration glue. Now that we have looked at interface design, let's look at

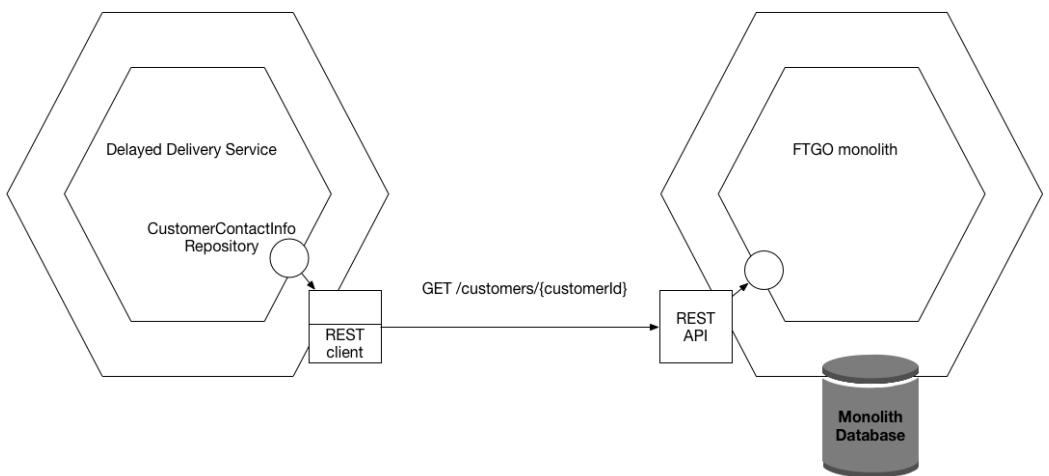
interaction styles and IPC mechanisms.

Picking an interaction style and IPC mechanism

An important design decision that you must make when designing the integration glue is selecting the interaction styles and IPC mechanisms that enable the service and the monolith to collaborate. As I described in chapter 3, there are several interaction styles and IPC mechanisms to choose from. Which one you should use depends on what the service or monolith (a.k.a. party) needs to query or update the other party.

If one party needs to query data owned by the other party there are several options. One option is, as figure 12.9 shows, for the adapter that implements the repository interface to invoke an API of the data provider. This API will typically use a request/reply interaction style, such as REST or gRPC. For example, the Delayed Delivery Service might retrieve the customer contact info by invoking a REST API implemented by the FTGO monolith.

Figure 12.9. The adapter that implements the CustomerContactInfoRepository interface invokes the monolith's REST API to retrieve the customer information.



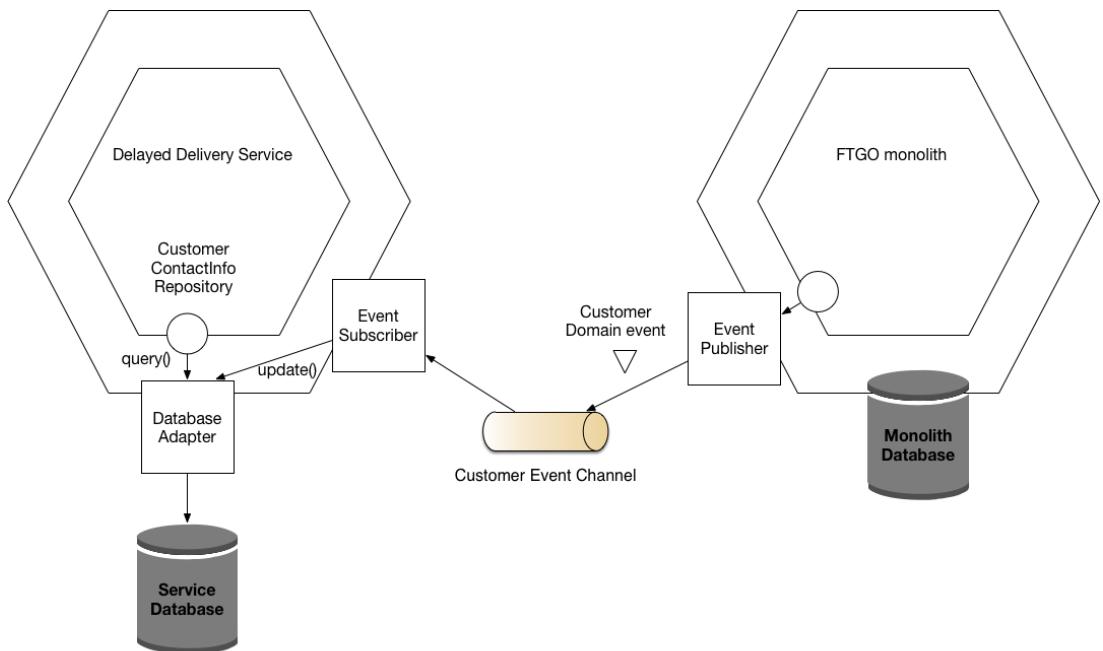
In this example, the Delayed Delivery Service's domain logic retrieves the customer contact info by invoking `CustomerContactInfoRepository` interface. The implementation of this interface invokes the monolith's REST API.

An important benefit of querying data by invoking a query API is its simplicity. The main drawback is that it is potentially inefficient. A consumer might need to make a large number of requests. A provider might return a large amount of data. Another drawback is that it reduces availability since it's synchronous IPC. As a result, it might not be practical to use a query API.

An alternative approach is for the data consumer to maintain a replica of the data, as is shown in figure 12.10. The replica is essentially a CQRS view. It data consumer keeps

the replica up to date by subscribing to domain events published by the data provider.

Figure 12.10. The integration glue replicates data from the monolith the service. The monolith publishes domain events and an event handler implemented by the service update's the service's database.



Using a replica has several benefits. It avoids the overhead of repeatedly querying the data provider. Instead, as I discussed when describing CQRS in chapter 7, you can design the replica to support efficient queries. One drawback of using a replica, however, is the complexity of maintaining the replica. A potential challenge is, as I describe later on in this section, the need to modify the monolith to publish domain events.

Now that we have discussed how to do queries, let's now consider how to do updates. One challenge with performing updates is the need to maintain data consistency across the service and monolith. The party making the update request (ie. the requestor) has or needs to update its database. It is, therefore, essential that both updates happen. The solution is for the service and monolith to communicate using transactional messaging implemented by a framework, such as Eventuate Tram. In simple scenarios, the requestor can simply send a notification message or publish an event to trigger an update. In more complex scenarios, the requestor must use a saga to maintain data consistency. Later in section [“Maintaining data consistency across a service and a monolith”](#) I'll discuss the implications of using sagas.

Implementing an anti-corruption layer

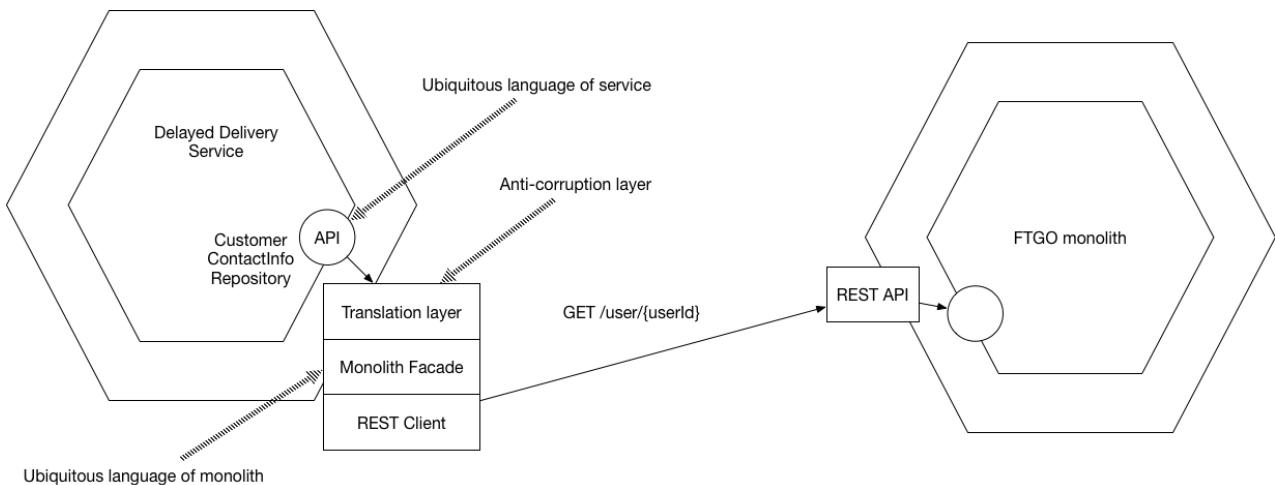
Let's imagine that you are implementing a new feature as a brand new service. Since you aren't constrained by the monolith's code base, you can use modern development techniques, such as DDD, and develop a pristine new domain model. Also, because the FTGO monolith's domain is poorly defined and somewhat out of date, you will probably model concepts differently. As a result, your service's domain model will have different class names, field names, and field values. For example, the Delayed Delivery Service has a `Delivery` entity with narrowly focussed responsibilities whereas the FTGO monolith has an `Order` entity with an excessive number of responsibilities. Because the two domain models are different, you must implement what DDD calls an anti-corruption layer (ACL) in order for the service to communicate with the monolith.

Pattern: Anti-corruption layer

A software layer that translates between two different domain models in order to prevent concepts from one model polluting another.

The goal of an ACL is to prevent a legacy monolith's domain model from polluting a service's domain model. It's a layer of code that translates between the different domain models. For example, as figure 12.11 shows, the Delayed Delivery Service has a `CustomerContactInfoRepository` interface, which defines a `findCustomerContactInfo()` method that returns a `CustomerContactInfo`. The class that implements the `CustomerContactInfoRepository` interface must translate between the ubiquitous language of the Delayed Delivery Service and that of the FTGO monolith.

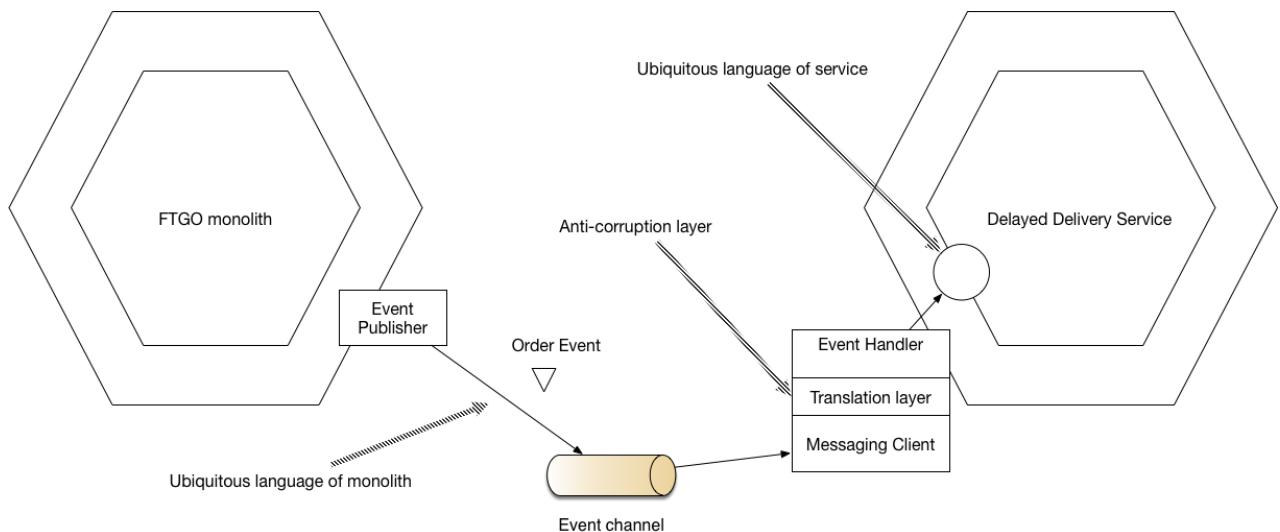
Figure 12.11. A service adapter that invokes the monolith must translate between the service's domain model and monolith's domain model.



The implementation of `findCustomerContactInfo()` invokes the FTGO monolith to retrieve the customer information and translates the response to a `CustomerContactInfo`. In this example, the translation is quite simple but in other scenarios, it could be quite complex and involve, for example, mapping values, such as status codes.

An event subscriber, which consumes domain events, also has an ACL. Domain events are part of the publisher's domain model. An event handler must translate domain events to the subscriber's domain model. For example, as figure 12.12 shows, the FTGO monolith publishes Order domain events. The Delivery Service has an event handler, which subscribes to those events.

Figure 12.12. An event handler must translate from the event publisher's domain model to the subscriber's domain model.



The event handler must translate domain events from the monolith's domain language to that of the `Delivery Service`. It might need to map class and attribute names and potentially attribute values.

It's not just services that use an anti-corruption layer. A monolith also uses an ACL when invoking the service and when subscribing to domain events published by a service. For example, FTGO monolith schedules a delivery by sending a notification message to the `Delivery Service`. It sends the notification by invoking a method on the `DeliveryService` interface. The implementation class translates its parameters into a message that the `Delivery Service` understands.

How the monolith publishes and subscribes to domain events

Domain events are an important collaboration mechanism. It's straightforward for a newly developed service to publish and consume events. It can simply use one of the

mechanisms that I described in chapter 3, such the Eventuate Tram framework. A service might even publish events using event sourcing, which I described in chapter 6. It's potentially challenging, however, to change the monolith to publish and consume events. Let's look at why.

There are a couple of different ways that a monolith can publish domain events. One approach is to use the same domain event publishing mechanism used by the services. You simply find all of the places in the code that change a particular entity and insert a call to a event publishing API. The problem with this approach, however, is that changing a monolith is not always easy. It might be time consuming and error-prone to locate all of the places and insert calls to publish events. And to make matters worse, some of the monolith's business logic might consist of stored procedures, which cannot easily publish domain events.

Another approach is to publish of domain events at the database level. You can, for example, use either transaction logic tailing or polling, which I described in chapter 3. A key benefit of using transaction tailing is that you do not have to change the monolith. The main drawback of publishing events at the database level, is that it is often difficult to identify the reason for update and publish the appropriate high-level business event. As a result, the service will typically publish events representing changes to tables rather than business entities.

Fortunately, it's usually easier for the monolith to subscribe to domain events published a services. Quite often, you can simply write event handlers using a framework, such as Eventuate Tram. But sometimes, it's even challenging for the monolith to subscribe to events. For example, the monolith might be written in a language that doesn't have a message broker client. In this situation, you will need to write a small 'helper' application that subscribe to events and updates the monolith's database directly.

Now that we have looked at how to design the integration glue that enables a service and the monolith to collaborate, let's now look at another challenge that you might face when migrating to microservices: maintaining data consistency across a service and a monolith.

12.3.2 Maintaining data consistency across a service and a monolith

When you develop a service, you might find it challenging to maintain data consistency across the service and the monolith. A service operation might need to update data in the monolith or a monolith operation might need to update data in the service. For example, let's imagine that you extracted the Restaurant Order Service from the monolith. You would need to change the monolith's order management operations, such as `createOrder()` and `cancelOrder()`, to use sagas in order to keep the `RestaurantOrder` consistent with the Order.

The problem with using sagas, however, is that the monolith might not be a willing participant. As I described in chapter 4, sagas must use compensating transactions to undo changes. The `Create Order Saga`, for example, includes a compensating

transaction that marks an Order as rejected if it is rejected by the Restaurant Order Service. The problem with compensating transactions in the monolith is that you might need to make numerous and time consuming changes to the monolith in order to support them. The monolithic might also need to implement countermeasures to handle the lack of isolation between sagas. The cost of these code changes can be a huge obstacle to extracting a service.

Key saga terminology

I cover sagas in chapter 4. Here are some key terms:

- Saga - a sequence of local transactions coordinated through asynchronous messaging
- Compensating transaction - a transaction that undoes the updates made by a local transaction
- Countermeasure - a design technique used to handle the lack of isolation between sagas.
- Semantic lock - a countermeasure that consists of setting a flag that is in record that is being updated by a saga
- Compensatable transaction - a transaction that needs a compensating transaction because one of the transactions that follows it in the saga can fail
- Pivot transaction - a transaction that is the saga's go/no-go point. If it succeeds then the saga will run to completion
- Retriable transaction - a transaction that follows the pivot transaction and is guaranteed to succeed

Fortunately, many sagas are straightforward to implement. If the monolith's transactions are either, as I describe in chapter 4, *pivot transactions* or *retriable transactions* then implementing sagas should be straightforward. You might even be able to simplify implementation by carefully ordering the sequence of service extractions so that the monolith's transactions never need to be compensatable. Alternatively, it might be relatively to change the monolith to support compensating transactions. To understand why implementing compensating transactions in the monolith is sometimes challenging, let's look at some examples, beginning with a particularly troublesome one.

The challenge of changing the monolith to support compensatable transactions

Let's dig into the problem of compensating transactions that you will need to solve when extracting the Restaurant Order Service from the monolith. This refactoring involves splitting the Order entity and creating a RestaurantOrder entity in the Restaurant Order Service. It impacts numerous commands implemented by the monolith including `createOrder()`.

The monolith implements the `createOrder()` command as a single ACID transaction consisting of the following steps:

1. Validate order details
2. Verify that the consumer can place an order
3. Authorize consumer's credit card
4. Create an Order

You need to replace this ACID transaction with a saga consisting of the following

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

steps:

1. In the monolith:
 - a. Create an Order in a APPROVAL_PENDING state
 - b. Verify that the consumer can place an order
2. In the Restaurant Order Service:
 - a. Validate order details
 - b. Create a PENDING RestaurantOrder
3. In the monolith:
 - a. authorize consumer's credit card
 - b. change state of the Order to APPROVED
4. In the Restaurant Order Service:
 - a. change the state of the RestaurantOrder to APPROVED

This saga is similar to the `CreateOrderSaga`, which I described in chapter 4. It consists of four local transactions, two in the monolith and two in the Restaurant Order Service. The first transaction creates an Order in the APPROVAL_PENDING state. The second transaction creates a PENDING RestaurantOrder. The third transaction authorizes the Consumer credit card and changes the state of the order to CREATED. The fourth and final transaction changes the state of the RestaurantOrder to CREATED.

The challenge with implementing this saga is that the first step, which creates the Order must be compensatable. That's because the second local transaction, which occurs in the Restaurant Order Service, might fail and require the monolith to undo the updates performed by the first local transaction. As a result, the Order entity needs to have a APPROVAL_PENDING, which is a semantic lock countermeasure, which I described chapter 4 that indicates that an Order is in the process of being created.

The problem with introducing a new Order entity state is that it potentially requires widespread changes to the monolith. You might need to change every place in the code that touches an Order entity. Making these kinds of widespread changes to the monolith is time consuming and not the best investment of development resources. It is also potentially risky since the monolith is often difficult to test.

Sagas don't always require the monolith to support compensatable transactions

Sagas are highly domain specific. Some, such as the one we just looked at, require the monolith to support compensating transactions. However, it's quite possible that when you extract a service, you might be able to design sagas that don't require the monolith to implement compensating transactions. That's because a monolith only needs to support compensating transactions if the transactions that follow the monolith's transaction can fail. If each of the monolith's transactions is either a pivot transaction or a retriable transaction, then the monolith never needs to execute a compensating transaction. As a result, you only need to make minimal changes to the monolith to support sagas.

For example, let's imagine that instead of extracting Restaurant Order Service you extract the Order Service. This refactoring involves splitting the Order entity and

creating a slimmed down Orderentity in the Order Service. It also impacts numerous commands including `createOrder()`, which is moved from the monolith to the Order Service. In order to extract the Order Service, you need to change the `createOrder()` command to use a saga consisting of the following steps:

1. Order Service:
 - a. create an Order in a APPROVAL_PENDING state
2. Monolith:
 - a. Consumer Service: verify that the consumer can place an order
 - b. Validate order details and create a RestaurantOrder
 - c. Authorize consumer's credit card
3. Order Service:
 - a. Change state of the Order to APPROVED

This saga consists of three local transactions, one in the monolith and two in the Order Service. The first transaction, which is in the Order Service, creates an Order in the APPROVAL_PENDING state. The second transaction, which is in the monolith, verifies that the consumer can place orders, authorizes their credit card and creates a RestaurantOrder. The third transaction, which is in the Order Service, changes the state of the Order to APPROVED.

The monolith's transaction is the saga's pivot transaction - the point no return for the saga. If the monolith's transaction completes then the saga will run until completion. Only the first and second steps of this saga can fail. The third transaction cannot fail and so the second transaction in the monolith never needs to be rolled back. As a result, all the complexity of supporting compensatable transactions is in the Order Service, which is much more testable than the monolith.

If all of the sagas that you need to write when extracting service have this structure then you will need to make far fewer changes to the monolith. What's more, it's possible to carefully sequence the extraction of services to ensure that the monolith's transactions are either pivot transactions or retriable transactions. Let's look at how to do that.

Sequencing the extraction of services to avoid implementing compensating transactions in the monolith

As we just saw, extracting the Restaurant Order Service requires the monolith to implement compensating transactions whereas extracting the Order Service does not. This suggests that the order in which you extract services matters. By carefully ordering the extraction of services, we can potentially avoid having to make widespread modifications to the monolith to support compensatable transactions. We can ensure that the monolith's transactions are either pivot transactions or retriable transactions. For example, if we first extract the Order Service from the FTGO monolith and then extract the Consumer Service, it is then straightforward to extract Restaurant Order Service. Let's take a closer look at how to do that.

The Consumer Service is responsible for managing the Consumer Service. If we extract this service the `createOrder()` command uses the following saga:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

1. Order Service: create an Order in a APPROVAL_PENDING state
2. Consumer Service: verify that the consumer can place an order
3. Monolith:
 - a. Validate order details and create a RestaurantOrder
 - b. Authorize consumer's credit card
4. Order Service: change state of the Order to APPROVED

In this saga, the monolith's transaction is the pivot transaction. The Order Service implements the compensatable transaction.

Now that we have extracted the Consumer Service, we can then extract the Restaurant Order Service. If we extract this service the `createOrder()` command uses the following saga:

1. Order Service: create an Order in a APPROVAL_PENDING state
2. Consumer Service: verify that the consumer can place an order
3. Restaurant Order Service validate order details and create a PENDING RestaurantOrder
4. Monolith: Authorize consumer's credit card
5. Restaurant Order Service change the state of the RestaurantOrder to APPROVED
6. Order Service: change state of the Order to APPROVED

In this saga, the monolith's transaction is still the pivot transaction. The Order Service and Restaurant Order Service implement the compensatable transactions.

We can even continue to refactor the monolith by extracting the Accounting Service. If we extract this service the `createOrder()` command uses the following saga:

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

As you can see, by carefully sequencing the extractions you can avoid using sagas that require making complex changes to the monolith. Let's now look at how to handle security when migrating to a microservice architecture.

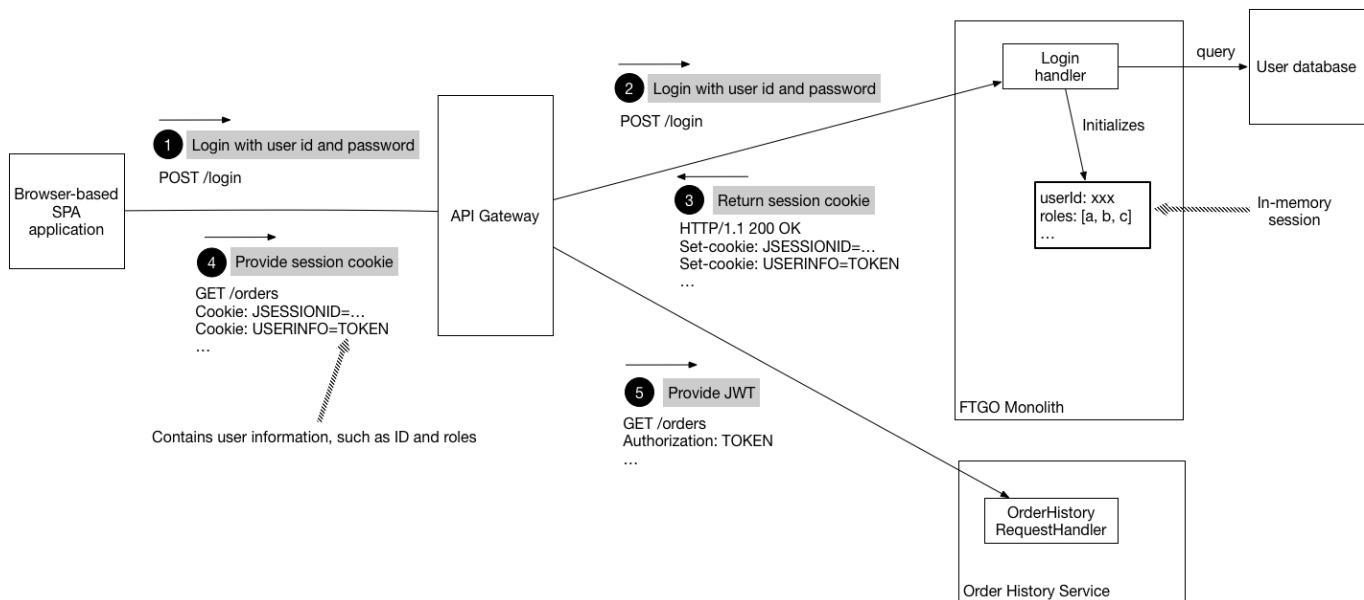
12.3.3 Handling authentication and authorization

Another design issue that you need to tackle when refactoring a monolithic application to a microservice architecture is adapting the monolith's security mechanism to support the services. In chapter {chapter-prod-ready}, I described how to handle security in a microservice architecture. A microservices-based application uses tokens, such as JSON Web tokens (JWT), to pass around user identity. That's quite different than a typical traditional, monolithic application that uses in-memory session state and passes around the user identity using a thread local. The challenge when transforming a monolith application to a microservice architecture is that you need to support both the

monolithic and JWT-based security mechanisms simultaneously.

Fortunately, there is a straightforward way to solve this problem that only require you to make only one small change to the monolith's login request handler. Figure 12.13 shows how this works. The login handler returns an additional cookie, which in this example I call `USERINFO`, that contains user information, such as the `user id` and `roles`. The browser includes that cookie in every request. The API gateway extracts the information from the cookie and includes in the HTTP requests that it makes to a service. As a result, each service has access to the needed user information.

Figure 12.13. The login handler is enhanced to set a `USERINFO` cookie which is a JWT containing user information. The API Gateway transfers the `USERINFO` cookie to an authorization header when it invokes a service.



The sequence of events is as follows:

1. The client makes a login request containing the user's credentials
2. The API Gateway routes the login request to the FTGO monolith
3. The monolith returns a response containing the JSESSIONID session cookie and the USERINFO cookie, which contains the user information, such as their ID and roles
4. The client makes a request, which includes the USERINFO cookie, in order to invoke an operation
5. The API Gateway validates the USERINFO cookie and includes it in the Authorization header of the request that it makes to the service. The service validates the USERINFO token and extracts the user information.

Let's look at the `LoginHandler` and the `API Gateway` in more detail.

The monolith's LoginHandler sets the USERINFO cookie

The LoginHandler processes the POST of the user's credentials. It authenticates the user and stores information about the user in the session. It is often implemented by a security framework, such as Spring Security or Passport for NodeJS. If the application is configured to use the default in-memory session then the HTTP response sets a session cookie, such as JSESSIONID. In order to support the migration to microservices, the LoginHandler must also set the USERINFO cookie containing the JWT that describes the user.

The API gateway maps the USERINFO cookie to the Authorization header

The API Gateway, as I described in chapter 8, is responsible for request routing and API Composition. It handles each request by making one or more requests to the monolith and the services. When the API gateway invokes a service, it validates the USERINFO cookie and passes it to the service in the HTTP request's Authorization header. By mapping the cookie to the Authorization header, the API gateway ensures that it passes the user identity to the service in a standard way that's independent of the type of client.

Eventually, we will most likely extract login and user management into services. But as you can see, by only making one small change to the monolith's login handler it's now possible for services to access user information. This enables you focus on developing services that provide the greatest value to business and delay extracting less valuable services, such as user management. Now that we have looked at how to handle security when refactoring to microservices, let's now look an example of implementing a new feature as a service.

12.4 Implementing a new feature as a service: handling mis-delivered orders

Let's imagine that you have been tasked with improving how FTGO handles mis-delivered orders. A growing number of customers has been complaining about how customer service handles orders not being delivered. The majority of orders are delivered on time, but from time to time, orders are either delivered late or not all. For example, the courier gets delayed by unexpectedly bad traffic and so the order is picked up and delivered late. Or perhaps, by the time the courier arrives at the restaurant it is closed and the delivery cannot be made. And to make matters worse, the first time customer service hears about the mis-delivery is when they receive an angry email from an unhappy customer.

A true story: my missing ice cream

One Saturday night I was feeling lazy and placed an order using a well known food delivery app to have ice cream delivered from Smitten. It never showed up. The only communication from the company was an email the next morning saying my order had been cancelled. I also got a voicemail from a very confused customer service agent who clearly didn't know what she was calling about. Perhaps the call

was prompted by one of my tweets describing what happened. Clearly, the delivery company had not established any mechanisms for properly handling inevitable mistakes.

The root cause for many of these delivery problems is the primitive delivery scheduling algorithm used by the FTGO application. A more sophisticated scheduler is under development but won't be finished for a few months. The interim solution is for FTGO to proactively handle delayed or cancelled orders by apologizing to the customer and in some cases offering compensation before the customer complains.

Your job is to implement a new feature that does the following:

1. notify the customer when their order will not be delivered on time
2. notify the customer when their order cannot be delivered because it can't be picked up before the restaurant closes
3. notify customer service when an order cannot be delivered on time so that they can proactively rectify the situation by compensating the customer
4. track delivery statistics

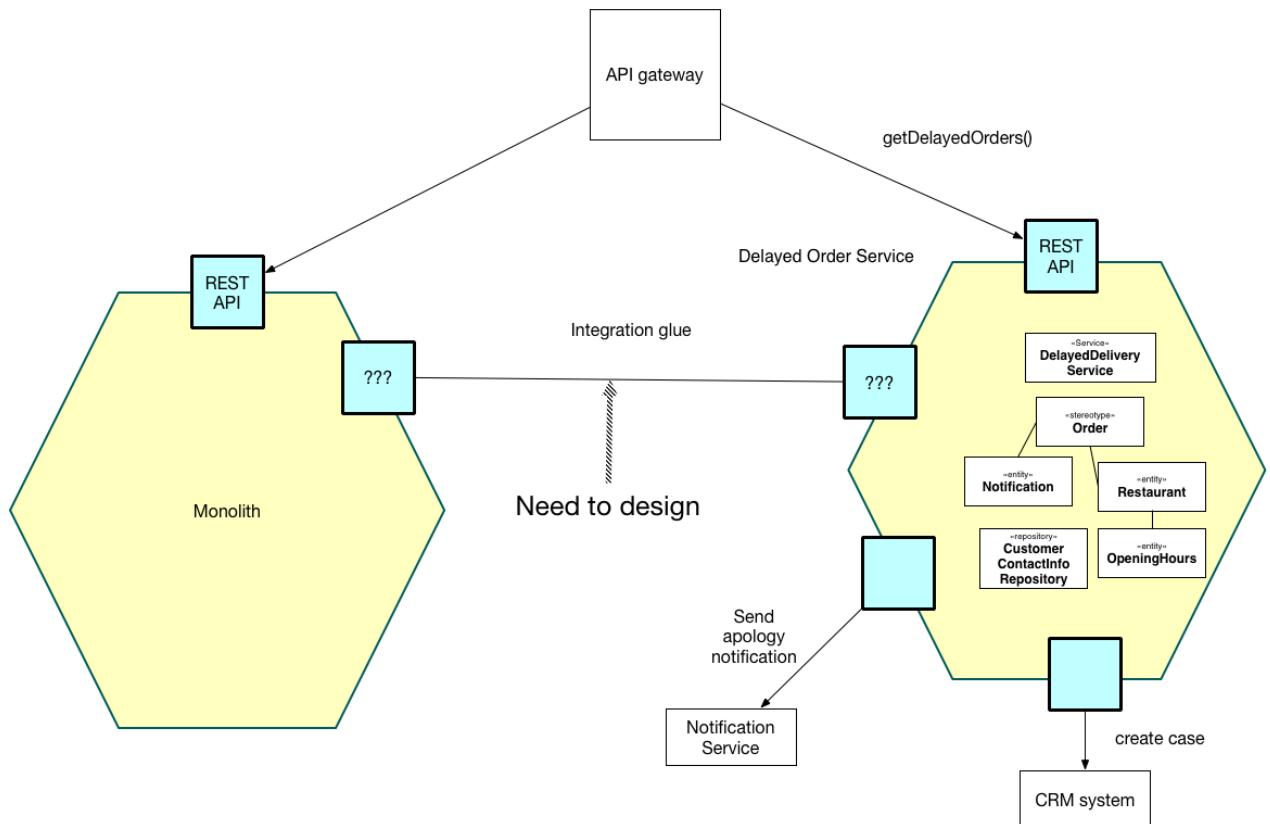
This new feature is fairly simple. The new code must track the state of each `Order` and if an `Order` can't be delivered as promised, notify the customer and customer support, by, for example, sending an email.

But how to, or perhaps more precisely, where to implement this new feature? One approach is to implement a new module in the monolith. The problem with this approach is that developing and testing this code will be difficult. What's more, this approach increases the size of the monolith and thereby makes monolith hell even worse. As I described earlier in section "[Implement new features as services](#)", when you are in a hole it's best to stop digging. Rather than make the monolith larger, a much better approach is to implement these new features as a service

12.4.1 The design of the DeDelayed Delivery Service

We'll implement this feature as service called the `Delayed Order Service`. Figure [12.14](#) shows the FTGO application's architecture after implementing this service. The application consists of the FTGO monolith, the new `Delayed Delivery Service` and an `API gateway`. The `Delayed Delivery Service` has API that defines a single query operation called `getDelayedOrders()`, which returns the currently delayed or undeliverable orders. The `API Gateway` routes `getDelayedOrders()` request to the service and all other requests to the monolith. The integration glue provides the `Delayed Order Service` with access to the monolith's data.

Figure 12.14. The design of the Delayed Delivery Service. The integration glue provides the Delayed Delivery Service with access to data owned by the monolith, such as the Order and Restaurant entities and the customer contact information.



The Delayed Order Service's domain model consists of various entities including `DelayedOrderNotification`, `Order`, and `Restaurant`. The core logic is implemented by the `DelayedOrderService` class. It's periodically invoked by a timer to find orders that won't be delivered on time. It does that by querying the `Orders` and `Restaurants`. If an `Order` can't be delivered on time the `DelayedOrderService` notifies the consumer and customer service.

The Delayed Order Service doesn't own the `Order`, and `Restaurant` entities. Instead, this data is replicated from the FTGO monolith. What's more the service doesn't store the customer contact information and instead retrieves it from the monolith. Let's look at the design of the integration glue that provides the Delayed Order Service with access to the monolith's data.

12.4.2 Designing the integration glue for the Delayed Delivery Service

Even though a service that implements a new feature defines its own entity classes, it usually accesses data that is owned by the monolith. The Delayed Delivery

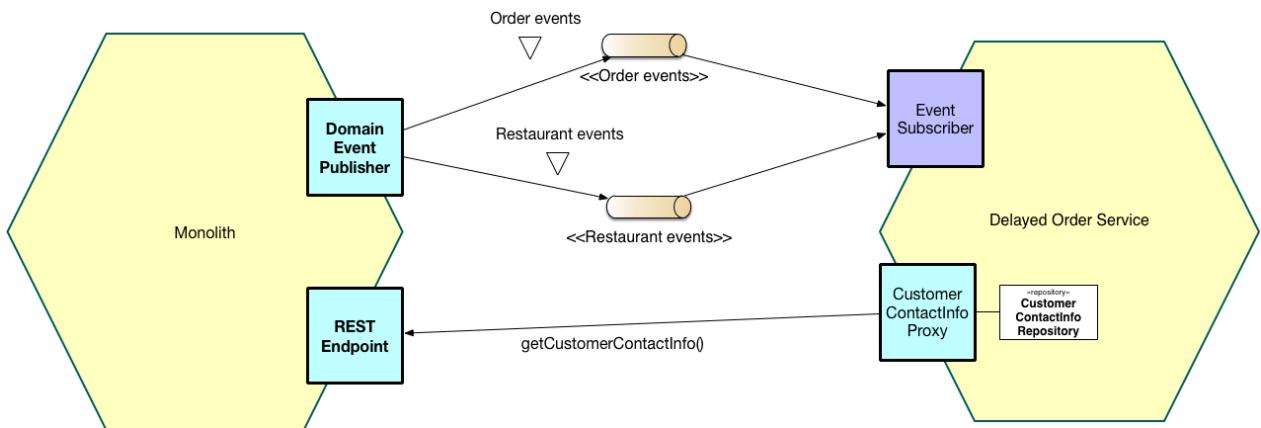
©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/microservices-patterns>

Service is no exception. It has an `DelayedOrderNotification` entity, which represents a notification that it has sent to the consumer. But, as I just mentioned, its `Order` and `Restaurant` entities replicate data from the FTGO monolith. It also needs to query user contact information in order to notify the user. Consequently, we need to implement integration glue that enables the `Delivery Service` to access the monolith's data.

Figure 12.15 shows the design of the integration glue. The FTGO monolith publishes `Order` and `Restaurant` domain events. The `Delivery Service` consumes these events and updates its replicas of those entities. The FTGO monolith implements a REST endpoint for querying the customer contact information. The `Delivery Service` calls this endpoint when it needs to notify a user that their order cannot be delivered on time.

Figure 12.15. The integration glue provides the Delayed Delivery Service with access to the data owned by the monolith.



Let's look at the design of each part of the integration starting with the REST API for retrieving customer contact information.

Querying customer contact information using the `CustomerContactInfoRepository`

As I described earlier in section [“Designing the integration glue”](#), there are a couple of different ways that a service, such as `Delayed Delivery Service` could read the monolith's data. The simplest option is for the `Delayed Order Service` to retrieve data using the monolith's query API. This approach makes sense when retrieving the User contact information. There aren't any latency or performance issues since the `Delayed Delivery Service` rarely needs to retrieve a user's contact information and the amount of data is quite small.

The `CustomerContactInfoRepository` is an interface that enables the `Delayed Delivery Service` to retrieve a consumer's contact info. It's implemented by a `CustomerContactInfoProxy`, which simply retrieves the user information by invoking the monolith's `getCustomerContactInfo()` REST endpoint.

Publishing and consuming Order and Restaurant domain events.

Unfortunately, it isn't practical for the Delayed Delivery Service to query the monolith for the state of all open Orders and the Restaurant hours. That's because it's inefficient to repeatedly transfer a large amount of data over the network. Consequently, the Delayed Delivery Service must use the second and more complex option and maintain a replica of the Orders and Restaurants by subscribing to events published by the monolith. It's important to remember that the replica is not a complete copy of the data from the monolith. It just stores a small subset of the attributes of the Order and Restaurant entities.

As described earlier in section "[How the monolith publishes and subscribes to domain events](#)", there are a couple of different ways that we can change the FTGO monolith so that it publishes Order and Restaurant domain events. One option is to modify all of the places in the monolith that update Orders and Restaurants to publish high-level domain events. The second option is to tail the transaction log to replica the changes as events. In this particular scenario, we simply need to synchronize the two databases. We don't require FTGO monolith to publish high-level domain events and so either approach is fine.

The Delayed Order Service implements event handlers that subscribe to events from the monolith and update its Order and Restaurant entities. The details of the event handlers depend on whether the monolith publishes specific high-level events or low-level change events. However, in either case, you can think of an event handler as translating an event in the monolith's bounded context to the update of an entity in the service's bounded context.

An important benefit of using a replica is that it enables the Delayed Order Service to efficiently query the orders and the restaurant opening hours. One drawback, however, is that it's more complex. Another drawback is that it requires the monolith to publish the necessary Order and Restaurant events. Fortunately, since the Delayed Delivery Service only needs what's essentially a subset of the columns of the ORDERS and RESTAURANT tables we shouldn't encounter the problems that I described in section "[How the monolith publishes and subscribes to domain events](#)".

Implementing new features, such as delayed order management, as a standalone service accelerates its development, testing and deployment. What's more, it enables you to implement the feature using a brand new technology stack instead of the monolith's older technology stack. It also stops the monolith from growing. Delayed order management management is just one of the many new features that are planned for the FTGO application. The FTGO team can implement many of these features as separate services.

Unfortunately, you cannot implement all changes as new service. Quite often you must make extensive changes to the monolith to implement new features or change existing features. Any development involving that the monolithic will mostly likely be slow and painful. If you want to accelerate the delivery of these features, you must break up the monolith by migrating functionality from the monolith into services. Let's look at how

to do that.

12.5 *Breaking apart the monolith: extracting delivery management*

In order to accelerate the delivery of features that are implemented by a monolith you need to break up the monolith into services. For example, let's imagine that you want to enhance FTGO delivery management by implementing a new routing algorithm. A major obstacle to developing delivery management is that it's entangled with order management and is part of the monolithic code base. Developing, testing and deploying delivery management is likely to be slow. In order to accelerate its development, you need to extract delivery management into a `Delivery Service`.

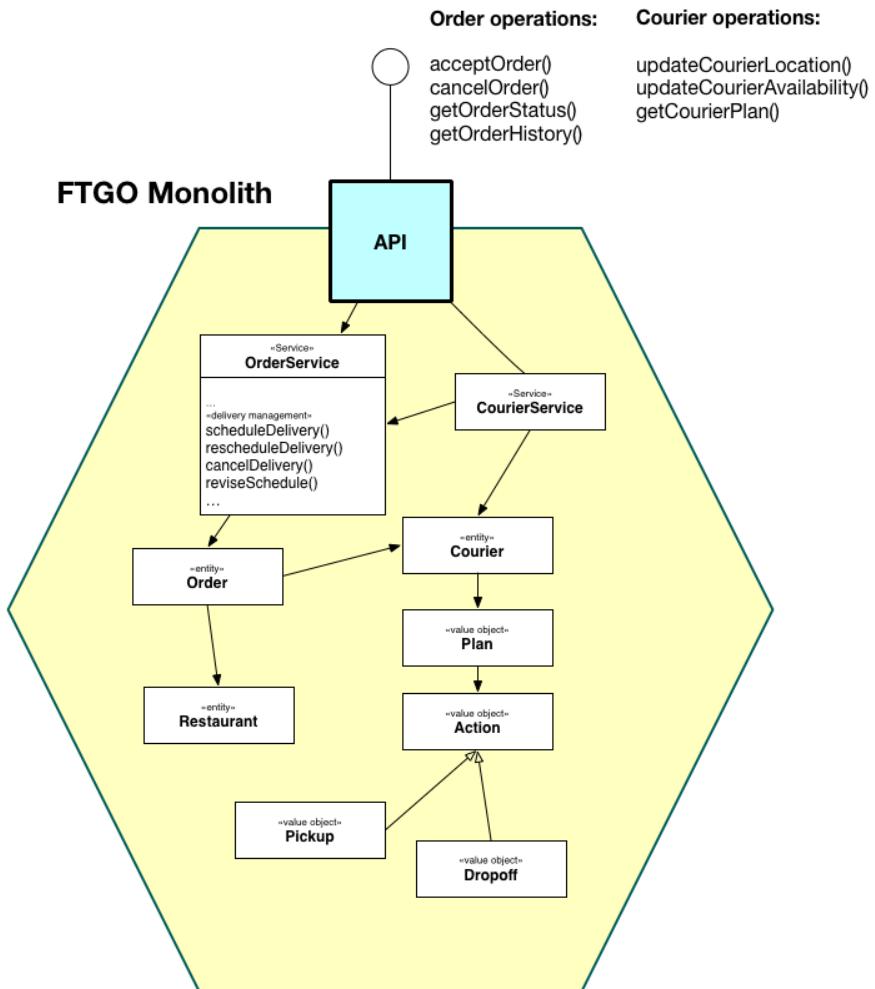
I start this section by describing delivery management and how it's currently embedded within the monolith. Next, I describe how the design of new, standalone `Delivery Service`, and its API. I then describe how the `Delivery Service` and the FTGO monolith collaborate. Finally, I describe some of the changes that we need to make to monolith to support the `Delivery Service`. Let's begin by reviewing the existing design.

12.5.1 *Overview the existing delivery management functionality*

Delivery management is responsible for scheduling the couriers that pick up orders at restaurants and deliver them to consumers. Each courier has a plan that is a schedule of pickup and deliver actions. A pickup action tells the `Courier` to pick up an order from a restaurant at a particular time. A deliver action tells the `Courier` to deliver an order to a consumer. The plans are revised whenever orders are placed, cancelled, or revised and as the location and availability of couriers changes.

Delivery management is one of the oldest parts of the FTGO application. As figure 12.16 shows, it's embedded within order management. Much of the code for managing deliveries is in `OrderService`. What's more, there is no explicit representation of a `Delivery`. It's embedded within the `Order` entity, which has various delivery-related fields, such as `scheduledPickupTime` and `scheduledDeliveryTime`.

Figure 12.16. Delivery management is entangled with order management within the FTGO monolith



Numerous commands implemented by the monolith invoke delivery management including:

- `acceptOrder()` - invoked when a restaurant accepts an order and commits to preparing it by a certain time. This operation invokes delivery management to schedule a delivery.
- `cancelOrder()` - invoked when a consumer cancels an order. If necessary, it cancels the delivery.
- `noteCourierLocationUpdated()` - invoked by the courier's mobile application to update the courier's location. It triggers the rescheduling of deliveries.
- `noteCourierAvailabilityChanged()` - invoked by the courier's mobile

application to update the courier's availability. It triggers the rescheduling of deliveries.

Also, various queries retrieve data maintained by delivery management including:

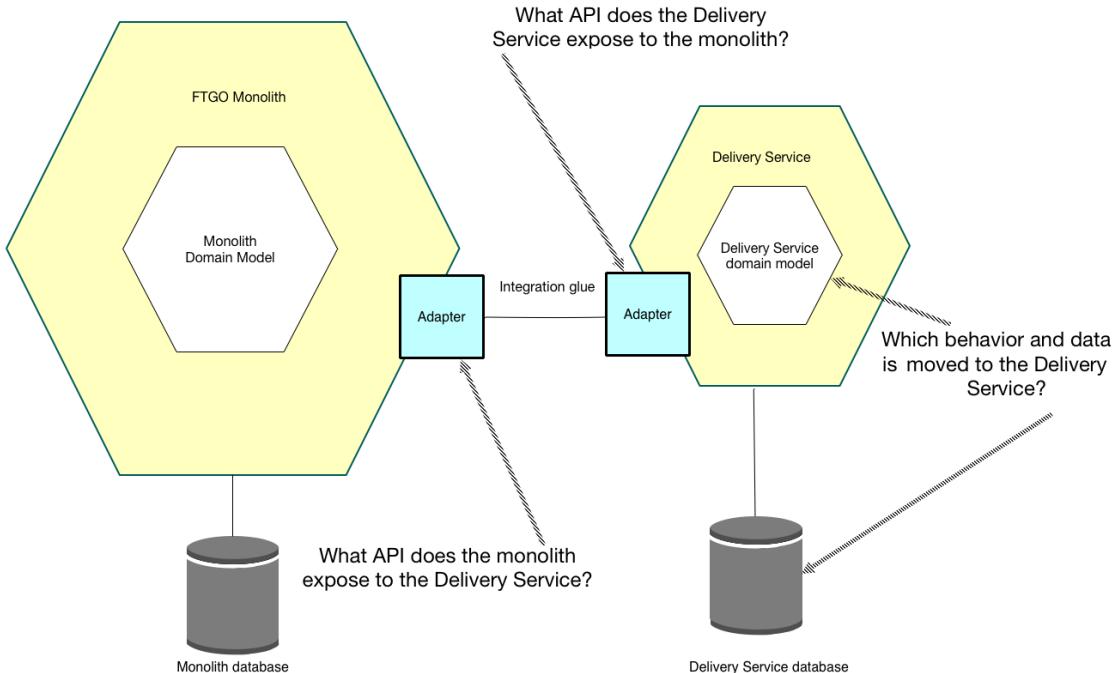
- `getCourierPlan()` - invoked by the courier's mobile application and returns the courier's plan
- `getOrderStatus()` - returns the order's status, which includes delivery related information, such as the assigned courier and the its ETA.
- `getOrderHistory()` - returns similar information as `getOrderStatus()` except about multiple orders

Quite often, what is extracted into a service is, as I described earlier in section [“Extract business capabilities into services”](#), an entire vertical slice with controllers at the top and database tables at the bottom. We could consider the Courier-related commands and queries to be part of delivery management. After all, delivery management creates the courier plans and is the primary consumer of the Courier location and availability information. However, in order to minimize the development effort we will leave those operations in the monolith and just extract the core of the algorithm. Consequently, the first iteration of the `Delivery Service` will not expose a publicly accessible API. Instead, it will only be invoked by the monolith. Let's now explore the design of the `Delivery Service`.

12.5.2 Overview of the Delivery Service

The proposed new `Delivery Service` is responsible for scheduling, rescheduling and cancelling deliveries. Figure [12.17](#) show a very high-level view of the architecture of the FTGO application after extracting the `Delivery Service`. The architecture consists of the FTGO monolith and `Delivery Service`. They collaborate using the integration glue, which consists of APIs in both the service and monolith. The `Delivery Service` has its own domain model and database.

Figure 12.17. The high-level view of the FTGO application after extracting the Delivery Service. The FTGO monolith and Delivery Service collaborate using the integration glue, which consists of APIs in each of them. The two key decisions that need to be made are which functionality and data are moved to the Delivery Service and how do the monolith and the Delivery Service collaborate via APIs.



In order to flesh out this architecture and determine the service's domain model, we need to answer the following questions:

1. Which behavior and data is moved to the Delivery Service?
2. What API does the Delivery Service expose to the monolith?
3. What API does the monolith expose to the Delivery Service?

These issues are inter-related since the distribution of responsibilities between the monolith and the service affects the APIs. For instance, the Delivery Service will need to invoke an API provided by the monolith to access the data in the monolith's database and vice versa. Later I'll describe the design of the integration glue that enables the Delivery Service and the FTGO monolith to collaborate. But first, let's look at the design of the Delivery Service's domain model.

12.5.3 Designing the Delivery Service domain model

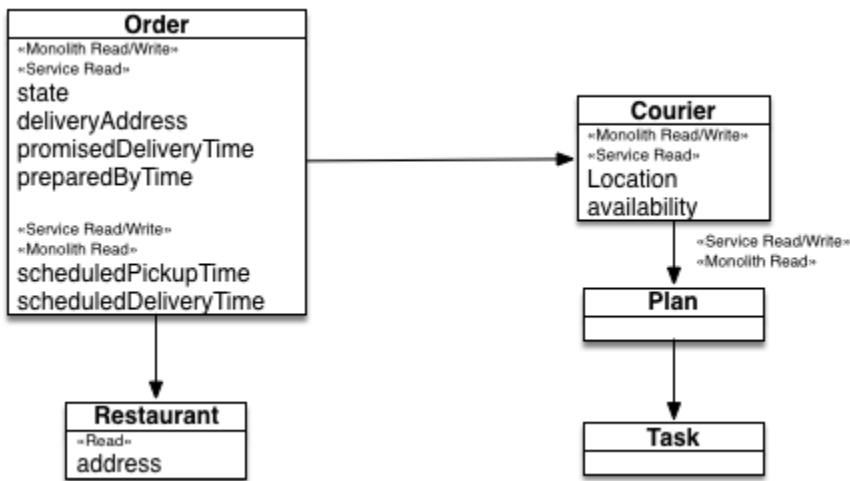
In order to be able to extract delivery management, we first need to identify the classes that implement it. Once we have done that we can then decide which classes to move to the Delivery Service to form its domain logic. In some cases, we will need to split

classes. We will also need to decide which data to replicate between the service and the monolith. Let's start by identifying the classes that implement delivery management.

Identifying which entities and their fields are part of delivery management

The first step in the process of designing the Delivery Service is to carefully review the delivery management code and identify the participating entities and their fields. Figure 12.18 shows the entities and fields that are part of delivery management. Some fields are inputs to the delivery scheduling algorithm and others are the outputs. The figure also shows which of those fields are also used by other functionality implemented by the monolith.

Figure 12.18. The entities and fields that are accessed by delivery management and other functionality implemented by the monolith. A field can be read or written or both. It can be accessed by delivery management, the monolith or both.



The delivery scheduling algorithm reads various attributes including the Orders' restaurant, `promisedDeliveryTime` and `deliveryAddress` and the Couriers' location, availability and current plans. It updates the Couriers' plans and the Orders' `scheduledPickupTime` and `scheduledDeliveryTime`. As you can see, the fields used by delivery management are also used by the monolith.

Deciding which data to migrate to the Delivery Service

Now that we have identified which entities and fields participate in delivery management, the next step is to decide which of them we should move to the service. In an ideal scenario, the data accessed by the service is used exclusively by the service and so we could simply move that data to the service and be done. Sadly, it is rarely that simple, and this situation is no exception. All of the entities and fields used by the delivery management are also used by other functionality implemented by the monolith.

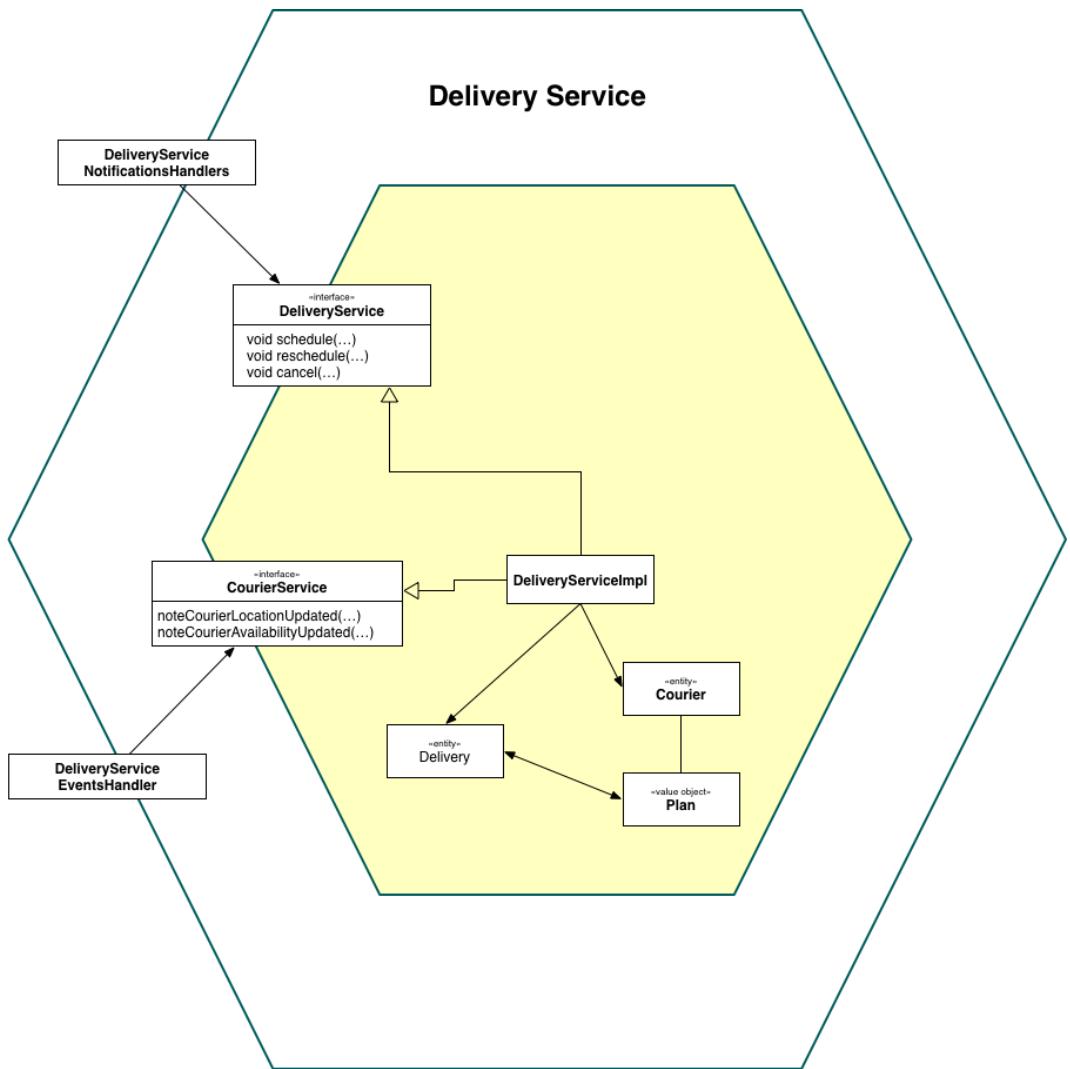
As a result, when determining which data to move to the service, we need to keep in mind two issues. The first is how does the service access the data that remains in the monolith? The second is how does the monolith access data that is moved to the service? Also, as I described earlier in section [“Designing how the service and the monolith collaborate”](#), we need to carefully consider how to maintain data consistency between the service and the monolith.

The essential responsibility of the Delivery Service is managing courier plans and updating the Order's scheduledPickupTime and scheduledDeliveryTime fields. It makes sense, therefore, for it to own those fields. We could also move the Courier.location and Courier.availability fields to the Delivery Service. However, since we are trying to make the smallest possible change, we will leave those fields in the monolith for now.

The design of the Delivery Service domain logic

Figure 12.19 shows the design of the Delivery Service's domain model. The core of the service consists of domain classes, such as Delivery and Courier. The DeliveryServiceImpl class is the entry point into the delivery management business logic. It implements the DeliveryService and CourierService interfaces, which are invoked by the DeliveryServiceEventsHandler and DeliveryServiceNotificationsHandlers, which I describe later in this section.

Figure 12.19. The design of the Delivery Service's domain model



The delivery management business logic is mostly code copied from the monolith. For example, we will copy the Order entity from the monolith to the Delivery Service, rename it to Delivery and delete all of fields except those used by delivery management. We will also copy the Courier entity and delete most of its fields. In order to develop the domain logic for the Delivery Service we will need to untangle the code from the monolith. We will need to break numerous dependencies, which is likely to be time consuming. Once again, its a lot easier to refactor code when using a statically typed language since the compiler will be your friend.

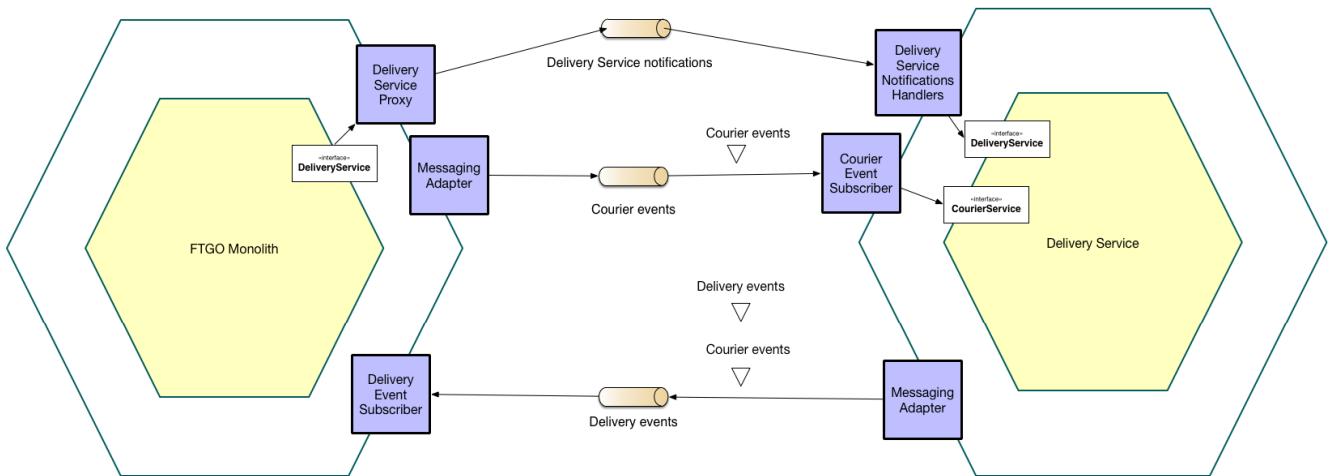
The Delivery Service is not a standalone service. Let's look at the design of the integration glue that enables the Delivery Service and the FTGO monolith to

collaborate.

12.5.4 The design of the Delivery Service integration glue

The FTGO monolith needs to invoke the Delivery Service to manage deliveries. The monolith also needs to exchange data with the Delivery Service. This collaboration is enabled by the integration glue. Figure 12.20 shows the design of the Delivery Service integration glue. The Delivery Service has a delivery management API. It also publishes Delivery and Courier domain events. The FTGO monolith publishes Courier domain events.

Figure 12.20. The design of the Delivery Service integration glue. The Delivery Service has a delivery management API. The service and the FTGO monolith synchronize data by exchanging domain events.



Let's look at the design of each part of the integration glue starting with the Delivery Service's API for managing deliveries.

The design of the Delivery Service API

The Delivery Service must provide an API that enables the monolith to schedule, revise and cancel deliveries. As you have seen throughout this book, the preferred approach is to use asynchronous messaging since it promotes loose coupling and increases availability. One approach is for the Delivery Service to subscribe to Order domain events published by the monolith. Depending on the type of the event, it creates, revises and cancels a Delivery. A benefit of this approach is that the monolith doesn't need to explicitly invoke the Delivery Service. The drawback of

relying on domain events is that it requires the Delivery Service to know how each Order event impacts the corresponding Delivery.

A better approach is for Delivery Service to implement a notification-based API that enables the monolith to explicitly tell the Delivery Service to create, revise and cancel deliveries. The Delivery Service's API consists of a message notification channel and three message types: ScheduleDelivery, ReviseDelivery or CancelDelivery. A notification message contains the Order information needed by the Delivery Service. For example, a ScheduleDelivery notification contains the pickup time and location and the delivery time and location. An important benefit of this approach is that the Delivery Service does not have detailed knowledge of the Order lifecycle. It's entirely focussed on managing deliveries and has no knowledge of orders.

This API isn't the only way that the Delivery Service and the FTGO monolith collaborate. They also need to exchange data.

How the Delivery Service accesses the FTGO monolith's data

The Delivery Service needs to access the Courier location and availability data, which is owned by the monolith. Since that's potentially a large amount of data, it's not practical for the service to repeatedly query the monolith. Instead, a better approach is for the monolith to replicate the data to the Delivery Service by publishing Courier domain events, CourierLocationUpdated and CourierAvailabilityUpdated. The Delivery Service has a CourierEventSubscriber, which subscribes to the domain events and updates its version of the Courier. It might also trigger the rescheduling of deliveries.

How the FTGO monolith accesses the Delivery Service data

The FTGO monolith needs to read the data that has been moved to the Delivery Service, such as the Courier plans. In theory, the monolith could query the service but that requires extensive changes to the monolith. For the time being, it's easier to leave the monolith's domain model and database schema unchanged and replicate data from the service back to the monolith.

The easiest way to accomplish this is for the Delivery Service to publish Courier and Delivery domain events. The service publishes a CourierPlanUpdated event when it updates a Courier's plan and DeliveryScheduleUpdate event when it updates a Delivery. The monolith consumes these domain events and updates its database. Now that we have looked at how the FTGO monolith and the Delivery Service interact, let's look at how to change the monolith.

12.5.5 *Changing the FTGO monolith to interact with the Delivery Service*

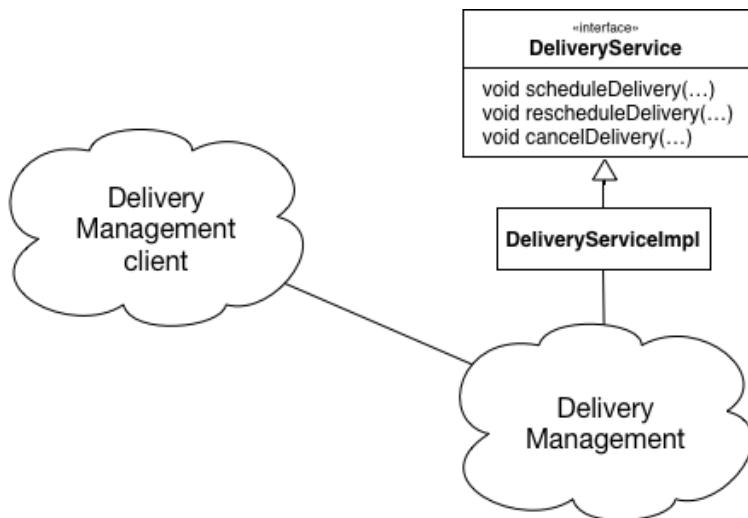
In many ways, implementing the Delivery Service is the easier part of the extraction process. Modifying the FTGO monolith is much more difficult. Fortunately, replicating data from the service back to the monolith reduces the size of the change. However, we

still need to change the monolith to manage deliveries by invoking the `DeliveryService`. Let's look at how to do that.

Defining a `DeliveryService` interface

The first step is to encapsulate the delivery management code with a Java interface corresponding to the messaging-based API I defined earlier. This interface, which is shown in figure 12.21, defines methods for scheduling, rescheduling and cancelling deliveries. Eventually, we will implement this interface with a proxy that sends messages to the delivery service. But initially, we'll implement this API with a class that calls the delivery management code.

Figure 12.21. The first step is to define `DeliveryService`, which is a coarse-grained, remotable API for invoking the delivery management logic

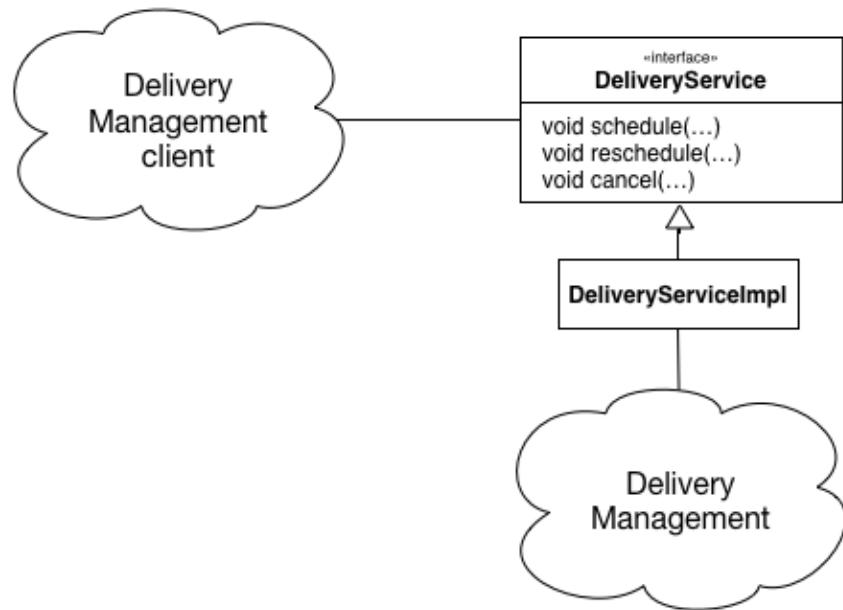


The `DeliveryService` interface is a coarse-grained interface that is well suited to being implemented by an IPC mechanism. It defines `schedule()`, `reschedule()`, and `cancel()` methods, which correspond to the notification message types I defined earlier.

Refactoring the monolith to call the `DeliveryService` interface

Next, as figure 12.22 shows, we need to identify all of the places in the FTGO monolith that invoke delivery management and change them to use the `DeliveryService` interface. This might take some time and is one of the most challenging aspects of extracting a service from the monolith.

Figure 12.22. The second step is to change the FTGO monolith to invoke delivery management via the DeliveryService interface

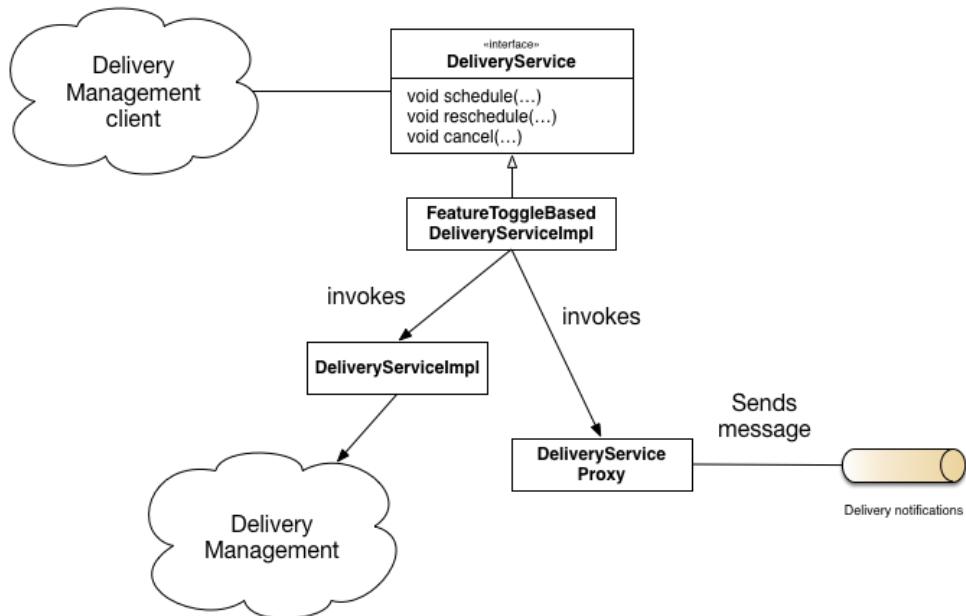


It certainly helps if the monolith is written in a statically typed language, such as Java, since the tools do a better job of identifying dependencies. If not, then hopefully you have some automated tests with sufficient coverage of the parts of the code that need to be changed.

Implementing the DeliveryService interface

The final step is to replace the `DeliveryServiceImpl` class with a proxy that sends notification messages to the standalone `Delivery Service`. However, rather than discard the existing implementation right away, we'll use a design, which is shown in figure 12.23, that enables monolith to dynamically switch between the existing implementation and the `Delivery Service`. We'll implement the `DeliveryService` interface with a class that uses a dynamic feature toggle to determine whether to invoke the existing implementation or the `Delivery Service`.

Figure 12.23. The final step is to implement the DeliveryService with a proxy class that class that sends messages the Delivery Service. A feature toggle controls whether the FTGO monolith uses the old implementation or the new Delivery Service.



Using a feature toggle significantly reduces the risk of rolling out the `Delivery Service`. We can deploy the `Delivery Service` and test it. And then, once we are sure that it works we can flip the toggle to route traffic to it. If we then discover that the `Delivery Service` isn't working as expected we can switch back to the old implementation.

About feature toggles

Feature toggles, a.k.a. known as feature flags, let you deploy code changes without necessarily releasing them to users. They also enable you to dynamically change the behavior of the application with deploying new code. The article by Martin Fowler ⁷⁶ is an excellent overview of the topic.

Once we are sure that the `Delivery Service` is working as expected we can then remove the delivery management code from the monolith.

The `Delivery Service` and the `Delayed Order Service` are examples of the services that the FTGO team will develop during their journey to the microservice architecture. Where they go next after implementing these services depends on the priorities of the business. One possible path is to extract the `Order History Service`, which I described in chapter 7. Extracting this service partially eliminates the need for the `Delivery Service` to replicate data back to the monolith.

⁷⁶ <https://martinfowler.com/articles/feature-toggles.html>

After implement the Order History Service the FTGO team could then extract the services in the order that I described section "Sequencing the extraction of services to avoid implementing compensating transactions in the monolith": Order Service, Consumer Service, Restaurant Order Service, etc. As the FTGO team extracts each service, the maintainability and testability of their application gradually improves and their development velocity increases.

12.6 Summary

- Before migrating to a microservice architecture it's important to be sure that your software delivery problems are a result of having outgrown your monolithic architecture. You might be able to accelerate delivery by improving your software development process.
- It's important to migrate to microservices by incrementally developing a strangler application. A strangler application is a new application consisting of microservices that you build around the existing monolithic application. You should demonstrate value early and often in order to ensure that the business supports the migration effort.
- A great way to introduce microservices into your architecture is to implement new features as services. It enables you to quickly and easily develop a feature using a modern technology and development process. It's a good way to quickly demonstrate the value of migrating to microservices.
- One way to break up the monolith is to separate presentation tier from the back end, which results in two smaller monoliths. Although, it's not a huge improvement it does mean that you can deploy each monolith independently. It allows, for example, the UI team to iterate more easily on the UI design without impacting the backend.
- The main way to break up the monolith is by incrementally migrating functionality from the monolith into services. It's important to focus on extracting the services that provide the most benefit. For example, you will accelerate development if you extract a service that implements functionality that is being actively developed.
- Newly developed services almost always have to interact with the monolith. A service often needs to access a monolith's data and invoke its functionality. The monolith sometimes needs access to a service's data and invoke its functionality. To implement this collaboration, you need to develop integration glue, which consists of inbound and outbound adapters in the monolith.
- In order to prevent the monolith's domain model from polluting the service's domain model, the integration glue should use an anti-corruption layer. It's a layer of software, which translates between domain models.
- One way to minimize the impact on the monolith of extracting a service is to replicate the data that was moved to the service back to the monolith's database. Since the monolith's schema is left unchanged, it eliminates the need to make potentially widespread changes to the monolith code base.

- Developing a service often requires you to implement sagas that involve the monolith. However, it can be challenging to implement a compensatable transaction that requires making widespread changes to the monolith. Consequently, you sometimes need to carefully sequence the extraction of services to avoid implementing compensatable transactions in the monolith.
- When refactoring to a microservice architecture, you need to simultaneously support the monolithic application's existing security mechanism, which is often based on an in-memory session, and the token-based, security mechanism used by the services. Fortunately, a simple solution is to modify the monolith's login handler to generate a cookie containing security token, which is then forwarded to the services by the API gateway.