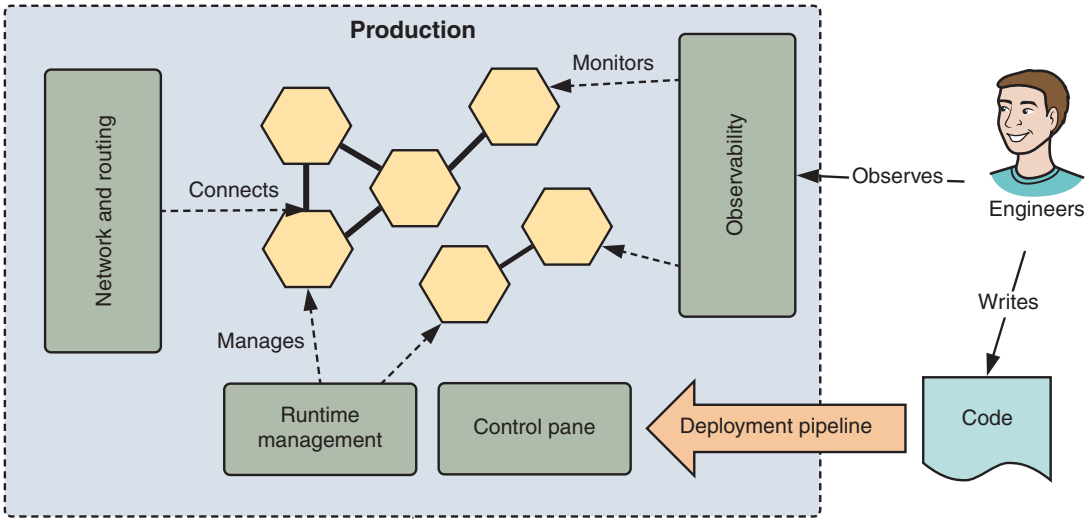
Microservices IN ACTION

Morgan Bruce
Paulo A. Pereira

MANNING

A microservice production environment



A microservice production environment has several components: a deployment target, a deployment pipeline, runtime management, networking features, and support for observability. In this book, we'll teach you about these components and how you can use them to build a stable, modern microservice application.

Microservices in Action

Microservices in Action

MORGAN BRUCE
PAULO A. PEREIRA

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Acquisitions editor:	Michael Stephens
Development editor:	Karen Miller
Technical development editor:	Karsten Strøbæk
Review editor:	Aleksander Dragosavljević
Project editor:	Anthony Calcara
Copy editor:	Carl Quesnel
Proofreader:	Keri Hales
Technical proofreader:	John Guthrie
Typesetter:	Happenstance Type-O-Rama
Cover designer:	Marija Tudor

ISBN 9781617294457
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 — DP — 23 22 21 20 19 18

brief contents

PART 1 THE LAY OF THE LAND 1

1 ■ Designing and running microservices 3
2 ■ Microservices at SimpleBank 28

PART 2 DESIGN 49

3 ■ Architecture of a microservice application 51
4 ■ Designing new features 75
5 ■ Transactions and queries in microservices 105
6 ■ Designing reliable services 129
7 ■ Building a reusable microservice framework 159

PART 3 DEPLOYMENT.....185

8 ■ Deploying microservices 187
9 ■ Deployment with containers and schedulers 214
10 ■ Building a delivery pipeline for microservices 243

PART 4 OBSERVABILITY AND OWNERSHIP 267

11 ■ Building a monitoring system 269
12 ■ Using logs and traces to understand behavior 296
13 ■ Building microservice teams 325

contents

preface xv
acknowledgments xvii
about this book xix
about the authors xxii
about the cover illustration xxiii

PART 1 THE LAY OF THE LAND 1

1 *Designing and running microservices* 3

1.1 What is a microservice application? 4
Scaling through decomposition 6 ▪ *Key principles* 7 ▪ *Who uses microservices?* ▪ *Why are microservices a good choice?* 12

1.2 What makes microservices challenging? 14
Design challenges ▪ *Operational challenges* 17

1.3 Microservice development lifecycle 18
Designing microservices ▪ *Deploying microservices* ▪ *Observing microservices* 24

1.4 Responsible and operationally aware engineering culture 26

2 *Microservices at SimpleBank 28*

2.1 What does SimpleBank do? 29

2.2 Are microservices the right choice? 30

Risk and inertia in financial software 31 ▪ Reducing friction and delivering sustainable value 31

2.3 Building a new feature 32

Identifying microservices by modeling the domain ▪ Service collaboration 3 ▪ Service choreography 37

2.4 Exposing services to the world 39

2.5 Taking your feature to production 40

Quality-controlled and automated deployment 42

Resilience 43Transparency 43

2.6 Scaling up microservice development 45

Technical divergence 45 ▪ Isolation 46

2.7 What's next? 47

PART 2 DESIGN.....49

3 *Architecture of a microservice application 51*

3.1 Architecture as a whole 52

From monolith to microservices 52 ▪ The role of an architect 54 ▪ Architectural principles 54 ▪ The four tiers of a microservice application 55

3.2 A microservice platform 56

Mapping your runtime platform 57

3.3 Services 58

Capabilities 58 ▪ Aggregation and higher order services 59 ▪ Critical and noncritical paths 60

3.4 Communication 60

When to use synchronous messages 61 ▪ When to use asynchronous messages 62 ▪ Asynchronous communication patterns 63 ▪ Locating other services 65

3.5The application boundary 66

API gateways 68 ▪ Backends for frontends 69 ▪ Consumer-driven gateways 70

3.6 Clients 71

Frontend monoliths 71 ▪ Micro-frontends 72

4 *Designing new features 75*

4.1 A new feature for SimpleBank 76

4.2 Scoping by business capabilities 78

Capabilities and domain modeling 78 ▪ Creating investment strategies 79 ▪ Nested contexts and services 86 ▪ Challenges and limitations 87

4.3 Scoping by use case 87

Placing investment strategy orders 88 ▪ Actions and stores 92 Orchestration and choreography 94

4.4 Scoping by volatility 94

4.5 Technical capabilities 96

Sending notifications 96 ▪ When to use technical capabilities 98

4.6 Dealing with ambiguity 99

Start with coarse-grained services 99 ▪ Prepare for further decomposition 100 ▪ Retirement and migration 100

4.7 Service ownership in organizations 103

5 *Transactions and queries in microservices 105*

5.1 Consistent transactions in distributed applications 106

Why can't you use distributed transactions? 107

5.2 Event-based communication 108

Events and choreography 109

5.3 Sagas 111

Choreographed sagas 112 ▪ Orchestrated sagas 115 Interwoven sagas 117 ▪ Consistency patterns 118 ▪ Event sourcing 119

5.4 Queries in a distributed world 120

Storing copies of data 122 ▪ Separating queries and commands 123 ▪ CQRS challenges 125 ▪ Analytics and reporting 127

5.5 Further reading 128

6 *Designing reliable services 129*
6.1 Defining reliability 130
6.2 What could go wrong? 132
Sources of failure 133 ▪ Cascading failures 136
6.3 Designing reliable communication 139
Retries 140 ▪ Fallbacks 143 ▪ Timeouts 145 ▪ Circuit breakers 146 ▪ Asynchronous communication 149
6.4 Maximizing service reliability 150
Load balancing and service health 150 ▪ Rate limits 152
Validating reliability and fault tolerance 152
6.5 Safety by default 156
Frameworks 156 ▪ Service mesh 157

7 *Building a reusable microservice framework 159*
7.1 A microservice chassis 160
7.2 What's the purpose of a microservice chassis? 163
Reduced risk 164 ▪ Faster bootstrapping 164
7.3 Designing a chassis 165
Service discovery 167 ▪ Observability 171 ▪ Balancing and limiting 177
7.4 Exploring the feature implemented using the chassis 180
7.5 Wasn't heterogeneity one of the promises of microservices? 182

PART 3 DEPLOYMENT 185

8 *Deploying microservices 187*
8.1 Why is deployment important? 188
Stability and availability 189
8.2 A microservice production environment 189
Features of a microservice production environment 190
Automation and speed 191

8.3 Deploying a service, the quick way 191
Service startup 192 ▪ Provisioning a virtual machine 192 Run multiple instances of your service 194 ▪ Adding a load balancer 196 ▪ What have you learned? 198

8.4 Building service artifacts 199
What's in an artifact? 200 ▪ Immutability 201 ▪ Types of service artifacts 202 ▪ Configuration 206

8.5 Service to host models 207
Single service to host 207 ▪ Multiple static services per host 208 ▪ Multiple scheduled services per host 209

8.6 Deploying services without downtime 210
Canaries and rolling deploys on GCE 211

9 *Deployment with containers and schedulers 214*

9.1 Containerizing a service 215
Working with images 216 ▪ Building your image 218 Running containers 220 ▪ Storing an image 223

9.2 Deploying to a cluster 224
Designing and running pods 226 ▪ Load balancing 228 A quick look under the hood 230 ▪ Health checks 233 Deploying a new version 235 ▪ Rolling back 241 Connecting multiple services 241

10 *Building a delivery pipeline for microservices 243*

10.1 Making deploys boring 244
A deployment pipeline 244

10.2 Building a pipeline with Jenkins 246
Configuring a build pipeline 247 ▪ Building your image 251 Running tests 252 ▪ Publishing artifacts 254 ▪ Deploying to staging 255 ▪ Staging environments 258 ▪ Deploying to production 259

10.3 Building reusable pipeline steps 262
Procedural versus declarative build pipelines 263

10.4 Techniques for low-impact deployment and feature release 264
Dark launches 264 ▪ Feature flags 265

PART 4 OBSERVABILITY AND OWNERSHIP 267

11 *Building a monitoring system* 269

11.1 A robust monitoring stack 270

Good monitoring is layered 270 ▪ *Golden signals* 272 *Types of metrics* 273 ▪ *Recommended practices* 274

11.2 Monitoring SimpleBank with Prometheus and Grafana 275

Setting up your metric collection infrastructure 276 ▪ *Collecting infrastructure metrics—RabbitMQ* 282 ▪ *Instrumenting SimpleBank's place order* 285 ▪ *Setting up alerts* 287

11.3 Raising sensible and actionable alerts 291

Who needs to know when something is wrong? 292 ▪ *Symptoms, not causes* 292

11.4 Observing the whole application 293

12 *Using logs and traces to understand behavior* 296

12.1 Understanding behavior across services 297

12.2 Generating consistent, structured, human-readable logs 300

Useful information to include in log entries 300 ▪ *Structure and readability* 301

12.3 Setting up a logging infrastructure for SimpleBank 303

ELK- and Fluentd-based solution 304 ▪ *Setting up your logging solution* 306 ▪ *Configure what logs to collect* 308 ▪ *Finding a needle in the haystack* 311 ▪ *Logging the right information* 313

12.4 Tracing interactions between services 313

Correlating requests: traces and spans 314 ▪ *Setting up tracing in your services* 315

12.5 Visualizing traces 320

13 *Building microservice teams* 325

13.1 Building effective teams 326

Conway's Law 327 ▪ *Principles for effective teams* 328

13.2 Team models 330

Grouping by function 330 ▪ *Grouping across functions* 332 *Setting team boundaries* 334 ▪ *Infrastructure, platform, and product* 335 ▪ *Who's on-call?* 337 ▪ *Sharing knowledge* 338

13.3 Recommended practices for microservice teams 340

Drivers of change in microservices 340 ▪ The role of architecture 341 ▪ Homogeneity versus technical flexibility 343 ▪ Open source model 343 ▪ Design review 345 ▪ Living documentation 346 ▪ Answering questions about your application 347

13.4 Further reading 347

appendix Installing Jenkins on Minikube 349

index 357

preface

Over the past five years, the microservice architectural style—structuring applications as fine-grained, loosely coupled, and independently deployable services—has become increasingly popular and increasingly feasible for engineering teams, regardless of company size.

For us, working on microservice projects at Onfido was a revelation, and this book records many of the things we learned along the way. By breaking apart our product, we could ship faster and with less friction, instead of tripping over each other's toes in a large, monolithic codebase. A microservice approach helps engineers build applications that can evolve over time, even as product complexity and team size grow.

Originally, we set out to write a book about our real-world experience running microservice applications. As we scoped the book, that mission evolved, and we decided to distill our experience of the full application lifecycle—designing, deploying, and operating microservices—into a broad and practical review. We've picked tools to illustrate these techniques—such as Kubernetes and Docker—that are popular and go hand in hand with microservice best practice, but we hope that you can apply the lessons within regardless of which language and tools you ultimately use to build applications.

We sincerely hope you find this book a valuable reference and guide—and that the knowledge, advice, and examples within help you build great products and applications with microservices.

acknowledgments

In its evolution over the past year and a half, this book has grown from an idea to write a small book on deploying services to a substantial work covering a wide swath of microservice development topics—from design to communication to deployment and operation. It has been our privilege to work with so many talented people in delivering a book that we truly hope will be useful, both to those who are starting to adopt this type of architecture and to those who already are using it.

I would like to thank my family, in particular Rosa and Beatriz, my wife and daughter, who put up with the absences of a husband and father. I would also like to thank Morgan, my coauthor and colleague. He has been crucial in providing guidance and clarity from day one. Thank you!

—Paulo

This book wouldn't have been possible to write without the patience and support of my family, who gracefully tolerated far too many weekends, evenings, and holidays with me sitting in front of a laptop. I'd also like to thank my parents, Heather and Allan, who taught me a love of reading, without which I wouldn't be writing this today. And lastly, thanks Paulo! You encouraged me to start this project with you, and although the way was sometimes challenging, I've learned so much from the journey.

—Morgan

Together, we would like to thank:

- Karen and Dan, our development editors, who were tireless, week after week, in providing support and advice to help us write the best possible book
- Karsten Strøbæk, our technical development editor, for his critical eye and generous feedback

- Michael Stephens, for his faith in us, and Marjan Bace, for his help in shaping our book into something compelling for Manning's readers
- The many other people we've worked with at Manning, who formed such a professional and talented team, without whom this book would have never been possible
- Lastly, our reviewers, whose feedback and help improving our book we deeply appreciated, including Akshat Paul, Al Krinker, Andrew Miles, Andy Miles, Antonio Pessolano, Bachir Chihani, Christian Bach, Christian Thoudahl, Vittal Damaraju, Deepak Bhaskaran, Evangelos Bardis, John Guthrie, Lorenzo De Leon, Łukasz Witczak, Maciej Jurkowski, Mike Jensen, Shobha Iyer, Srihari Sridharan, Steven Parr, Thorsten Weber, and Tiago Boldt Sousa

about this book

Microservices in Action is a practical book about building and deploying microservice-based applications. Written for developers and architects with a solid grasp of service-oriented development, it tackles the challenge of putting microservices into production. You'll begin with an in-depth overview of microservice design principles, building on your knowledge of traditional systems. Then you'll start creating a reliable road to production. You'll explore examples using Kubernetes, Docker, and Google Container Engine as you learn to build clusters and maintain them after deployment.

The techniques in this book should apply to developing microservices in most popular programming languages. We decided to use Python as the primary language for this book because its low-ceremony style and terse syntax lend themselves to clear and explicit examples. Don't worry if you're not too familiar with Python—we'll guide you through running the examples.

How this book is organized: a roadmap

Part 1 of this book gives a brief introduction to microservices, exploring the properties and benefits of microservice-based systems and the challenges you may face in their development.

Chapter 1 introduces the microservice architecture. We examine the benefits and drawbacks of the microservice approach and explain the key principles of microservice development. Lastly, we introduce the design and deployment challenges we'll cover throughout this book.

Chapter 2 applies the microservice approach to an example domain—SimpleBank. We design a new feature with microservices and examine how to make that feature ready for production.

In part 2, we explore the architecture and design of microservice applications.

Chapter 3 walks through the architecture of a microservice application, covering four layers: platform, service, boundary, and client. The goal of this chapter is to give the reader a big-picture model that they can use when working to understand any microservice system.

Chapter 4 covers one of the hardest parts of microservice design: how to decide on service responsibilities. This chapter lays out four approaches to modeling—business capabilities, use cases, technical capabilities, and volatility—and, using examples from SimpleBank, explores how to make good design decisions, even when boundaries are ambiguous.

Chapter 5 explores how to write business logic in distributed systems, where transactional guarantees no longer apply. We introduce the reader to different transaction patterns, such as sagas, and query patterns, such as API composition and CQRS.

Chapter 6 covers reliability. Distributed systems can be more fragile than monolithic applications, and communication between microservices requires careful consideration to avoid availability issues, downtime, and cascading failures. Using examples in Python, we explore common techniques for maximizing application resiliency, such as rate limits, circuit breakers, health checks, and retries.

In chapter 7, you'll learn how to design a reusable microservice framework. Consistent practices across microservices improve overall application quality and reliability and reduce time to development for new services. We provide working examples in Python.

In part 3, we look at deployment best practices for microservices.

Chapter 8 emphasizes the importance of automated continuous delivery in microservice applications. Within this chapter, we take a single service to production—on Google Compute Engine—and from that example learn about the importance of immutable artifacts and the pros and cons of different microservice deployment models.

Chapter 9 introduces Kubernetes, a container scheduling platform. Containers, combined with a scheduler like Kubernetes, are a natural and elegant fit for running microservices at scale. Using Minikube, you'll learn how to package a microservice and deploy it seamlessly to Kubernetes.

In chapter 10, you'll build on the example in the previous chapter to construct an end-to-end delivery pipeline using Jenkins. You'll script a pipeline with Jenkins and Groovy that takes new commits to production rapidly and reliably. You'll also learn how to apply consistent deployment practices to a microservice fleet.

In this book's final part, we explore observability and the human side of microservices.

Chapter 11 will walk you through the development of a monitoring system for microservices, using StatsD, Prometheus, and Grafana to collect and aggregate metrics to produce dashboards and alerts. We'll also discuss good practices for alert management and avoiding alert fatigue.

Chapter 12 builds on the work in the previous chapter to include logs and traces. Getting rich, real-time, and searchable information from our microservices helps us

understand them, diagnose issues, and improve them in the future. Examples in this chapter use Elasticsearch, Kibana, and Jaeger.

Lastly, chapter 13 takes a slight left turn to explore the people side of microservice development. People implement software: building great software is about effective collaboration as much as implementation choices. We'll examine the principles that make microservice teams effective and explore the psychological and practical implications of the microservice architectural approach on good engineering practices.

About the code

This book contains many examples of source code, both in numbered listings and inline with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes code is also **`in bold`**, either to highlight specific lines or to differentiate entered commands from the resulting output.

In many cases, we've reformatted the original source code; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➥). Additionally, we've often removed comments in the source code from the listings when we've described the code in the text. Code annotations accompany many of the listings, highlighting important concepts.

The source code within this book is available on the book's website at https://www.manning.com/books/microservices-in-action and on the Github repository at https://github.com/morganjbruce/microservices-in-action.

You can find instructions on running examples throughout the book. We typically use Docker and/or Docker Compose to simplify running examples. The appendix covers configuring Jenkins, used in chapter 10, to run smoothly on a local deployment of Kubernetes.

Book forum

Purchase of *Microservices in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum, go to https://forums.manning.com/forums/microservices-in-action. You can also learn more about Manning's forums and the rules of conduct at https://forums.manning.com/forums/about.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

Morgan Bruce has significant experience in building complex applications, with particular expertise in the finance and identity verification industries, where accuracy, resilience, and security are crucial. As an engineering leader, he has worked on large-scale refactoring and rearchitecture efforts. He also has firsthand experience leading the evolution from a monolithic application to a robust microservice architecture.

Paulo A. Pereira is currently leading a team involved in reshaping a monolith into microservices, while dealing with the constraints of evolving a system where security and accuracy are of the utmost importance. Enthusiastic about choosing the right tool for the job and combining different languages and paradigms, he is currently exploring functional programming, mainly via Elixir. Paulo also authored *Elixir Cookbook* and was one of the technical reviewers for *Learning Elixir* and for *Mastering Elixir*.

about the cover illustration

The figure on the cover of *Microservices in Action* is captioned "Habit of a Lady of China in 1700." The illustration is taken from Thomas Jefferys' *A Collection of the Dresses of Different Nations, Ancient and Modern* (four volumes), London, published between 1757 and 1772. The title page states that these are hand-colored copperplate engravings, heightened with gum arabic.

Thomas Jefferys (1719–1771) was called "Geographer to King George III." He was an English cartographer who was the leading map supplier of his day. He engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. His work as a mapmaker sparked an interest in local dress customs of the lands he surveyed and mapped, which are brilliantly displayed in this collection. Fascination with faraway lands and travel for pleasure were relatively new phenomena in the late 18th century, and collections such as this one were popular, introducing both the tourist and the armchair traveler to the inhabitants of other countries.

The diversity of the drawings in Jefferys' volumes speaks vividly of the uniqueness and individuality of the world's nations some 200 years ago. Dress codes have changed since then, and much of the diversity by region and country, so rich at the time, has faded away. It's now often hard to tell the inhabitants of one continent from another. Perhaps, trying to view it optimistically, we've traded a cultural and visual diversity for a more varied personal life—or a more varied and interesting intellectual and technical life.

At a time when it's difficult to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Jeffreys' pictures.

Part 1

The lay of the land

This part introduces microservice architecture, explores the properties and benefits of microservice applications, and presents some of the challenges you'll face in developing microservice applications. We'll also introduce SimpleBank, a fictional company whose attempts to build a microservice application will be the common thread in many examples used in this book.

1 Designing and running microservices

This chapter covers

- Defining a microservice application
- The challenges of a microservices approach
- Approaches to designing a microservice application
- Approaches to running microservices successfully

Software developers strive to craft effective and timely solutions to complex problems. The first problem you usually try to solve is: What does your customer want? If you're skilled (or lucky), you get that right. But your efforts rarely stop there. Your successful application continues to grow: you debug issues; you build new features; you keep it available and running smoothly.

Even the most disciplined teams can struggle to sustain their early pace and agility in the face of a growing application. At worst, your once simple and stable product becomes both intractable and delicate. Instead of sustainably delivering more value to your customers, you're fatigued from outages, anxious about releasing, and too slow to deliver new features or fixes. Neither your customers nor your developers are happy.

Microservices promise a better way to sustainably deliver business impact. Rather than a single monolithic unit, applications built using microservices are made up of

loosely coupled, autonomous services. By building services that do one thing well, you can avoid the inertia and entropy of large applications. Even in existing applications, you can progressively extract functionality into independent services to make your whole system more maintainable.

When we started working with microservices, we quickly realized that building smaller and more self-contained services was only one part of running a stable and business-critical application. After all, any successful application will spend much more of its life in production than in a code editor. To deliver value with microservices, our team couldn't be focused on build alone. We needed to be skilled at operations: deployment, observation, and diagnosis.

1.1 What is a microservice application?

A microservice application is a collection of autonomous services, each of which does one thing well, that work together to perform more intricate operations. Instead of a single complex system, you build and manage a suite of relatively simple services that might interact in complex ways. These services collaborate with each other through technology-agnostic messaging protocols, either point-to-point or asynchronously.

This might seem like a simple idea, but it has striking implications for reducing friction in the development of complex systems. Classical software engineering practice advocates *high cohesion* and *loose coupling* as desirable properties of a well-engineered system. A system that has these properties will be easier to maintain and more malleable in the face of change.

Cohesion is the degree to which elements of a certain module belong together, whereas coupling is the degree to which one element knows about the inner workings of another. Robert C. Martin's Single Responsibility Principle is a useful way to consider the former:

> *Gather together the things that change for the same reasons. Separate those things that change for different reasons.*

In a monolithic application, you try to design for these properties at a class, module, or library level. In a microservice application, you aim instead to attain these properties at the level of independently deployable units of functionality. A single microservice should be highly cohesive: it should be responsible for some single capability within an application. Likewise, the less that each service knows about the inner workings of other services, the easier it is to make changes to one service—or capability—without forcing changes to others.

To get a better picture of how a microservice application fits together, let's start by considering some of the features of an online investment tool:

- Opening an account
- Depositing and withdrawing money
- Placing orders to buy or sell positions in financial products (for example, shares)
- Modeling risk and making financial predictions

Let's explore the process of selling shares:

1. A user creates an order to sell some shares of a stock from their account.
2. This position is reserved on their account, so it can't be sold multiple times.

3 It costs money to place an order on the market—the account is charged a fee.
4 The system needs to communicate that order to the appropriate stock market.

Figure 1.1 shows how placing that sell order might look as part of a microservice application.

You can observe three key characteristics of microservices in figure 1.1:

- Each microservice is *responsible for a single capability. This might be business related or represent a shared technical capability, such as integration with a third party (for example, the stock exchange).*
- A microservice *owns its data store,* if it has one. This reduces coupling between services because other services can only access data they don't own through the interface that a service provides.
- Microservices themselves, not the messaging mechanism that connects them nor another piece of software, are *responsible for choreography and collaboration*—the sequencing of messages and actions to perform some useful activity.

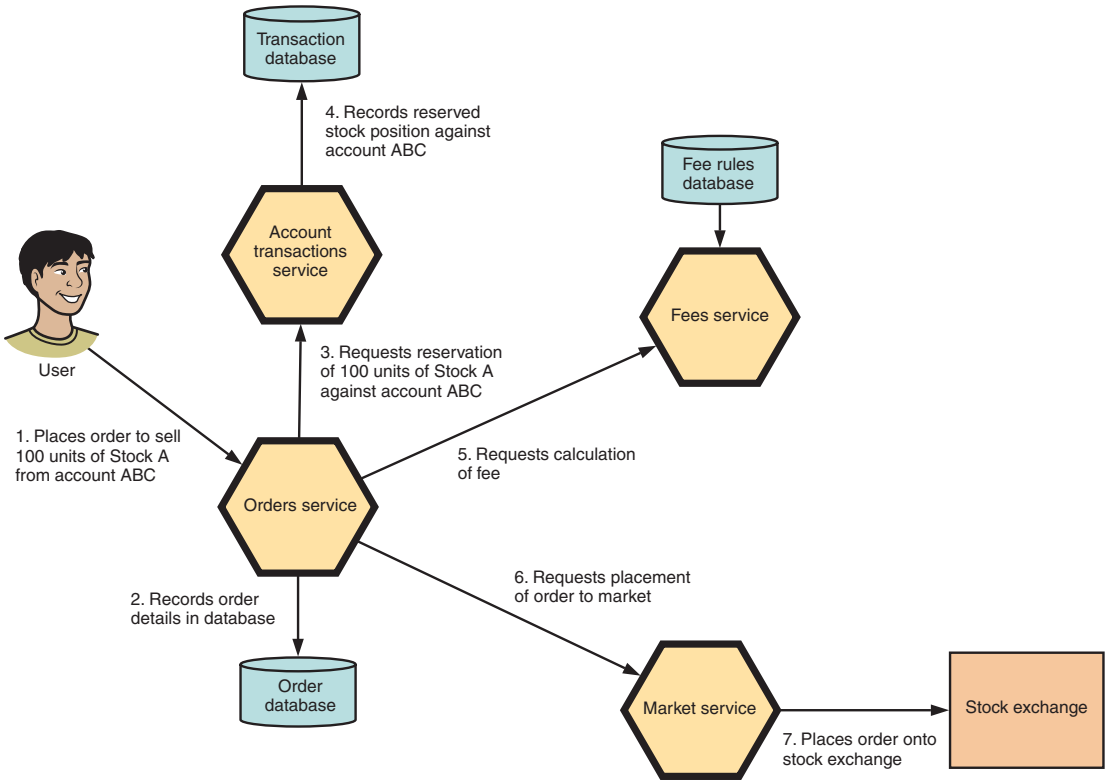


Figure 1.1 The flow of communication through microservices in an application that allows users to sell positions in financial shares

In addition to these three characteristics, you can identify two more fundamental attributes of microservices:

- Each microservice can be *deployed independently*. Without this, a microservice application would still be monolithic at the point of deployment.
- A microservice is *replaceable*. Having a single capability places natural bounds on size; likewise, it makes the individual responsibility, or role, of a service easy to comprehend.

The idea that microservices are responsible for coordinating actions in a system is the crucial difference between this approach and traditional service-oriented architectures (SOAs). Those types of systems often used enterprise service buses (ESBs) or more complex orchestration standards to externalize messaging and process orchestration from applications themselves. In that model, services often lacked cohesion, as business logic was increasingly added to the service bus, rather than the services themselves.

It's interesting to think about how decoupling functionality in the online investment system helps you be more flexible in the face of changing requirements. Imagine that you need to change how fees are calculated. You could make and release those changes to the *fees* service without any change to its upstream or downstream services. Or imagine an entirely new requirement: when an order is placed, you need to alert your risk team if it doesn't match normal trading patterns. It'd be easy to build a new microservice to perform that operation based on an event raised by the *orders* service without changing the rest of the system.

1.1.1 Scaling through decomposition

You also can consider how microservices allow you to scale an application. In *The Art of Scalability*, Abbott and Fisher define three dimensions of scale as the scale cube (figure 1.2).

Monolithic applications typically scale through horizontal duplication: deploying multiple, identical instances of the application. This is also known as cookie-cutter, or X-axis, scaling. Conversely, microservice applications are an example of Y-axis scaling, where you decompose a system to address the unique scaling needs of different functionality.

> **NOTE** The Z axis refers to horizontal data partitions: sharding. You can apply sharding to either approach — microservices or monolithic applications — but we won't be exploring that topic in this book.

Let's revisit the investment tool as an example, with the following characteristics:

- Financial predictions might be computationally onerous and are rarely done.
- Complex regulatory and business rules may govern investment accounts.
- Market trading may happen in extremely large volumes, while also relying on minimizing latency.

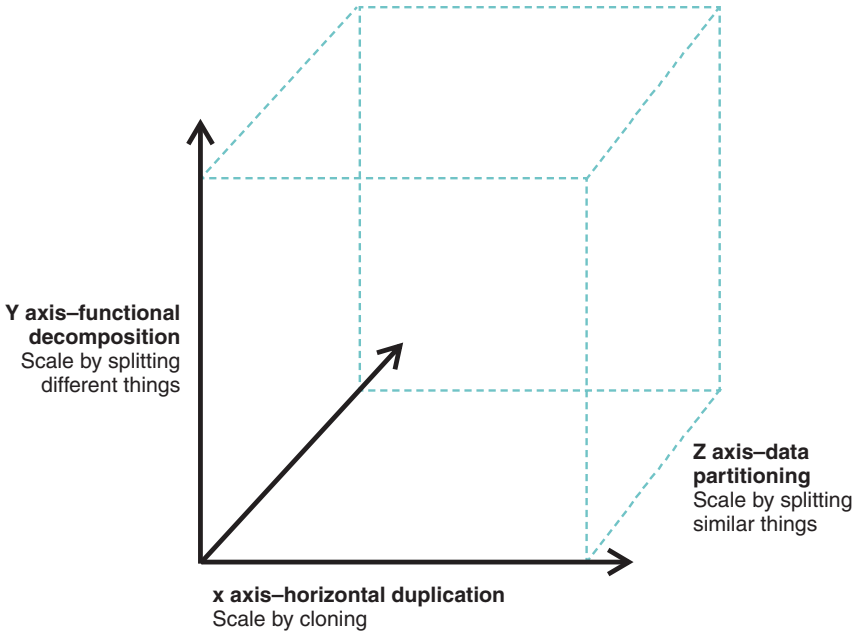


Figure 1.2 The three dimensions of scaling an application

If you build features as microservices that meet the requirements of these characteristics, you can choose the ideal technical tools to solve each problem, rather than trying to fit square pegs into round holes. Likewise, autonomy and independent deployment mean you can manage the microservices' underlying resource needs separately. Interestingly, this also implies a natural way to limit failure: if your financial prediction service fails, that failure is unlikely to cascade to the market trading or investment account services.

Microservice applications have some interesting technical properties:

- Building services along the lines of single capabilities places natural bounds on size and responsibility.
- Autonomy allows you to develop, deploy, and scale services independently.

1.1.2 Key principles

Five cultural and architectural principles underpin microservices development:

- Autonomy
- Resilience
- Transparency
- Automation
- Alignment

These principles should drive your technical and organizational decisions when you're building and running a microservice application. Let's explore each of them.

AUTONOMY

We've established that microservices are *autonomous*—each service operates *and changes independently of others*. To ensure that autonomy, you need to design your services so they are:

- *Loosely coupled*—By interacting through clearly defined interfaces, or through published events, each microservice remains independent of the internal implementation of its collaborators. For example, the orders service we introduced earlier shouldn't be aware of the implementation of the account transactions service. This is illustrated in figure 1.3.
- *Independently deployable*—Services will be developed in parallel, often by multiple teams. Being forced to deploy them in lockstep or in an orchestrated formation would result in risky and anxious deployments. Ideally, you want to use your smaller services to enable rapid, frequent, and small releases.

Autonomy is also cultural. It's vital that you delegate accountability for and ownership of services to teams responsible for delivering business impact. As we've established, organizational design has an influence on system design. Clear service ownership allows teams to build iteratively and make decisions based on their local context and goals. Likewise, this model is ideal for promoting end-to-end ownership, where a team is responsible for a service in both development and production.

NOTE In chapter 13, we'll discuss developing responsible and autonomous engineering teams and why this is crucial when working with microservices.

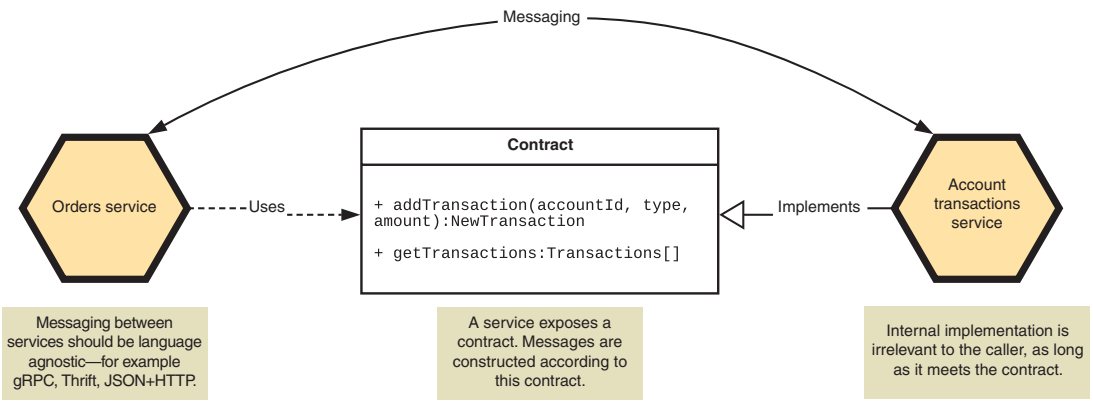


Figure 1.3 You can loosely couple services by having them communicate through defined contracts that hide implementation details.

Resilience

Microservices are a natural mechanism for isolating failure: if you deploy them independently, application or infrastructure failure may only affect part of your system. Likewise, being able to deploy smaller bits of functionality should help you change your system more gradually, rather than releasing a risky big bang of new functionality.

Consider the investment tool again. If the market service is unavailable, it won't be able to place the order to market. But a user can still request the order, and the service can pick it up later when the downstream functionality becomes available.

Although splitting your application into multiple services can isolate failure, it also will multiply points of failure. In addition, you'll need to account for what happens when failure *does* occur to prevent cascades. This involves both design—favoring asynchronous interaction where possible and using circuit breakers and timeouts appropriately—and operations—using provable continuous delivery techniques and robustly monitoring system activity.

Transparency

Most importantly, you need to know when a failure has occurred, and rather than one system, a microservice application depends on the interaction and behavior of multiple services, possibly built by different teams. At any point, your system should be transparent and observable to ensure that you both observe and diagnose problems.

Every service in your application will produce business, operational, and infrastructure metrics; application logs; and request traces. As a result, you'll need to make sense of a huge amount of data.

Automation

It might seem counterintuitive to alleviate the pain of a growing application by building a multitude of services. It's true that microservices are a more complex architecture than building a single application. By embracing automation and seeking consistency in the infrastructure *between* services, you can significantly reduce the cost of managing this additional complexity. You need to use automation to ensure the correctness of deployments and system operation.

It's not a coincidence that the popularity of microservice architecture parallels both the increasing mainstream adoption of DevOps techniques, especially *infrastructure-as-code*, and the rise of infrastructure environments that are fully programmable through APIs (such as AWS or Azure). These two trends have done a lot to make microservices feasible for smaller teams.

Alignment

Lastly, it's critical that you align your development efforts in the right way. You should aim to structure your services, and therefore your teams, around business concepts. This leads to higher cohesion.

To understand why this is important, consider the alternative. Many traditional SOAs deployed the technical tiers of an application separately—UI, business logic, integration, data. Figure 1.4 compares SOA and microservice architecture.

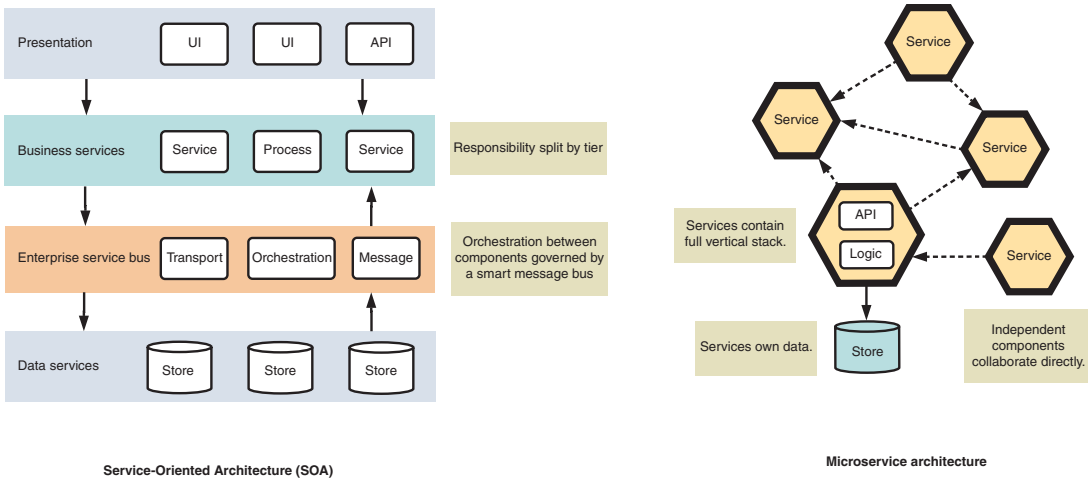


Figure 1.4 SOA versus microservice architecture

This use of *horizontal decomposition* in SOA is problematic, because cohesive functionality becomes spread across multiple systems. New features may require coordinated releases to multiple services and may become unacceptably coupled to others at the same level of technical abstraction.

A microservice architecture, on the other hand, should be biased toward vertical decomposition; each service should align to a single business capability, encapsulating all relevant technical layers.

> **NOTE** In rare instances, it might make sense to build a service that implements a technical capability, such as integration with a third-party service, if multiple services require it.

You also should be mindful of the consumers of your services. To ensure a stable system, you need to ensure you're developing patiently and maintaining backwards compatibility—whether explicitly or by running multiple versions of a service—to ensure that you don't force other teams to upgrade or break complex interactions between services.

Working with these five principles in mind will help you develop microservices well, leading to systems that are highly amenable to change, scalable, and stable.

1.1.3 Who uses microservices?

Many organizations have successfully built and deployed microservices, across many domains: in media (*The Guardian*); content distribution (SoundCloud, Netflix); transport and logistics (Hailo, Uber); e-commerce (Amazon, Gilt, Zalando); banking (Monzo); and social media (Twitter).

Most of these companies took a monolith-first approach.[1] They started by building a single large application, then progressively moved to microservices in response to growth pressures they faced. These pressures are outlined in Table 1.1.

Table 1.1 Pressures of growth on a software system

Pressure	Description
Volume	The volume of activity that a system performs may outgrow the capacity of original technology choices.
New features	New features may not be cohesive with existing features, or different technologies may be better at solving problems.
Engineering team growth	As a team grows larger, lines of communication increase. New developers spend more time comprehending the existing system and less time adding product value.
Technical debt	Increased complexity in a system—including debt from previous build decisions—increases the difficulty of making changes.
International distribution	International distribution may lead to data consistency, availability, and latency challenges.

For example, Hailo wanted to expand internationally—which would've been challenging with their original architecture—but also increase their pace of feature delivery.[2] SoundCloud wanted to be more productive, as the complexity of their original monolithic application was holding them back.[3] Sometimes, the shift coincided with a change in business priority: Netflix famously moved from physical DVD distribution to content streaming. Some of these companies completely decommissioned their original monolith. But for many, this is an ongoing process, with a monolith surrounded by a constellation of smaller services.

As microservice architecture has been more widely popularized—and as early adopters have open sourced, blogged, and presented the practices that worked for them—teams have increasingly begun greenfield projects using microservices, rather than building a single application first. For example, Monzo started with microservices as part of its mission to build a better and more scalable bank.[4]

---

[1] Martin Fowler expands on this pattern: "MonolithFirst," June 3, 2015, http://martinfowler.com/bliki/MonolithFirst.html.

[2] See Matt Heath, "A Long Journey into a Microservice World," *Medium*, May 30, 2015, http://mng.bz/XAOG.

[3] See Phil Calçado, "How we ended up with microservices," September 8, 2015, http://mng.bz/Qzhi.

[4] See Matt Heath, "Building microservice architectures in Go," June 18, 2015, http://mng.bz/9L83.

1.1.4 Why are microservices a good choice?

Plenty of successful businesses are built on monolithic software—Basecamp,[5] StackOverflow, and Etsy spring to mind. And in monolithic applications, a wealth of orthodox, long-established software development practice and knowledge exists. Why choose microservices?

TECHNICAL HETEROGENEITY LEADS TO MICROSERVICES

In some companies, technical heterogeneity makes microservices an obvious choice. At Onfido, we started building microservices when we introduced a product driven by machine learning—not a great fit for our original Ruby stack! Even if you're not fully committed to a microservice approach, applying microservice principles gives you a greater range of technical choices to solve business problems. Nevertheless, it's not always so clear-cut.

DEVELOPMENT FRICTION INCREASES AS COMPLEX SYSTEMS GROW

It comes down to the nature of complex systems. At the beginning of the chapter, we mentioned that software developers strive to craft effective and timely solutions to complex problems. But the software systems we build are inherently complex. No methodology or architecture can eliminate the essential complexity at the heart of such a system.

But that's no reason to get downhearted! You can ensure that the development approaches you take result in *good* complex systems, free from *accidental* complexity.

Take a moment and consider what you're trying to achieve as an enterprise software developer. Dan North puts it well:

> *The goal of software development is to sustainably minimize lead time to positive business impact.*

The hard part in complex software systems is to deliver sustainable value in the face of change: to continue to deliver with agility, pace, and safety even as the system becomes larger and more complex. Therefore, we believe a good complex system is one where two factors are minimized throughout the system's lifecycle: friction and risk.

Friction and risk limit your velocity and agility, and therefore your ability to deliver business impact. As a monolith grows, the following factors may lead to friction:

- Change cycles are coupled together, leading to higher coordination barriers and higher risk of regression.
- Soft module and context boundaries invite chaos in undisciplined teams, leading to tight or unanticipated coupling between components.
- Size alone can be painful: continuous integration jobs and releases—even local application startup—become slower.

These qualities aren't true for all monoliths, but unfortunately they're true for most that we've encountered. Likewise, these types of challenges are a common thread in the stories of the companies we mentioned.

---

[5] David Heinemeier Hansson coined the term "Majestic Monolith" to describe how 37signals built Basecamp: *Signal v. Noise*, February 29, 2016, http://mng.bz/1p3I.

Microservices reduce friction and risk

Microservices help reduce friction and risk in three ways:

- Isolating and minimizing dependencies at build time
- Allowing developers to reason about cohesive individual components, rather than an entire system
- Enabling the continuous delivery of small, independent changes

Isolating and minimizing dependencies at build time—whether between teams or on existing code—allows developers to move faster. Development can move in parallel, with reduced long-term dependency on past decisions made in a monolithic application. Technical debt is naturally limited to service boundaries.

Microservices are individually easier to build and reason about than monolithic applications. This is beneficial for the productivity of development in a growing organization. It also provides a compelling and flexible paradigm for coping with increased scale or smoothly introducing new technologies.

Small services are also a great enabler of continuous delivery. Deployments in large applications can be risky and involve lengthy regression and verification cycles. By deploying smaller elements of functionality, you better isolate changes to your active system, reducing the potential risk of an individual deployment.

At this point, we can come to two conclusions:

- Developing small, autonomous services can reduce friction in the development of long-running complex systems.
- By delivering cohesive and independent pieces of functionality, you can build a system that's malleable and resilient in the face of change, helping you to deliver sustainable business impact with reduced risk.

That doesn't mean everyone should build microservices. It'd be wonderful if there was an objective answer to the question "Do I need microservices?" but unfortunately you can only say "It depends"—on your team, on your company, and on the nature of the system you're building. If the scope of your system is trivial, then it's unlikely you'll gain benefits that outweigh the added complexity of building and running this type of fine-grained application. But if you've faced any of the challenges we mentioned earlier in this section, then microservices are a compelling solution.

A cautionary tale

We once heard a story about a microservice implementation gone wrong. The startup in question had begun to scale, and the CTO had decided that the only solution was to rebuild the application as microservices. If you're not worried by that sentence, you should be!

The engineering team set out to rebuild their application. This took them five months, during which time they released zero new features, nor did they release any of their microservices to production. The team proceeded to launch their new microservice application

(continued)

during the busiest month for the business, causing absolute chaos and necessitating a rollback to the original monolith.

This type of migration gives microservices a bad name. Few businesses have the luxury of a feature freeze for several months nor can they indulge a big-bang launch of a new architecture. Although the sample set is small, most successful microservice migrations that we've observed have been piecemeal, balancing architectural vision with business needs, priorities, and resource constraints. Although it'll take longer and require more engineering effort, hopefully you'll never recognize your team being mentioned in a cautionary tale!

1.2 What makes microservices challenging?

Let's dig a little deeper and explore the costs and complexity of designing and running microservices. Microservices aren't the only architecture that have promised nirvana through decomposition and distribution, but those past attempts, such as SOA,[6] are widely considered unsuccessful. No technique is a silver bullet. For example, as we've mentioned, microservices drastically increase the number of moving parts in a system. By distributing functionality and data ownership across multiple autonomous services, you likewise distribute responsibility for stability and sane operation of your application.

You'll encounter many challenges when designing and running a microservice application:

- *Scoping and identifying microservices* requires substantial domain knowledge.
- The right *boundaries and contracts* between services are difficult to identify and, once you've established them, can be time-consuming to change.
- Microservices are *distributed systems* and therefore require different assumptions to be made about state, consistency, and network reliability.
- By distributing system components across networks, and increasing technical heterogeneity, microservices introduce *new modes of failure*.
- It's more challenging to understand and verify what should happen in normal operation.

1.2.1 Design challenges

How do these challenges impact the design and runtime phases of microservice development? Earlier we introduced the five key principles underlying microservice development. The first of those was *autonomy*. For your services to be autonomous, you need to design them such that, together, they're loosely coupled, and, individually, they encapsulate highly cohesive elements of functionality. This is an evolutionary process. The

[6] SOA is a wooly term. Although many principles of SOA are similar to microservices, the definition of the former is inextricably associated with heavyweight, enterprise vendor tools, such as ESBs.

scope of your services may change over time, and you'll often choose to carve out new functionality from—or even retire—existing services.

Making those choices is challenging, and even more so at the start of developing an application! The primary driver of loose coupling is the boundaries you establish between services; getting those wrong will lead to services that are resistant to change and, overall, a less malleable and flexible application.

Scoping microservices requires domain knowledge

Each microservice is responsible for a single capability. Identifying these capabilities requires knowledge of the business domain of your application. Early in an application's lifetime, your domain knowledge might be at best incomplete, or at worst, incorrect.

Inadequate understanding of your problem domain can result in poor design choices. In a microservice application, the increased rigidity of a service boundary when compared to a module within a monolithic application means the downstream cost of poor scoping decisions is likely to be higher:

- You may need to refactor across multiple distinct codebases.
- You may need to migrate data from one service's database to another.
- You may not have identified implicit dependencies between services, which could lead to errors or incompatibility on deployment.

These activities are illustrated in figure 1.5.

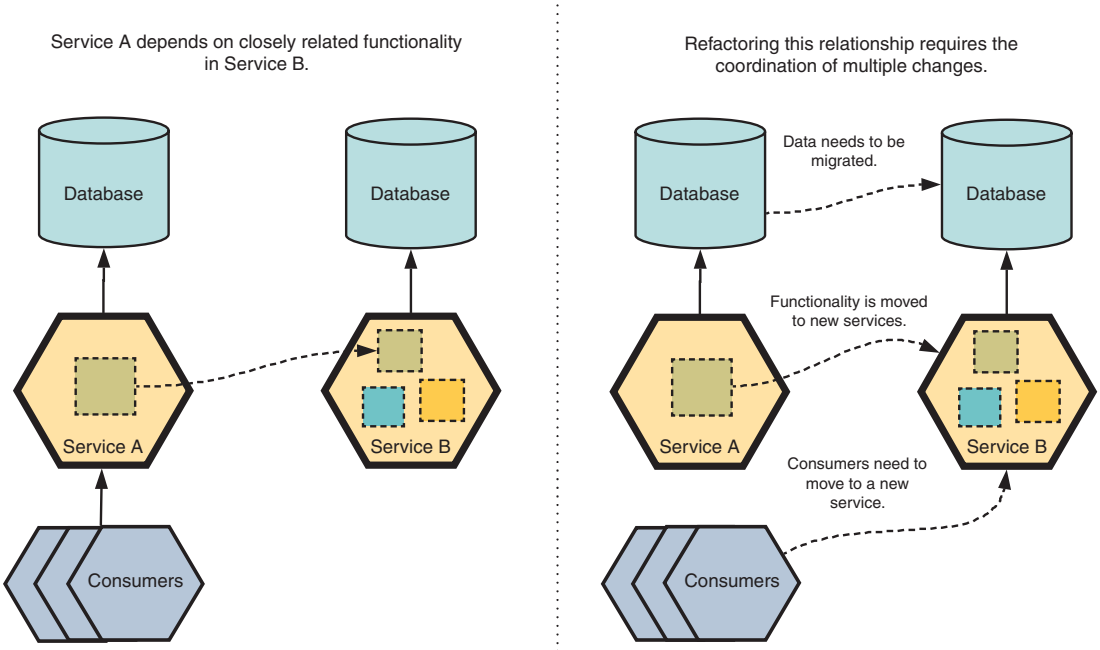


Figure 1.5 Incorrect service scoping decisions may require complex and costly refactoring across service boundaries.

But making design decisions based on insufficient domain knowledge is hardly unique to microservices! The difference is in the impact of those decisions.

NOTE In chapters 2 and 4, we'll discuss best practices for identifying and scoping services, using an example application.

MAINTAINING CONTRACTS BETWEEN SERVICES

Each microservice should be independent of the implementation of other services. This enables technical heterogeneity and autonomy. For this to work, each microservice should expose a *contract*—analogous to an interface in object-oriented design—defining the messages it expects to receive and respond with. A good contract should be

- *Complete*—Defines the full scope of an interaction
- *Succinct*—Takes in no more information than is necessary, so that consumers can construct messages within reasonable bounds
- *Predictable*—Accurately reflects the real behavior of any implementation

Anyone who's designed an API might know how hard these properties are to achieve. Contracts become the glue between services. Over time, contracts may need to evolve while also needing to maintain backwards compatibility for existing collaborators. These twin tensions—between stability and change—are challenging to navigate.

MICROSERVICE APPLICATIONS ARE DESIGNED BY TEAMS

In larger organizations, it's likely that multiple teams will build and run a microservice application, each taking responsibility for different microservices. Each team may have its own goals, way of working, and delivery lifecycle. It can be difficult to design a cohesive system when you also need to reconcile the timelines and priorities of other independent teams. Coordinating the development of any substantial microservice application therefore will require the agreement and reconciliation of priorities and practices across multiple teams.

MICROSERVICE APPLICATIONS ARE DISTRIBUTED SYSTEMS

Designing microservice applications means designing distributed systems. Many fallacies occur in the design of distributed systems,[7] including

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- Transport cost is zero.

Clearly, assumptions you might make in nondistributed systems—such as the speed and reliability of method calls—are no longer appropriate and can lead to poor, unstable implementation. You must consider latency, reliability, and the consistency of state across your application.

---

[7] See Arnon Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained," https://pages.cs.wisc.edu/~zuyu/files/fallacies.pdf.

Once the application is distributed—where the application's underlying state data is spread across a multitude of places—consistency becomes challenging. You may not have guarantees of the order of operations. It won't be possible to maintain ACID-like transactional guarantees when actions take place across multiple services. This will affect design at the application level: you'll need to consider how a service might operate in an inconsistent state and how to roll back in the event of transaction failure.

1.2.2 Operational challenges

A microservice approach will inherently multiply the possible points of failure in a system. To illustrate this, let's return to the investment tool we mentioned earlier. Figure 1.6 identifies possible points of failure in this application. You can see that something could go wrong in multiple places, and that could affect the normal processing of an order.

Consider the questions you might need to answer when this application is in production:

- If something goes wrong and your user's order isn't placed, how would you determine where the fault occurred?
- How do you deploy a new version of a service without affecting order placement?
- How do you know which services were meant to be called?
- How do you test that this behavior is working correctly across multiple services?
- What happens if a service is unavailable?

Rather than eliminating risk, microservices move that cost to later in the lifecycle of your system: reducing friction in development but increasing the complexity of how you deploy, verify, and observe your application in operation.

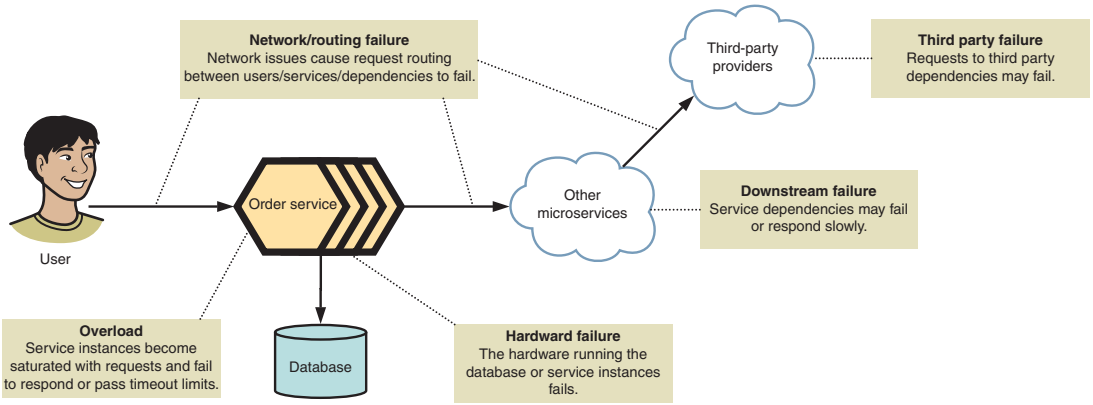


Figure 1.6 Possible points of failure when placing a sell order

A microservices approach suggests an evolutionary approach to system design: you can add new features independently without changing existing services. This minimizes the cost and risk of change.

But in a decoupled system that constantly changes, it can be extremely difficult to keep track of the big picture, which makes issue diagnosis and support more challenging. When something goes wrong, you need to have some way of tracing how the system *did* behave (what services it called, in which order, and what the outcome was), but you also need some way of knowing how the system *should have* behaved.

Ultimately, you face two operational challenges in microservices: observability and multiple points of failure. Let's focus on each of those in turn.

OBSERVABILITY IS DIFFICULT TO ACHIEVE

We touched on the importance of transparency back in section 1.1.2. But why is it harder in microservice applications? It's harder because you need to understand the big picture. You need to assemble that big picture from multiple jigsaw pieces, to correlate and link together the data each service produces to ensure you understand what each service does within the wider context of delivering some business output. Individual service logs provide a partial view of system operation, which is helpful, but you need to use both a microscope and a wide-angle lens to understand the system in full.

Likewise, because you're running multiple applications, depending on how you choose to deploy them, a less obvious correlation may exist between underlying infrastructural metrics—like memory and CPU usage—and the application. These metrics are still useful but are less of a focus than they might be in a monolithic system.

MULTIPLYING SERVICES MULTIPLIES POINTS OF FAILURE

We're probably not being too pessimistic if we say that everything that can fail will fail. It's important that you start with that mindset: if you assume weakness and fragility in the multiple services forming your system, that can better inform how you design, deploy, and monitor that system—rather than getting too surprised when something does go wrong.

You need to consider how your system will continue operating despite the failures of individual components. This implies that, individually, services will need to become more robust—considering error checking, failover, and recovery—but also that the whole system should act reliably, even when individual components are never 100% reliable.

1.3 Microservice development lifecycle

At an individual level, each microservice should look familiar to you—even if it's a bit smaller. To build a microservice, you'll use many of the same frameworks and techniques that you'd normally apply in building an application: web application frameworks, SQL databases, unit tests, libraries, and so on.

At a system level, choosing a microservice architecture will have a significant impact on how you design and run your application. Throughout this book, we'll focus on these

three key stages in the development lifecycle of a microservice application: designing services, deploying them to production, and observing their behavior. This cycle is illustrated in figure 1.7.

Making well-reasoned decisions in each of these three stages will help you build applications that are resilient, even in the face of changing requirements and increasing complexity. Let's walk through each stage and consider the steps you'll take to deliver an application with microservices.

1.3.1 Designing microservices

You'll need to make several design decisions when building a microservice application that you wouldn't have encountered building monolithic apps. The latter often follow well-known patterns or frameworks, such as three-tier architecture or model-view controller (MVC). But techniques for designing microservices are still in their relative infancy. You'll need to consider

- Whether to start with a monolith or commit to microservices up front
- The overall architecture of your application and the façade it presents to outside consumers
- How to identify and scope the boundaries of your services
- How your services communicate with each other, whether synchronously or asynchronously
- How to achieve resiliency in services

That's quite a lot of ground to cover. For now, we'll touch on each of these considerations so you can see why paying attention to all of them is vital to a well-designed microservice application.

MONOLITH FIRST?

You'll find two opposing trends to starting with microservices: monolith first or microservices only. Advocates of the former reason that you should always start with a monolith, as you won't understand the component boundaries in your system at an early stage, and the cost of getting these wrong is much higher in a microservice application. On the other hand, the boundaries you choose in a monolith aren't necessarily the same ones you'd choose in a well-designed microservice application.

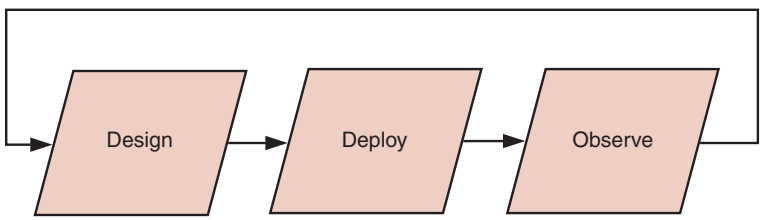


Figure 1.7 The key iterative stages—design, deploy, and observe—in the microservice development lifecycle

Although the speed of development may be slower to begin with, microservices will reduce friction and risk in future development. Likewise, as tooling and frameworks mature, microservices best practice is becoming increasingly less daunting to pick up. Either way you want to go, the advice in this book should be useful, regardless of whether you're thinking of migrating away from your monolith or starting afresh.

SCOPING SERVICES

Choosing the right level of responsibility for each service—its scope—is one of the most difficult challenges in designing a microservice application. You'll need to model services based on the business capabilities they provide to an organization.

Let's extend the example from the beginning of this chapter. How might your services change if you wanted to introduce a new, special type of order? You have three options to solve this problem (figure 1.8):

1. Extend the existing service interface
2. Add a new service endpoint
3. Add a new service

Each of these options has pros and cons that will impact the cohesiveness and coupling between services in your application.

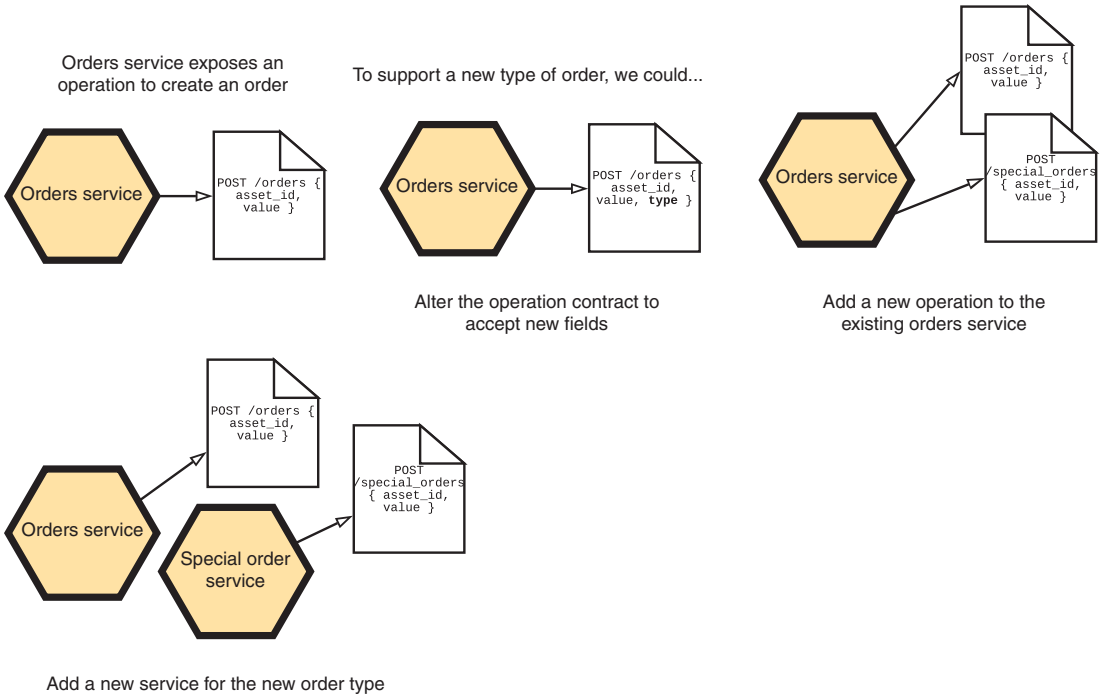


Figure 1.8. To scope functionality, you need to make decisions about whether capabilities belong in existing services or if you need to design new services.

NOTE In chapters 2 and 4, we'll explore service scoping and how to make optimal decisions about service responsibility.

COMMUNICATION

Communication between services may be asynchronous or synchronous. Although synchronous systems are easier to reason through, asynchronous systems are highly decoupled—reducing the risk of change—and potentially more resilient. But the complexity of such a system is high. In a microservice application, you need to balance synchronous and asynchronous messaging to choreograph and coordinate the actions of multiple microservices effectively.

RESILIENCY

In a distributed system, a service can't trust its collaborators, not necessarily because they're coded poorly or because of human error, but because you can't safely assume the network between or behavior of those services is reliable or predictable. Services need to be resilient in the face of failure. To achieve this, you need to design your services to work defensively by backing off in the event of errors, limiting request rates from poor collaborators, and dynamically finding healthy services.

1.3.2 Deploying microservices

Development and operations must be closely intertwined when building microservices. It's not going to work if you build something and throw it over the fence for someone else to deploy and operate it. In a system composed of numerous, autonomous services, if you build it, you should run it. Understanding how your services run will in turn help you make better design decisions as your system grows.

Remember, what's special about your application is the business impact it delivers. That emerges from collaboration between multiple services. In fact, you could standardize or abstract away anything outside of the unique capability each service offers—ensuring teams are focused on business value. Ultimately, you should reach a stage where there's no ceremony involved in deploying a new service. Without this, you'll invest all your energy in plumbing, rather than creating value for customers.

In this book, we'll teach you how to construct a reliable road to production for existing and new services. The cost of deploying new services must be negligible to enable rapid innovation. Likewise, you should standardize this process to simplify system operation and ensure consistency across services. To achieve this, you'll need to

- Standardize microservice deployment artifacts
- Implement continuous delivery pipelines

We've heard reliable deployment described as boring, not in the sense that it's unexciting, but that it's incident-free. Unfortunately, we've seen too many teams where the opposite is true: deploying software is stressful and encourages unhealthy all-hands-on-deck behavior. This is bad enough for one service—if you're deploying any number

of services, the anxiety alone will drive you mad! Let's look at how these steps lead to stable and reliable microservice deployments.

STANDARDIZE MICROSERVICE DEPLOYMENT ARTIFACTS

It often seems like every language and framework has its own deployment tool. Python has Fabric, Ruby has Capistrano, Elixir has exrm, and so on. And then the deployment environment itself is complex:

- What server does an application run on?
- What are the application's dependencies on other tools?
- How do you start that application?

At runtime, an application's dependencies (figure 1.9) are broad and might include libraries, binaries and OS packages (such as ImageMagick or libc), and OS processes (such as cron or fluentd).

Technically, heterogeneity is a fantastic benefit of service autonomy. But it doesn't make life easy for deployment. Without consistency, you won't be able to standardize your approach to taking services to production, which increases the cost of managing deployments and introducing new technology. At worst, each team reinvents the wheel, coming up with different approaches for managing dependencies, packing builds, getting them onto servers, and operating the application itself.

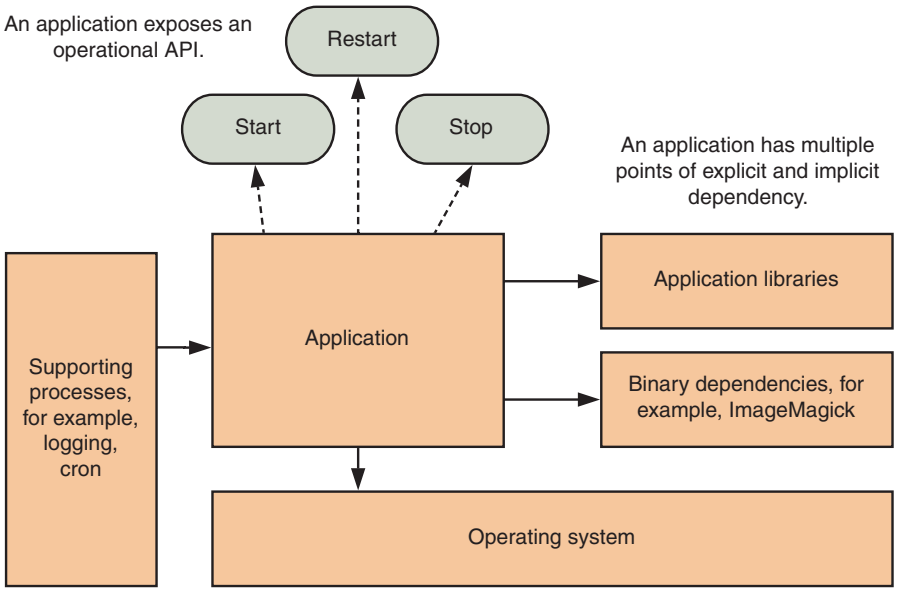


Figure 1.9 An application exposes an operational API and has many types of dependencies, including libraries, binary dependencies, and supporting processes.

Our experience suggests the best tools for this job are *containers*. A container is an operating system-level virtualization method that supports running isolated systems on a host, each with its own network and process space, sharing the same kernel. A container is quicker to build and quicker to start up than a virtual machine (seconds, rather than minutes). You can run multiple containers on one machine, which simplifies local development and can help to optimize resource usage in cloud environments.

Containers standardize the packaging of an application, and the runtime interface to it, and provide immutability of both operating environment and code. This makes them powerful building blocks for higher level composition. By using them, you can define and isolate the full execution environment of any service.

Although many implementations of containers are available (and the concept exists outside of Linux, such as jails in FreeBSD and zones in Solaris), the most mature and approachable tooling that we've used so far is Docker. We'll use that tool later in this book.

IMPLEMENT CONTINUOUS DELIVERY PIPELINES

Continuous delivery is a practice in which developers produce software that they can reliably release to production at any time. Imagine a factory production line: to continuously deliver software, you build similar pipelines to take your code from commit to live operation. Figure 1.10 illustrates a simple pipeline. Each stage of the pipeline provides feedback to the development team on the correctness of their code.

Earlier, we mentioned that microservices are an ideal enabler of continuous delivery because their smaller size means you can develop them quickly and release them independently. But continuous delivery doesn't automatically follow from developing microservices. To continuously deliver software, you need to focus on two goals:

- Building a set of validations that your software has to pass through. At each stage of your deployment process, you should be able to prove the correctness of your code.
- Automating the pipeline that delivers your code from commit to production.

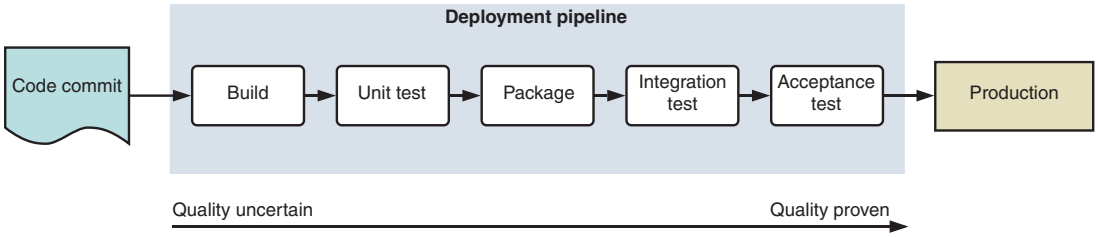


Figure 1.10 A high-level deployment pipeline for a microservice

Building a provably correct deployment pipeline will allow developers to work safely and at pace as they iteratively develop services. Such a pipeline is a repeatable and reliable process for delivering new features. Ideally, you should be able to standardize the validations and steps in your pipeline and use them across multiple services, further reducing the cost of deploying new services.

Continuous delivery also reduces risk, because the quality of the software produced and the team's agility in delivering changes are both increased. From a product perspective, this may mean you can work in a leaner fashion—rapidly validating your assumptions and iterating on them.

> **NOTE** In part 3, we'll build a continuous delivery pipeline using the Pipeline feature of the freely available Jenkins continuous integration tool. We'll also explore different deployment patterns, such as canaries and blue-green deployments.

1.3.3 Observing microservices

We've discussed transparency and observability throughout this chapter. In production, you need to know what's going on. The importance of this is twofold:

- You want to proactively identify and refactor fragile implementation in your system.
- You need to understand how your system is behaving.

Thorough monitoring is significantly more difficult in a microservice application because single transactions may span multiple distinct services; technically heterogeneous services might produce data in irreconcilable formats; and the total volume of operational data is likely to be much higher than that of a single monolithic application. But if you're able to understand how your system operates—and observe that closely—despite this complexity, you'll be better placed to make effective changes to your system.

IDENTIFY AND REFACTOR POTENTIALLY FRAGILE IMPLEMENTATION

Systems will fail, whether because of bugs introduced, runtime errors, network failures, or hardware problems.[8] Over time, the cost of eliminating unknown bugs and errors becomes higher than the cost of being able to react quickly and effectively when they occur.

Monitoring and alerting systems allow you to diagnose problems and determine what causes failures. You may have automated mechanisms reacting to the alerts that'll spawn new container instances in different data centers or react to load issues by increasing the number of running instances of a service.

To minimize the consequences of those failures, and prevent them cascading throughout the system, you need to be able to architect dependencies between services in ways

[8] You even have to watch out for squirrels: Rich Miller, "Surviving Electric Squirrels and UPS Failures," *DataCenter Knowledge*, July 9, 2012, http://mng.bz/rmbF.

that'll allow for partial degradation. One service going down shouldn't bring down the whole application. It's important to think about the possible failure points of your applications, recognize that failure will always happen, and prepare accordingly.

UNDERSTAND BEHAVIOR ACROSS HUNDREDS OF SERVICES

You need to prioritize transparency in design and implementation to understand behavior across your services. Collecting logs and metrics—and unifying them for analytical and alerting purposes—allows you to build a single source of truth to resort to when monitoring and investigating the behavior of your system.

As we mentioned in section 1.3.2, you can standardize and abstract anything outside of the unique capability each service offers. You can think of each service as an onion. At the center of that onion, you have the unique business capability offered by that service. Surrounding that, you have layers of instrumentation—business metrics, application logs, operational metrics, and infrastructure metrics—that make that capability observable. You can then trace each request to the system through these layers. You'd then push the data you collected from these layers to an operational data store for analytics and alerting. This is illustrated in figure 1.11.

> **NOTE** In part 4 of this book, we'll discuss how to build a monitoring system for microservices, collect appropriate data, and use that data to produce a live model for a complex microservice application.

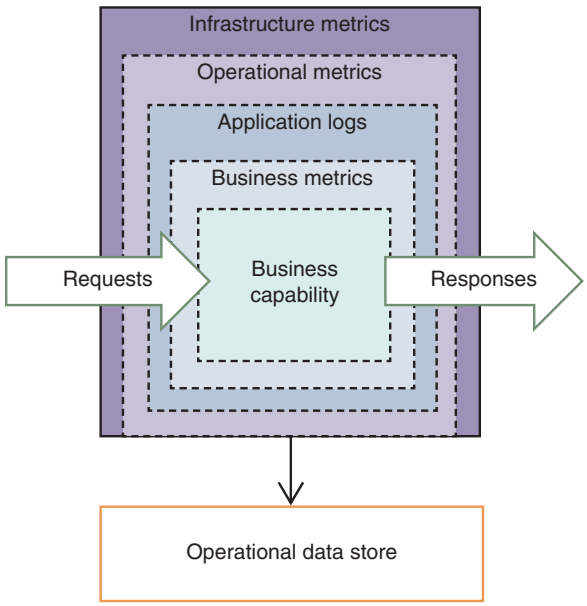


Figure 1.11 A business capability microservice surrounded by layers of instrumentation, through which pass requests to the microservice and its responses, with data collected from the process going to an operational data store

1.4 Responsible and operationally aware engineering culture

It'd be a mistake to examine the technical nature of microservices in isolation from how an engineering team works to develop them. Building an application out of small, independent services will drastically change how an organization approaches engineering, so guiding the culture and priorities of your team will be a significant factor in whether you successfully deliver a microservice application.

It can be difficult to separate cause and effect in organizations that have successfully built microservices. Was the development of fine-grained services a logical outcome of their organizational structure and the behavior of their teams? Or did that structure and behavior arise from their experiences building fine-grained services?

The answer is a bit of both. A long-running system isn't only an accumulation of features requested, designed, and built. It also reflects the preferences, opinions, and objectives of its builders and operators. Conway's Law expresses this to some degree:

> *organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.*

"Constrained" might suggest that these communication structures will limit and constrict the effective development of a system. In fact, microservices practice implies the opposite: that a powerful way to avoid friction and tension in building systems is to design an organization in the shape of the system you intend to build.

Deliberate symbiosis with organizational structure is one example of common microservices practice. To be able to realize benefits from microservices and adequately manage their complexity, you need to develop working principles and practices that are effective for that type of application, rather than using the same techniques that you used to build monoliths.

Summary

- Microservices are both an architectural style and a set of cultural practices, underpinned by five key principles: autonomy, resilience, transparency, automation, and alignment.
- Microservices reduce friction in development, enabling autonomy, technical flexibility, and loose coupling.
- Designing microservices can be challenging because of the need for adequate domain knowledge and balancing priorities across teams.
- Services expose contracts to other services. Good contracts are succinct, complete, and predictable.
- Complexity in long-running software systems is unavoidable, but you can deliver value sustainably in these systems if you make choices that minimize friction and risk.
- Reliably incident-free ("boring") deployment reduces the risk of microservices by making releases automated and provable.

- Containers abstract away differences between services at runtime, simplifying large-scale management of heterogeneous microservices.
- Failure is inevitable: microservices need to be transparent and observable for teams to proactively manage, understand, and own service operation ... and the lack thereof.
- Teams adopting microservices need to be operationally mature and focus on the entire lifecycle of a service, not only on the design and build stages.

2 Microservices at SimpleBank

This chapter covers

- Introducing SimpleBank, a company adopting microservices
- Designing a new feature with microservices
- How to expose microservice-based features to the world
- Ensuring features are production ready
- Challenges faced in scaling up microservice development

In Chapter 1, you learned about the key principles of microservices and why they're a compelling approach for sustainably delivering software value. We also introduced the design and development practices that underpin microservices development. In this chapter, we'll explore how you can apply those principles and practices to developing new product features with microservices.

Over the course of this chapter, we'll introduce the fictitious company of SimpleBank. They're a company with big plans to change the world of investment, and you're working for them as an engineer. The engineering team at SimpleBank wants to be able to deliver new features rapidly while ensuring scalability and

stability—after all, they're dealing with people's money! Microservices might be exactly what they need.

Building and running an application made up of independently deployable and autonomous services is a vastly different challenge from building that application as a single monolithic unit. We'll begin by considering why a microservice architecture might be a good fit for SimpleBank and then walk you through the design of a new feature using microservices. Finally, we'll identify the steps needed to develop that proof of concept into a production-grade application. Let's get started.

2.1 What does SimpleBank do?

The team at SimpleBank wants to make smart financial investment available to everyone, no matter how much money they have. They believe that buying shares, selling funds, or trading currency should be as simple as opening a savings account.

That's a compelling mission, but not an easy one. Financial products have multiple dimensions of complexity: SimpleBank will need to make sense of market rules and intricate regulations, as well as integrate with existing industry systems, all while meeting stringent accuracy requirements.

In the previous chapter, we identified some of the functionality that SimpleBank could offer its customers: opening accounts, managing payments, placing orders, and modeling risk. Let's expand on those possibilities and look at how they might fit within the wider domain of an investment tool. Figure 2.1 illustrates the different elements of this domain.

As the figure shows, an investment tool will need to do more than offer customer-facing features, like the ability to open accounts and manage a financial portfolio. It also will need to manage custody, which is how the bank holds assets on behalf of customers and moves them in or out of their possession, and manufacture, which is the creation of financial products appropriate to customer needs.

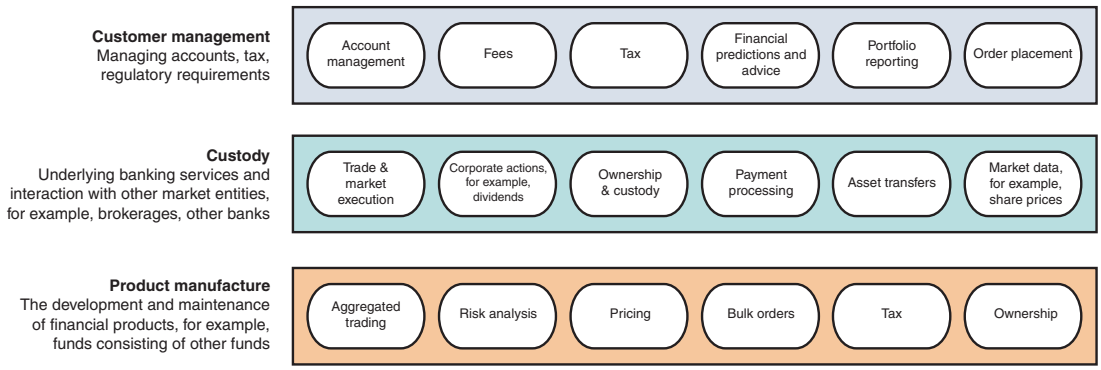


Figure 2.1 A high-level (and by no means exhaustive) model of functionality that SimpleBank might build

As you can see, it's not so simple! You can begin to see some of the business capabilities that SimpleBank might implement: portfolio management, market data integrations, order management, fund manufacture, and portfolio analysis. Each of the business areas identified might consist of any number of services that collaborate with each other or services in other areas.

This type of high-level domain model is a useful first step when approaching any system, but it's crucial when building microservices. Without understanding your domain, you might make incorrect decisions about the boundaries of your services. You don't want to build services that are *anemic—existing only to perform trivial create, read, update, delete (CRUD) operations. These often become a source of tight coupling within an application. At the same time, you want to avoid pushing too much responsibility into a single service. Less cohesive services make software changes slower and riskier—exactly what you're trying to avoid.*

Lastly, without this perspective, you might fall prey to overengineering—choosing microservices where they're not justified by the real complexity of your product or domain.

2.2 Are microservices the right choice?

The engineers at SimpleBank believe that microservices are the best choice to tackle the complexity of their domain and be flexible in the face of complex and changing requirements. They anticipate that as their business grows, microservices will reduce the risk of individual software changes, leading to a better product and happier customers.

As an example, let's say they need to process every buy or sell transaction to calculate tax implications. But tax rules work differently in every country—and those rules tend to change frequently. In a monolithic application, you'd need to make coordinated, time-sensitive releases to the entire platform, even if you only wanted to make changes for one country. In a microservice application, you could build autonomous tax-handling services (whether by country, type of tax, or type of account) and deploy changes to them independently.

Is SimpleBank making the right choice? Architecting software always involves tension between pragmatism and idealism—balancing product needs, the pressures of growth, and the capabilities of a team. Poor choices may not be immediately apparent, as the needs of a system vary over its lifetime. Table 2.1 expands on the factors to consider when choosing microservices.

Table 2.1 Factors to consider when choosing a microservice architecture

Factor	Impact
Domain complexity	It's difficult to objectively evaluate the complexity of a domain, but microservices can address complexity in systems driven by competing pressures, such as regulatory requirements and market breadth.
Technical requirements	You can build different components of a system using different programming languages (and associated technical ecosystems). Microservices enable heterogeneous technical choices.

Table 2.1 Factors to consider when choosing a microservice architecture *(continued)*

Factor	Impact
Organizational growth	Rapidly growing engineering organizations may benefit from microservices because lowering dependency on existing codebases enables rapid ramp-up and productivity for new engineers.
Team knowledge	Many engineers lack experience in microservices and distributed systems. If the team lacks confidence or knowledge, it may be appropriate to build a proof-of-concept microservice before fully committing to implementation.

Using these factors, you can evaluate whether microservices will help you deliver sustainable value in the face of increasing application complexity.

2.2.1 Risk and inertia in financial software

Let's take a moment to look at how SimpleBank's competitors build software. Most banks aren't ahead of the curve in terms of technological innovation. There's an element of inertia that's typical of larger organizations, although that's not unique to the finance industry. Two primary factors limit innovation and flexibility:

- *Aversion to risk*—Financial companies are heavily regulated and tend to build top-down systems of change control to avoid risk by limiting the frequency and impact of software changes.
- *Reliance on complex legacy systems*—Most core banking systems were built pre-1970. In addition, mergers, acquisitions, and outsourcing have led to software systems that are poorly integrated and contain substantial technical debt.

But limiting change and relying on existing systems hasn't prevented software problems from leading to pain for customers or the finance companies themselves. The Royal Bank of Scotland was fined £56 million in 2014 when an outage caused payments to fail for 6.5 million customers. That's on top of the £250 million it was already spending every year on its IT systems.[1]

That approach also hasn't led to better products. Financial technology startups, such as Monzo and Transferwise, are building features at a pace most banks can only dream of.

2.2.2 Reducing friction and delivering sustainable value

Can you do any better? By any measure, the banking industry is a complex and competitive domain. A bank needs to be both resilient and agile, even when the lifetime of a banking system is measured in decades. The increasing size of a monolithic application is antithetical to this goal. If a bank wants to launch a new product, it shouldn't be

[1] See Sean Farrell and Carmen Fishwick, "RBS could take until weekend to make 600,000 missing payments after glitch," *The Guardian*, June 17, 2015, http://mng.bz/kxQY, and Chad Bray, "Royal Bank of Scotland Fined $88 Million Over Technology Failure," *Dealbook*, *The New York Times*, November 20, 2014, http://mng.bz/hn8D.

bogged down by the legacy of previous builds[2] or require outsize effort and investment to prevent regression in existing functionality.

A well-designed microservice architecture can solve these challenges. As we established earlier, this type of architecture avoids many of the characteristics that, in monolithic applications, slow velocity in development. Individual teams can move forward with increased confidence as

- Change cycles are decoupled from other teams.
- Interaction between collaborating components is disciplined.
- Continuous delivery of small, isolated changes limits the risk of breaking functionality.

These factors reduce friction in the development of a complex system but maintain resiliency. As such, they reduce risk without stifling innovation through bureaucracy.

This isn't only a short-term solution. Microservices aid engineering teams in delivering sustainable value throughout the lifecycle of an application by placing natural bounds on the conceptual and implementation complexity of individual components.

2.3 Building a new feature

Now that we've established that microservices are a good choice for SimpleBank, let's look at how it might use them to build new features. Building a minimum viable product—an MVP—is a great first step to ensure that a team understands the constraints and requirements of the microservices style. We'll start by exploring one of the features that SimpleBank needs to build and the design choices the team will make, working through the lifecycle we illustrated in chapter 1 (figure 2.2).

In chapter 1, we touched on how services might collaborate to place a sell order. An overview of this process is shown in figure 2.3.

Let's look at how you'd approach building this feature. You need to answer several questions:

- Which services do you need to build?
- How do those services collaborate with each other?
- How do you expose their functionality to the world?

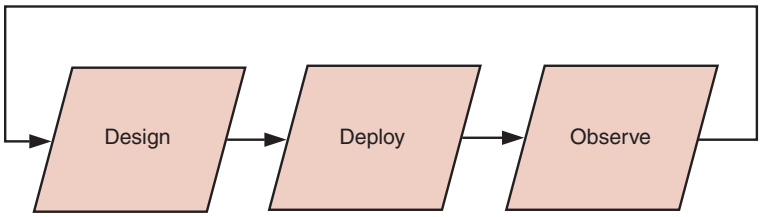


Figure 2.2 The key iterative stages—design, deploy, and observe—in the microservice development lifecycle

[2] How bad can it get? I once encountered a financial software company that maintained over 10 distinct monolithic codebases, each surpassing 2 million lines of code!

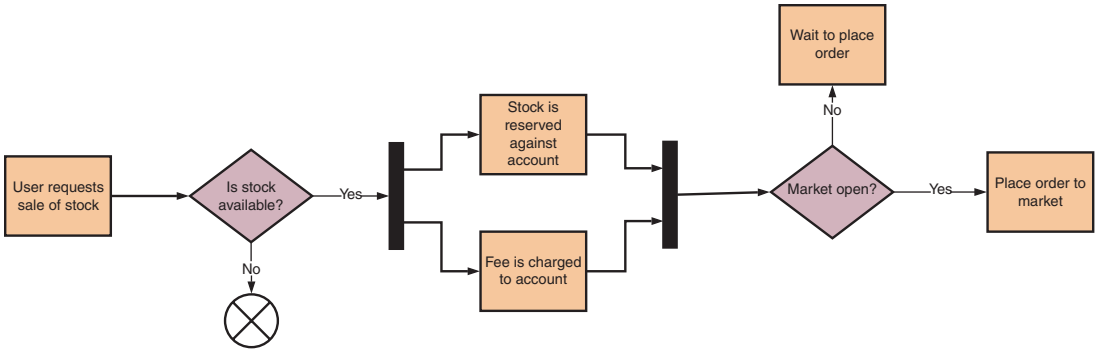


Figure 2.3 The process of placing an order to sell a financial position from an account at SimpleBank

These may be similar to the questions you might ask yourself when designing a feature in a monolithic application, but they have different implications. For example, the effort required to deploy a new service is inherently higher than creating a new module. In scoping microservices, you need to ensure that the benefits of dividing up your system aren't outweighed by added complexity.

> **NOTE** As the application evolves, these questions will take on added dimensions. Later, we'll also ask whether to add functionality to existing services or carve those services up. We'll explore this further in chapters 4 and 5.

As we discussed earlier, each service should be responsible for a single capability. Your first step will be to identify the distinct business capabilities you want to implement and the relationship between those capabilities.

2.3.1 Identifying microservices by modeling the domain

To identify the business capabilities you want, you need to develop your understanding of the domain where you're building software. This is normally the hard work of product discovery or business analysis: research; prototyping; and talking to customers, colleagues, or other end users.

Let's start by exploring the order placement example from figure 2.3. What value are you trying to deliver? At a high level, a customer wants to be able to place an order. So, an obvious business capability will be the ability to store and manage the state of those orders. This is your first microservice candidate.

Continuing our exploration of the example, you can identify other functionalities your application needs to offer. To sell something, you need to own it, so you need some way of representing a customer's current holdings resulting from the transactions that have occurred against their account. Your system needs to send an order to a broker—the application needs to be able to interact with that third party. In fact, this one feature,

placing a sell order, will require SimpleBank's application to support all of the following functionality:

- Record the status and history of sell orders
- Charge fees to the customer for placing an order
- Record transactions against the customer's account
- Place an order onto a market
- Provide valuation of holdings and order to customer

It's not a given that each function maps to a single microservice. You need to determine which functions are cohesive—they belong together. For example, transactions resulting from orders will be similar to transactions resulting from other events, such as dividends being paid on a share. Together, a group of functions forms a capability that one service may offer.

Let's map these functions to business capabilities—what the business does. You can see this mapping in figure 2.4. Some functions cross multiple domains, such as fees.

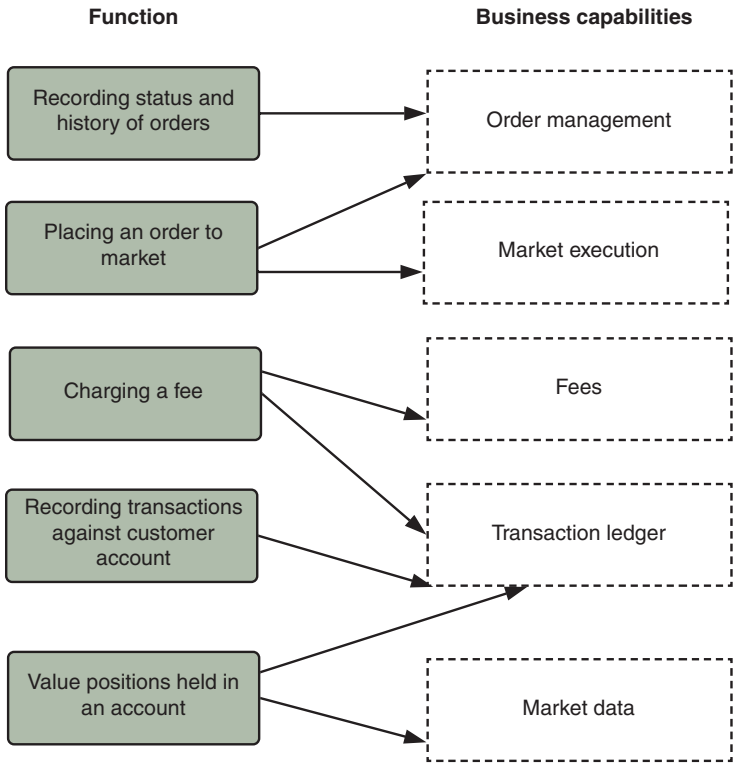


Figure 2.4 The relationship between application functionality and capabilities within SimpleBank's business

You can start by mapping these capabilities directly to microservices. Each service should reflect a capability that the business offers—this results in a good balance of size versus responsibility. You also should consider what would drive a microservice to change in the future—whether it truly has single responsibility. For example, you could argue that market execution is a subset of order management and therefore shouldn't be a separate service. But the drivers for change in that area are the behavior and scope of the markets you're supporting, whereas order management relates more closely to the types of product and the account being used to trade. These two areas don't change together. By separating them, you isolate areas of volatility and maximize cohesiveness (figure 2.5).

Some microservice practitioners would argue that microservices should more closely reflect single functions, rather than single capabilities. Some have even suggested that microservices are "append only" and that it's always better to write new services than to add to existing ones.

We disagree. Decomposing too much can lead to services that lack cohesiveness and tight coupling between closely related collaborators. Likewise, deploying and monitoring many services might be beyond the abilities of the engineering team in the early days of a microservice implementation. A useful rule of thumb is to err on the side of larger services; it's often easier to carve out functionality later if it becomes more specialized or more clearly belongs in an independent service.

Lastly, keep in mind that understanding your domain isn't a one-off process! Over time, you'll continue to iterate on your understanding of the domain; your users' needs will change, and your product will continue to evolve. As this understanding changes, your system itself will change to meet those needs. Luckily, as we discussed in chapter 1, coping with changing needs and requirements is a strength of the microservices approach.

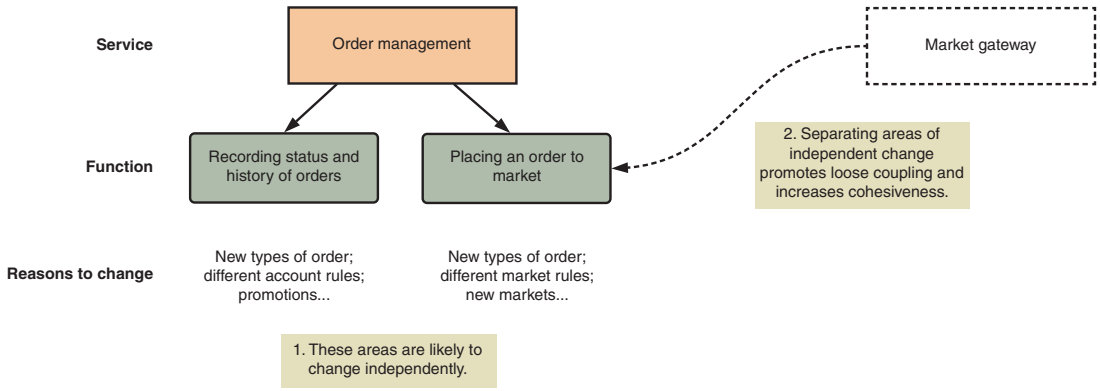


Figure 2.5 Services should isolate reasons to change to promote loose coupling and single responsibility.

2.3.2 Service collaboration

We've identified several microservice candidates. These services need to collaborate with each other to do something useful for SimpleBank's customers.

As you may already know, service collaboration can be either point-to-point or event-driven. Point-to-point communication is typically synchronous, whereas event-driven communication is asynchronous. Many microservice applications begin by using synchronous communication. The motivations for doing so are twofold:

- Synchronous calls are typically simpler and more explicit to reason through than asynchronous interaction. That said, don't fall into the trap of thinking they share the same characteristics as local, in-process function calls—requests across a network are significantly slower and more unreliable.
- Most, if not all, programming ecosystems already support a simple, language-agnostic transport mechanism with wide developer mindshare: HTTP, which is mainly used for synchronous calls but you can also use asynchronously.

Consider SimpleBank's order placement process. The orders service is responsible for recording and placing an order to market. To do this, it needs to interact with your market, fees, and account transaction services. This collaboration is illustrated in figure 2.6.

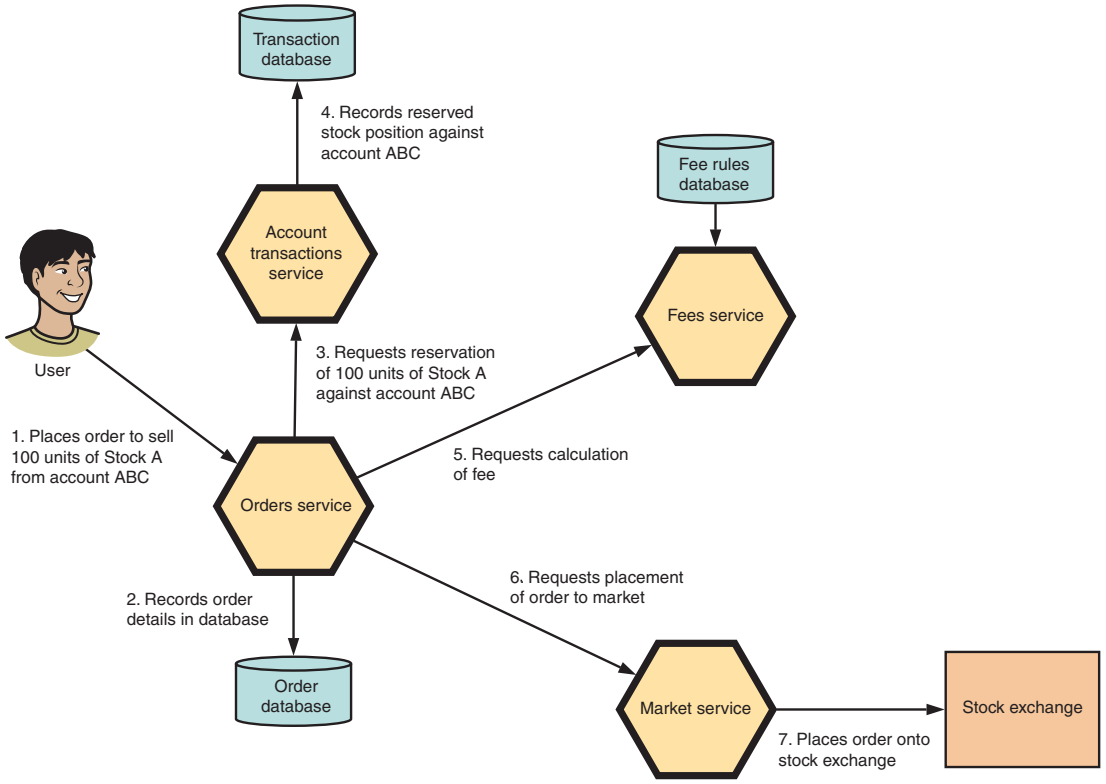


Figure 2.6 The orders service orchestrates the behavior of several other services to place an order to market.

Earlier, we pointed out that microservices should be autonomous, and to achieve that, services should be loosely coupled. You achieve this partly through the design of your services, "[gathering] together the things that change for the same reasons" to minimize the chance that changes to one service require changes to its upstream or downstream collaborators. You also need to consider *service contracts* and *service responsibility*.

Service contracts

The messages that each service accepts, and the responses it returns, form a contract between that service and the services that rely on it, which you can call *upstream collaborators*. Contracts allow each service to be treated as a black box by its collaborators: you send a request and you get something back. If that happens without errors, the service is doing what it's meant to do.

Although the implementation of a service may change over time, maintaining contract-level compatibility ensures two things:

1. Those changes are less likely to break consumers.
2. Dependencies between services are explicitly identifiable and manageable.

In our experience, contracts are often implicit in naïve or early microservice implementations; they're suggested by documentation and practice, rather than explicitly codified. As the number of services grows, you can realize significant benefit from standardizing the interfaces between them in a machine-readable format. For example, REST APIs may use Swagger/OpenAPI. As well as aiding the conformance testing of individual services, publishing standardized contracts will help engineers within an organization understand how to use available services.

Service responsibility

You can see in figure 2.6 that the orders service has a lot of responsibility. It directly orchestrates the actions of every other service involved in the process of placing an order. This is conceptually simple, but it has downsides. At worst, our other services become anemic, with many dumb services controlled by a small number of smart services, and those smart services grow larger

This approach can lead to tighter coupling. If you want to introduce a new part of this process—let's say you want to notify a customer's account manager when a large order is placed—you're forced to deploy new changes to the orders service. This increases the cost of change. In theory, if the orders service doesn't need to synchronously confirm the result of an action—only that it's received a request—then it shouldn't need to have any knowledge of those downstream actions.

2.3.3 Service choreography

Within a microservice application, services will naturally have differing levels of responsibility. But you should balance orchestration with *choreography*. In a choreographed system, a service doesn't need to directly command and trigger actions in other services. Instead, each service owns specific responsibilities, which it performs in reaction to other events.

Let's revisit the earlier design and make a few tweaks:

1. When someone creates an order, the market might not currently be open. Therefore, you need to record what status an order is in: created or placed. Placement of an order doesn't need to be synchronous.
2. You'll only charge a fee once an order is placed, so charging fees doesn't need to be synchronous. In fact, it should happen in reaction to the market service, rather than being orchestrated by the orders service.

Figure 2.7 illustrates the changed design. Adding events adds an architectural concern: you need some way of storing them and exposing them to other applications. We'd recommend using a message queue for that purpose, such as RabbitMQ or SQS.

In this design, we've removed the following responsibility from the orders service:

- *Charging fees* — The orders service has no awareness that a fee is being charged once an order is being placed to market.
- *Placing orders* — The orders service has no direct interaction with the market service. You could easily replace this with a different implementation, or even a service per market, without needing to change the orders service itself.

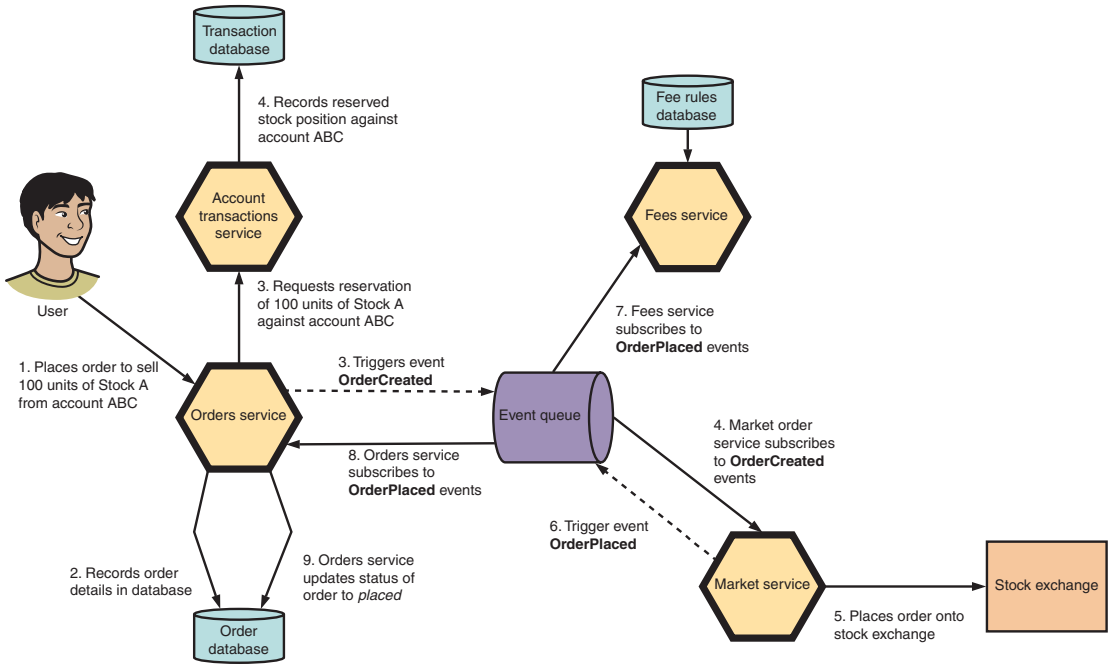


Figure 2.7 You choreograph the behavior of other services through events, reducing the coordinating role of the orders service. Note that some actions, for example, the two actions numbered "3.," happen concurrently.

The orders service itself also reacts to the behavior of other services by subscribing to the `OrderPlaced` event emitted by the market service. You can easily extend this to further requirements; for example, the orders service might subscribe to `TradeExecuted` events to record when the sale has been completed on the market or `OrderExpired` events if the sale can't be made within a certain timeframe.

This setup is more complex than the original synchronous collaboration. But by favoring choreography where possible, you'll build services that are highly decoupled and therefore independently deployable and amenable to change. These benefits do come at a cost: a message queue is another piece of infrastructure to manage and scale and itself can become a single point of failure.

The design we've come up with also has some benefit in terms of resiliency. For example, failure in the market service is isolated from failure in the orders service. If placing an order fails, you can replay that event[3] later, once the service is available, or expire it if too much time passes. On the other hand, it's now more difficult to trace the full activity of the system, which you'll need to consider when you think about how to monitor these services in production.

2.4 Exposing services to the world

So far, we've explored how services collaborate to achieve some business goal. How do you expose this functionality to a real user application?

SimpleBank wants to build both web and mobile products. To do this, the engineering team have decided to build an API gateway as a façade over these services. This abstracts away backend concerns from the consuming application, ensuring it doesn't need to have any awareness of underlying microservices, or how those services interact with each other to deliver functionality. An API gateway delegates requests to underlying services and transforms or combines their responses as appropriate to the needs of a public API.

Imagine the user interface of a place order screen. It has four key functions:

- Displaying information about the current holdings within a customer's account, including both quantity and value
- Displaying market data showing prices and market movements for a holding
- Inputting orders, including cost calculation
- Requesting execution of those orders against the specified holdings

Figure 2.8 illustrates how an API gateway serves that functionality, and how that gateway collaborates with underlying services.

The API gateway pattern is elegant but has a few downsides. Because it acts as a single composition point for multiple services, it'll become large and possibly unwieldy. It may be a temptation to add business logic in the gateway, rather than treating it as a proxy alone. It can suffer from trying to be all things to all applications: whereas a mobile customer application may want a smaller, cut-down payload, but an internal administration web application might require significantly more data. It can be hard to balance these competing forces while building a cohesive API.

[3] Assuming the queue itself is persistent.

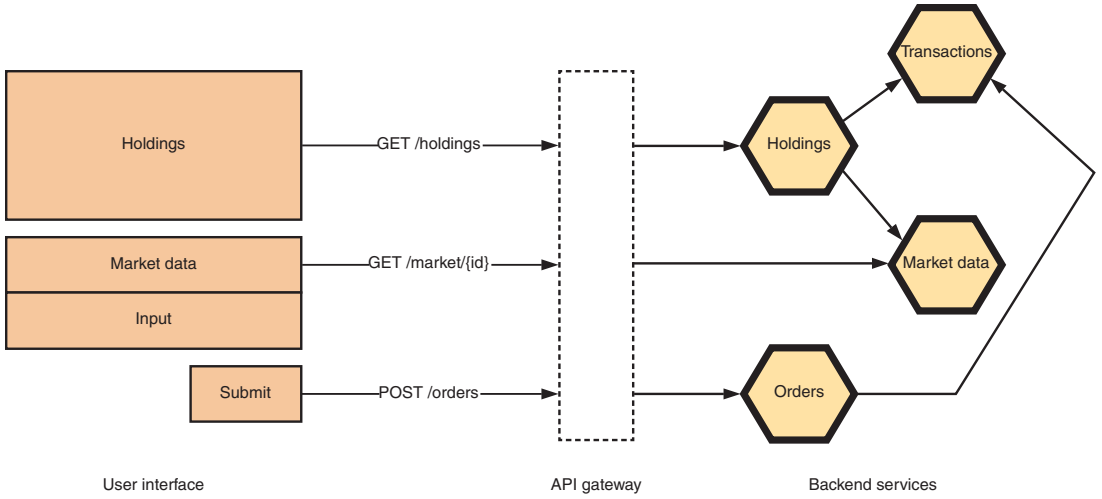


Figure 2.8 A user interface, such as a web page or mobile app, interacts with the REST API that an API gateway exposes. The gateway provides a façade over underlying microservices and proxies requests to appropriate backend services.

> **NOTE** We'll revisit the API gateway pattern and discuss alternative approaches in chapter 3.

2.5 Taking your feature to production

You've designed a feature for SimpleBank that involves the interaction of multiple services, an event queue, and an API gateway. Let's say you've taken the next step: you've built those services and now the CEO is pushing you to get them into production.

In public clouds like AWS, Azure, or GCE, the obvious solution is to deploy each service to a group of virtual machines. You could use load balancers to spread load evenly across instances of each web-facing service, or you could use a managed event queue, such as AWS's Simple Queue Service, to distribute events between services.

> **NOTE** An in-depth discussion of effective infrastructure automation and management is outside the scope of this book. Most cloud providers provide this capability through custom tooling, such as AWS's CloudFormation or Elastic Beanstalk. Alternatively, you could consider open source tools, such as Chef or Terraform.

Anyway—you compiled that code, FTP'd it onto those VMs, got the databases up and running, and tried some test requests. This took a few days. Figure 2.9 shows your production infrastructure.

For a few weeks, that didn't work too badly. You made a few changes and pushed out the new code. But soon you started to run into trouble. It was hard to tell if the services

were working as expected. Worse, you were the only person at SimpleBank who knew how to release a new version. Even worse than that, the guy who wrote the transaction service went on vacation for a few weeks, and no one knew how the service was deployed. These services would have a *bus factor* of 1—suggesting they wouldn't survive the disappearance of any team member.

> **DEFINITION** bus factor is a measurement of the risk of knowledge not being shared between multiple team members, from the phrase "in case they get hit by a bus." It's also known as *truck factor*. The lower bus factors are, the worse they are.

Something was definitely wrong. You remembered that in your last job at GiantBank, the infrastructure team managed releases. You'd log a ticket, argue back and forth, and after a few weeks, you'd have what you needed…or sometimes not, so you'd log another ticket. That doesn't seem like the right approach either. In fact, you were glad that using microservices allowed you to manage deployment.

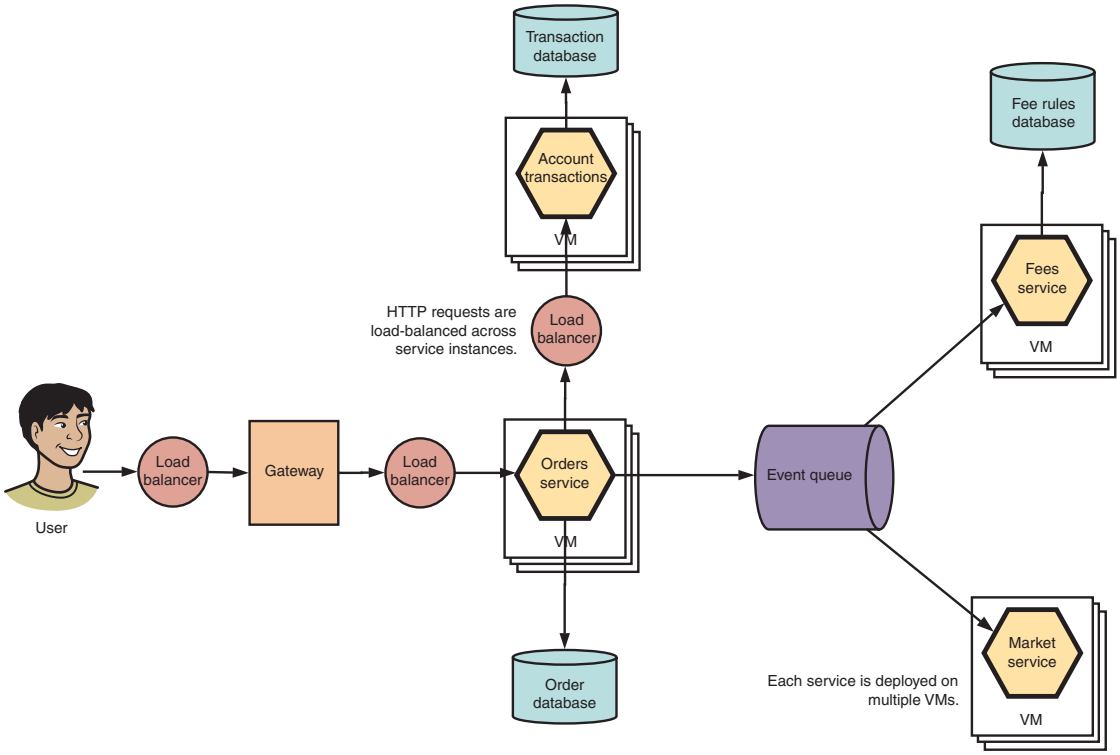


Figure 2.9 In a simple microservices deployment, requests to each service are load balanced across multiple instances, running across multiple virtual machines. Likewise, multiple instances of a service may subscribe to a queue.

It's safe to say that your services weren't ready for production. Running microservices requires a level of operational awareness and maturity from an engineering team beyond what's typical in a monolithic application. You can only say a service is production ready if you can confidently trust it to serve production workloads.

How can you be confident a service is trustworthy? Let's start with a list of questions you might need to consider to achieve production readiness:

- *Reliability*—Is your service available and error free? Can you rely on your deployment process to push out new features without introducing instability or defects?
- *Scalability*—Do you understand the resource and capacity needs of a service? How will you maintain responsiveness under load?
- *Transparency*—Can you observe a service in operation through logs and metrics? If something goes wrong, is someone notified?
- *Fault tolerance*—Have you mitigated single points of failure? How do you cope with the failure of other service dependencies?

At this early stage in the lifetime of a microservice application, you need to establish three fundamentals:

- Quality-controlled and automated deployments
- Resilience
- Transparency

Let's examine how these fundamentals will help you address the problems that SimpleBank has encountered.

2.5.1 Quality-controlled and automated deployment

You'll lose the added development speed you gain from microservices if you can't get them to production rapidly and reliably. The pain of unstable deployments—such as introducing a serious error—will eliminate those speed gains.

Traditional organizations often seek stability by introducing (often bureaucratic) change control and approval processes. They're designed to manage and limit change. This isn't an unreasonable impulse: if changes introduce most bugs[4]—costing the company thousands (or millions) of dollars of engineering effort and lost revenue—then you should closely control those changes.

In a microservice architecture, this won't work, because the system will be in a state of continuous evolution; it's this freedom that gives rise to tangible innovation. But to ensure that freedom doesn't lead to errors and outages, you need to be able to trust your development process and deployment. Equally, to enable such freedom in the first place, you also need to minimize the effort required to release a new service

[4] "SRE has found that roughly 70% of outages are due to changes in a live system." Benjamin Treynor Sloss, Chapter 1, *Site Reliability Engineering*, 2017, O'Reilly Media, http://mng.bz/7Mm4.

or change an existing one. You can achieve stability through standardization and automation:

- *You should standardize the development process.* You should review code changes, write appropriate tests, and maintain version control of the source code. We hope this doesn't surprise anyone!
- *You should standardize and automate the deployment process.* You should thoroughly validate the delivery of a code change to production, and it should require minimal intervention from an engineer. This is a deployment pipeline.

2.5.2 Resilience

Ensuring a software system is resilient in the face of failure is a complicated task. The infrastructure underpinning your systems is inherently unreliable; even if your code is perfect, network calls will fail and servers will go down. As part of designing a service, you need to consider how it and its dependencies may fail and proactively work to avoid—or minimize the impact of—those failure scenarios.

Table 2.2 examines the potential areas of risk in the system that SimpleBank has deployed. You can see that even a relatively simple microservice application introduces several areas of potential risk and complexity.

Table 2.2 Areas of risk in SimpleBank's microservice application

Area	Possible failures
Hardware	Hosts, data center components, physical network
Communication between services	Network, firewall, DNS errors
Dependencies	Timeouts, external dependencies, internal failures, for example, supporting databases

NOTE Chapter 6 will investigate techniques for maximizing service resilience.

2.5.3 Transparency

The behavior and state of a microservice should be *observable*: at any time, you should be able to determine whether the service is healthy and whether it's processing its workload in the way you expect. If something affects a key metric—say, orders are taking too long to be placed to market—this should send an actionable alert to the engineering team.

We'll illustrate this with an example. Last week, there was an outage at SimpleBank. A customer called and told you she was unable to submit orders. Quick investigation turned up that this was affecting every customer: requests made to the order creation service were timing out. Figure 2.10 illustrates the possible points of failure within that service.

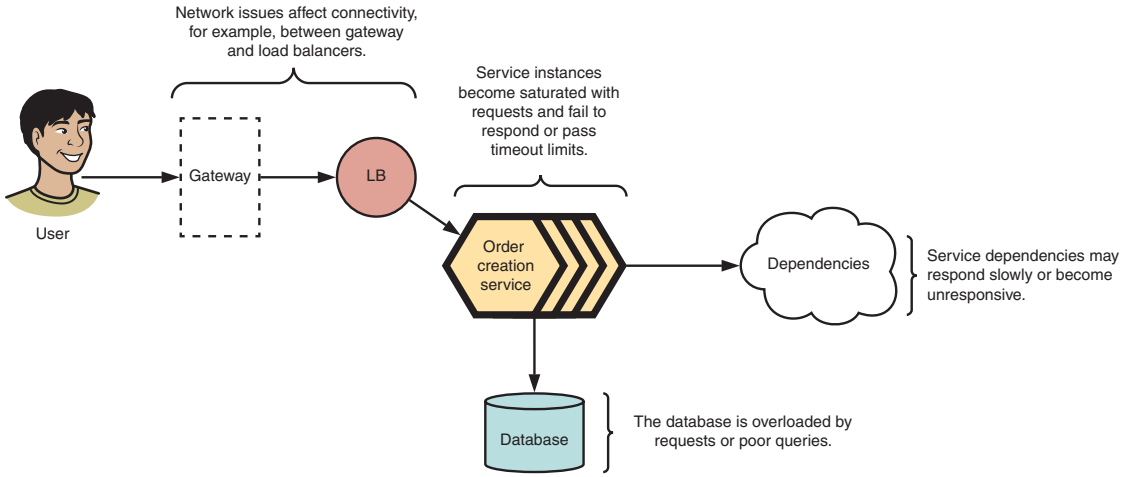


Figure 2.10 A service timeout may be due to several underlying reasons: network issues, problems with service-internal dependencies—such as databases—or unhealthy behavior from other services.

It was clear that you had a major operational problem: you lacked logging to determine exactly what went wrong and where things were falling apart. Through manual testing, you managed to isolate the problem: the account transaction service was unresponsive. Meanwhile, your customers had been unable to place orders for several hours. They weren't happy.

To avoid such problems in the future, you need to add thorough instrumentation to your microservices. Collecting data about application activity—at all layers—is vital to understanding the present and past operational behavior of a microservice application.

As a first step, SimpleBank set up infrastructure to aggregate the basic logs that your services produced, sending them to a service that allowed you to tag and search them.[5] Figure 2.11 illustrates this approach. By doing this, the next time a service failed, the engineering team could use those logs to identify the point where the system began to fail and diagnose the issue precisely where it occurred.

But inadequate logging wasn't the only problem. It was embarrassing that SimpleBank only identified an issue once a customer called. The company should have had alerting in place to ensure that each service was meeting its responsibilities and service goals.

In such cases, in its most simple form, you should have a recurring heartbeat check that happening on each service to alert the team if a service becomes completely unresponsive. Beyond that, a team should commit to operational guarantees for each service. For example, for a critical service, you might aim for 95% of requests to return in under 100ms with 99.99% uptime. Failing to meet these thresholds should result in alerts being sent to the service owners.

[5] Several managed services exist for log aggregation, including Loggly, Splunk, and Sumo Logic. You also can run this function in-house using the well-known ELK (Elasticsearch, Logstash, Kibana) tool stack.

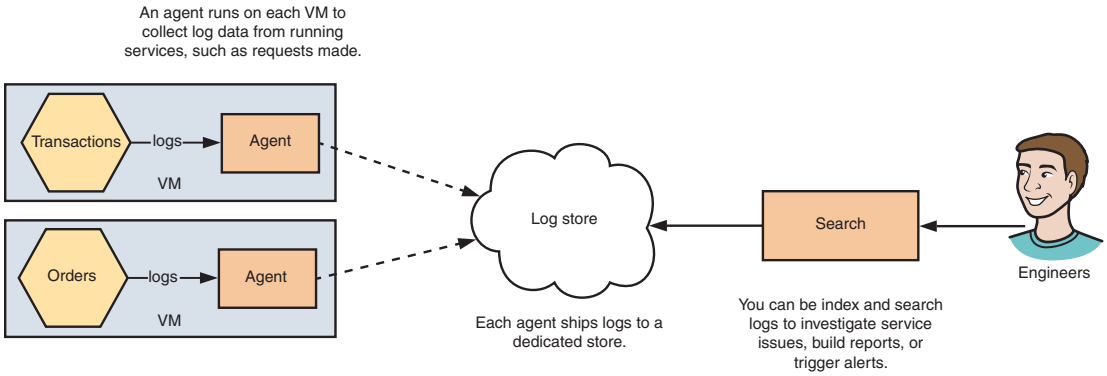


Figure 2.11 You install a logging collection agent on each instance. This ships application log data to a central repository where you can index, search, and analyze it further.

Building thorough monitoring for a microservice application is a complex task. The depth of monitoring you apply will evolve as your system increases in complexity and number of services. As well as the operational metrics and logging we've described, a mature microservice monitoring solution will address business metrics, interservice tracing, and infrastructure metrics. If you are to trust your services, you need to constantly work at making sense of that data.

> **NOTE** In part 4 of this book, we'll discuss monitoring in detail, and how to use tools like Prometheus to trigger alerts and build health dashboards for microservices.

2.6 Scaling up microservice development

The technical flexibility of microservices is a blessing for the speed of development and the effective scalability of a system. But that same flexibility also leads to organizational challenges that change the nature of how an engineering team works at scale. You'll quickly encounter two challenges: *technical divergence* and *isolation*.

2.6.1 Technical divergence

Imagine SimpleBank has built a large microservice system of say 1,000 services. A small team of engineers owns each service, with each team using their preferred languages, their favorite tools, their own deployment scripts, their favored design principles, their preferred external libraries,[6] and so on.

Take a moment to recoil in terror at the sheer weight of effort involved in maintaining and supporting so many different approaches. Although microservices make it possible to choose different languages and frameworks for different services, it's easy to

[6] Unfortunately, this isn't only an issue in microservices, although it's exacerbated by hard component boundaries and explicit service ownership. Earlier in my career, I encountered a single Ruby project that used six different HTTP client libraries!

see that without choosing reasonable standards and limits, the system will become an unimaginable and fragile sprawl.

It's easy to see this frustration emerge on a smaller scale. Consider two services—account transactions and orders—that two different teams own. The first service produces well-structured log output for every request, including helpful diagnostic information such as timings, a request ID, and the currently released revision ID:

```
service=api
git_commit=d670460b4b4aece5915caf5c68d12f560a9fe3e4
request_id=55f10e07-ec6c
request_ip=1.2.3.4
request_path=/users
response_status=500
error_id=a323-da321
parameters={ id: 1 }
user_id=123
timing_total_ms=223
```

The second service produces anemic messages in a difficult to parse format:

```
Processed /users in 223ms with response 500
```

You can see that even in this simple example of log message format, consistency and standardization would make it easier to adequately diagnose issues and trace requests across multiple services. It's crucial to agree on reasonable standards at all layers of your microservice system to manage divergence and sprawl.

2.6.2 Isolation

In chapter 1, we mentioned Conway's Law. In an organization that works with microservices, the inverse of this law is likely to be true: the structure of the company is determined by the architecture of its product.

This suggests that development teams will increasingly reflect microservices: they'll be highly specialized to do one thing well. Each team will own and be accountable for several closely related microservices. Taken collectively, the developers will know everything there is to know about a system, but individually they'll have a narrow area of specialization. As SimpleBank's customer base and product complexity grow, this specialization will deepen.

This configuration can be immensely challenging. Microservices have limited value by themselves and don't function in isolation. Therefore, these independent teams must collaborate closely to build an application that runs seamlessly, even though their goals as a team likely relate to their own narrower area of ownership. Likewise, a narrow focus may tempt a team to optimize for their local problems and preferences, rather than the needs of the whole organization. At its worst, this could lead to conflict between teams, in turn leading to slower deployment and a less reliable product.

2.7 What's next?

In this chapter, we established that microservices were a good fit for SimpleBank, designed a new feature, and considered how you might make that feature production ready. We hope this case study has shown that a microservice-driven approach to application development is both compelling and challenging!

Throughout the rest of this book, we'll teach you the techniques and tools you need to know to run a great microservice application. Although microservices can lead to both flexible and highly productive development, running multiple distributed services is much more demanding than running a single application. To avoid instability, you need be able to design and deploy services that are production ready: transparent, fault-tolerant, reliable, and scalable.

In part 2, we'll focus on design. Effectively designing a system of distributed, interdependent services requires careful consideration of your system domain and how those services interact. Being able to identify the right boundaries between responsibilities—and therefore build highly cohesive and loosely coupled services—is one of the most valuable skills for any microservice practitioner.

Summary

- Microservices are highly applicable in systems with multiple dimensions of complexity—for example, breadth of product offering, global deployment, and regulatory pressures.
- It's crucial to understand the product domain when designing microservices.
- Service interactions may be orchestrated or choreographed. The latter adds complexity but can lead to a more loosely coupled system.
- API gateways are a common pattern for abstracting away the complexity of a microservice architecture for front-end or external consumers.
- You can say a service is production ready if you can trust it to serve production workloads.
- You can be more confident in a service if you can reliably deploy and monitor it.
- Service monitoring should include log aggregation and service-level health checks.
- Microservices can fail because of problems with hardware, communication, and dependencies, not just defects in code.
- Collecting business metrics, logs, and interservice traces is vital to understanding the present and past operational behavior of a microservice application.
- Technical divergence and isolation will become increasingly challenging for an engineering organization as the number of microservices (and supporting teams) increases.
- Avoiding divergence and isolation requires standards and best practices to be similar across multiple teams, regardless of technical underpinnings.

Part 2

Design

In this part of the book, we'll explore the design of microservice applications. We'll start with a big-picture view—the architecture of an entire application—and then drill down to explore how to scope services and connect them together. You'll learn how to design services that are reliable and a microservice framework that's reusable.

3 Architecture of a microservice application

This chapter covers

- The big picture view of a microservice application
- The four tiers of microservice architecture: platform, service, boundary, and client
- Patterns for service communication
- Designing API gateways and consumer-driven façades as application boundaries

In chapter 2, we designed a new feature for SimpleBank as a set of microservices and discovered that deep understanding of the application domain is one of the keys to a successful implementation. In this chapter, we'll look at the bigger picture and consider the design and architecture of an entire application made up of microservices. We can't give you a deep understanding of the domain your own application lives in, but we can show you how having such an understanding will help you build a system that's flexible enough to grow and evolve over time.

You'll see how a microservice application is typically designed to have four tiers—platform, service, boundary, and client—and you'll learn what they are and how they combine to deliver customer-facing applications. We'll also highlight the role of an event backbone in building a large-scale microservice application and

discuss different patterns for building application boundaries, such as API gateways. Lastly, we'll touch on recent trends in building user interfaces for microservice applications, such as micro-frontends and frontend composition.

3.1 Architecture as a whole

As a software designer, you want to build software that's amenable to change. Many forces put pressure on your software: new requirements, defects, market demands, new customers, growth, and so on. Ideally, you can respond to these pressures at a steady pace and with confidence. For you to be able to do that, your development approach should reduce friction and minimize risk.

Your engineering organization will want to remove any roadblocks to development as time goes by and the system evolves. You want to be able to quickly and seamlessly replace any system's component that becomes obsolete. You want to have teams in place that can become completely autonomous and responsible for portions of a larger system. And you want those teams to coexist without the need for constant synchronization and without blocking other teams. For that, you need to think about architecture: your plan for building an application.

3.1.1 From monolith to microservices

With a monolithic application, your primary deliverable is a single application. That application is split horizontally into different technical layers—in a typical three-tier application, they'd be *data*, *logic*, and *presentation* (figure 3.1)—and vertically into different business domains. Patterns like MVC and frameworks like Rails and Django reflect the three-tier model. Each tier provides services to the tier above: the data tier provides persistent state; the logic tier executes useful work; and the presentation layer presents the results back to the end user.

An individual microservice is similar to a monolith: it stores data, performs some business logic, and returns data and outcomes to consumers through APIs. Each microservice owns a business or technical capability of the application and interacts with other microservices to execute work. Figure 3.2 illustrates the high-level architecture of an individual service.

> **NOTE** Chapter 4 discusses microservice scoping—how to define the boundaries and responsibilities of a microservice—in detail.

In a monolithic application, your architecture is limited to the boundaries of the application itself. In a microservice application, you're planning for something that'll keep evolving both in size and breadth. Think of it like a city: building a monolith is like building a skyscraper; whereas building a microservice application is like building a neighborhood: you need to build infrastructure (plumbing, roads, cables) and plan for growth (zone for small businesses versus houses).

This analogy highlights the importance of considering not only the components themselves, but also the way they connect, where they're placed, and how you can build them concurrently. You want your plan to encourage growth along good lines, rather than dictate or enforce a certain structure on your overall application.

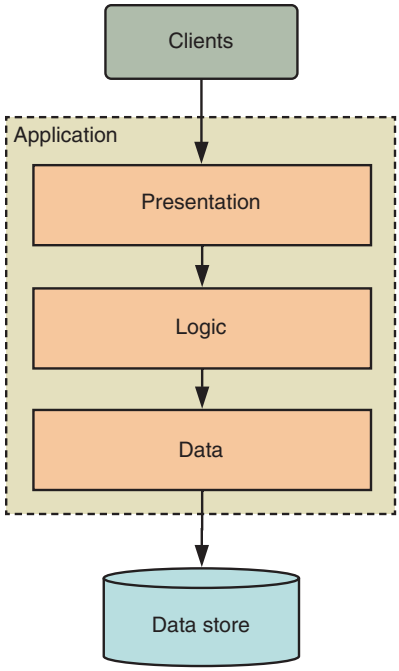


Figure 3.1 The architecture of a typical three-tier monolithic application

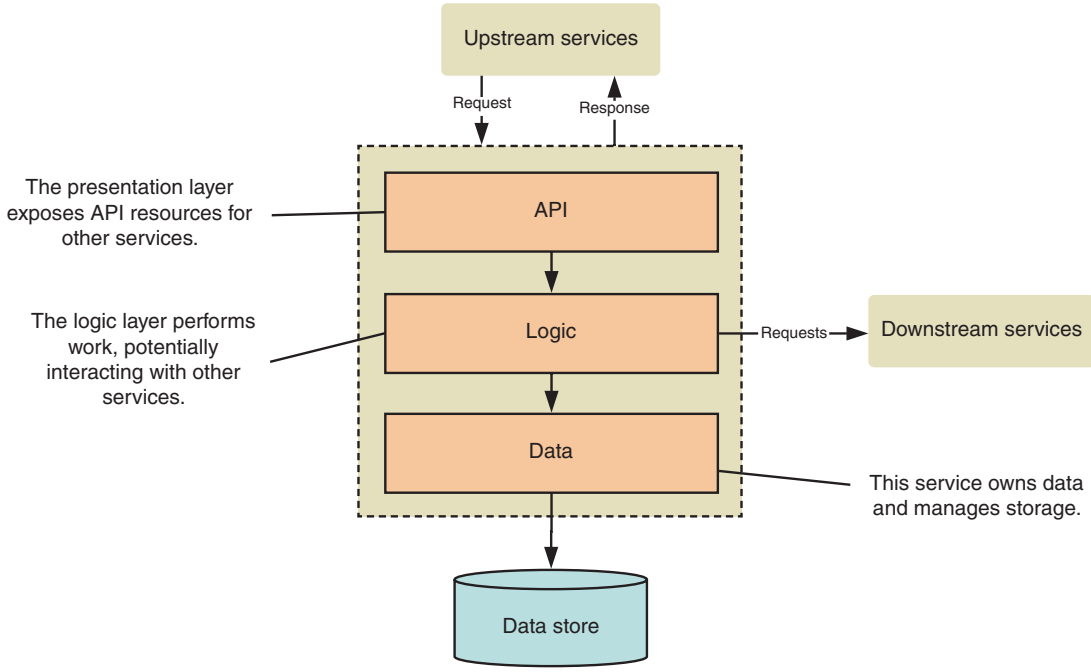


Figure 3.2 The high-level architecture of an individual microservice

Mostly importantly, you don't run microservices in isolation; each microservice lives in an environment that enables you to build, deploy, and run it, in concert with other microservices. Your application architecture should encompass that whole environment.

3.1.2 The role of an architect

Where do software architects fit in? Many enterprises employ software architects, although the effectiveness of and the approach to this role varies wildly.

Microservice applications enable rapid change: they evolve over time as teams build new services, decommission existing services, refactor existing functionality, and so on. As an architect or technical lead, your job is to enable evolution, rather than dictate design. If the microservice application is a city, then you're a planner for the city council.

An architect's role is to make sure the technical foundations of the application support a fast pace and fluidity. An architect should have a global perspective and make sure the global needs of the application are met, guiding its evolution so that

- The application is aligned to the wider strategic goals of the organization.
- Teams share a common set of technical values and expectations.
- Cross-cutting concerns—such as observability, deployment, and interservice communication—meet the needs of multiple teams.
- The whole application is flexible and malleable in the face of change.

To achieve these things, an architect should guide development in two ways:

- *Principles*—Guidelines that the team should follow to achieve higher level technical or organizational goals
- *Conceptual models*—High-level models of system relationships and application-level patterns

3.1.3 Architectural principles

Principles are guidelines (or sometimes rules) that teams should follow to achieve higher level goals. They inform team practice. Figure 3.3 illustrates this model. For example, if your product goal is to sell to privacy- and security-sensitive enterprises, you might set the following principles:

- Development practices must comply with recognized external standards (for example, ISO 27001).
- All data must be portable and stored with retention limits in mind.
- Personal information must be clearly tracked and traceable through the application.

Principles are flexible. They can and should change to reflect the priorities of the business and the technical evolution of your application. For example, early development might prioritize validating product-market fit, whereas a more mature application might require a focus on performance and scalability.

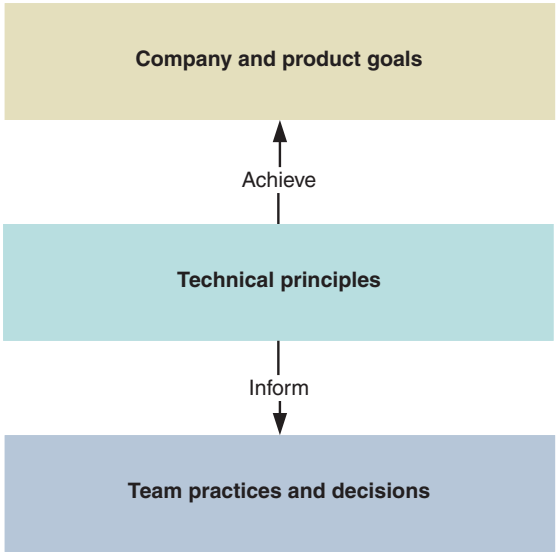


Figure 3.3 An architectural approach based on technical principles

3.1.4 The four tiers of a microservice application

Architecture should reflect a clear high-level conceptual model. A model is a useful tool for reasoning about an application's technical structure. A multi-tiered model, like the three-tier model outlined in figure 3.1, is a common approach to application structure, reflecting layers of abstraction and responsibility within an overall system.

In the rest of this chapter, we'll explore a four-tier model for a microservice application:

- *Platform* — A microservice platform provides tooling, infrastructure, and high-level primitives to support the rapid development, operation, and deployment of microservices. A mature platform layer enables engineers to focus on building features, not plumbing.
- *Services* — In this tier, the services that you build interact with each other to provide business and technical capabilities, supported by the underlying platform.
- *Boundary* — Clients will interact with your application through a defined boundary that exposes underlying functionality to meet the needs of outside consumers.
- *Client* — Client applications, such as websites and mobile applications, interact with your microservice backend.

Figure 3.4 illustrates these architectural layers. You should be able to apply them to any microservice application, regardless of underlying technology choices.

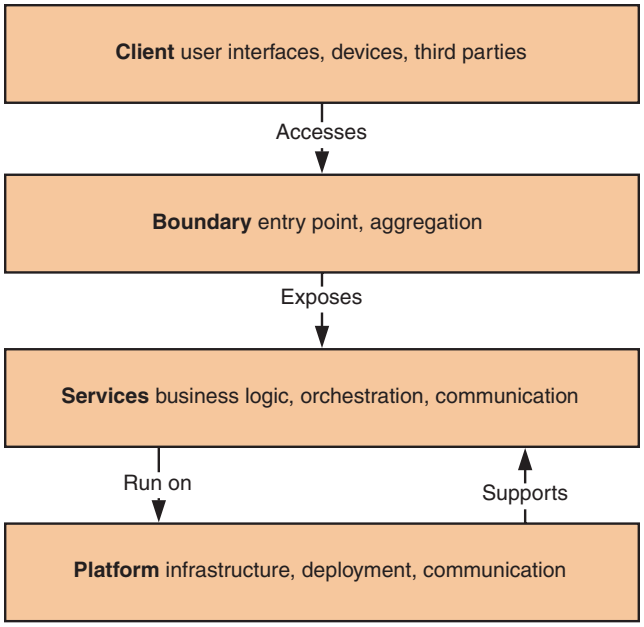


Figure 3.4 A four-tiered model of microservice application architecture

Each layer is built on the capabilities of the layers below; for example, individual services take advantage of deployment pipelines, infrastructure, and communication mechanisms that the underlying microservice platform provides. A well-designed microservice application requires sophistication and investment at all layers.

Great! So now you have a model you can work with. In the next five sections, we'll walk through each layer in this architectural model and discuss how it contributes to building sustainable, flexible, and evolutionary microservice applications.

3.2 A microservice platform

Microservices don't live in isolation. A microservice is supported by infrastructure:

- A deployment target where services are run, including infrastructure primitives, such as load balancers and virtual machines
- Logging and monitoring aggregation to observe service operation
- Consistent and repeatable deployment pipelines to test and release new services and versions
- Support for secure operation, such as network controls, secret management, and application hardening
- Communication channels and service discovery to support service interaction

Figure 3.5 illustrates these capabilities and how they relate to the service layer of the application. If each microservice is a house, then the platform provides roads, water, electricity, and telephone cables.

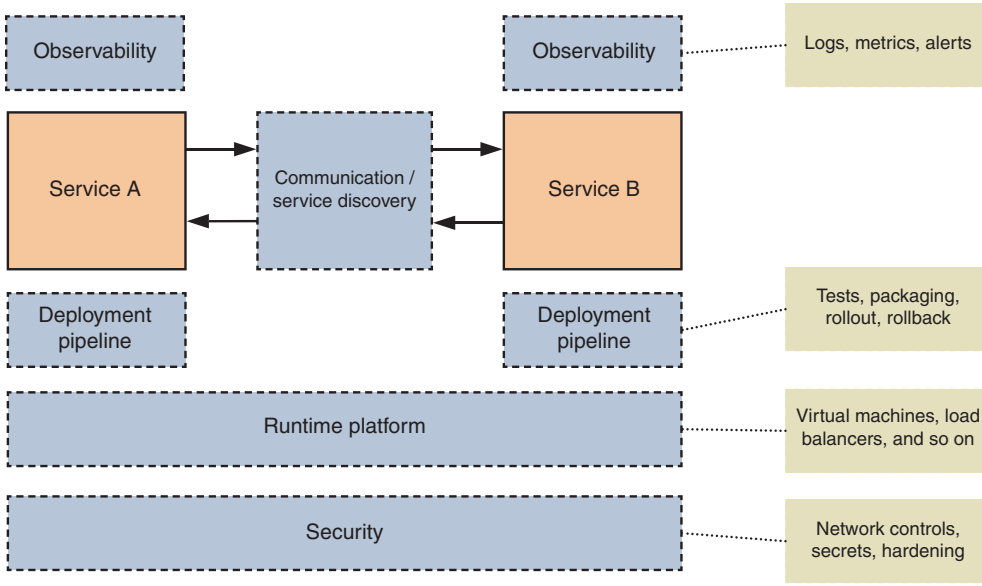


Figure 3.5 The capabilities of a microservice platform

A robust platform layer decreases overall implementation cost, increases overall stability, and enables rapid service development. Without this platform, product developers would need to repeatedly write plumbing code themselves, taking energy away from delivering new features and business impact. The average developer shouldn't need to be an expert in the intricacies of every layer of the application. Ultimately, a semi-independent, specialist team can develop the platform layer to meet the needs of multiple teams working in the service layer of the application.

3.2.1 Mapping your runtime platform

A microservice platform will help you be confident that you can trust the services your team writes to serve production workloads and be resilient, transparent, and scalable. Figure 3.6 maps out a runtime platform for a microservice.

A runtime platform (or deployment target)—for example, a cloud environment like AWS or a platform as a service (PaaS) like Heroku—provides infrastructure primitives necessary to run multiple service instances and route requests between them. In addition, it provides mechanisms for providing configuration—secrets and environment-specific variables—to service instances.

You build the other elements of a microservice platform on top of this foundation. Observability tools collect and correlate data from services and underlying infrastructure. Deployment pipelines manage the upgrade (or rollback) of this stack.

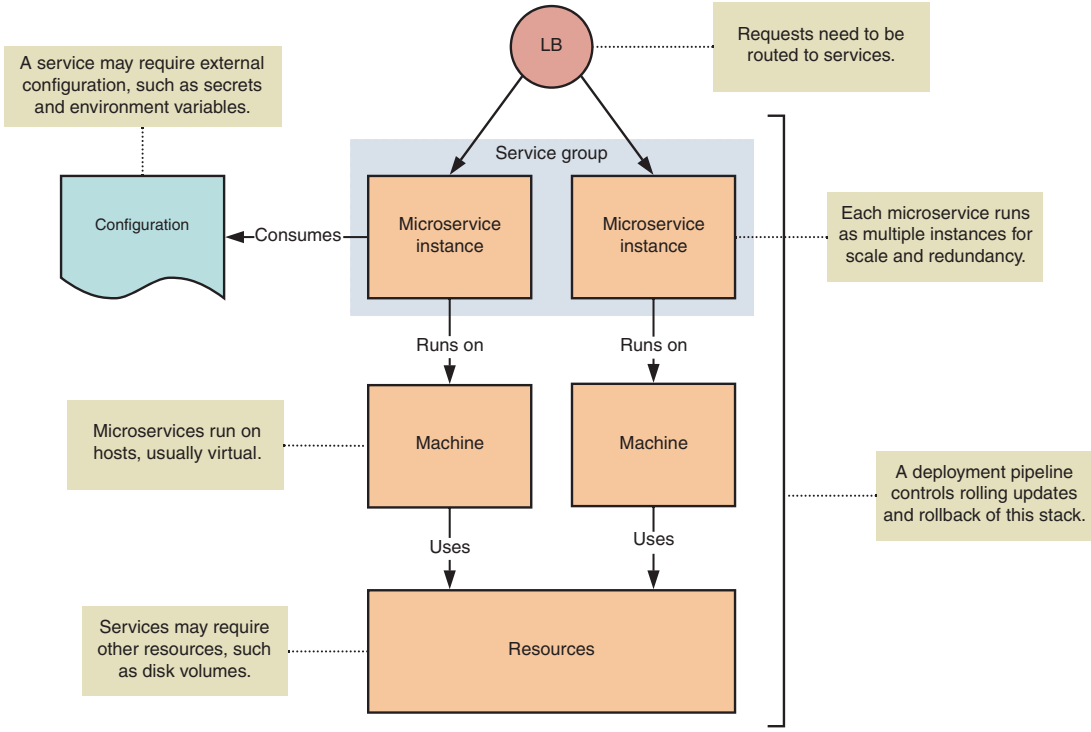


Figure 3.6 A deployment configuration for a microservice running in a typical cloud environment

3.3 Services

The service layer has perhaps the most self-explanatory name—this is where your services live. At this tier, services interact to perform useful work, relying on the underlying platform abstractions for reliable operation and communication and exposing their work through the boundary layer to application clients. We also consider components that are logically internal to a service, such as data stores, to be part of this tier.

The structure of your service tier will differ widely depending on the nature of your business. In this section, we'll discuss some of the common patterns you'll encounter:

- Business and technical capabilities
- Aggregation and higher order services
- Services on critical and noncritical paths

3.3.1 Capabilities

The services you write will implement different capabilities:

- A *business capability* is something that an organization does to generate value and meet business goals. Microservices that you scope to business capabilities directly reflect business goals.

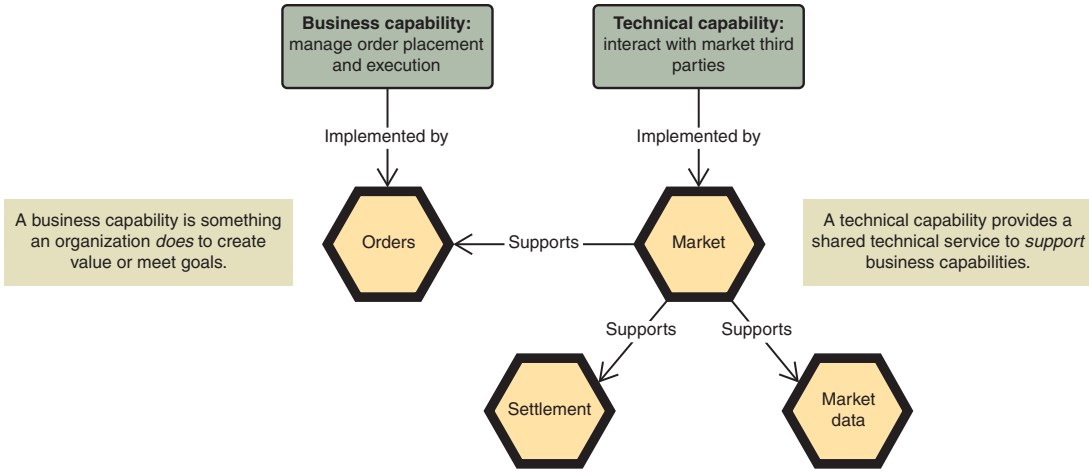


Figure 3.7 Microservices implementing business or technical capabilities

- A *technical capability* supports other services by implementing a shared technical feature.

Figure 3.7 compares these two types of capability. SimpleBank's orders service exposes a capability for managing order execution—this is a business capability. The market service is a technical capability; it provides a gateway to a third party that other services (such as exposing market information or settling trades) can reuse.

> **NOTE** We'll explore when to use business and technical capabilities and how you map them to individual services in the next chapter.

3.3.2 Aggregation and higher order services

In the early days of a microservice application, your services are likely to be *flat*; each service is likely to have a similar level of responsibility. For example, the services in chapter 2—orders, fees, transactions, and accounts—are scoped at a roughly equivalent level of abstraction.

As the application grows, you'll encounter two pressures on the growth of services:

- Aggregating data from multiple services to serve client requests for denormalized data (for example, returning orders and fees together)
- Providing specialized business logic that takes advantage of underlying capabilities (for example, placing a specific type of order)

Over time, these two pressures will lead to a hierarchy of services. Services that are closer to the system boundary will interact with several services to aggregate their output—let's call those *aggregators* (figure 3.8). In addition, specialized services may act as *coordinators* for the work of multiple lower order services.

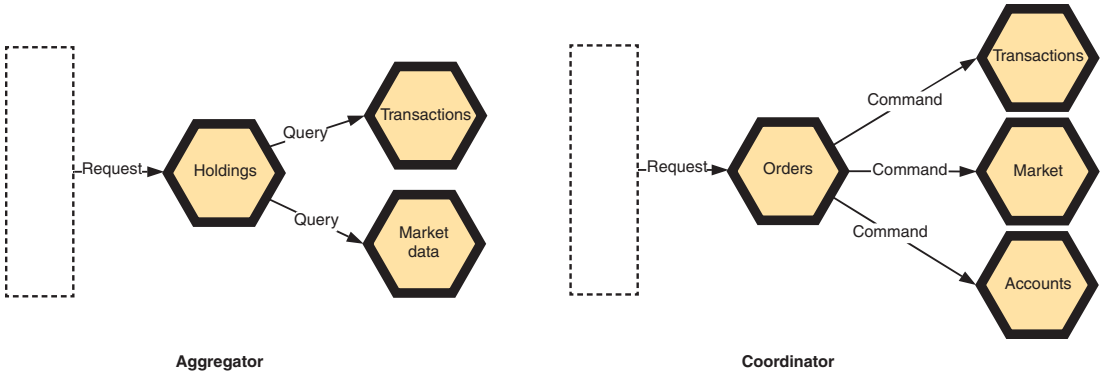


Figure 3.8 An aggregator serves queries by joining data from underlying services, and a coordinator orchestrates behavior by issuing commands to downstream services.

The challenge you'll face is to determine when new data requirements or new application behavior requires a new service, rather than changes to an existing service. Creating a new service increases overall complexity and may result in tight coupling, but adding functionality to an existing service may make it less cohesive and more difficult to replace. That would bend a fundamental microservice principle.

3.3.3 Critical and noncritical paths

As your system evolves, some functions will naturally become more critical to your customer needs—and the successful operation of your business—than others. For example, at SimpleBank, the orders service is on the critical path for order placement. Without this service operating correctly, you can't execute customer orders. Conversely, other services are less important; if the customer profile service is unavailable, it's less likely to affect a critical, revenue-generating component of your offering. Figure 3.9 illustrates example paths at SimpleBank.

This is a double-edged sword. The more services on a critical path, the more likely failure will occur. Because no service is 100% reliable, the cumulative reliability of a service is the product of the reliability of its dependencies.

But microservices allow you to clearly identify these paths and treat them independently, investing more engineering effort to maximize the resiliency and scalability of these paths than you invest in less crucial system areas.

3.4 Communication

Communication is a fundamental element of a microservice application. Microservices communicate with each other to perform useful work. Your chosen methods for microservices to instruct and request action from other microservices determine the shape of the application you build.

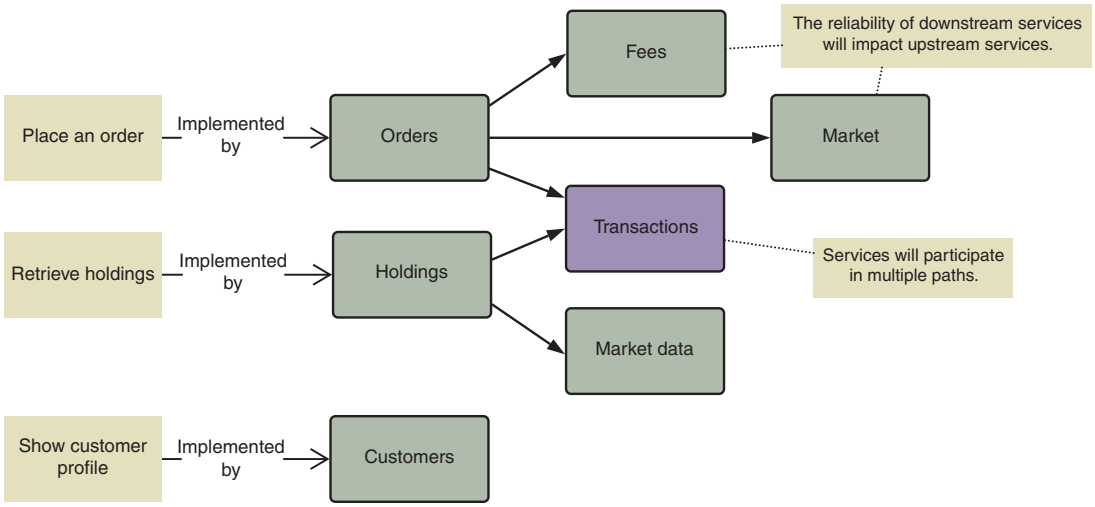


Figure 3.9 Chains of services serve capabilities. Many services will participate in multiple paths.

> **TIP** Network communication is also a primary source of unreliability in a microservice application. In chapter 6, we'll explore techniques for maximizing the reliability of service-to-service communication.

Communication isn't an independent architectural layer, but we've pulled this out into a separate section because it blurs the boundary between the service and platform layers. Some elements—such as communication brokers—are part of the platform layer. But services themselves are responsible for constructing and sending messages. You want to build smart endpoints but dumb pipes.

In this section, we'll discuss common patterns for microservice communication and how they impact the flexibility and evolution of a microservice application. Most mature microservice applications will mix both synchronous and asynchronous interaction styles.

3.4.1 When to use synchronous messages

Synchronous messages are often the first design approach that comes to mind. They're well-suited to scenarios where an action's results—or acknowledgement of success or failure—are required before proceeding with another action.

Figure 3.10 illustrates a request–response pattern for synchronous messages. The first service constructs an appropriate message to a collaborator, which the application sends using a transport mechanism, such as HTTP. The destination service receives this message and responds accordingly.

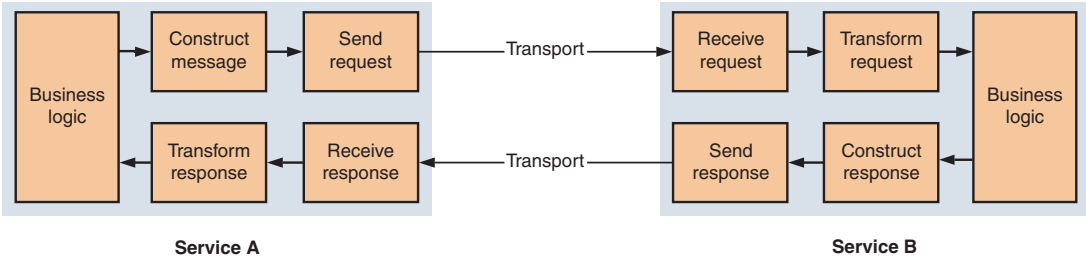


Figure 3.10 A synchronous request–response lifecycle between two communicating services

CHOOSING A TRANSPORT

The choice of transport—RESTful HTTP, an RPC library, or something else—will impact the design of your services. Each transport has different properties of latency, language support, and strictness. For example, gRPC provides generated client/server API contracts using Protobufs, whereas HTTP is agnostic to the context of messages. Across your application, using a single method of synchronous transport has economies of scale; it's easier to reason through, monitor, and support with tooling.

Separation of concerns within microservices is also important. You should separate your choice of transport mechanism from the business logic of your service, which shouldn't need to know about HTTP status codes or gRPC response streams. Doing so makes it easier to swap out different mechanisms in the future if your application's needs evolve.

DRAWBACKS

Synchronous messages have limitations:

- They create tighter coupling between services, as services must be aware of their collaborators.
- They don't have a strong model for broadcast or publish-subscribe models, limiting your capability to perform parallel work.
- They block code execution while waiting on responses. In a thread- or process-based server model, this can exhaust capacity and trigger cascading failures.
- Overuse of synchronous messages can build deep dependency chains, which increases the overall fragility of a call path.

3.4.2 When to use asynchronous messages

An asynchronous style of messaging is more flexible. By announcing events, you make it easy to extend the system to handle new requirements, because services no longer need to have knowledge of their downstream consumers. New services can consume existing events without changing existing services.

> **TIP** Events represent post-hoc state changes. `OrderCreated`, `OrderPlaced`, and `OrderCanceled` are examples of events that the SimpleBank orders service might emit.

This style enables more fluid evolution and creates looser coupling between services. This does come at a cost: asynchronous interactions are more difficult to reason through, because overall system behavior is no longer explicitly encoded into linear sequences. System behavior will become increasingly *emergent*—developing unpredictably from interactions between services—requiring investment in monitoring to adequately trace what's happening.

> **NOTE** Events enable different styles of persistence and querying, such as event sourcing and command query responsibility segregation (CQRS). These aren't a prerequisite for microservices but have some synergies with a microservice approach. We'll explore them in chapter 5.

Asynchronous messaging typically requires a *communication broker*, an independent system component that receives events and distributes them to event consumers. This is sometimes called an *event backbone*, which indicates how central to your application this component becomes (figure 3.11). Tools commonly used as brokers include Kafka, RabbitMQ, and Redis. The semantics of these tools differ: Kafka specializes in high-volume, replayable event storage, whereas RabbitMQ provides higher level messaging middleware (based on the AMQP protocol (https://www.amqp.org/)).

3.4.3 Asynchronous communication patterns

Let's look at the two most common event-based patterns: job queue and publish-subscribe. You'll encounter these patterns a lot when architecting microservices—most higher level interaction patterns are built on one of these two primitives.

JOB QUEUE

In this pattern, workers take jobs from a queue and execute them (figure 3.12). A job should only be processed once, regardless of how many worker instances you operate. This pattern is also known as winner takes all.

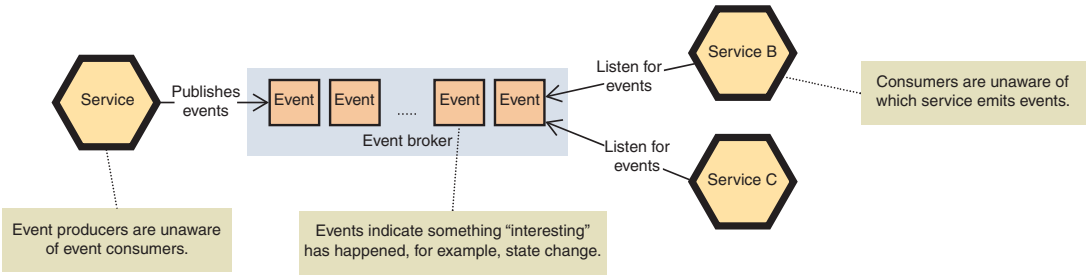


Figure 3.11 Event-driven asynchronous communication between services

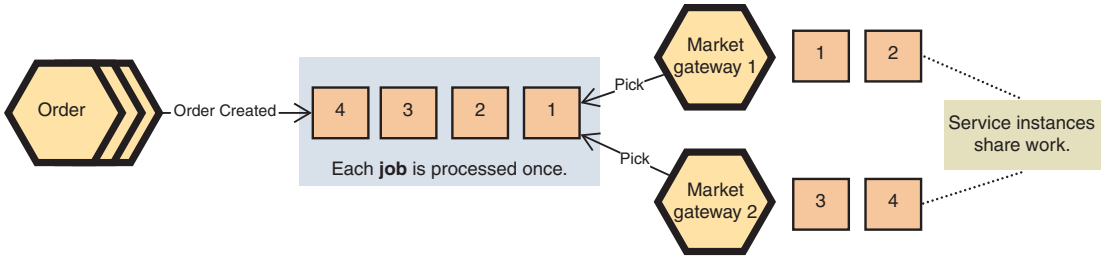


Figure 3.12 A job queue distributes work to 1 to *n* consumers

Your market gateway could operate in this fashion. Each order that the orders service creates will trigger an `OrderCreated` event, which will be queued for the market gateway service to place it. This pattern is useful where

- A 1:1 relationship exists between an event and work to be done in response to that event.
- The work that needs to be done is complex or time-consuming, so it should be done out-of-band from the triggering event.

By default, this approach doesn't require sophisticated event delivery. Many task queue libraries are available that use commodity data stores, such as Redis (Resque, Celery, Sidekiq) or SQL databases.

PUBLISH-SUBSCRIBE

In publish-subscribe, services trigger events for arbitrary listeners. All listeners that receive the event act on it appropriately. In some ways, this is the ideal microservice pattern: a service can send arbitrary events out into the world without caring who acts on them (figure 3.13).

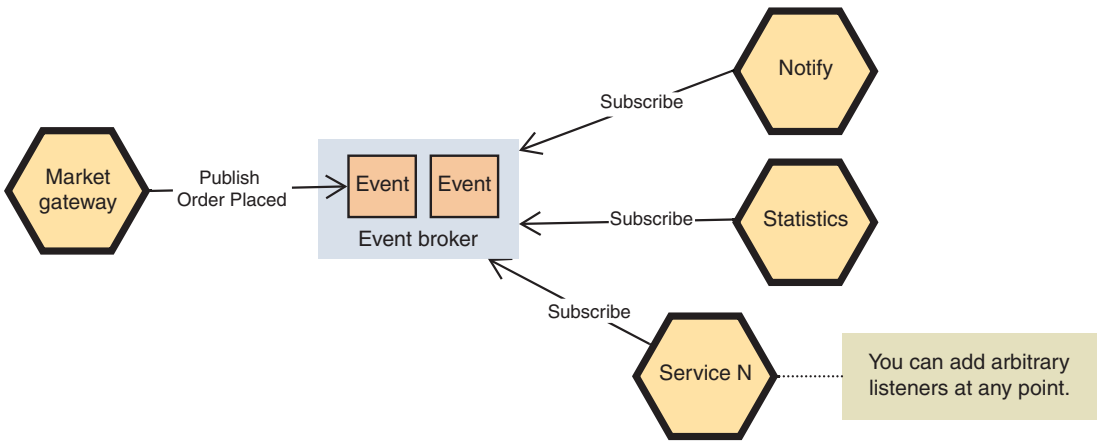


Figure 3.13 How publish-subscribe sends events out to subscribers

For example, imagine you need to trigger other downstream actions once an order has been placed. You might send a push notification to the customer or use it to feed your order statistics and recommendation feature. These features can all listen for the same event.

3.4.4 Locating other services

To wrap up this section, let's take a moment to examine *service discovery*. For services to communicate, they need to be able to *discover* each other. The platform layer should offer this capability.

A rudimentary approach to service discovery is to use load balancers (figure 3.14). For example, an elastic load balancer (ELB) on AWS is assigned a DNS name and manages health checking of underlying nodes, based on their membership in a group of virtual machines (an auto-scaling group on AWS).

This works but doesn't handle more complex scenarios. What if you want to route traffic to different versions of your code to enable canary deployments or dark launches, or if you want to route traffic across different data centers?

A more sophisticated approach is to use a registry, such as Consul (https://www.consul.io). Service instances announce themselves to a registry, which provides an API—either through DNS or a custom mechanism for resolving requests for those services. Figure 3.15 illustrates this approach.

Your service discovery needs will depend on the complexity of your deployed application's topology. More complex deployments, such as geographical distribution, require more robust service discovery architecture.[1]

NOTE When you deploy to Kubernetes in chapter 9, you'll learn about *services*, the mechanism that Kubernetes uses to provide discovery.

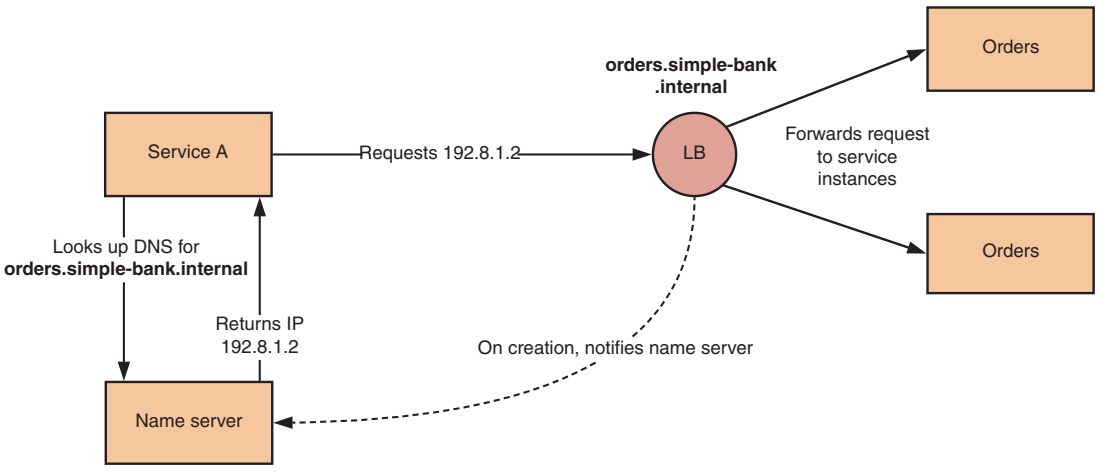


Figure 3.14 Service discovery using load balancers and known DNS names

[1] bit.ly/2o86ShQ is a great place to start if you're interested in further exploring different types of proxies and load balancing.

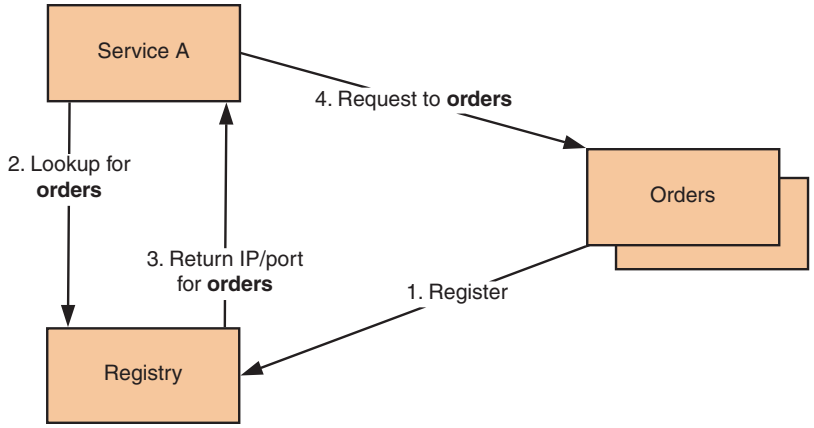


Figure 3.15 Service discovery using a service registry as a source of truth

3.5 The application boundary

A boundary layer provides a façade over the complex interactions of your internal services. Clients, such as mobile apps, web-based user interfaces, or IoT devices, may interact with a microservice application. (You might build these clients yourself, or third parties consuming a public API to your application may build them.) For example, SimpleBank has internal admin tools, an investment website, iOS and Android apps, and a public API, as depicted in figure 3.16.

The boundary layer provides an *abstraction* over internal complexity and change (figure 3.17). For example, you might provide a consistent interface for a client to list all historic orders, but, over time, you might completely refactor the internal implementation of that functionality. Without this layer, clients would require too much knowledge of individual services, becoming tightly coupled to your system implementation.

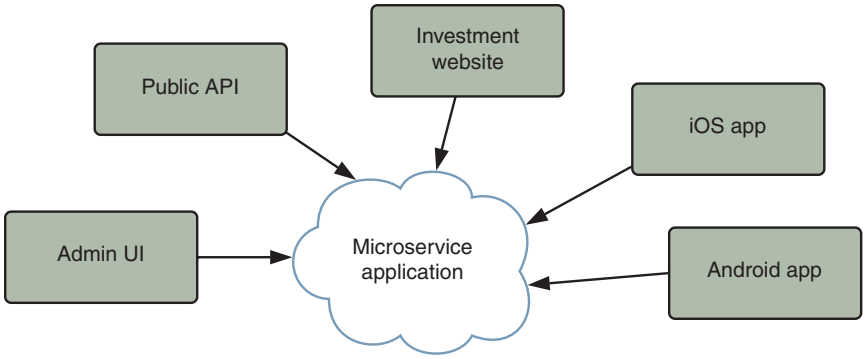


Figure 3.16 Client applications at SimpleBank

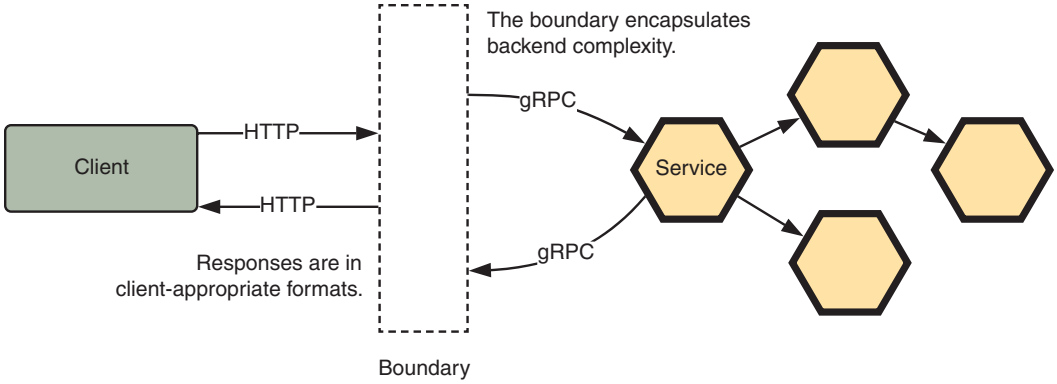


Figure 3.17 A boundary provides a façade over the service layer to hide internal complexity from a consumer.

Second, the boundary tier provides *access* to data and functionality using a transport and content type appropriate to the consumer. For example, whereas services might communicate between each other with gRPC, a façade can expose an HTTP API to external consumers, which is much more appropriate for external applications to consume.

Combining these roles allows your application to become a black box, performing whatever (unknown to the client) operations to deliver functionality. You also can make changes to the service layer with more confidence, because the client interfaces with it through a single point.

The boundary layer also may implement other client-facing capabilities:

- *Authentication and authorization* — To verify the identity and claims of an API client
- *Rate limiting* — To provide defense against client abuse
- *Caching* — To reduce overall load on the backend
- Collect *logs and metrics* — To allow analysis and monitoring of client requests

Placing these *edge capabilities* in the boundary layer provides clear separation of concerns — without a boundary, backend services would need to individually implement these concerns, increasing their complexity.

You might also use boundaries within your service tier to separate domains. For example, an order placement process might consist of several services, but only one of those services should expose an entry point that other domains can access (figure 3.18).

> **NOTE** Internal service boundaries often reflect *bounded contexts*: cohesive, bounded subsets of the overall application domain. We'll explore them more in the next chapter.

That provides an overview for how you can use boundaries. Let's get more specific and explore three different (albeit related) patterns for application boundaries: API gateways, backends for frontends, and consumer-driven gateways.

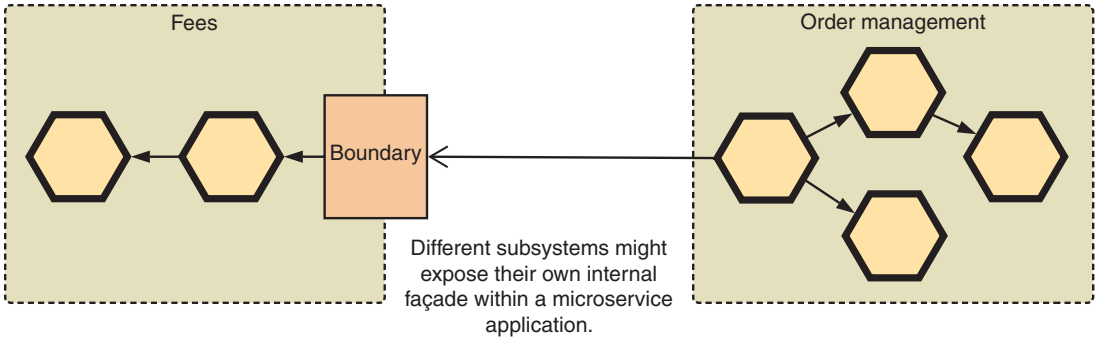


Figure 3.18 Boundaries might be present between different contexts within a microservice application.

3.5.1 API gateways

We introduced the API gateway pattern in chapter 2. An API gateway provides a single client-entry point over a service-oriented backend. It proxies requests to underlying services and transforms their responses. An API gateway might handle other cross-cutting client concerns, such as authentication and request signing.

> **TIP** API gateways that are available include such open source options as Mashape's Kong, as well as commercial offerings, such as AWS API Gateway.

Figure 3.19 illustrates an API gateway. The gateway authenticates a request, and if that succeeds, it proxies the request to an appropriate backend service. It transforms the results it receives so that when it returns them, they're palatable for your consuming clients.

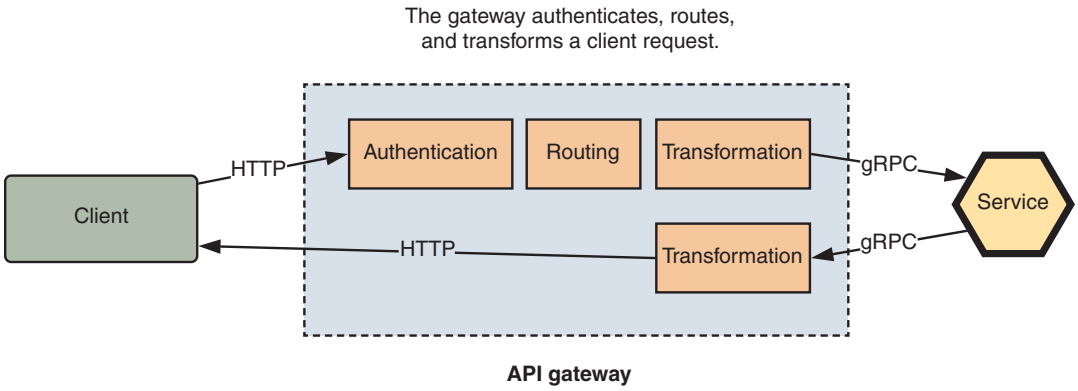


Figure 3.19 An API gateway serving a client request

A gateway also allows you to minimize the exposed area of your system from a security perspective by deploying internal services in a private network and restricting ingress to all but the gateway.

> **WARNING** Sometimes an API gateway might perform API composition: composing responses from multiple services into a single response. The line between this and service layer aggregation is fuzzy. It's best to be cautious and try to avoid business logic bleeding into the gateway itself, which can overly increase coupling between the gateway and underlying services.

3.5.2 Backends for frontends

The backends for frontends (BFF) pattern is a variation on the API gateway approach. Although the API gateway approach is elegant, it has a few downsides. If the API gateway acts as a composition point for multiple applications, it'll begin to take on more responsibility.

For example, imagine you serve both desktop and mobile applications. Mobile devices have different needs, displaying less data with less available bandwidth, and different user features, such as location and context awareness. In practice, this means desktop and mobile API needs diverge, which increases the breadth of functionality you need to integrate into a gateway. Different needs, such as the amount of data (and therefore payload size) returned for a given resource, may also conflict. It can be hard to balance these competing forces while building a cohesive and optimized API.

In a BFF approach, you use an API gateway for each consuming client type. To take the earlier example from SimpleBank, each user service they offered would have a unique gateway (figure 3.20).

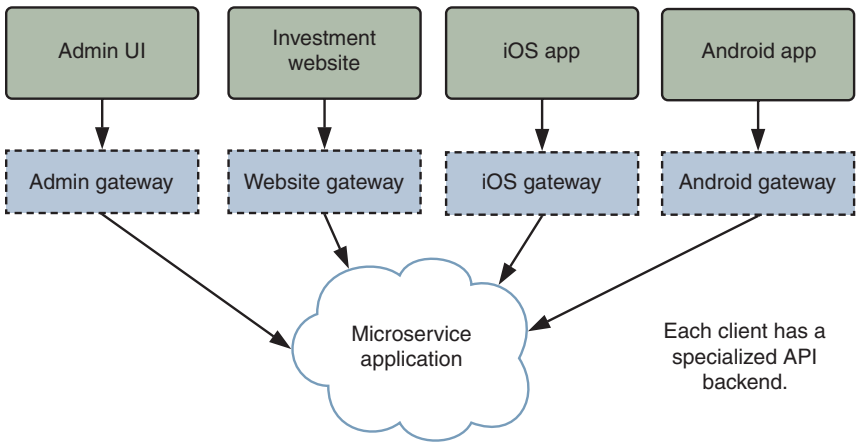


Figure 3.20 The backends for frontends pattern for SimpleBank's client applications

Doing so allows the gateway to be highly specific and responsive to the needs of its consumer without bloat or conflict. This results in smaller, simpler gateways and more focused development.

3.5.3 Consumer-driven gateways

In both previous patterns, the API gateway determines the structure of the data it returns to your consumer. To serve different clients, you might build unique backends. Let's flip this around. What if you could build a gateway that allowed consumers to express exactly what data they needed from your service? Think of this like an evolution of the BFF approach: rather than building multiple APIs, you can build a single "super-set" API that allows consumers to define the shape of response they require.

You can achieve this using GraphQL. GraphQL is a query language for APIs that allows consumers to specify which data fields they want and to multiplex different resources into a single request. For example, you might expose the following schema for SimpleBank clients.

Listing 3.1 Basic GraphQL schema for SimpleBank

```
type Account {
  id: ID!
  name: String!
  currentHoldings: [Holding]!
  orders: [Order]!
}

type Order {
  id: ID!
  status: String!
  asset: Asset!
  quantity: Float!
}

type Holding {
  asset: Asset!
  quantity: Float!
}

type Asset {
  id: ID!
  name: String!
  type: String!
  price: Float!
}

type Root {
  accounts: [Account]!
  account(id: ID): Account
}

schema: {
  query: Root
}
```

The ! indicates the field is non-nullable.

An account contains lists of Holding and Order.

Returns all accounts or an account by its ID

The schema has a single entry point for queries.

This schema exposes a customer's accounts, as well as orders and holdings against each of those accounts. Clients then execute queries against this schema. If a mobile app screen shows holdings and outstanding orders for an account, you could retrieve that data in a single request, as shown in the following listing.

Listing 3.2 Request body using GraphQL

```
{
  account(id: "101") {
    orders
    currentHoldings
  }
}
```

Filters by account ID

Requests specific member fields in response

In the backend, your GraphQL server would act like an API gateway, proxying and composing that data from multiple backend services (in this case, orders and holdings). We won't drill into GraphQL in further detail in this book, but if you're interested, the official documentation (http://graphql.org/) is a great place to start. We've also had some success using Apollo (https://www.apollographql.com/) to provide a GraphQL API façade over RESTful backend services.

3.6 Clients

The client tier, like the presentation layer in the three-tier architecture, presents to your users an interface to your application. Separating this layer from those below it allows you to develop user interfaces in a granular fashion and to serve the needs of different types of clients. This also means you can develop the frontend independently from backend features. As mentioned in the previous section, your application may need to serve many different clients—mobile devices, websites, both internal and external—each with different technology choices and constraints.

It's unusual for a single microservice to serve its own user interface. Typically, the functionality exposed to a given set of users is broader than the capabilities of a single service. For example, administrative staff at SimpleBank might deal with order management, account setup, reconciliation, tax, and so on. And this comes with cross-cutting concerns—authentication, audit logging, user management—that are clearly not the responsibility of an orders or account setup service.

3.6.1 Frontend monoliths

Your backend is straightforward to split into independently deployable and maintainable services—well, relatively, you still have another 10 chapters to go. But this can be challenging to achieve on the frontend. A typical frontend over a microservice application might still be a monolith that's deployed and changed as a single unit (figure 3.21). Specialist frontends, particularly mobile applications, often demand dedicated teams, making end-to-end feature ownership difficult to practically achieve.

> **NOTE** We'll talk more about end-to-end ownership (and why it's desirable and beneficial when developing microservice applications) in chapter 13.

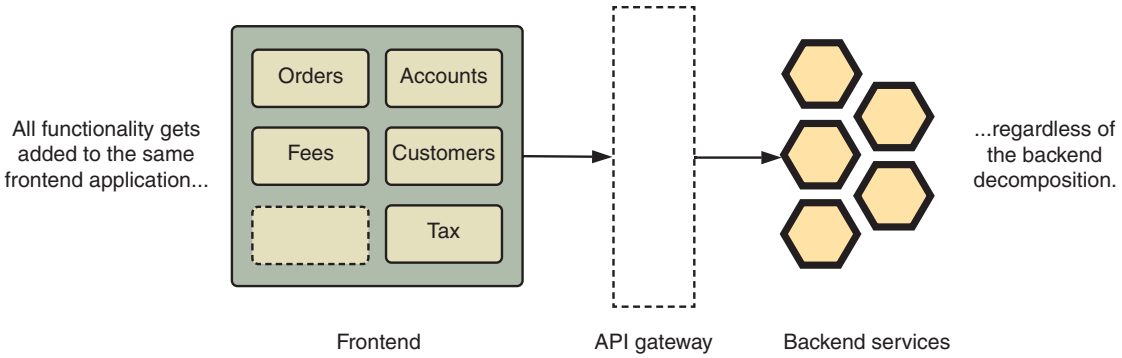


Figure 3.21 A typical frontend client in a microservice application can become monolithic.

3.6.2 Micro-frontends

As frontend applications grow larger, they begin to encounter the same coordination and friction issues that plague large-scale backend development. It'd be great if you could split frontend development in the same way you can split your backend services. An emerging trend in web applications is micro-frontends—serving fragments of a UI as independently packaged and deployable components that you can compose together. Figure 3.22 illustrates this approach.

This would allow each microservice team to deliver functionality end to end. For example, if you had an orders team, it could independently deliver both order management microservices and the web interface required to place and manage orders.

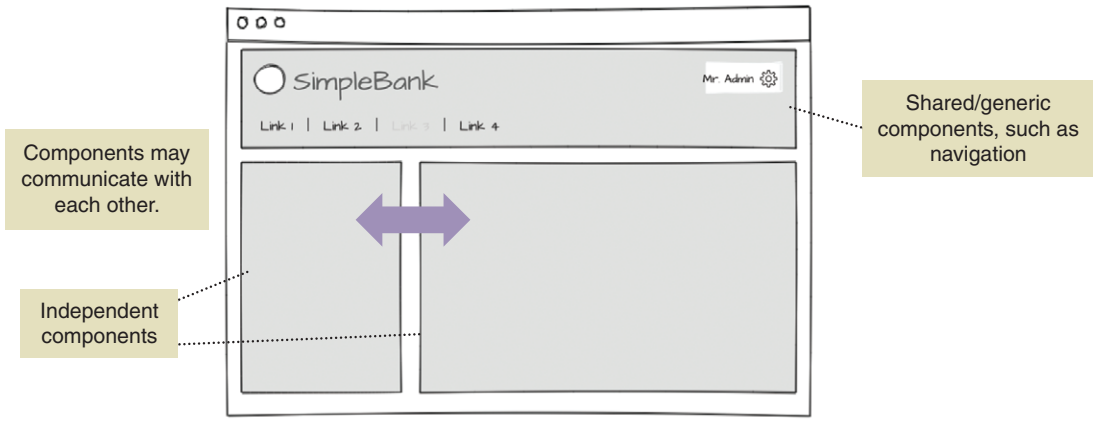


Figure 3.22 A user interface composed from independent fragments

Although promising, this approach has many challenges:

- Visual and interaction consistency across different components requires nontrivial effort to build and maintain common components and design principles.
- Bundle size (and therefore load time) can be difficult to manage when loading JavaScript code from multiple sources.
- Interface reloads and redraws can cause overall performance to suffer.

Micro-frontends aren't yet commonplace, but people are using several different technical approaches in the wild, including

- Serving UI fragments as web components with a clear, event-driven API
- Integrating fragments using client-side includes
- Using iframes to serve micro-apps into separate screen sections
- Integrating components at the cache layer using edge side includes (ESI)

If you're interested in learning more, Micro Frontends (https://micro-frontends.org/) and Zalando's Project Mosaic (https://www.mosaic9.org/) are great starting points.

Summary

- Individually, microservices are similar internally to monolithic applications.
- A microservice application is like a neighborhood: its final shape isn't prescribed but instead guided by principles and a high-level conceptual model.
- The principles that guide microservice architecture reflect organizational goals and inform team practices.
- Your architectural plan should encourage growth along good lines, rather than dictate approaches for your overall application.
- A microservice application consists of four layers: platform, service, boundary, and client.
- The platform layer provides tooling, plumbing, and infrastructure to support the development of product-oriented microservices.
- Synchronous communication is often the first choice in a microservice application and is best suited to command-type interactions, but it has drawbacks and can increase coupling and fragility.
- Asynchronous communication is more flexible and amenable to rapid system evolution, at the cost of added complexity.
- Common asynchronous communication patterns include queues and publish-subscribe.
- The boundary layer provides a façade over your microservice application that's appropriate for external consumers.

- Common types of boundaries include API gateways and consumer-driven gateways, such as GraphQL.
- Client applications, such as websites and mobile applications, interact with your mobile backend through the boundary layer.
- Clients risk becoming monolithic, but techniques are beginning to emerge for applying microservice principles to frontend applications.

4 Designing new features

This chapter covers

- Scoping microservices based on business capabilities and use cases
- When to scope microservices to reflect technical capabilities
- Making design choices when service boundaries are unclear
- Scoping effectively when multiple teams own microservices

Designing a new feature in a microservice application requires careful and well-reasoned scoping of microservices. You need to decide when to build new services or extend existing services, where boundaries lie between those services, and how those services should collaborate.

Well-designed services have three key characteristics: they're responsible for a single capability, independently deployable, and replaceable. If your microservices have the wrong boundaries, or are too small, they can become tightly coupled, making them challenging to deploy independently or replace. Tight coupling increases the impact, and therefore the risk, of change. If your services are too large—taking

on too much responsibility—they become less cohesive, increasing friction in ongoing development.

Even if you get it right the first time, you need to keep in mind that the requirements and needs of most complex software applications will evolve over time, and approaches that worked early in that application's lifetime may not always remain suitable. No design is perfect forever.

You'll face additional challenges in longer running applications (and larger engineering organizations). Your services may rely on a web of dependencies managed by multiple teams—as an engineer in one team, you'll need to design cohesive functionality while relying on services that won't necessarily be under your control. And you'll need to know when to retire and migrate away from services that no longer meet the needs of the wider system.

In this chapter, we'll walk you through designing a new feature using microservices. We'll use that example to explore techniques and practices that you can use to guide the design of maintainable microservices in both new and longer running microservice applications.

4.1 A new feature for SimpleBank

Remember SimpleBank? The team is doing well—customers love their product! But SimpleBank has discovered that most of those customers don't want to pick their own investments—they'd much rather have SimpleBank do the hard work for them. Let's take this problem and work out how to solve it with a microservice application. In the next few sections, we'll develop the design in four stages:

1. *Understanding* the business problem, use cases, and potential solution
2. *Identifying* the different entities and business capabilities your services should support
3. *Scoping* services that are responsible for those capabilities
4. *Validating* your design against current and potential future requirements

This will build on the small collection of services we explored in chapters 2 and 3: orders, market gateway, account transactions, fees, market data, and holdings.

First, let's understand the business problem you're trying to solve. In the real world, you could carry out the discovery and analysis of business problems using several techniques, such as market research, customer interviews, or impact mapping. As well as understanding the problem, you'd need to decide whether it was one your company should solve. Luckily, this isn't a book about product management—you can skip that part.

> **NOTE** We haven't tried to extrapolate a general approach to understanding business problems—that's another book altogether.

Ultimately, SimpleBank's customers want to invest money, either up front or on a regular basis, and see their wealth increase, either over a defined period or to meet a

specific goal, such as a deposit on a house. Currently, SimpleBank's customers need to choose how their money is invested—even if they don't have a clue about investing. An uninformed investor might choose an asset based on high predicted returns, without realizing that higher returns typically mean significantly higher risk.

To solve this problem, SimpleBank could make investment decisions on the customer's behalf by allowing the customer to choose a premade investment strategy. An investment strategy consists of proportions of different asset types—bonds, shares, funds, and so on—designed for a certain level of risk and investment timeline. When a customer adds money to their account, SimpleBank will automatically invest that money in line with this strategy. This setup is summarized in figure 4.1.

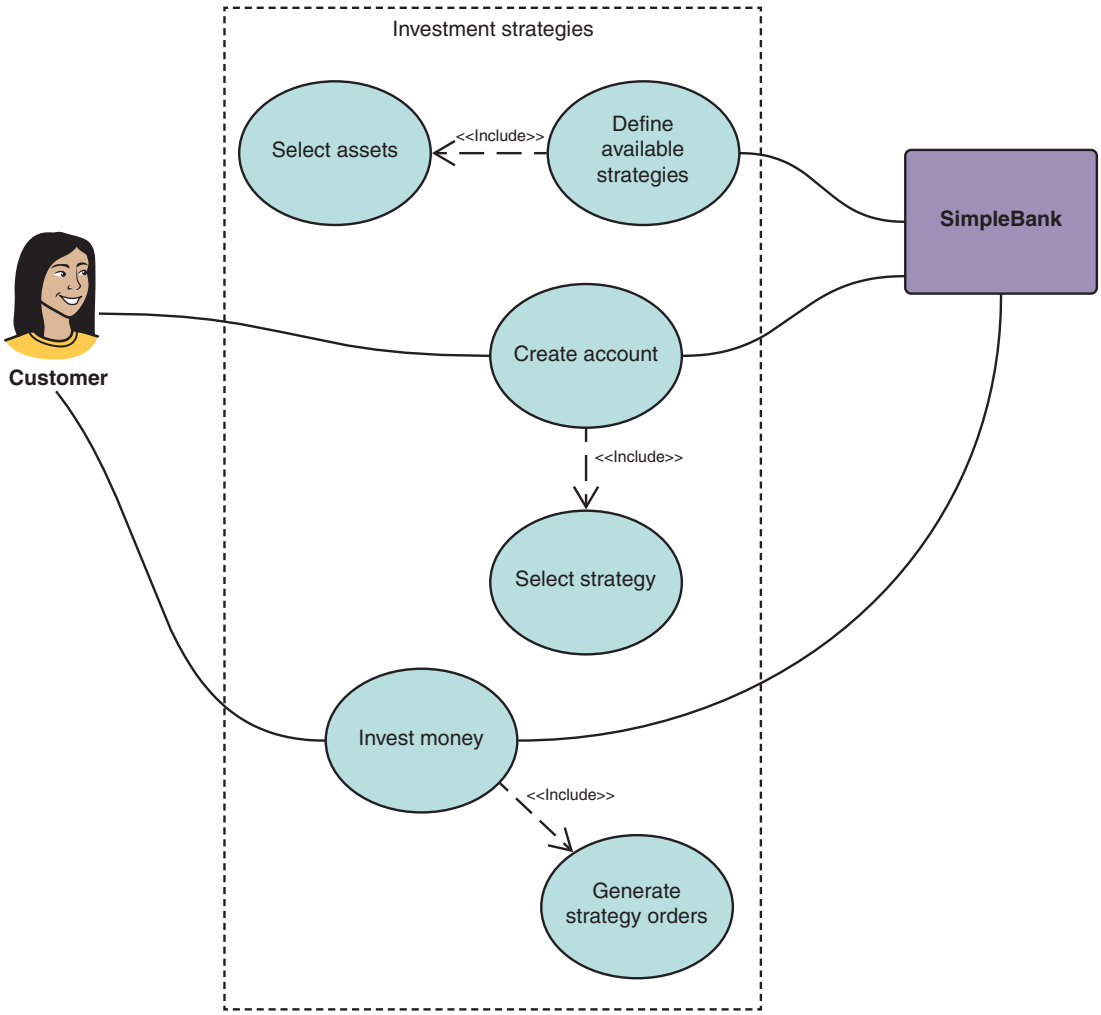


Figure 4.1 Potential use cases to support defining and selecting investment strategies

Based on figure 4.1, you can start to identify the use cases you need to satisfy to solve this problem:

- SimpleBank must be able to create and update available strategies.
- A customer must be able to create an account and elect an appropriate investment strategy.
- A customer must be able to invest money using a strategy, and investing in a strategy generates appropriate orders.

Over the next few sections, we'll explore these use cases. When identifying use cases in your own domain, you may prefer to use a more structured and exhaustive approach, such as behavior-driven development (BDD) scenarios. What's important is that you start to establish a concrete understanding of the problem, which you then can use to validate an acceptable solution.

4.2 Scoping by business capabilities

After you've identified your business requirements, your next step is to identify the technical solution: which features you need to build and how you'll support them with existing and new microservices. Choosing the right scope and purpose for each microservice is essential to building a successful and maintainable microservice application.

This process is called service scoping. It's also known as decomposition or partitioning. Breaking apart an application into services is challenging—as much art as science. In the following sections, we'll explore three strategies for scoping services:

- *By business capability or bounded context*—Services should correspond to relatively coarse-grained, but cohesive, areas of business functionality.
- *By use case*—Services should be verbs that reflect actions that will occur in a system.
- *By volatility*—Services should encapsulate areas where change is likely to occur in the future.

You don't necessarily use these approaches in isolation; in many microservice applications, you'll combine scoping strategies to design services appropriate to different scenarios and requirements.

4.2.1 Capabilities and domain modeling

A business capability is something that an organization does to generate value and meet business goals. Microservices that are scoped to business capabilities directly reflect business goals. In commercial software development, these goals are usually the primary drivers of change within a system; therefore, it's natural to structure the system to encapsulate those areas of change. You've seen several business capabilities implemented in services so far: order management, transaction ledgers, charging fees, and placing orders to market (figure 4.2).

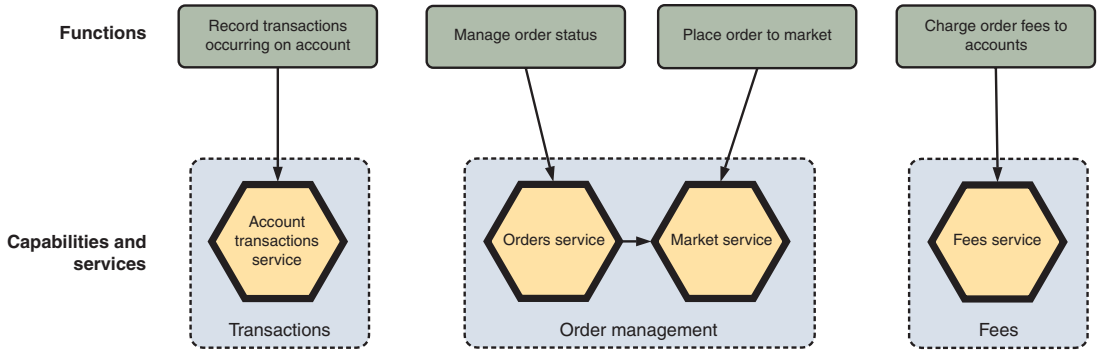


Figure 4.2 Functions that existing microservices provide and their relationship to business capabilities performed by SimpleBank

Business capabilities are closely related to a domain-driven design approach. Domain-driven design (DDD) was popularized by Eric Evans' book of the same name and focuses on building systems that reflect a shared, evolving view, or model, of a real-world domain.[1] One of the most useful concepts that Evans introduced was the notion of a *bounded context*. Any given solution within a domain might consist of multiple bounded contexts; the models inside each context are highly cohesive and have the same view of the real world. Each context has a strong and explicit boundary between it and other contexts.

Bounded contexts are cohesive units with a clear scope and an explicit external boundary. This makes them a natural starting point for scoping services. Each context demarcates the boundaries between different areas of your solution. This often has a close correspondence with organizational boundaries; for example, an e-commerce company will have different needs—and different teams—for shipping versus customer payments.

To begin with, a context typically maps directly to a service and an area of business capability. As the business grows and becomes more complex, you may end up breaking a context down into multiple subcapabilities, many of which you'll implement as independent, collaborating services. From the perspective of a client, though, the context may still appear as a single logical service.

> **TIP** The API gateway pattern we discussed in chapter 3 can be useful for establishing boundaries between different contexts (and underlying groups of services) within your application.

4.2.2 Creating investment strategies

You can design services to support creating investment strategies using a business capability approach. You might want to get a sketch pad to work through this one. To help you work through this example and give the use case more shape, we've wireframed what the UI for this feature might look like in figure 4.3.

---

[1] Although many of the implementation patterns—repositories, aggregates, and factories—are quite specific to object-oriented programming, many of Evans' analysis techniques—such as ubiquitous language—are useful in any programming paradigm.

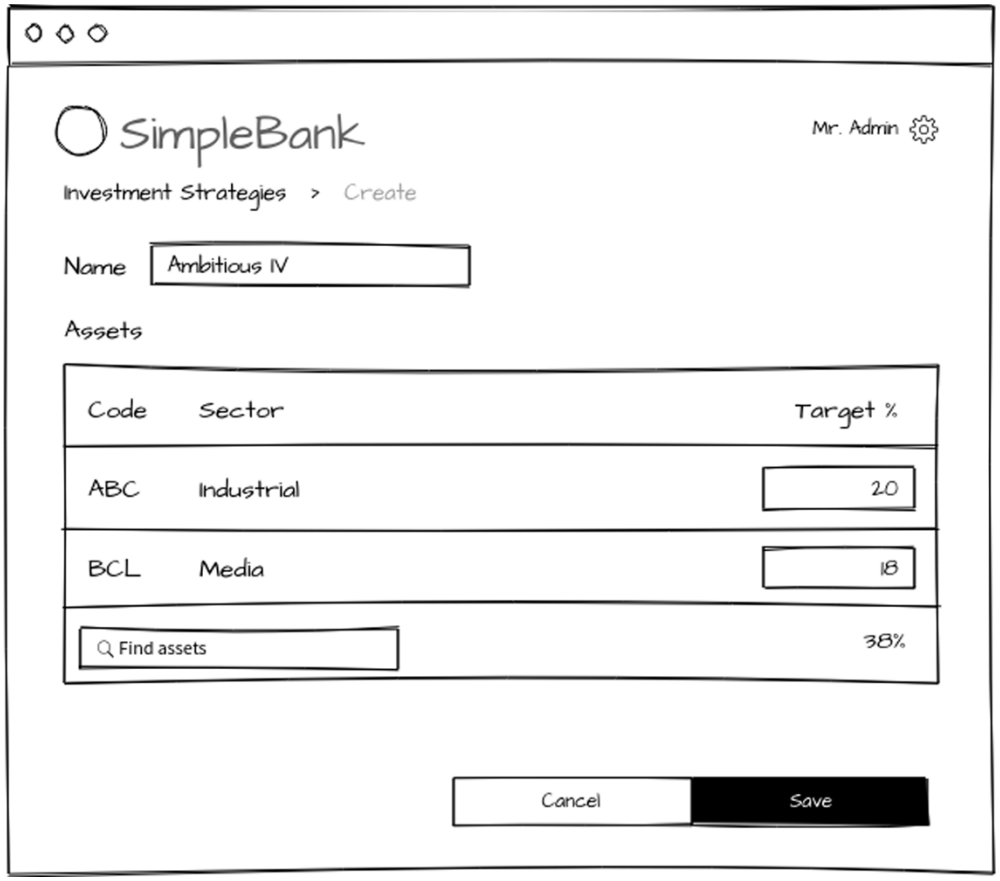


Figure 4.3 A user interface for an admin user to create new investment strategies

To design services by business capability, it's best to start with a domain model: some description of the functions your business performs in your bounded context(s) and the entities that are involved. From figure 4.3, you've probably identified these already. A simple investment strategy has two components: a name and a set of assets, each with a percentage allocation. An administrative staff member at SimpleBank will create a strategy. We've drafted those entities in figure 4.4.

The design of these entities helps you understand the data your services own and persist. Only three entities, and it already looks like you've identified (at least) two new services: user management and asset information. The user and asset entities are both part of distinct bounded contexts:

- *User management* — This covers features like sign-up, authentication, and authorization. In a banking environment, authorization for different resources and functionality is subject to strict controls for security, regulatory, and privacy reasons.

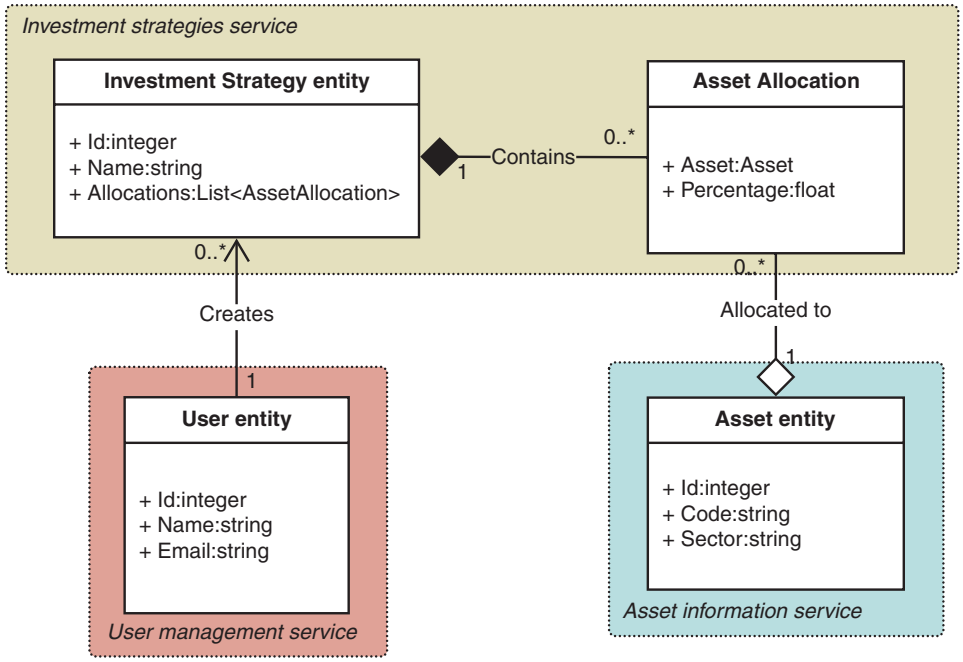


Figure 4.4 The first draft of a domain model made up of entities to support the creation of investment strategies

- *Asset information* — This covers integration with third-party providers of market data, such as asset prices, categories, classification, and financial performance. This capability would include asset search, as required by your user interface (figure 4.3).

Interestingly, these different domains reflect the organization of SimpleBank itself. A dedicated operational team manages asset data; likewise, user management. This comparability is desirable, as it means your services will reflect real-world lines of cross-team communication.

More on that later — let's get back to investment strategies. You know that you can associate them with customer accounts and use them to generate orders. Accounts and orders are both distinct bounded contexts, but investment strategies don't belong in either one. When strategies change, the change is unlikely to affect accounts or orders themselves. Conversely, adding investment strategies to either of those existing services will hamper their replaceability, making them less cohesive and less amenable to change.

These factors indicate that investment strategies are a distinct business capability, requiring a new service. Figure 4.5 illustrates the relationships between this context and your existing capabilities.

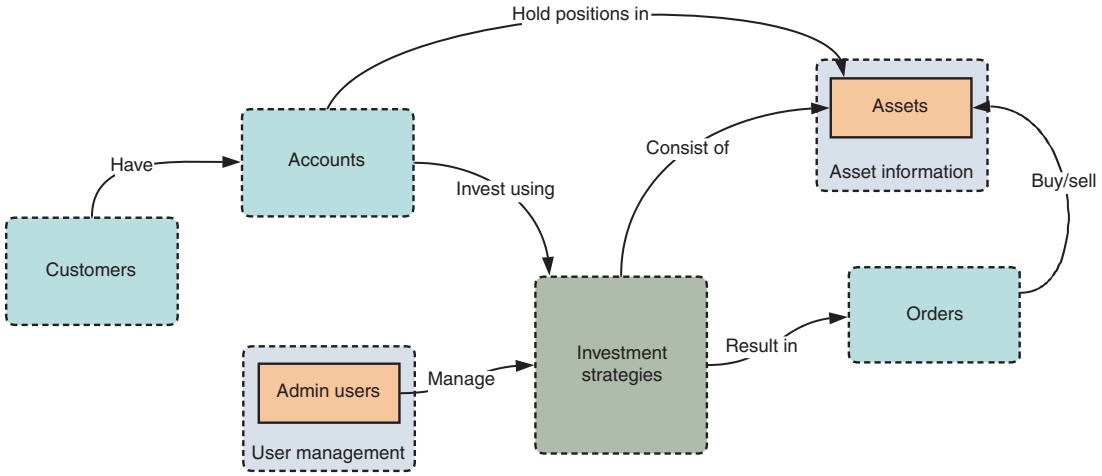


Figure 4.5 Relationships between your new business capability and other bounded contexts within the SimpleBank application

You can see that some contexts are aware of information that belongs to other contexts. Some entities within your context are shared: they're conceptually the same but carry unique associations or behavior within different contexts. For example, you use assets in multiple ways:

- The strategy context records the allocation of assets to different strategies.
- The orders context manages the purchase and sale of assets.
- The asset context stores fundamental asset information for use by multiple contexts, such as pricing and categorization.

The model we've drawn out in figure 4.5 doesn't tell you much about the behavior of a service; it only tells you the business scope your services cover. Now that you have a firmer idea of where your service boundaries lie, you can draft out the contract that your service offers to other services or end users.

> **NOTE** Don't worry too much about what technology you'll use for communication at this stage. The examples in this chapter could easily apply to any point-to-point messaging approach.

First, your investment strategies service needs to expose methods for creating and retrieving investment strategies. Other services or your UI can then access this data. Let's draft out an endpoint that allows creating an investment strategy. The example shown in listing 4.1 uses the OpenAPI specification (formerly known as Swagger), which is a popular technique for designing and documenting REST API interfaces. If you're interested in learning more, the Github page for the OpenAPI specification[2] is a good place to start.

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

Listing 4.1 API for the investment strategies service

```
openapi: "3.0.0"
info:
  title: Investment Strategies
servers:
  - url: https://investment-strategies.simplebank.internal
paths:
  /strategies:
    post:
      summary: Create an investment strategy
      operationId: createInvestmentStrategy
      requestBody:
        description: New strategy to create
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/NewInvestmentStrategy'
      responses:
        '201':
          description: Created strategy
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/InvestmentStrategy'
components:
  schemas:
    NewInvestmentStrategy:
      required:
        - name
        - assets
      properties:
        name:
          type: string
        assets:
          type: array
          items:
            $ref: '#/components/schemas/AssetAllocation'
    AssetAllocation:
      required:
        - assetId
        - percentage
      properties:
        assetId:
          type: string
        percentage:
          type: number
          format: float
    InvestmentStrategy:
      allOf:
```

Starts with some metadata about your API

Defines a "POST /strategies" path

The body of this request should be the new investment strategy.

Refers to a location elsewhere in the document: the components key

Defines the response type in the components section

Defines reusable data types

A new investment strategy type

Contains a list of assets of type AssetAllocation

```
    - $ref: '#/components/schemas/NewInvestmentStrategy'
    - required:
      - id
      - createdByUserId
      - createdAt
      properties:
        id:
          type: integer
          format: int64
        createdByUserId:
          type: integer
          format: int64
        createdAt:
          type: string
          format: date-time
```

The InvestmentStrategy type extends the NewInvestmentStrategy and adds fields, based on your entity model.

If you're going to use strategies again later—and you are—you'll need to retrieve them. Immediately under your `paths:` element in listing 4.1, add the code in the following listing.

Listing 4.2 API for retrieving strategies from the investment strategies service

The path for retrieving an investment strategy

```
/strategies/{id}:
  get:
    description: Returns an investment strategy by ID
    operationId: findInvestmentStrategy
    parameters:
      - name: id
        in: path
        description: ID of strategy to fetch
        required: true
        schema:
          type: integer
          format: int64
    responses:
      '200':
        description: investment strategy
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/InvestmentStrategy'
```

Defines the format of the ID

Returns an investment strategy

You also should consider what events this service should emit. An event-based model aids in decoupling services from each other, ensuring that you can choreograph long-term interactions, rather than explicitly orchestrate them.

WARNING Anticipating future use cases of a service is one of the most difficult elements of service design. But building flexible APIs and integration points between services reduces the need for future rework and coordination between teams.

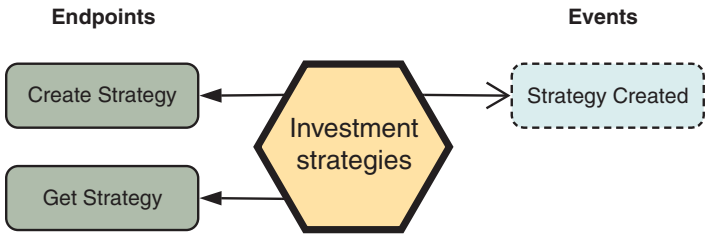


Figure 4.6 The inbound and outbound contract of your investment strategies microservice

For example, imagine that creating a strategy will trigger email notifications to potentially interested customers. This is separate from the scope of the investment strategies service itself; it has no knowledge of customers (or their preferences). This is an ideal use case for events. If a `POST` to `/strategies` post-hoc triggers an event—let's call it `StrategyCreated`—then arbitrary microservices can listen for that event and act appropriately. Figure 4.6 illustrates the full scope of your service's API.

Great work—you've identified all the capabilities that you require to support this use case. To see how this fits together, you can map the investment strategies service and the other capabilities you've identified to the wireframe (figure 4.7).

Let's summarize what you've done so far:

1. For a sample problem, you've identified functions the business performs to generate value and the natural seams between different areas of SimpleBank's business domain.
2. You've used that knowledge to identify boundaries within your microservice application, identifying entities and responsibility for different capabilities.
3. You've scoped your system into services that reflect those domain boundaries.

Figure 4.7 Identified capabilities and services mapped to how they'd support functionality in the create investment strategy user interface

This approach results in services that are relatively stable, cohesive, oriented to business value, and loosely coupled.

4.2.3 Nested contexts and services

Each bounded context provides an API to other contexts, while encapsulating internal operation. Let's take asset information as an example (figure 4.8):

- It exposes methods that other contexts can use, such as searching for and retrieving assets.
- Third-party integrations or specialist teams within SimpleBank populate asset data.

The private/public divide provides a useful mechanism for service evolution. Early in a system's lifecycle, you might choose to build coarser services, representing a high-level boundary. Over time, you might decompose services further, exposing behavior from nested contexts. Doing this maintains replaceability and high cohesion, even as business logic increases in complexity.

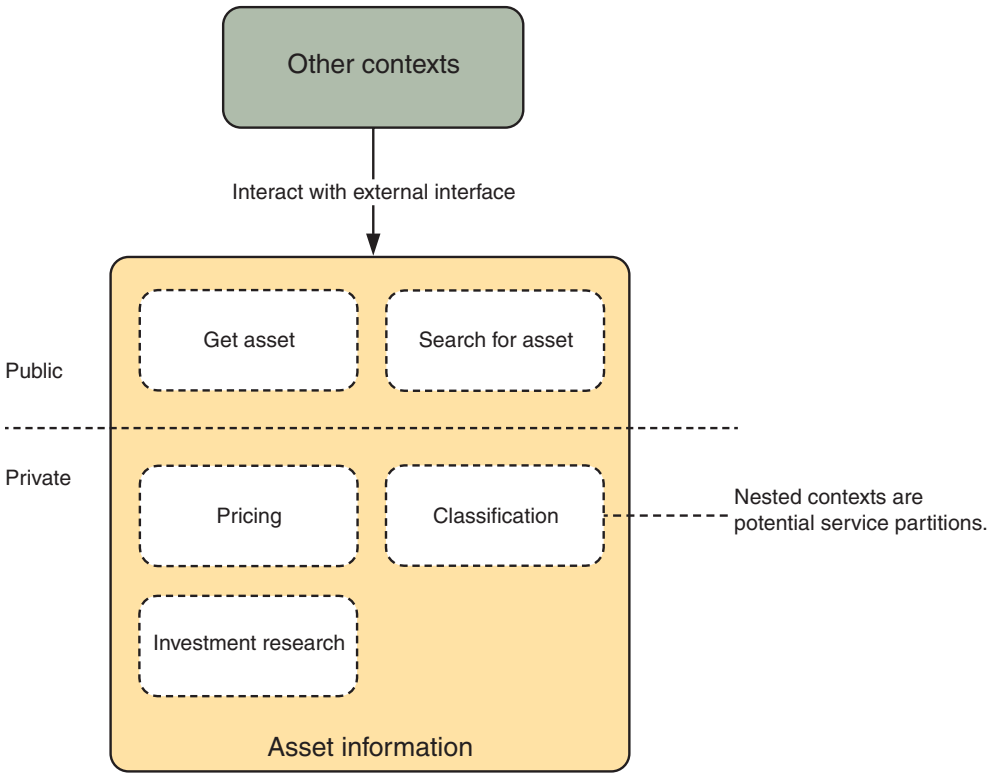


Figure 4.8 A context exposes an external interface and may itself contain nested contexts

4.2.4 Challenges and limitations

In the previous sections, you identified the natural seams within the organization's business domain and applied them to partition your services. This approach is effective because it maps services to the functional structure of a business—directly reflecting the domain in which an organization operates. But it's not perfect.

Requires substantial business knowledge

Partitioning by business capabilities requires having significant understanding of the business or problem domain. This can be difficult. If you don't have enough information—or you've made the wrong assumptions—you can't be completely certain you're making the right design decisions. Understanding the needs of any business problem is a complex, time-consuming and iterative process.

This problem isn't unique to microservices, but misunderstanding the business scope—and reflecting it incorrectly in your services—can incur higher refactoring costs in this architecture, because both data and behavior can require time-consuming migration between services.

Coarse-grained services keep growing

Similarly, a business capability approach is biased toward the initial development of coarse-grained services that cover a large business boundary—for example, orders, accounts, or assets. New requirements increase the breadth and depth of that area, increasing the scope of the service's responsibility. These new reasons to change can violate the single responsibility principle. It'll be necessary to partition that service further to maintain an acceptable level of cohesion and replaceability.

> **WARNING** Service teams sometimes add functionality to existing microservices because it's easy—a deployable unit already exists—rather than investing more time to create a new service or repartition the existing service appropriately. Although teams sometimes need to make pragmatic decisions, they need to exercise discipline to minimize this source of technical debt.

4.3 Scoping by use case

So far, your services have been nouns, oriented around objects and things that exist within the business domain. An alternative approach to scoping is to identify verbs, or use cases within your application, and build services to match those responsibilities. For example, an e-commerce site might implement a complex sign-up flow as a microservice that interacts with other services, such as user profile, welcome notifications, and special offers.

This approach can be useful when

- A capability doesn't clearly belong in one domain or interacts with multiple domains.
- The use case being implemented is complex, and placing it in another service would violate single responsibility.

Let's apply this approach to SimpleBank to understand how it differs from noun-oriented decomposition. Get your pencil and paper ready!

4.3.1 Placing investment strategy orders

A customer can invest money into an investment strategy. This will generate appropriate orders; for example, if the customer invests $1,000, and the strategy specifies 20% should be invested in Stock ABC, an order will be generated to purchase $200 of ABC.

This raises several questions:

1. How does SimpleBank accept money for investment? Let's assume a customer can make an investment by external payment (for example, a credit card or bank transfer).
2. Which service is responsible for generating orders against a strategy? How does this relate to your existing orders and investment strategies services?
3. How do you keep track of orders made against strategies?

You could build this capability into your existing investment strategies service.

But placing orders might unnecessarily widen the scope of responsibility that the service encapsulates. Likewise, the capability doesn't make sense to add to the orders service. Coupling all possible sources of orders to that service would give it too many reasons to change.

You can draft out an independent service for this use case as a starting point—call it PlaceStrategyOrders. Figure 4.9 sketches out how you'd expect this service to behave.

Consider the input to this service. For orders to be placed, this service needs three things: the account placing them, the strategy to use, and the amount to invest. You can formalize that input, as shown in the following listing.

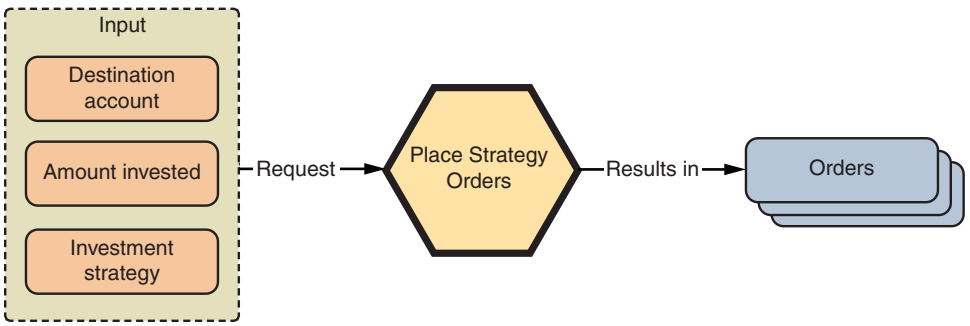


Figure 4.9 Expected behavior of a proposed PlaceStrategyOrders service

Listing 4.3 Draft input for PlaceStrategyOrders

```
paths:
  /strategies/{id}/orders:                    <-- Executing an order is a subresource
    post:                                         of an investment strategy.
      summary: Place strategy orders
      operationId: PlaceStrategyOrders
      parameters:
        - name: id
          in: path
          description: ID of strategy to order against
          required: true
          schema:
            type: integer
            format: int64
      requestBody:
        description: Details of order
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/StrategyOrder'
components:
  schemas:
    StrategyOrder:
      required:                               <-- An order requires a destination account
        - destinationAccountId                    and an investment amount.
        - amount
      properties:
        destinationAccountId:
          type: integer
          format: int64
        amount:
          type: number
          format: decimal
```

This is elegant but a little too simple. If you assume your payment is coming from an external source, you can't execute orders until those funds are available. It doesn't make sense for PlaceStrategyOrders to handle receipt of funds—this is clearly a distinct business capability. Instead, you can link placing strategy orders to a payment, as follows.

Listing 4.4 Using payment ID for PlaceStrategyOrders

```
components:
  schemas:
    StrategyOrder:
      required:
        - destinationAccountId
        - amount
        - paymentId                           <-- Your new required field: paymentId
      properties:
        destinationAccountId:
```

```
      type: integer
      format: int64
    amount:
      type: number
      format: decimal
  paymentId:
      type: integer
      format: int64
```

This anticipates the existence of a new service capability: payments. This capability should support

- Initiating payments by users
- Processing those payments by interacting with third-party payment systems
- Updating account positions at SimpleBank

Because you know that payments aren't instantaneous, you'd expect this service to trigger asynchronous events that other services can listen for, such as PaymentCompleted. Figure 4.10 illustrates this payments capability.

From the perspective of PlaceStrategyOrders, it doesn't matter how you implement the payments capability, as long as something implements the interface the consumer expects. It might be a single service—Payments—or a collection of action-oriented services, for example, CompleteBankTransfer.

You can summarize what you've designed so far in a sequence diagram (figure 4.11).

There's one missing element in this diagram: getting these orders to market. As mentioned, although this service generates orders, this capability clearly doesn't belong within your existing orders service. The orders service exposes behavior that multiple consumers can use, including this new service (figure 4.12); although the source of orders differs, the process of placing them remains the same.

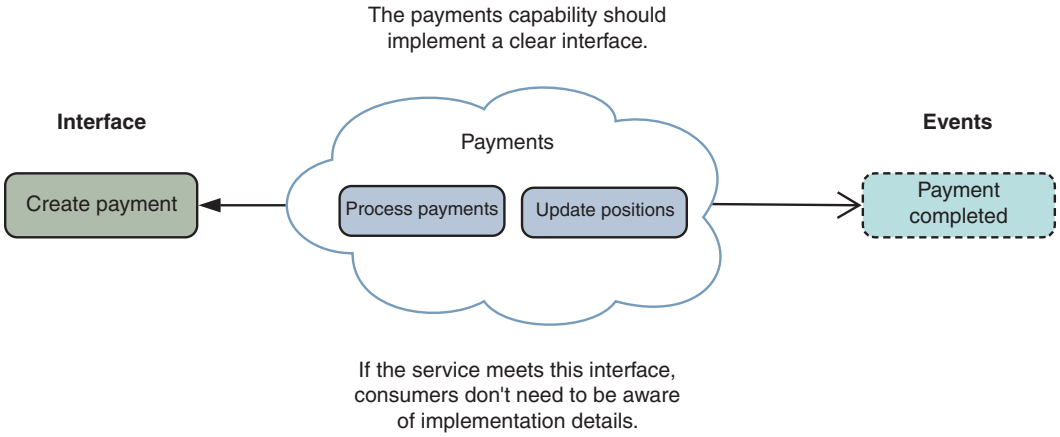


Figure 4.10 The interface that your proposed payments capability expects

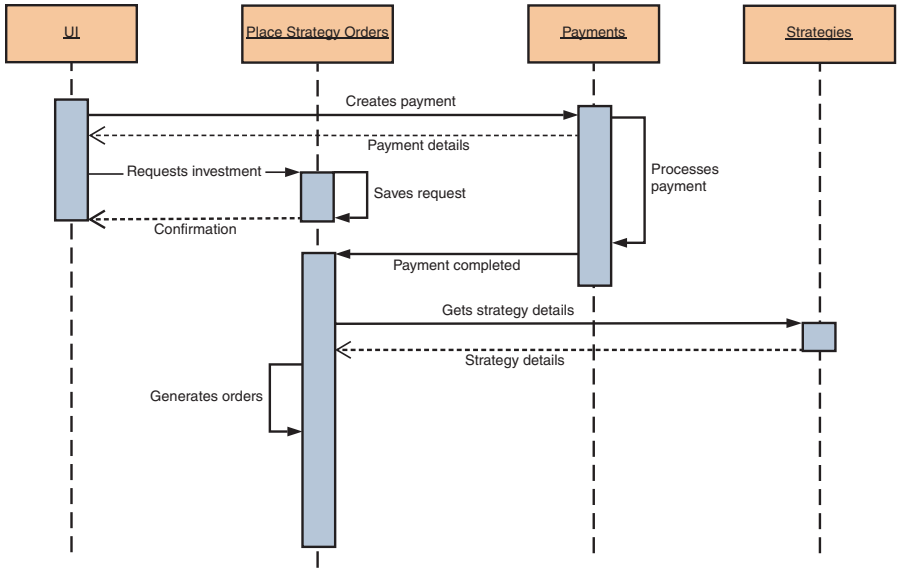


Figure 4.11 The process of creating a payment and making an investment using the proposed PlaceStrategyOrders service

Lastly, you need to persist the link between these orders and the strategy and investment that created them. PlaceStrategyOrders should be responsible for storing any request it receives—it clearly owns this data. Therefore, you should record any order IDs within the strategy order service to preserve that foreign key relationship. You could also record the order source ID—the ID of this investment strategy investment request—within the orders service itself, although it seems less likely you'd query data in that direction.

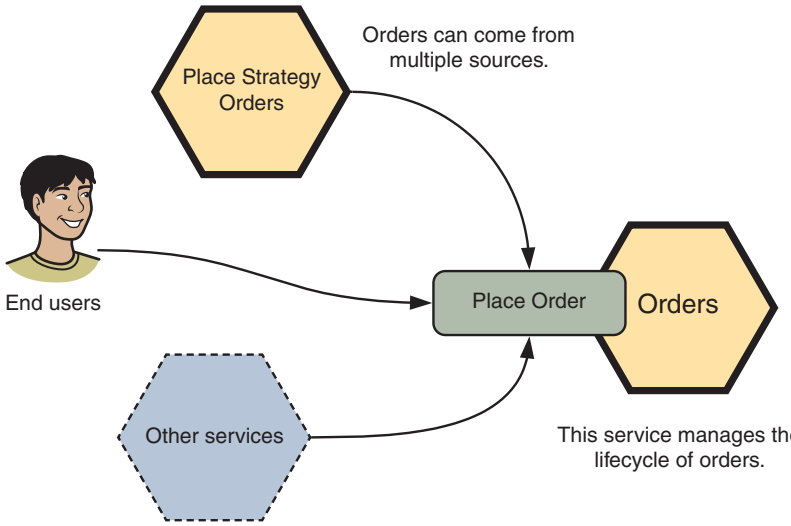


Figure 4.12 Your orders service provides an API that multiple other services within your system can consume.

The orders service emits `OrderCompleted` events when an order has been completed. Your strategy orders service can listen for these events to reflect that status against the overall investment request.

You can add the orders service and tie this all together as shown in figure 4.13.

Great! You've designed another new service. Unlike the previous section, you designed a service that closely represented a specific complex use case, rather than a broad capability.

This resulted in a service that was responsible for a single capability, replaceable, and independently deployable, meeting your desired characteristics for well-scoped microservices. In contrast, unlike if you'd focused on business capabilities, the tight focus of this service on a single use case limits potential for reuse in other use cases in the future. This inflexibility suggests that fine-grained use case services are best used in tandem with coarser grained services, rather than alone.

4.3.2 Actions and stores

We've identified an interesting pattern in the above examples: multiple higher level microservices access a coarse-grained underlying business capability. This is especially prevalent in a verb-oriented approach, as the data needs of different actions often overlap.

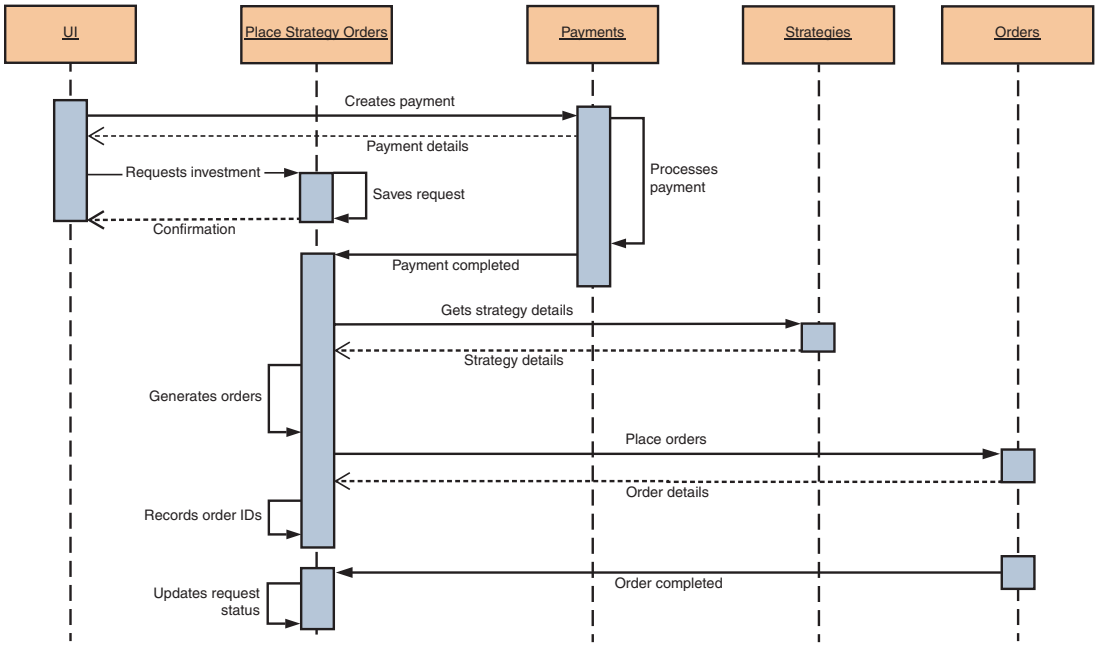


Figure 4.13 The full process of creating an investment strategy order using your new PlaceStrategyOrders service

For example, imagine you have two actions: update order and cancel order. Both operations operate against the same underlying order state, so neither can exclusively own that state itself, and you need to reconcile that conflict somewhere. In the previous examples, the orders service took care of the problem. This service is the ultimate owner of that subset of your application's persistent state.

This pattern is similar[3] to Bob Martin's clean architecture[4] or Alistair Cockburn's hexagonal architecture. In those models, the core of an application consists of two layers:

- *Entities*—Enterprisewide business objects and rules
- *Use cases*—Application-specific operations that direct entities to achieve the goals of the use case

Around those layers, you use interface adapters to connect these business-logic concerns to application-level implementation concerns, such as particular web frameworks or database libraries. Similarly, at an intraservice level, your use cases (or actions) interact with underlying entities (or stores) to generate some useful outcome. You then wrap them in a façade, such as an API gateway, to map from your underlying service-to-service representations to an output friendly to an external consumer (for example, a RESTful API). Figure 4.14 sketches out that arrangement.

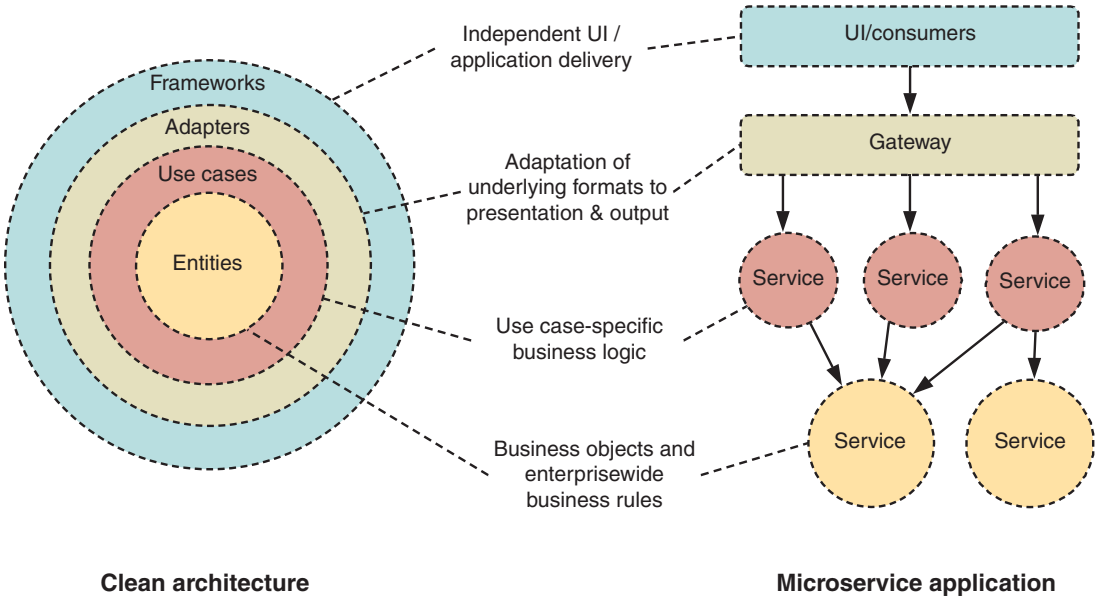


Figure 4.14 The architecture of a microservice application compared to Bob Martin's clean architecture

[3] But not quite the same: Martin's architecture is concerned with implementation detail independence within object-oriented applications (for example, keeping business logic independent of the data storage solution), which isn't particularly relevant at the intraservice level.

[4] For a more detailed explanation of Martin's clean architecture, see Uncle Bob, "The Clean Architecture," August 13, 2012, http://mng.bz/LJB4.

This architecture is conceptually elegant, but you need to apply it judiciously in a microservice system. Treating underlying capabilities as, first and foremost, stores of persistent state can lead to anemic, "dumb" services. These services fail to be truly autonomous because they can't take any action without being mediated by another, higher level service. This architecture also increases the number of remote calls and the length of the service chain you need to perform any useful action.

This approach also risks tight coupling between actions and underlying stores, hampering your ability to deploy services independently. To avoid these pitfalls, we recommend you design microservices from the inside out, building useful coarse-grained capabilities *before* building fine-grained action-oriented services.

4.3.3 Orchestration and choreography

In chapter 2, we discussed the difference between orchestration and choreography in service interaction. A bias toward choreography tends to result in more flexible, autonomous, and maintainable services. Figure 4.15 illustrates the difference between these approaches.

If you scope services by use case, you might find yourself writing services that explicitly orchestrate the behavior of several other services. This isn't always ideal:

- Orchestration can increase coupling between services and increase the risk of dependent deployments.
- Underlying services can become anemic and lack purpose, as the orchestrating service takes on more and more responsibility for useful business output.

When designing services that reflect use cases, it's important to consider their place within a broader chain of responsibilities. For example, the PlaceStrategyOrders service you designed earlier both orchestrates behavior (placing orders) and reacts to other events (payment processing). Taking a balanced approach to choosing orchestration or choreography reduces the risk of building services that lack autonomy.

4.4 Scoping by volatility

In an ideal world, you could build any feature by combining existing microservices. This might sound impractical, but it's interesting to consider how you maximize the reusability—and therefore long-term utility—of the services you build.

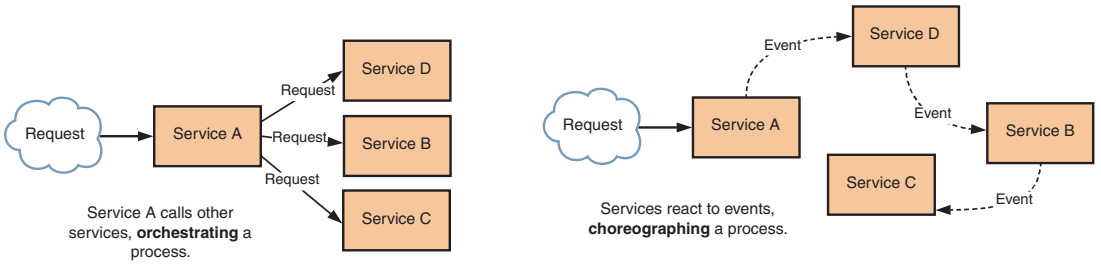


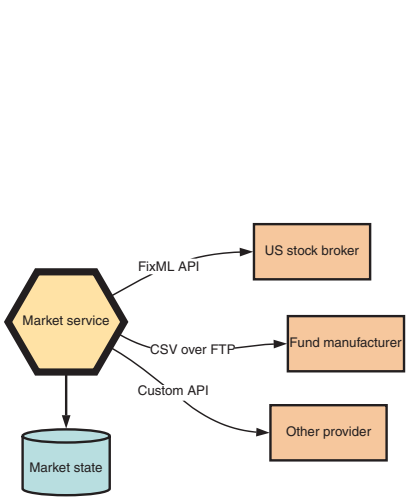
Figure 4.15 Orchestration versus choreography in service interaction

So far, we've taken a predominantly functional approach to decomposing services. This approach is effective but has limitations. Functional decomposition is biased toward the present needs of an application and doesn't explicitly consider how that application might evolve. A purely functional approach can constrain the future growth of a system by resulting in services that are inflexible in the face of new or evolving requirements, thereby increasing the risk of change.

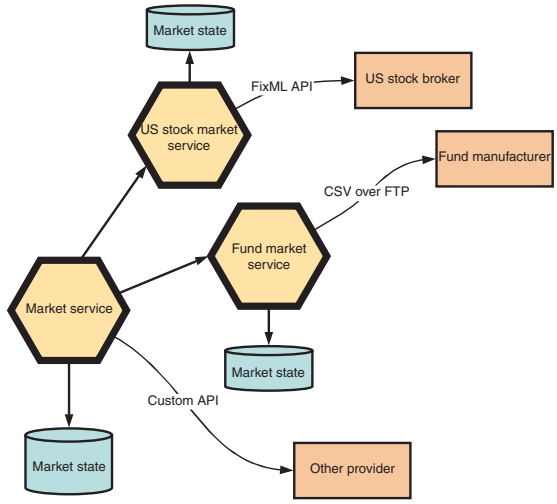
Therefore, as well as considering the functionality of your system, you should consider where that application is likely to change in the future. This is known as volatility. By encapsulating areas that are likely to change, you help to ensure that uncertainty in one area doesn't negatively impact other areas of the application. You can find an analogy to this in the stable dependencies principle in object-oriented programming: "a package should only depend on packages that are more stable than it is."

SimpleBank's business domain has multiple axes of volatility. For example, placing an order to market is volatile: different orders need to go to different markets; SimpleBank might have different APIs to each market (for example, through a broker, direct to an exchange); and those markets might change as SimpleBank broadens its offering of financial assets.

Tightly coupling market interaction as part of the orders service would lead to a high degree of instability. Instead, you'd split the market service and ultimately build multiple services to meet the needs of each market. Figure 4.16 illustrates this approach.



The market service encapsulates potential change in placing orders to market.



Over time, we might decompose further to encapsulate subareas of volatility.

Figure 4.16 The market service encapsulates change in how SimpleBank communicates with different financial market providers. Over time, this might evolve into multiple services.

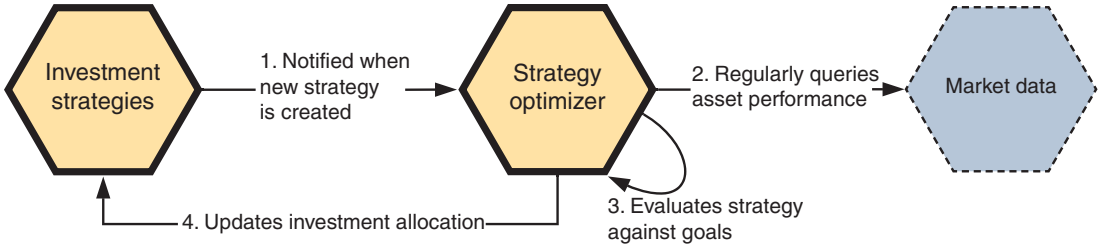


Figure 4.17 Partitioning a distinct area of system volatility—investment strategy optimization—as a separate service

Let's take one more example: imagine you have more than one type of investment strategy. Perhaps you have strategies that are optimized by deep learning: the performance of assets on the market should drive adjustments to future strategy allocations.

Adding this complex behavior to your InvestmentStrategies service would significantly broaden its reasons to change—reducing cohesiveness. Instead, you should add new services with responsibility for that behavior—as you can see in figure 4.17. By doing this, you can develop and release these services independently without unnecessary coupling between different features or rates of change.

Ultimately, good architecture strikes a balance between the current and future needs of an application. If microservices are too narrowly scoped, you might find the cost of change becomes higher in the future as you become increasingly constrained by earlier assumptions about the limits of your system. On the flipside, you should always be careful to keep YAGNI—"you aren't gonna need it"—in mind. You may not always have the luxury of time (or money) to anticipate and meet every possible future permutation of your application.

4.5 Technical capabilities

The services you've designed so far have reflected actions or entities that map closely to your business capabilities, such as placing orders. These business-oriented services are the primary type you'll build in any microservice application.

You can also design services that reflect technical capabilities. A technical capability indirectly contributes to a business outcome by supporting other microservices. Common examples of technical capabilities include integration with third-party systems and cross-cutting technical concerns, such as sending notifications.

4.5.1 Sending notifications

Let's work through an example. Imagine that SimpleBank would like to notify a customer—perhaps through email—whenever a payment has been completed. Your

first instinct might be to build that code within your payments service (or services). But that approach has three problems:

1. The payments service has no awareness of customer contact details or preferences. You'd need to extend its interface to include customer contact data (pushing that obligation on to service consumers) or query another service.
2. Other parts of your application might send notifications as well. You can easily picture other features—orders, account setup, marketing—that might trigger emails.
3. Customers might not even want to receive emails: they might prefer SMS or push notifications…or even physical mail.

The first and second points suggest that this should be a separate service; the third point suggests you might need multiple services—one to handle each type of notification. Figure 4.18 sketches that out. Your notification services can listen to the `PaymentCompleted` event that your payments service emits.

You can configure your group of notification services to listen to any events—from any service—that should result in a notification. Each service will need to be aware of a customer's contact preferences and details to send notifications. You could store that information in a separate service, such as a customers service, or have each service own it. This area has hidden dimensions of complexity; for example, many customers might own the payment's destination account, triggering multiple notifications.

You may have realized that the notification services are also responsible for generating appropriate message content based on each event, which suggests they could grow significantly in the future, in line with the potential number of notifications. It eventually might be necessary to split message content from message delivery to reduce this complexity.

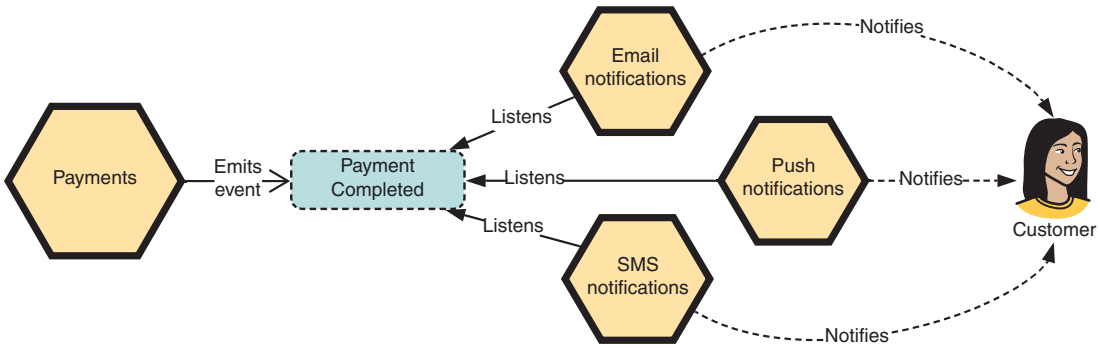


Figure 4.18 Supporting technical microservices for notifications

This example illustrates that implementing technical capabilities maximizes reusability while simplifying your business services, decoupling them from nontrivial technical concerns.

4.5.2 When to use technical capabilities

You should use a technical capability to support and simplify other microservices, limiting the size and complexity of your business capabilities. Partitioning these capabilities is desirable when

- Including the capability within a business-oriented service will make that service unreasonably complex, complicating any future replacement.
- A technical capability is required by multiple services—for example, sending email notifications.
- A technical capability changes independently of the business capability—for example, a nontrivial third-party integration.

Encapsulating these capabilities in separate services captures axes of volatility—areas that are likely to change independently—and maximizes service reusability.

In certain scenarios, it's unwise to partition a technical capability. In some situations, extracting a capability will reduce the cohesiveness of a service. For example, in classic SOA, systems were often decomposed horizontally, in the belief that splitting data storage from business functionality would maximize reusability. Figure 4.19 illustrates how requests would be serviced in this approach.

Unfortunately, the intended reusability came at a high cost. Splitting those layers of an application led to tight coupling between different deployable units, as delivering individual features required simultaneous change across multiple applications (figure 4.20). When you have to coordinate changes to distinct components, this leads to error-prone, lock-step deployments—a distributed monolith.

If you focus on business capabilities first, you'll avoid these pitfalls. But you should carefully scope any technical capability to ensure it's truly autonomous and independent from other services.

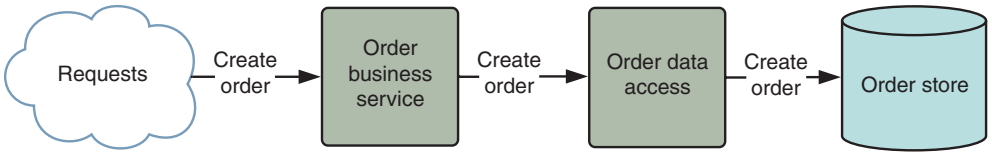


Figure 4.19 Lifecycle of a create order request in a horizontally partitioned service application

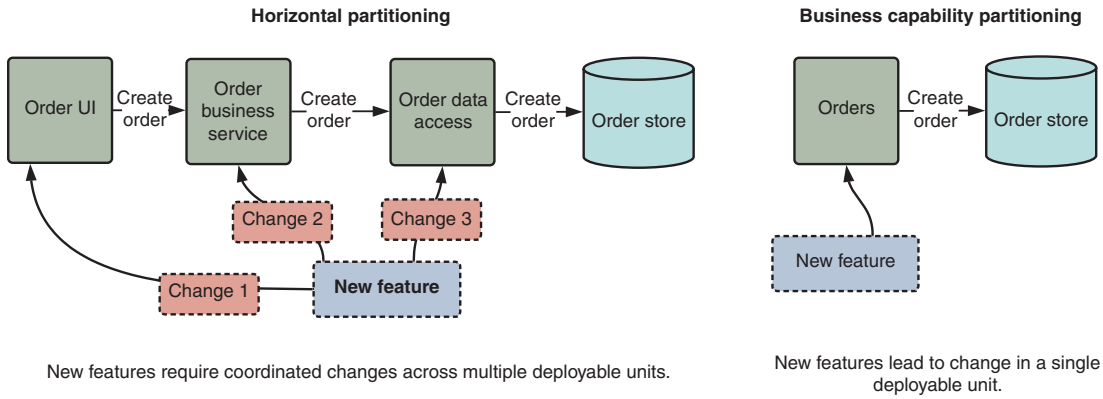


Figure 4.20 The impact of change in a horizontally partitioned service versus a service scoped to business capability

4.6 Dealing with ambiguity

Scoping microservices is as much art as science. A large part of software design is finding effective ways to achieve the best solution when faced with ambiguity:

- Your understanding of the problem domain might be incomplete or incorrect. Understanding the needs of any business problem is a complex, time-consuming, and iterative process.
- You need to anticipate how you might need to use a service in the future, rather than only right now. But you'll often run into tension between short-term feature needs and long-term service malleability.

Suboptimal service partitioning in microservices can be costly: it adds friction to development and extra effort to refactoring.

> **NOTE** Understanding a business domain is hardly unique to microservices—or even the engineering process itself. Most modern product engineering methodologies aim to maintain flexibility and agility when facing an evolving understanding of requirements. For that reason, we strongly recommend following an iterative and lean development process when building a microservice application.

4.6.1 Start with coarse-grained services

In this section, we'll explore a few approaches you can use to make practical service decisions when the right solution isn't obvious. To start, we've talked a lot about the importance of keeping the responsibility of a service focused, cohesive, and limited, so what I'm about to say might sound a little counterintuitive. Sometimes, when in doubt about service boundaries, it's better to build larger services.

If you err on the side of building services that are too small, it can lead to tight coupling between different services that should be combined in one service. This indicates you've decomposed a business capability too far, making responsibility unclear and making it more difficult—and costly—to refactor this element of functionality.

If instead you combine that functionality into a larger service, you reduce the cost of future refactoring, as well as avoiding intractable cross-service dependencies. Likewise, one of the most expensive costs you'll incur in a microservice application is changing a public interface; reducing the breadth of interfaces between components aids in maintaining flexibility, especially in early stages of development.

Understand that making a service larger also incurs a cost, because larger services become more resistant to change and difficult to replace. But at the beginning of its life, a service will be small. The costs associated with a service being too large are less than the costs of complexity that decomposing too far introduces. You need to carefully observe both service size and complexity to ensure you're not building more monoliths.

Here, it's useful to apply a key principle of lean software development: decide as late as possible. Because building a service incurs cost in both implementation and operation, avoiding premature decomposition when faced with uncertainty can give you time to develop your understanding of the problem space. It also will ensure you're making well-informed decisions about the shape of the application as it grows.

4.6.2 Prepare for further decomposition

The modeling and scoping techniques from earlier in this chapter will help you identify when a service has become too large. Often, you'll be able to identify possible seams quite early in the lifetime of a service. If so, you should endeavor to design your service internals to reflect them, whether through class and namespace design or as a separate library.

Maintaining disciplined internal module boundaries, with a clear public API, is generally sound software design. In a microservice, it reduces the cost of future refactoring by reducing the chance that code becomes highly coupled and difficult to untangle. That said, be careful—an API that's well-designed in the context of a code library may not always be ideal as the interface to a microservice.

4.6.3 Retirement and migration

We've talked about planning for future decomposition, but we also should talk about service retirement. Microservice development requires a certain degree of ruthlessness. It's important to remember that it's what your application does that matters, not the code. Over time—and especially if you start with larger services—you'll find it necessary to either carve out new microservices from existing services or retire microservices altogether.

This process can be difficult. Most importantly, you need to ensure that consuming services don't get broken *and* that they migrate in a timely way to any replacement service.

To carve out new services, you should apply the expand-migrate-contract pattern. Imagine you're carving out a new service from your orders service. When you first built the orders service, you were confident that it'd fit the needs of all order types, so you built it as a single service. But one order type has turned out to be different from the others, and supporting it has bloated your original service.

First, you need to expand—pulling the target functionality into a new service (figure 4.21). Next, you need to migrate consumers of your old service to the new service (figure 4.22). If access is through an API gateway, you can redirect appropriate requests to your new service.

But if other services call the orders service, you need to migrate those usages. Telling other teams to migrate doesn't always work (competing priorities, release cycles, and risk). Instead, you need to either make sure your new service is compelling—make people want to invest effort in migration—or do that migration for them.

To complete the process, you have one last step. Finally, you can contract the original service, removing the now obsolete code (figure 4.23).

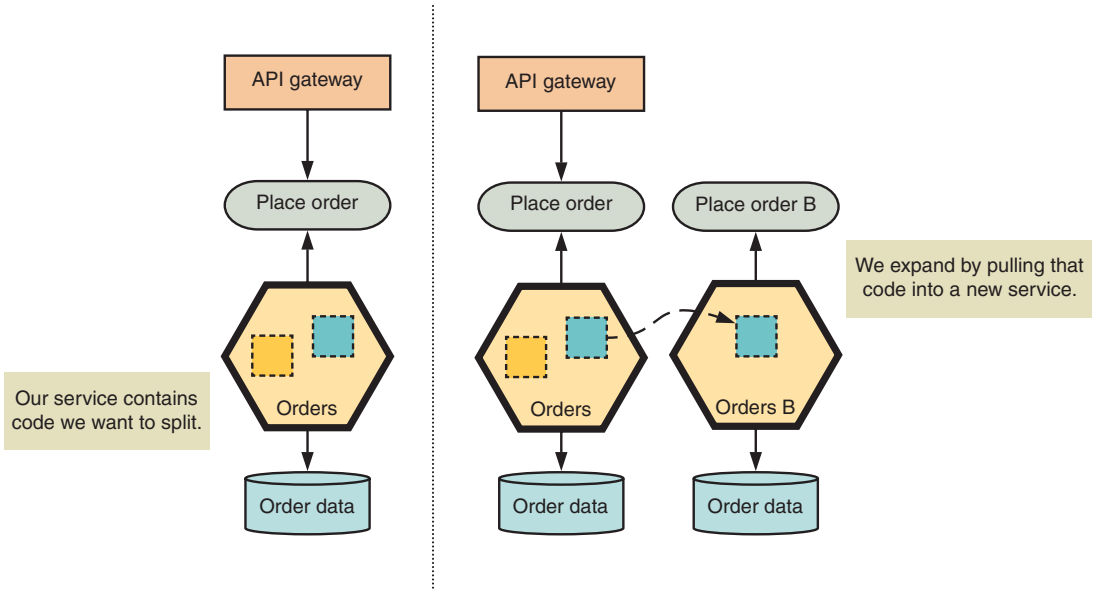


Figure 4.21 Expanding functionality from one service into a new service

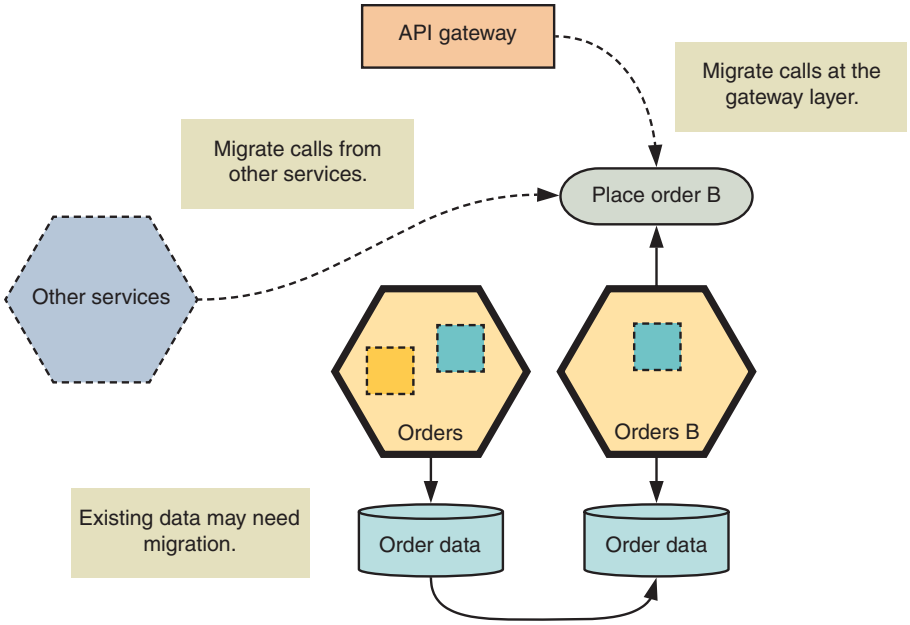


Figure 4.22 Migrating existing consumers to the new service

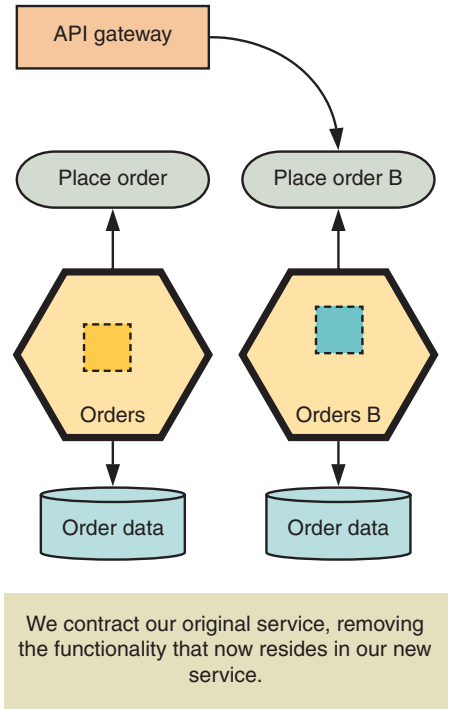


Figure 4.23 In the final state of your service migration, you've contracted your service to remove functionality that now resides in the new service

Great, you made it! This measured, multistep process systematically retires or migrates functionality while reducing the risk of breaking existing service consumers.

4.7 Service ownership in organizations

The examples so far have mostly assumed that a single team is responsible for building and changing microservices. In a large organization, different teams will own different microservices. This isn't a bad thing—it's an important part of scaling as an engineering team.

As we pointed out earlier, bounded contexts themselves are an effective way of splitting application ownership across different teams in an organization. Forming teams that own services in specific bounded contexts takes advantage of the inverse version of Conway's Law: if systems reflect the organizational structure that produces them, you can attain a desirable system architecture by first shaping the structure and responsibilities of your organization. Figure 4.24 illustrates how SimpleBank might organize its engineering teams around the services and bounded contexts you've identified so far.

Splitting ownership and delivery of services across teams has three implications:

- *Limited control*—You might not have full control over the interface or performance of your service dependencies. For example, payments are vital to placing investment strategy orders, but the team model in figure 4.24 means that another team is responsible for the behavior of that dependency.
- *Design constraints*—The needs of consuming services will constrain your service contracts; you need to ensure service changes don't leave consumers behind. Likewise, the possibilities that other existing services offer will constrain your potential designs.
- *Multispeed development*—Services that different teams own will evolve and change at different rates, depending on that team's size, efficiency, and priorities. A feature request from the investment team to the customers team may not make it to the top of the customers team's priority list.

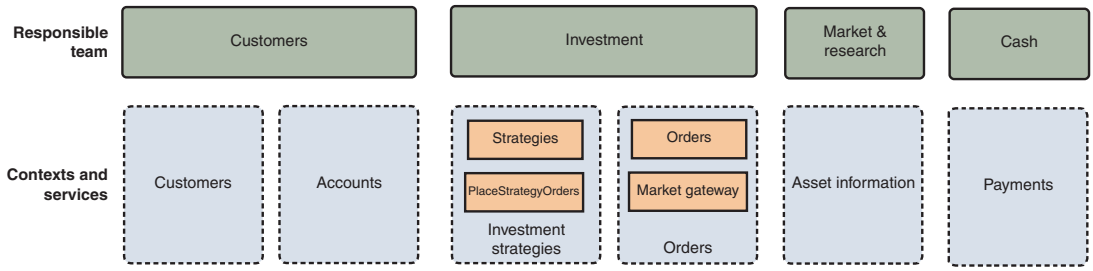


Figure 4.24 A possible model of service and capability ownership by different engineering teams as the size of SimpleBank's engineering organization grows

These implications can present an immense challenge, but applying a few tactics can help:

- *Openness*—Ensuring that all engineers can view and change all code reduces protectiveness, helps different teams understand each other's work, and can reduce blockers.
- *Explicit interfaces*—Providing explicit, documented interfaces for services reduces communication overhead and improves overall quality.
- *Worry less about DRY*—A microservice approach is biased toward delivery pace, rather than efficiency. Although engineers want to practice DRY (don't repeat yourself), you should expect some duplication of work in a microservice approach.
- *Clear expectations*—Teams should set clear expectations about the expected performance, availability, and characteristics of their production services.

These sorts of tactics touch on the people side of microservices. This is a substantial topic by itself, which we'll explore in depth in the final chapter of this book.

Summary

- You scope services through a process of understanding the business problem, identifying entities and use cases, and partitioning service responsibility.
- You can partition services in several ways: by business capability, use case, or volatility. And you can combine these approaches.
- Good scoping decisions result in services that meet three key microservice characteristics: responsible for a single capability, replaceable, and independently deployable.
- Bounded contexts often align with service boundaries and provide a useful way of considering future service evolution.
- By considering areas of volatility, you can encapsulate areas that change together and increase future amenability to change.
- Poor scoping decisions can become costly to rectify, as the effort involved in refactoring is higher across multiple codebases.
- Services may also encapsulate technical capabilities, which simplify and support business capabilities and maximize reusability.
- If service boundaries are ambiguous, you should err on the side of coarse-grained services but use internal modules to actively prepare for future decomposition.
- Retiring services is challenging, but you'll need to do it as a microservice application evolves.
- Splitting ownership across teams is necessary in larger organizations but causes new problems: limited control, design constraints, and multispeed development.
- Code openness, explicit interfaces, continual communication, and a relaxed approach to the DRY principle can alleviate tension between teams.

5 Transactions and queries in microservices

This chapter covers

- The challenges of consistency in a distributed application
- Synchronous and asynchronous communication
- Using sagas to develop business logic across multiple services
- API composition and CQRS for microservice queries

Many monolithic applications rely on transactions to guarantee consistency and isolation when changing application state. Obtaining these properties is straightforward: an application typically interacts with a single database, with strong consistency guarantees, using frameworks that provide support for starting, committing, or rolling back transactional operations. Each logical transaction might involve several distinct entities; for example, placing an order will update transactions, reserve stock positions, and charge fees.

You're not so lucky in a microservice application. As you learned earlier, each independent service is responsible for a specific capability. Data ownership is decentralized,

ensuring a single owner for each "source of truth." This level of decoupling helps you gain autonomy, but you sacrifice some of the safety you were previously afforded, making consistency an application-level problem. Decentralized data ownership also makes retrieving data more complex. Queries that previously used database-level joins now require calls to multiple services. This is acceptable for some use cases but painful for large data sets.

Availability also impacts your application design. Interactions between services might fail, causing business processes to halt, leaving your system in an inconsistent state.

In this chapter, you'll learn how to use *sagas* to coordinate complex transactions across multiple services and explore best practices for efficiently querying data. Along the way, we'll examine different types of event-based architectures, such as event sourcing, and their applicability to microservice applications.

5.1 Consistent transactions in distributed applications

Imagine you're a customer at SimpleBank and you want to sell some stock. If you recall chapter 2, this involves several operations (figure 5.1):

1. You create an order.
2. The application validates and reserves the stock position.
3. The application charges you a fee.
4. The application places the order to the market.

From your perspective as a customer, this operation appears to be atomic: charging a fee, reserving stock, and creating an order happen at the same time, and you can't sell stock that you don't have or sell a stock you do have more than once.

In many monolithic applications,[1] those requirements are easy to meet: you can wrap your database operations in an ACID transaction and rest easy in the knowledge that errors will cause an invalid state to be rolled back.

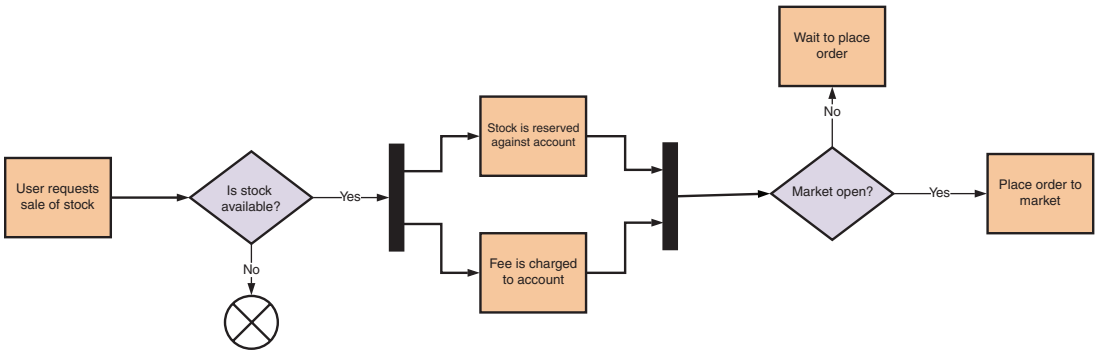


Figure 5.1 Placing a sell order

[1] At least, those with a typical three-tier architecture and a single persistent data store.

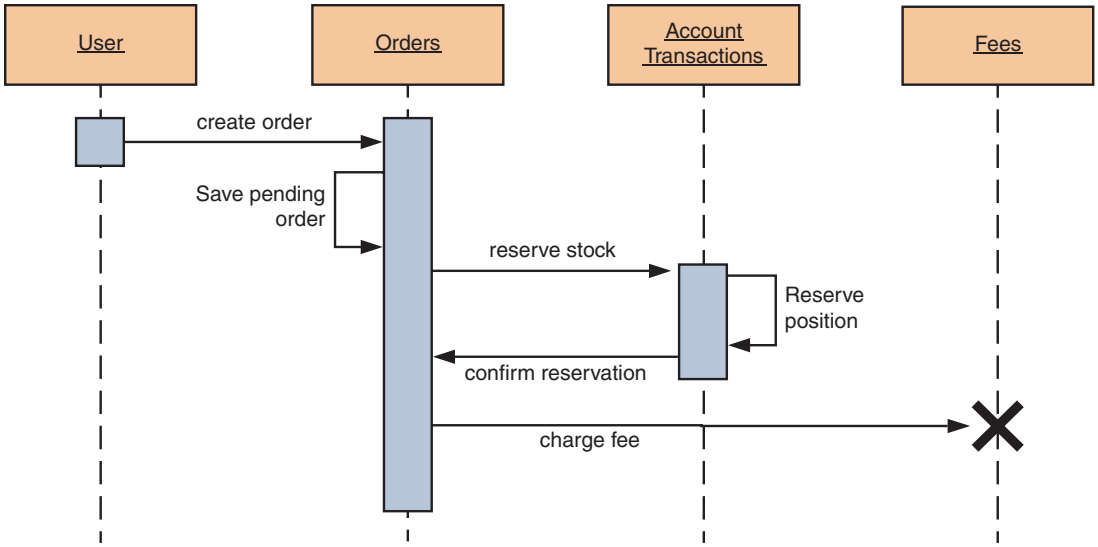


Figure 5.2 Failure occurs when charging a fee in your cross-service order placement process

By contrast, in your microservice application, each of the actions in figure 5.1 is performed by a distinct service responsible for a subset of application state. Decentralized data ownership helps ensure services are independent and loosely coupled, but it forces you to build application-level mechanisms to maintain overall data consistency.

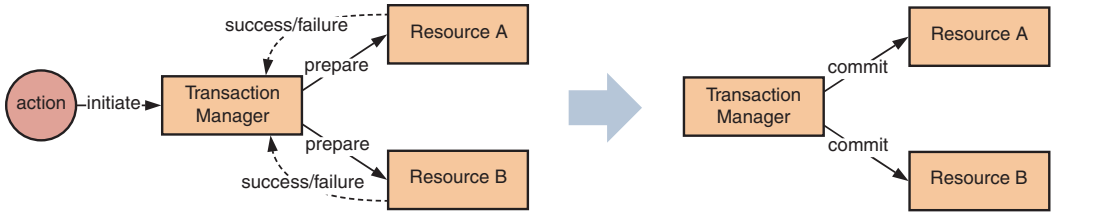
Let's say an orders service is responsible for coordinating the process of selling a stock. It calls account transactions to reserve stock and then the fees service to charge the customer. But that transaction fails. (See figure 5.2.)

At this stage, your system is in an inconsistent state: stock is reserved, an order is created, but you haven't charged the customer. You can't leave it like this—so the implementation of orders needs to initiate corrective action, instructing the account transactions service to compensate and remove the stock reservation. This might look simple, but it becomes increasingly complex when many services are involved, transactions are long-running, or an action triggers further interleaved downstream transactions.

5.1.1 Why can't you use distributed transactions?

Faced with this problem, your first impulse might be to design a system that achieves transactional guarantees across multiple services. A common approach is to use the two-phase commit (2PC) protocol.[2] In this approach, you use a transaction manager to split operations across multiple resources into two phases: prepare and commit (figure 5.3).

[2] See https://en.wikipedia.org/wiki/Two-phase_commit_protocol for more information.



In the **prepare** phase, the transaction manager instructs resources to prepare their relevant action.

In the **commit** phase, the transaction manager instructs resources to commit or abort the prepared action.

Figure 5.3 The prepare and commit phases of a 2PC protocol

This sounds great—like what you're used to. Unfortunately, this approach is flawed. First, 2PC implies synchronicity of communication between the transaction manager and resources. If a resource is unavailable, the transaction can't be committed and must roll back. This in turn increases the volume of retries and decreases the availability of the overall system. To support asynchronous service interactions, you would need to support 2PC with services *and* the messaging layer between them, limiting your technical choices.

> **NOTE** In a microservice application, availability is the product of all microservices involved in processing a given action. Because no service is 100% reliable, involving more services lessens overall reliability, increasing the probability of failure. We'll explore this in detail in the next chapter.

Handing off significant orchestration responsibility to a transaction manager also violates one of the core principles of microservices: service autonomy. At worst, you'd end up with dumb services representing CRUD operations against data, with transaction managers wholly encapsulating the interesting behavior of your system.

Finally, a distributed transaction places a lock on the resources under transaction to ensure isolation. This makes it inappropriate for long-running operations, as it increases the risk of contention and deadlock. What should you do instead?

5.2 Event-based communication

Earlier in this book, we discussed using events emitted by services as a communication mechanism. Asynchronous events aid in decoupling services from each other and increase overall system availability, but they also encourage service authors to think in terms of *eventual consistency*. In an eventually consistent system, you design complex outcomes to result from several independent local transactions over time, which leads you to explicitly design underlying resources to represent tentative states. From the perspective of Eric Brewer's CAP theorem,[3] this design approach prioritizes the availability of underlying data.

[3] Consistency, availability, partition tolerance—see Eric Brewer, "CAP Twelve Years Later: How the "Rules" Have Changed," *InfoQ*, May 30, 2012, http://mng.bz/HGA3, for more information.

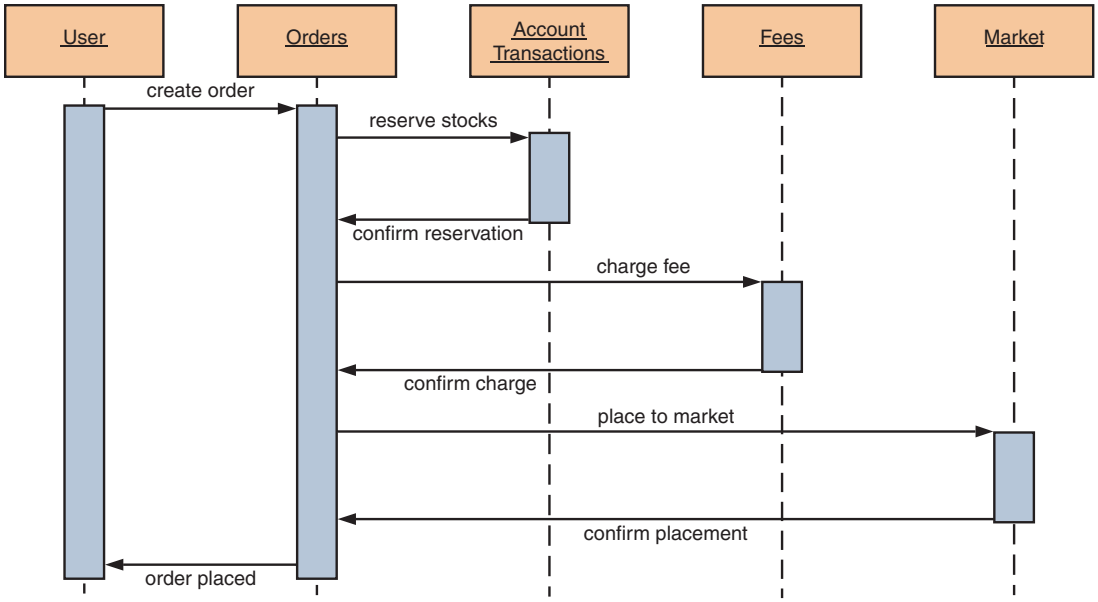


Figure 5.4 The synchronous process of placing a sell order

To illustrate the difference between a synchronous and an asynchronous approach, let's return to the sell order example. In a synchronous approach (figure 5.4), the orders service orchestrates the behavior of other services, invoking a sequence of steps until the order is placed to the market. If any steps fail, the orders service is responsible for initiating rollback action with other services, such as reversing the charge.
In this approach, the orders service takes on substantial responsibility:

- It knows which services it needs to call, as well as their order.
- It needs to know what to do in case any downstream service produces an error or can't proceed due to business rules.

Although this type of interaction is easy to reason through—as the call graph is logical and sequential—this level of responsibility tightly couples the orders service to other services, limiting its independence and increasing the difficulty of making future changes.

5.2.1 Events and choreography

You can redesign this scenario to use events (figure 5.5). Each service subscribes to events that interest it to know when it must perform some work:

1. When the user issues a sell request via the UI, the application publishes an `OrderRequested` event.
2. The orders service picks up this event, processes it, and publishes back to the event queue an `OrderCreated` event.

3. Both the transaction and fees services then pick up this event. Each one of them performs its work and publishes back events to notify about the completion.
4. The market service in turn is waiting for a pair of events notifying it of the charging of fees and the reservation of stocks. When both arrive, it knows it can place the order against the stock exchange. Once that's finished, the market service publishes a final event back to the queue.

Events allow you to take an optimistic approach to availability. For example, if the fees service were down, the orders service would still be able to create orders. When the fees service came back online, it could continue processing a backlog of events. You can extend this to rollback: if the fees service fails to charge because of insufficient funds, it could emit a `ChargeFailed` event, which other services would then consume to cancel order placement.

This interaction is *choreographed*: each service reacts to events, acting independently without knowledge of the overall outcome of the process. These services are like dancers: they know the steps and what to do in each section of a musical piece, and they react accordingly without you needing to explicitly invoke or command them. In turn, this design decouples services from each other, increasing their independence and making it easier to deploy changes independently.

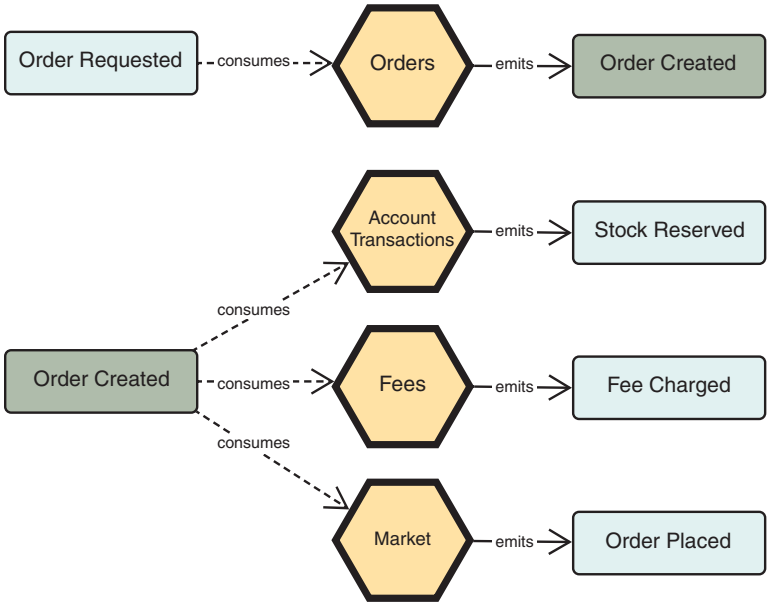


Figure 5.5 Services consuming and emitting events for order placement

Events and the monolith

An event-oriented approach to service communication shines when migrating a monolithic application to microservices. By emitting events from the monolith, you consume them in microservices that you're developing in parallel. This way, you can build new features without tightly coupling your monolith to your new services.

Think about it: you emit an event, and that's the only change you need to implement on the monolith to make an external system work alongside the current one, lowering risk and enabling safer experimentation on new services.

5.3 Sagas

The choreographed approach is a basic example of the *saga* pattern. A saga is a coordinated series of local transactions; a previous step triggers each step in the saga.

The concept itself significantly predates the microservice approach. Hector Garcia-Molina and Kenneth Salem originally described sagas in a 1987 paper[4] as an approach toward long-lived transactions in database systems. As with distributed transactions, locking in long-lived transactions reduces availability—a saga solves this as a sequence of interleaved, individual transactions.

As each local transaction is atomic—but not the saga as a whole—a developer must write their code to ensure that the system ultimately reaches a consistent state, even if individual transactions fail. Pat Helland's famous paper, "Life Beyond Distributed Transactions,"[5] suggests that you can think of this as uncertainty—an interaction across multiple services may not have a guaranteed outcome. In a distributed transaction, you manage uncertainty using locks on data; without transactions, you manage uncertainty through semantically appropriate workflows that confirm, cancel, or compensate for actions as they occur.

Before we talk about sell orders and services, let's look at a simple real-world saga: purchasing a cup of coffee.[6] Typically, this might involve four steps: ordering, payment, preparation, and delivery (figure 5.6). In the normal outcome, the customer pays for and receives the coffee they ordered.

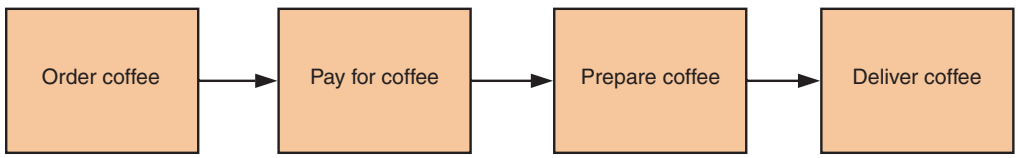


Figure 5.6 The process of purchasing a cup of coffee

[4] See Hector Garcia-Molina and Kenneth Salem, "Sagas," http://mng.bz/Qdot, for the original paper.

[5] See Pat Helland, "Life Beyond Distributed Transactions," *acmqueue*, December 12, 2016, http://queue.acm.org/detail.cfm?id=3025012.

[6] Adapted from Gregor Hohpe, "Compensating Action," *Enterprise Integration Patterns*, http://mng.bz/5FcG.

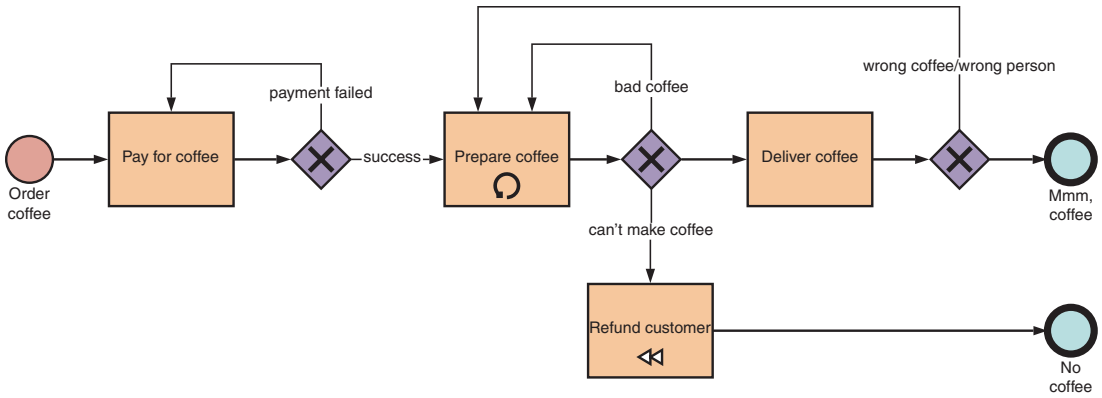


Figure 5.7 Purchasing a cup of coffee with compensating actions

This can go wrong! The coffee shop machine might break; the barista might make a cappuccino, but I wanted a flat white; they might give my coffee to the wrong customer; and so on. If one of these events occurs, the barista will naturally compensate: they might make my coffee again or refund my payment (figure 5.7). In most cases, I'll eventually get my coffee.

You use compensating actions in sagas to undo previous operations and return your system to a more consistent state. The system isn't guaranteed to be returned to the *original* state; the appropriate actions depend on business semantics. This design approach makes writing business logic more complex—because you need to consider a wide range of potential scenarios—but is a great tool for building reliable interactions between distributed services.

5.3.1 Choreographed sagas

Let's return to the earlier example—sell orders—to better understand how you can apply the saga pattern to your microservices. The actions in this saga are choreographed: each action, T_x, is performed in response to another, but without an overall conductor or orchestrator. You can break this task into five subtasks:

- T1—Create the order.
- T2—Reserve the stock position, which the account transaction service implements.
- T3—Calculate and charge the fee, which the fees service implements.
- T4—Place the purchase order to the market, which the market service implements.
- T5—Update the status of the order to be placed.

Figure 5.8 illustrates the optimistic—most likely—path of this interaction.

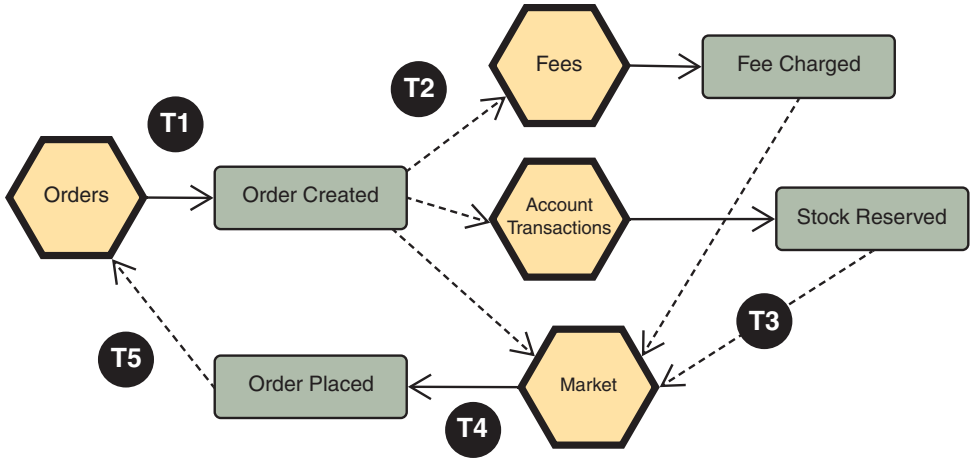


Figure 5.8 A saga for processing a sell order

Let's explain the five steps of this process:

1. The orders service performs T1 and emits an OrderCreated event.
2. The fees, account transactions, and market services consume this event.
3. The fees and account transactions services perform appropriate actions (T2 and T3) and emit events, and the market service consumes them.
4. When the prerequisites for the order are met, the market service places the order (T4) to the market and emits an OrderPlaced event.
5. Lastly, the orders service consumes that event and updates the status of the order (T5).

Each of these tasks might fail—in which case, your application should roll back to a sane, consistent state. Each of your tasks has a compensating action:

- C1—Cancel the order that the customer created.
- C2—Reverse the reservation of stock positions.
- C3—Revert the fee charge, refunding the customer.
- C4—Cancel the order placed to market.
- C5—Reverse the state of the order.

What triggers these actions? You guessed it—events! For example, imagine that placing the order to market fails. The market service will cancel the order by emitting an event—OrderFailed—that each other service involved in this saga consumes. When receiving the event, each service will act appropriately: the orders service will cancel the customer's order; the transaction service will cancel the stock reservation; and the fees service will reverse the fee charged, executing actions C1, C2, and C3, respectively. This is shown in figure 5.9.

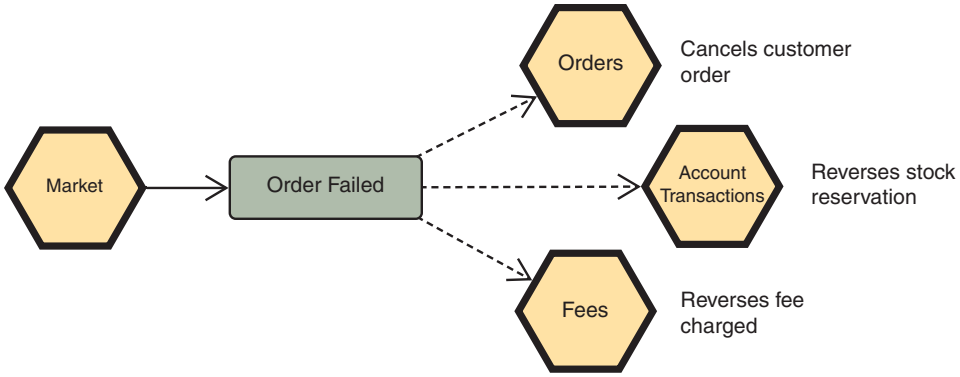


Figure 5.9 The market service emits a failure event is to initiate a rollback process across multiple services.

This form of rollback is intended to make the system *semantically*, not mathematically consistent. Your system on rollback of an operation may not be able to return to the exact same initial state. Imagine one of the tasks executed on calculating the fees was sending out an email. You can't unsend an email, so you'd instead send another one acknowledging the error and saying the amount that the fees service had charged was deposited back to the account.

Every action involved in a process might have one or more appropriate compensating actions. This approach adds to system complexity—both in anticipating scenarios and in coding for them and testing them—especially because the more services involved in an interaction, the greater the possible intricacy of rolling back.

Anticipating failure scenarios is a crucial part of building services that reflect real-world circumstance, rather than operating in isolation. When designing microservices, you need to take compensation into account to ensure that the wider application is resilient.

ADVANTAGES AND DRAWBACKS

The choreographed style of interaction is helpful because participating services don't need to explicitly know about each other, which ensures they're loosely coupled. In turn, this increases the autonomy of each service. Unfortunately, it's not perfect.

No single piece of your code knows how to execute a sell order. This can make validation challenging, spreading those rules across multiple distinct services. It also increases the complexity of state management: each service needs to reflect distinct states in the processing of an order. For example, the orders service must track whether an order has been created, placed, canceled, rejected, and so on. This additional complexity increases the difficulty of reasoning about your system.

Choreography also introduces cyclic dependencies between services: the orders service emits events that the market service consumes, but, in turn, it also consumes events that the market service emits. These types of dependencies can lead to release time coupling between services.

Generally, when opting for an asynchronous communication style, you must invest in monitoring and tracing to be able to follow the execution flow of your system. In case

of an error, or if you need to debug a distributed system, the monitoring and tracing capabilities act as a flight recorder. You should have all that happens stored there so you can later investigate every single event to make sense of what happened in a multitude of systems. This capability is crucial for choreographed interactions.

> **NOTE** Chapters 11 and 12 will explore how to achieve observability through logging, tracing, and monitoring in microservice applications.

A choreographed approach makes it difficult to know how far along a process is. Likewise, the order of rollback might be important; this isn't guaranteed by choreography, which has looser time guarantees than an orchestrated or synchronous approach. For simple, near-instant workflows, knowing where you're at is often irrelevant, but many business processes aren't instant—they might take multiple days and involve disparate systems, people, and organizations.

5.3.2 Orchestrated sagas

Instead of choreography, you can use *orchestration* to implement sagas. In an orchestrated saga, a service takes on the role of orchestrator (or coordinator): a process that executes and tracks the outcome of a saga across multiple services. An orchestrator might be an independent service—recall the verb-oriented services from chapter 4—or a capability of an existing service.

The sole responsibility of the orchestrator is to manage the execution of the saga. It may interact with participants in the saga via asynchronous events or request/response messages. Most importantly, it should track the state of execution for each stage in the process; this is sometimes called the *saga log*.

Let's make the orders service a saga coordinator. Figure 5.10 illustrates the happy path where a customer places an order successfully.

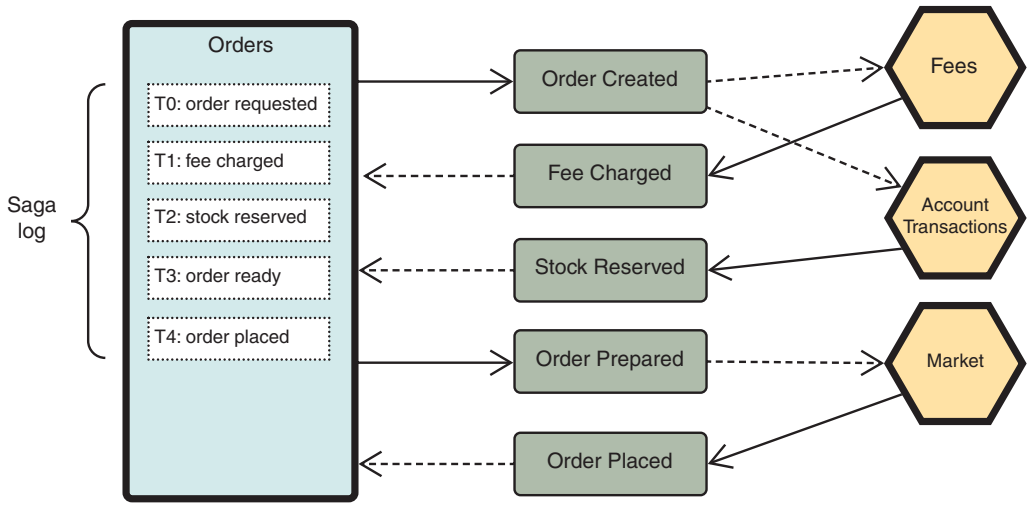


Figure 5.10 An orchestrated saga for placing an order

You'll quickly see the key difference between this and the choreographed example from figure 5.8: the orders service tracks the execution of each substep in the process of placing an order. It's useful to think of the coordinator as a state machine: a series of states and transitions between those states. Each response from a collaborator triggers a state change, moving the orchestrator toward the saga outcome.

As you know, a saga won't always be successful. In an orchestrated saga, the coordinator is responsible for initiating appropriate reconciliation actions to return the entities affected by the failed transaction to a valid, consistent state.

Like you did earlier, imagine the market service can't place the order to market. The orchestrating service will initiate compensating actions:

1. It'll issue a request to the account transaction service to reverse the lock placed on the holdings to be sold.
2. It'll issue a request to cancel the fee that was charged to the customer.
3. It may change the state of the order to reflect the outcome of the saga—for example, to rejected or failed. This depends on the business logic (and whether failed orders should be shown to the customer or retried).

In turn, the orchestrator also could track the outcome of actions 1 and 2. Figure 5.11 illustrates this failure scenario.

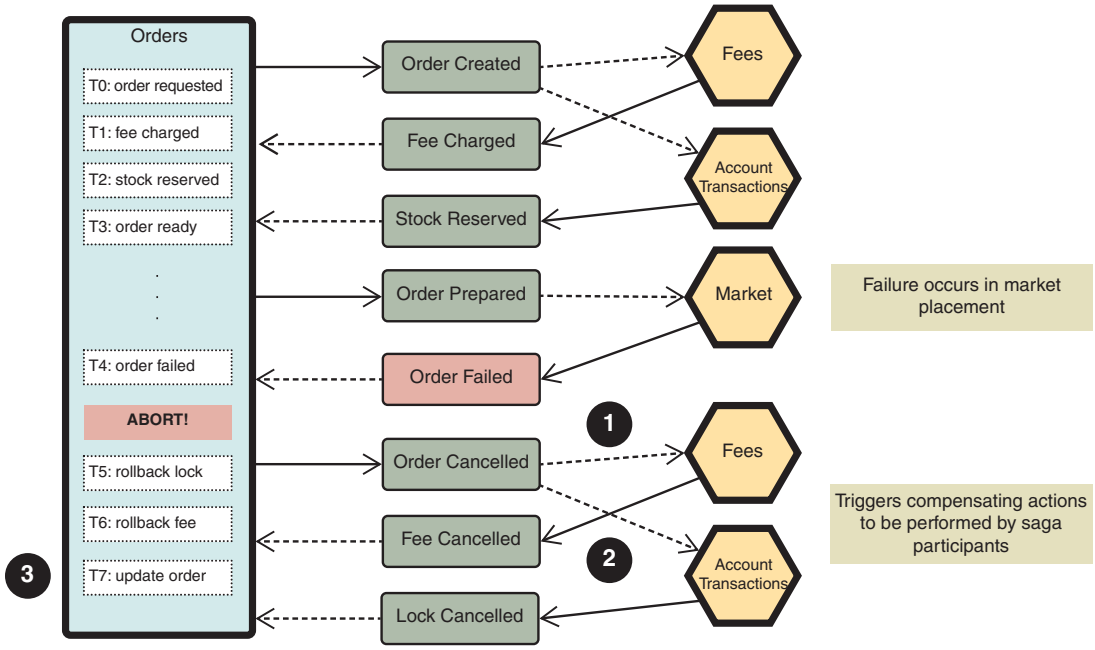


Figure 5.11 In this unsuccessful saga, a failure by the market service results in the orchestrator triggering compensating actions.

> **TIP** Don't forget that compensating actions might not all happen instantaneously or at the same time. For example, if the fee was charged to a customer's debit card, it might take a week for their bank to reverse the charge.

But if the desired actions you want to happen can fail, the compensating actions—or the orchestrator itself—also could fail. You should design compensating actions to be safe to retry without unintentional side effects (for example, double refunds). At worst, repeated failure during rollback might require manual intervention. Thorough error monitoring should catch these scenarios.

Advantages and drawbacks

Centralizing the saga's sequencing logic in a single service makes it significantly easier to reason about the outcome and progress of that saga, as well as change the sequencing in one place. In turn, this can simplify individual services, reducing the complexity of states they need to manage, because that logic moves to the coordinator.

This approach does run the risk of moving too much logic to the coordinator. At worst, this makes the other services anemic wrappers for data storage, rather than autonomous and independently responsible business capabilities.

Many microservice practitioners advocate peer-to-peer choreography over orchestration, as they see this approach to reflect the "smart endpoints, dumb pipes" aim of microservice architecture, in contrast to the heavy workflow tools (such as WS-BPEL) people often used in enterprise SOA. But orchestrated approaches are becoming increasingly popular in the community, especially for building long-running interactions, as seen by the popularity of projects like Netflix Conductor and AWS Step Workflows.

5.3.3 Interwoven sagas

Unlike ACID transactions, sagas aren't isolated. The result of each local transaction is immediately visible to other transactions affecting that entity. This visibility means that a given entity might get simultaneously involved in multiple, concurrent sagas. As such, you need to design your business logic to expect and handle intermediate states. The complexity of the interleaving required primarily depends on the nature of the underlying business logic.

For now, imagine that a customer placed an order by accident and wanted to cancel it. If they issued their request before the order was placed to market, the order placement saga would still be in progress, and this new instruction would potentially need to interrupt it (figure 5.12).

Three common strategies for handling interwoven sagas are available: short-circuiting, locking, and interruption.

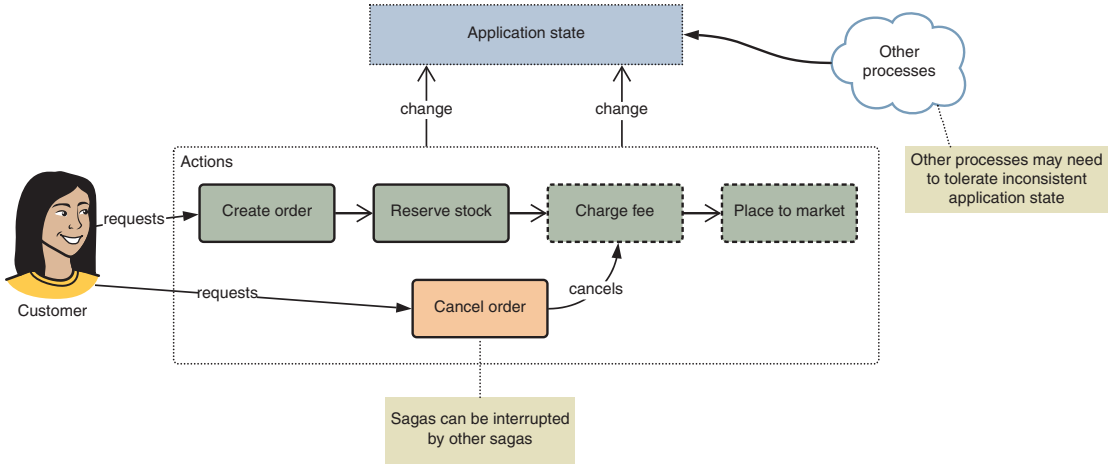


Figure 5.12 Steps in sagas may be interwoven

SHORT-CIRCUITING

You could prevent the new saga from being initiated while the order is still within another saga. For example, the customer couldn't cancel the order until after the market service attempted to place it to the market. This isn't great for a user but is probably the easiest strategy!

LOCKING

You could use locks to control access to an entity. Different sagas that want to change the state of the entity would wait to obtain the lock. You've already seen an example of this in action: you place a reservation—or lock—on a stock balance to ensure that a customer can't sell a holding twice if it's involved in an active order.

This can lead to deadlocks if multiple sagas block each other trying to access the lock, requiring you to implement deadlock monitoring and timeouts to make sure the system doesn't grind to a halt.

INTERRUPTION

Lastly, you could choose to interrupt the actions taking place. For example, you could update the order status to "failed." When receiving a message to send an order to market, the market gateway could revalidate the latest order status to ensure the order was still valid to send, and in this case it would see a "failed" status. This approach increases the complexity of business logic but avoids the risk of deadlocks.

5.3.4 Consistency patterns

Although sagas rely heavily on compensating actions, they're not the only approach you might use to achieve consistency in service interactions. So far, we've encountered two patterns for dealing with failure: compensating actions (refund my coffee payment) and retries (try to make the coffee again). Table 5.1 outlines other strategies.

Table 5.1 Consistency strategies in microservice applications

#	Name	Strategy
1	Compensating action	Perform an action that undoes prior action(s)
2	Retry	Retry until success or timeout
3	Ignore	Do nothing in the event of errors
4	Restart	Reset to the original state and start again
5	Tentative operation	Perform a tentative operation and confirm (or cancel) later

The use of these strategies will depend on the business semantics of your service interaction. For example, when processing a large data set, it might make sense to ignore individual failures (applying strategy #3), because the cost of processing the overall data set is large. When interacting with a warehouse—for example, to fulfill orders—it'd be reasonable to place a tentative hold (strategy #5) on a stock item in a customer's basket to reduce the possibility of overselling.

5.3.5 Event sourcing

So far, we've assumed that entity state and events are distinct: the former is stored in an appropriate transactional store, whereas the latter are published independently (figure 5.13).

An alternative to this approach is the *event sourcing* pattern: rather than publishing events *about* entity state, you represent state entirely as a sequence of events that have happened to an object. To get the state of an entity at a specific time, you aggregate events before that date. For example, imagine your orders service:

- In the traditional persistence approaches we've assumed so far, a database would store the latest state of the order.
- In event sourcing, you'd store the events that happened to change the state of the order. You could materialize the current state of the order by replaying those events.

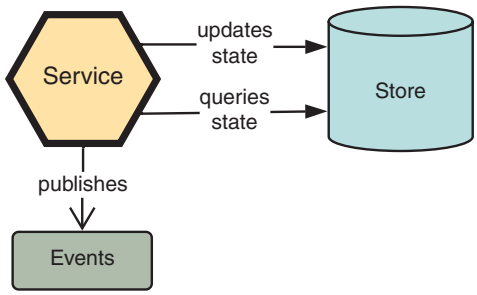


Figure 5.13 A service storing state in a data store and publishing events, in two distinct actions

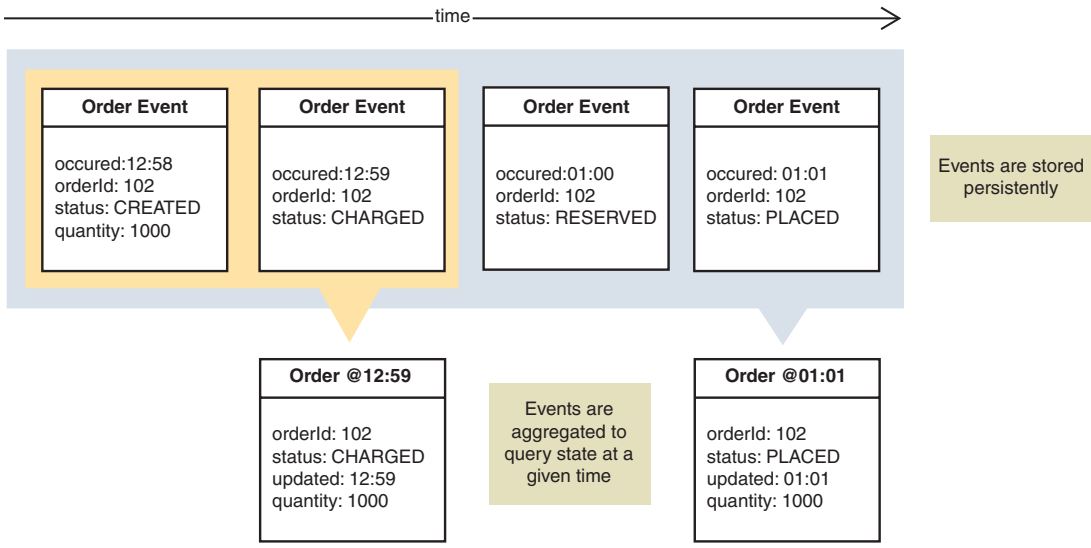


Figure 5.14 An order, stored as a sequence of events

Figure 5.14 illustrates the event sourcing approach for tracking an order's history.

This architecture solves a common problem in enterprise applications: understanding how you reached your current state. It removes the division between state and events; you don't need to stick events on top of your business logic, because your business logic inherently generates and manipulates events. On the other hand, it makes complex queries more difficult: you'd need to materialize views to perform joins or filter by field values, as your event storage format would only support retrieving entities by their primary key.

Event sourcing isn't a requirement for a microservice application, but using events to store application state can be a particularly elegant tool, especially for applications involving complex sagas where tracking the history of state transitions is vital. If you're interested in learning more about event sourcing, Nick Chamberlain's awesome-ddd list (https://github.com/heynickc/awesome-ddd) has a great collection of resources and further reading.

5.4 Queries in a distributed world

Decentralized data ownership also makes retrieving data more challenging, as it's no longer possible to aggregate related data at, or close to, the database level—for example, through joins. Presenting data from disparate services is often necessary at the UI layer of an application.

For example, imagine you're building an administrative UI that shows a list of customers, together with their current open orders. In a SQL database, you'd join these two tables in a single query, returning one dataset. In a microservice application, this *composition* would typically take place at the API level: a service or an API gateway could

perform this (figure 5.15). C*orrelation IDs* — roughly analogous to foreign keys in a relational database — identify relationships between data that each service owns; for example, each order would record the associated customer ID.

The two-step approach in figure 5.15 works well for single entities or small datasets but will scale poorly for bulk requests. If the first query returns N customers, then the second query will be performed N times, which could quickly get out of hand. If we were querying a SQL database, this would be trivial to solve with a join, but because our data is spread across multiple data stores, an easy solution like using a join isn't possible.

We could improve this query by introducing bulk request endpoints and paging, as in listing 5.1. Rather than getting every customer, you'd get the first page; rather than retrieving customer orders one-by-one, you could retrieve them with a list of IDs. You should note, though, that if each customer had thousands of orders, having to page those as well would add substantial overhead.

Listing 5.1 Different endpoints for data retrieval

```
/customers?page=1&size=20
/orders?customerIds=4,5,10,20
```

You should page large datasets.

You should retrieve children using "IN" semantics rather than individually.

API composition is simple and intuitive, and for many use cases, such as individual aggregates or small enumerables, the performance of this approach will be acceptable. For others, such as the following, performance will be inefficient and far from ideal:

- *Queries that return and join substantial data, such as reporting* — "I want all customer orders from the last year."
- *Queries that aggregate or perform analytics across multiple services* — "I want to know the average order value of emerging market stocks purchased by customers over 35."
- *Queries that aren't optimally supported by the service's own database* — For example, complex search patterns are often difficult to optimize in relational databases.

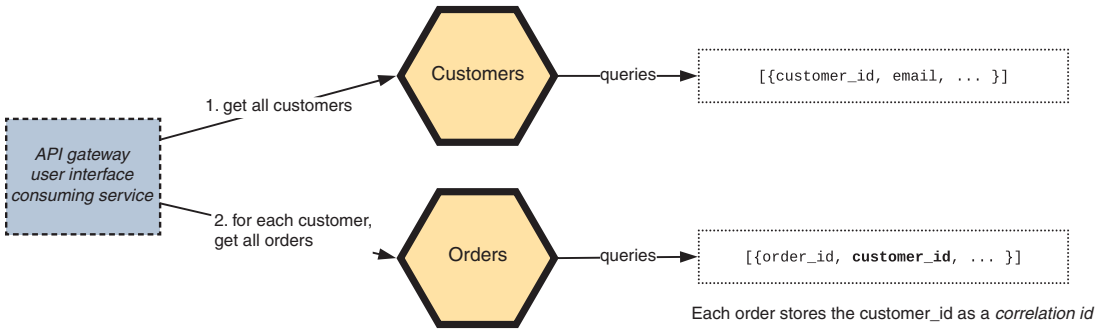


Figure 5.15 Data composition at the API level

Lastly, API composition is impacted by availability. Composition requires synchronous calls to underlying services, so the total availability of a query path is the product of the availability of all services involved in that path. For example, if the two services and the API gateway in figure 5.15 each have an availability of 99%, their availability when called together would be 99%^3: 97.02%. Over the next three sections, we'll discuss how you also can use events to build efficient queries in microservice applications.

NOTE We'll discuss service availability and reliability, and techniques for maximizing those properties in the following chapter.

5.4.1 Storing copies of data

You can elect to have services store or cache data that they receive from other services via events. For example, in figure 5.16, when the fees service receives an `OrderCreated` message, it might elect to store additional detail about the order, beyond the correlation ID. This service can now handle queries like "What was the value of this order?" without needing to retrieve that data with an additional call to the orders service.

This technique can be quite useful but risky:

- Maintaining multiple copies of data increases overall application and service complexity (and possibly, overall storage cost).
- Breaking schema changes in events is extremely tricky to manage, as services become increasingly coupled to event content.
- Cache invalidation is notoriously hard.[7]

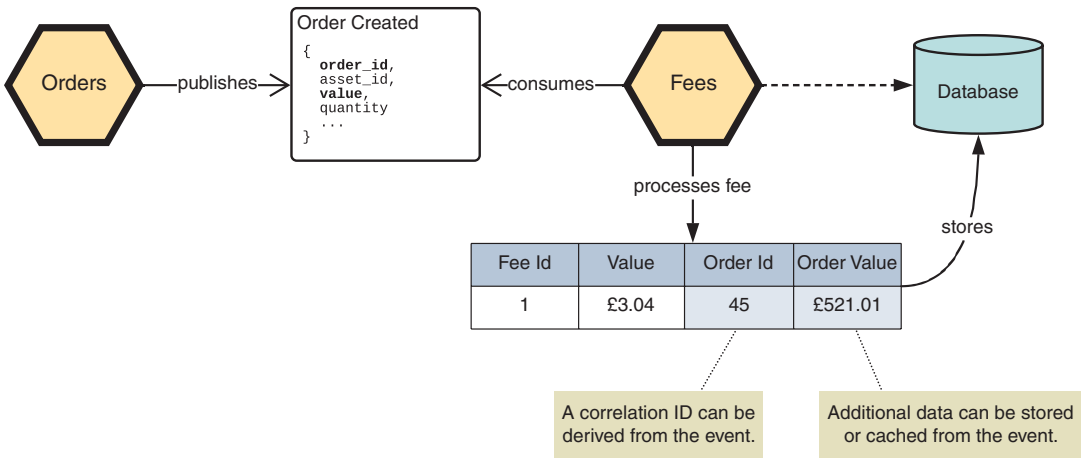


Figure 5.16 You can use events to share, and therefore replicate, state across multiple services

[7] See Martin Fowler, "TwoHardThings," July 14, 2009, https://martinfowler.com/bliki/TwoHardThings.html, and Mark Heath, "Troubleshooting Caching Problems," *SoundCode*, January 23, 2018, http://mng.bz/M2J7.

By maintaining canonical data in multiple locations—updated via asynchronous events, which could be delayed, or fail, or be delivered multiple times—you have to cope with eventual consistency and the chance that the copies of data you retrieve have become stale.

Whether it's fine for data to be stale sometimes is down to the business semantics of the particular feature. But it's a hard tradeoff. The CAP theorem[8] says that you can't have things both ways: you need to choose between availability—returning a successful result, without a guarantee that data is fresh—and consistency—returning the most recent state, or an error.

Guaranteeing consistency tends to result in increased coordination between systems—such as distributed locks—which hampers transaction speed. In contrast, a system that maximizes availability ultimately relies on compensating actions and retries—a lot like sagas. From an architectural perspective, availability is usually easier to achieve and, because of the reduced coordination cost, more amenable to building scalable applications.

Prioritizing availability

Building systems that prioritize availability might require you to avoid the instinctual, consistency-oriented solution to a problem. Even systems that seem like they should prioritize consistency often make availability tradeoffs to maximize successful use.

A great example is an automated teller machine (ATM)—prioritizing availability increases bank revenue. If an ATM can't connect to the bank backend, or the wider ATM network, it'll still allow withdrawals, but cap them, ensuring risk of overdraft is limited. If a withdrawal does place a customer in overdraft, the bank can recoup that with a fee.

A recent article from Eric Brewer—http://mng.bz/HGA3—has a great overview of this scenario.

5.4.2 Separating queries and commands

You can generalize the previous approach—using events to build views—further. In many systems, queries are substantially different from writes: whereas writes affect singular, highly normalized entities, queries often retrieve denormalized data from a range of sources. Some query patterns might benefit from completely different data stores than writes; for example, you might use PostgreSQL as a persistent transactional store but Elasticsearch for indexing search queries. The command-query responsibility segregation pattern (CQRS) is a general model for managing these scenarios by explicitly separating reads (queries) from writes (commands) within your system.[9]

[8] This is a fantastic, "plain English" explanation of the CAP theorem: ksat.me/a-plain-english-introduction-to-cap-theorem/, by Kaushik Sathupadi.

[9] You need to use CQRS if you implement an event-sourcing architecture.

NOTE We won't go into specific technical detail about implementing CQRS, but you can explore frameworks in many languages, such as Commanded (Elixir), CQRS.net (.NET), Lagom (Java and Scala), and Broadway (PHP).

CQRS ARCHITECTURE

Let's sketch out this architecture. In figure 5.17, you can see that CQRS partitions commands and queries:

- The *command* side of an application performs updates to a system—creates, updates and deletes. Commands emit events, either in-band or to a distinct event bus, such as RabbitMQ or Kafka.
- Event handlers consume events to build appropriate *query* or *read* models.
- A separate data store may support each side of the system.

You can apply this pattern both within services and across your whole application—using events to build dedicated query services that own and maintain complex views of application data. For example, imagine you wanted to aggregate order fees across your entire customer base, potentially slicing them by multiple attributes (for example, type of order, asset categories, payment method). This wouldn't be possible at a service level, because neither the fees, orders, nor customers service has all the data needed to filter those attributes.

Instead, as figure 5.18 illustrates, you could build a query service, CustomerOrders, to construct appropriate views. A query service is a good way to handle views that don't clearly belong to any other services, ensuring a reasonable separation of concerns.

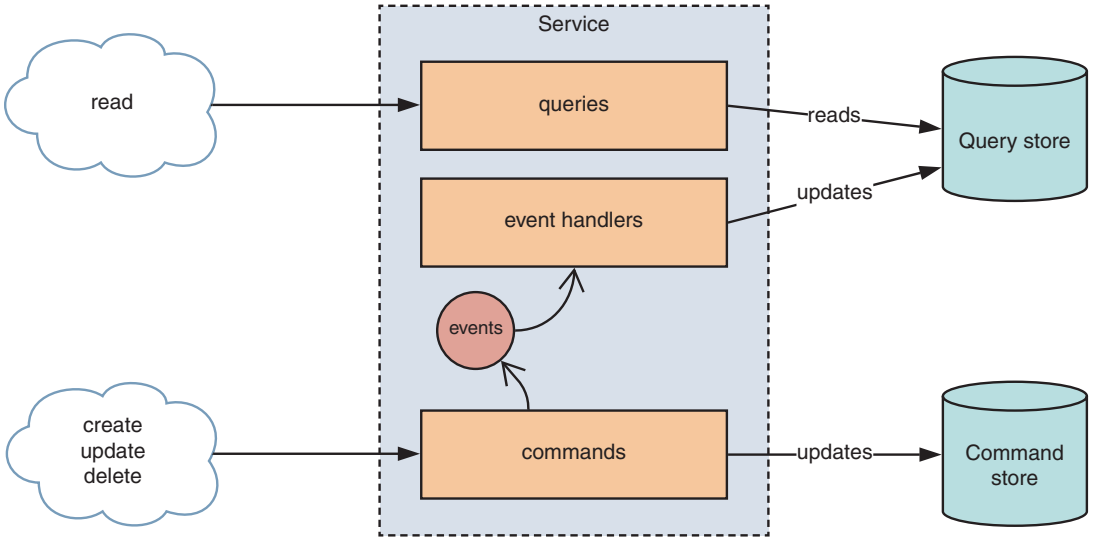


Figure 5.17 CQRS partitions a service into command and query sides, each accessing separate data stores.

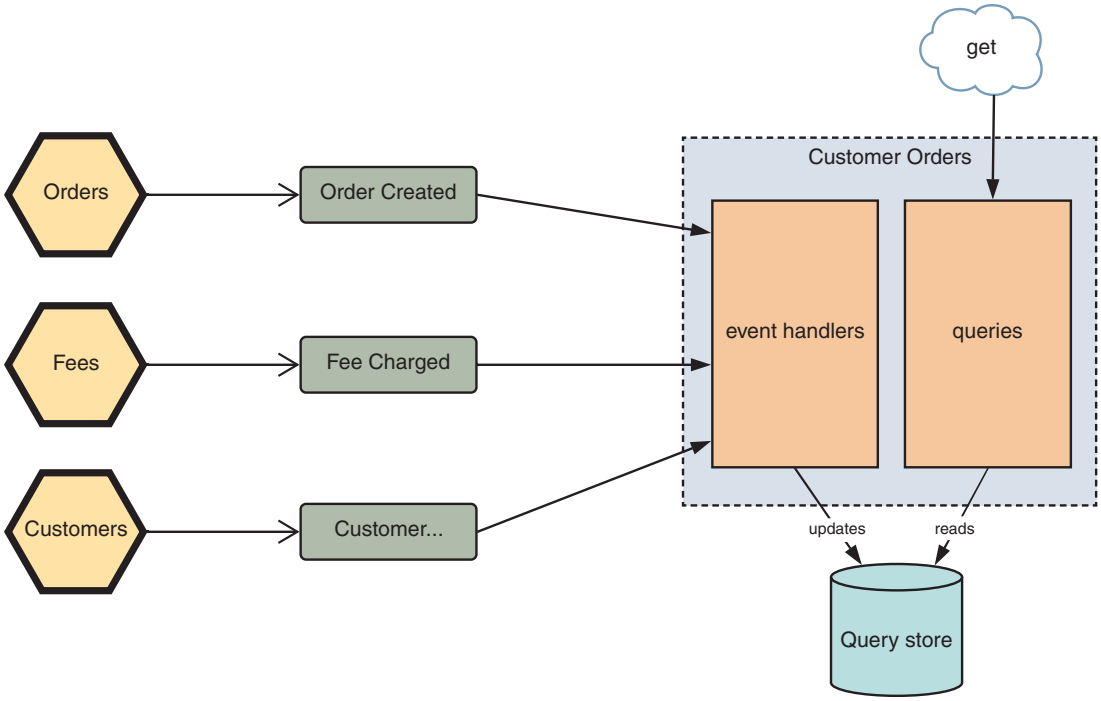


Figure 5.18 **Query services can construct complex views from events that multiple services emit.**

> **TIP** You don't need to use only CQRS within your application. Using different query styles in different scenarios can help achieve a good balance of complexity, implementation speed, and customer value.

So far, this all sounds great! In a microservices application, CQRS offers two key benefits:

- You can optimize the query model for specific queries to improve their performance and remove the need for cross-service joins.
- It aids in separation of concerns, both within services and at an application level.

But it's not without drawbacks. Let's explore those now.

5.4.3 CQRS challenges

Like the data caching example, CQRS requires you to consider eventual consistency because of *replication lag*: inherently, the command state of a service will be updated before the query state. Because events update query models, someone querying that data might receive an out of date view. This might be a frustrating user experience (figure 5.19). Imagine you update the value of an order, but on clicking Confirm, you see the details of the original order! Web UIs that use a POST/redirect/GET[10] pattern will often suffer from this problem.

[10] See https://en.wikipedia.org/wiki/Post/Redirect/Get for more information.

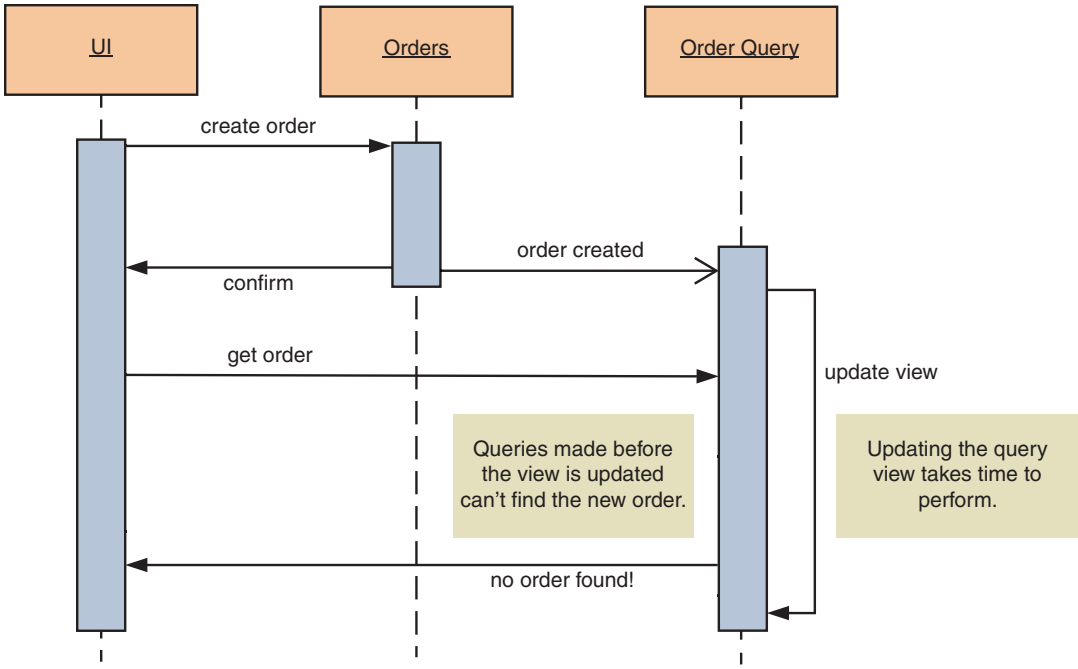


Figure 5.19 Lag in updating a query view leads to inconsistent results when making a request.

In some systems, this might not matter. For example, delayed updates are common for activity feeds[11]—if I post an update on Twitter, it doesn't matter if my followers don't all receive it at the same time. And in fact, attempting to achieve greater consistency can lead to substantial scalability challenges that might not be worth it.

In other systems, it'll be important to ensure you don't query invalid state. You can apply three strategies (figure 5.20) in these scenarios: optimistic updates, polling, or publish-subscribe.

OPTIMISTIC UPDATES

You could update the UI optimistically, based on the expected result of a command. If the command fails, you can roll back the UI state. For example, imagine you like a post on Instagram. The app will show a red heart before the Instagram backend saves that change. If that save fails, Instagram will roll back the optimistic UI change, and you'll have to like it again for it to show a red heart.

This approach relies on having—or being able to derive—all the information you need to update the UI from the command input, so it works best when working with simple entities.

[11] If you're interested in the architecture behind activity streams, https://github.com/tschellenbach/Stream-Framework is a good place to start.

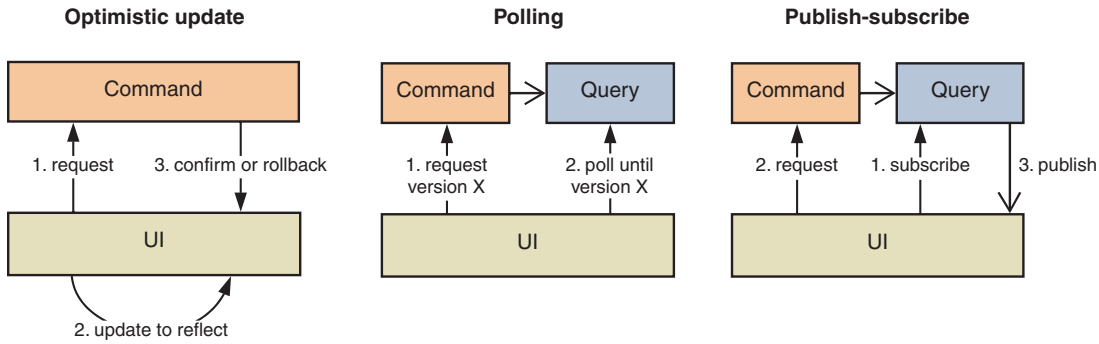


Figure 5.20 Three strategies for dealing with query-side replication lag in CQRS

POLLING

The UI could poll the query API until an expected change has occurred. When initiating a command, the client would set a version, such as a timestamp. For subsequent queries, the client would continue to poll until the version number was equal or greater to the version number specified, indicating that the query model had been updated to reflect the new state.

PUBLISH-SUBSCRIBE

Instead of polling for changes, a UI could subscribe to events on a query model—for example, over a web socket channel. In this case, the UI would only update when the read model published an "updated" event.

As you can see, it's challenging to reason through CQRS, and it requires a different mindset from what you'd have when dealing with normal CRUD APIs. But it can be useful in a microservice application. Done right, CQRS helps to ensure performance and availability in queries, even as you distribute data and responsibility across multiple distinct services and data stores.

5.4.4 Analytics and reporting

You can generalize the CQRS technique to other use cases, such as analytics and reporting. You can transform a stream of microservice events and store them in a data warehouse, such as Amazon Redshift or Google BigQuery (figure 5.21). A transformation stage may involve mapping events to the semantics and data model of the target warehouse or combining events with data from other microservices. If you don't yet know how you want to treat or query events, you can store them in commodity storage, such as Amazon S3, for later querying or reprocessing with big data tools such as Apache Spark or Presto.

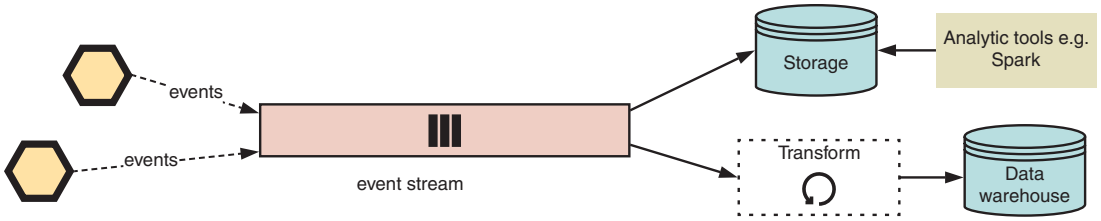


Figure 5.21 You can use microservice events to populate data warehouses or other analytic stores.

5.5 Further reading

We've covered a lot of ground in this chapter, but some topics, like sagas, event sourcing, and CQRS, can each fill entire books. In case you're interested in knowing more about those topics, we recommend the following books:

- *Reactive Application Development*, by Duncan K. DeVore, Sean Walsh, and Brian Hanafee, https://www.manning.com/books/reactive-application-development (ISBN 9781617292460)
- *Microservices Patterns*, by Chris Richardson, https://www.manning.com/books/microservices-patterns (ISBN 9781617294549)
- *Event Streams in Action*, by Alexander Dean, https://www.manning.com/books/event-streams-in-action (ISBN 9781617292347)

Summary

- ACID properties are difficult to achieve in interactions across multiple services; microservices require different approaches to achieve consistency.
- Coordination approaches, such as two-phase commit, introduce locking and don't scale well.
- An event-based architecture decouples independent components and provides a foundation for scalable business logic and queries in a microservice application.
- Biasing towards availability, rather than consistency, tends to lead to a more scalable architecture.
- Sagas are global actions composed from message-driven, independent local transactions. They achieve consistency by using compensating actions to roll back incorrect state.
- Anticipating failure scenarios is a crucial element of building services that reflect real-world circumstance, rather than operating in isolation.
- You typically implement queries across microservices by composing results from multiple APIs.
- Efficient complex queries should use the CQRS pattern to materialize read models, especially where those query patterns require alternative data stores.

6 Designing reliable services

This chapter covers

- The impact of service availability on application reliability
- Designing microservices that defend against faults in their dependencies
- Applying retries, rate limits, circuit breakers, health checks, and caching to mitigate interservice communication issues
- Applying safe communication standards across many services

No microservice is an island; each one plays a small part in a much larger system. Most services that you build will have other services that rely on them—upstream collaborators—and in turn themselves will depend on other services—downstream collaborators—to perform useful functions. For a service to reliably and consistently perform its job, it needs to be able to trust these collaborators.

This is easier said than done. Failures are inevitable in any complex system. An individual microservice might fail for a variety of reasons. Bugs can be introduced into code. Deployments can be unstable. Underlying infrastructure might let you down: resources

might be saturated by load; underlying nodes might become unhealthy; even entire data centers can fail. As we discussed in chapter 5, you can't even trust that the network between your services is reliable—believing otherwise is a well-known fallacy of distributed computing.[1] Lastly, human error can lead to major failures. For example, I'm writing this chapter a week after an engineer's mistake in running a maintenance script led to a severe outage in Amazon S3, affecting thousands of well-known websites.

It's impossible to eliminate failure in microservice applications—the cost of that would be infinite! Instead, your focus needs to be on designing microservices that are tolerant of dependency failures and able to gracefully recover from them or mitigate the impact of those failures on their own responsibilities.

In this chapter, we'll introduce the concept of service availability, discuss the impact of failure in microservice applications, and explore approaches to designing reliable communication between services. We'll also discuss two different tactics—frameworks and proxies—for ensuring all microservices in an application interact safely. Using these techniques will help you maximize the reliability of your microservice application—and keep your users happy.

6.1 Defining reliability

Let's start by figuring out how to measure the reliability of a microservice. Consider a simple microservice system: a service, *holdings*, calls two dependencies, *transactions* and *market-data*. Those services in turn call further dependencies. Figure 6.1 illustrates this relationship.

For any of those services, you can assume that they spent some time performing work successfully. This is known as *uptime*. Likewise, you can safely assume—because failure is inevitable—that they spent some time failing to complete work. This is known as *downtime*. You can use uptime and downtime to calculate *availability*: the percentage of operational time during which the service was working correctly. A service's availability is a measure of how reliable you can expect it to be.

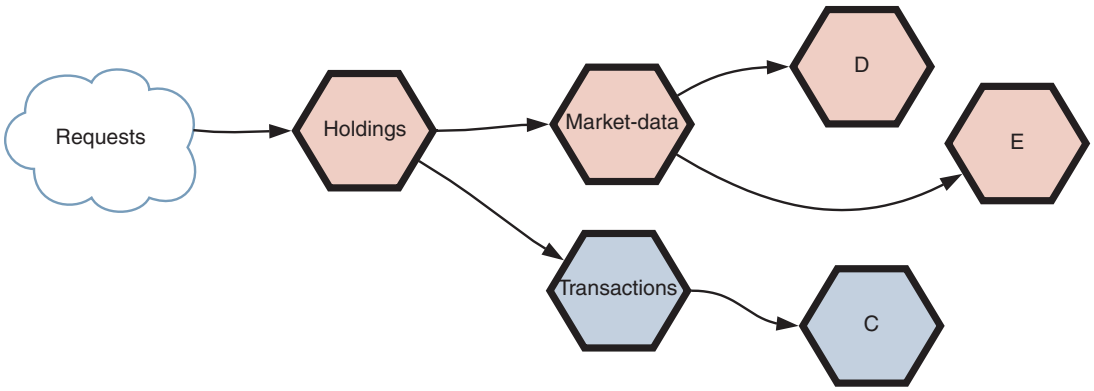


Figure 6.1 A simple microservice system, illustrating dependencies between collaborating services

[1] Peter Deutsch originally posited the eight fallacies of distributed computing in 1994. A good overview is available here: http://mng.bz/9T5F.

A typical shorthand for high availability is "nines:" for example, two nines is 99%, whereas five nines is 99.999%. It'd be highly unusual for critical production-facing services to be less reliable than this.

To illustrate how availability works, imagine that calls from holdings to market-data are successful 99.9% of the time. This might sound quite reliable, but downtime of 0.1% quickly becomes pronounced as volumes increase: only one failure per 1,000 requests, but 1,000 failures per million. These failures will directly affect your calling service unless you can design that service to mitigate the impact of dependency failure.

Microservice dependency chains can quickly become complex. If those dependencies can fail, what's the probability of failure within your whole system? You can treat your availability figure as the probability of a request being successful—by multiplying together the availability figures for the parts of the chain, you can estimate the failure rate across your entire system.

Say you expand the previous example to specify that you have six services with the same success rate for calls. For any request to your system, you can expect one of four outcomes: all services work correctly, one service fails, multiple services fail, or all services fail.

Because calls to each microservice are successful 99.9% of the time, combined reliability of the system will be $0.999^6 = 0.994 = 99.4\%$. Although this is a simple model, you can see that the whole application will always be less reliable than its independent components; the maximum availability you can achieve is a product of the availability of a service's dependencies.

To illustrate, imagine that service D's availability is degraded to 95%. Although this won't affect transactions—because it's not part of that call hierarchy—it will reduce the reliability of both market-data and holdings. Figure 6.2 illustrates this impact.

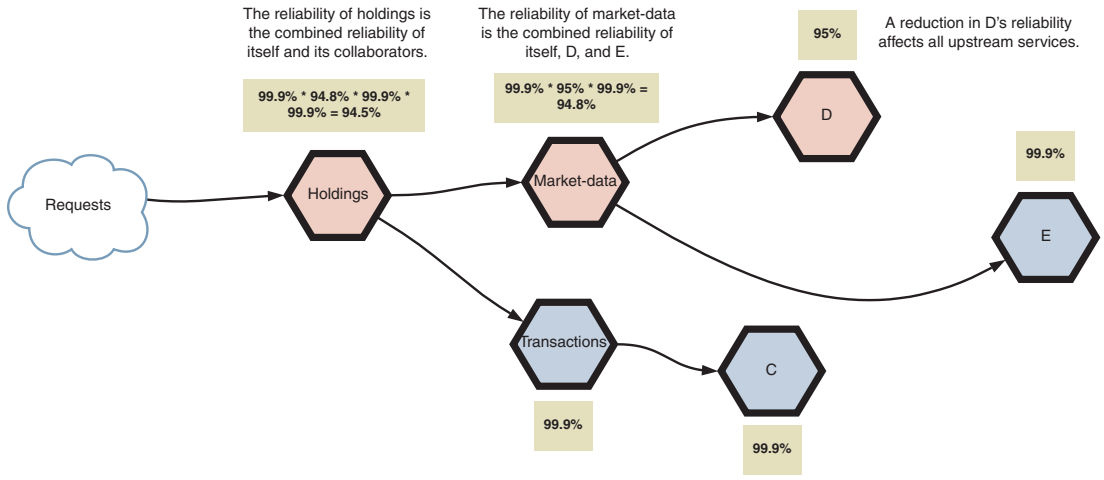


Figure 6.2 The impact of service dependency availability on reliability in a microservice application

It's crucial to maximize service availability—or isolate the impact of unreliability—to ensure the availability of your entire application. Measuring availability won't tell you how to make your services reliable, but it gives you a target to aim for or, more specifically, a goal to guide both the development of services and the expectations of consuming services and engineers.

> **NOTE** How do you monitor availability? We'll explore approaches to monitoring service availability in a microservice application in part 4 of this book.

If you can't trust the network, your hardware, other services, or even your own services to be 100% reliable, how can you maximize availability? You need to design defensively to meet three goals:

- Reduce the incidence of avoidable failures
- Limit the cascading and system-wide impact of unpredictable failures
- Recover quickly—and ideally automatically—when failures do occur

Achieving these goals will ultimately maximize the uptime and availability of your services.

6.2 What could go wrong?

As we've stated, failure is inevitable in a complex system. Over the lifetime of an application, it's incredibly likely that any catastrophe that could happen, will happen. Consequently, you need to understand the different types of failures that your application might be susceptible to. Understanding the nature of these risks and their likelihood is fundamental to both architecting appropriate mitigation strategies and reacting rapidly when incidents do occur.

Balancing risk and cost

It's important to be pragmatic: you can neither anticipate nor eliminate every possible cause of failure. When you're designing for resilience, you need to balance the risk of a failure against what you can reasonably defend against given time and cost constraints:

- The cost to design, build, deploy, and operate a defensive solution
- The nature of your business and expectations of your customers

To put that in perspective, consider the S3 outage I mentioned earlier. You could defend against that error by replicating data across multiple regions in AWS or across multiple clouds. But given that S3 failures of that magnitude are exceptionally rare, that solution wouldn't make economic sense for many organizations because it would significantly increase operational costs and complexity.

As a responsible service designer, you need to identify possible types of failure within your microservice application, rank them by anticipated frequency and impact, and decide how you'll mitigate their impact. In this section, we'll walk you through some

common failure scenarios in microservice applications and how they arise. We'll also explore cascading failures—a common catastrophic scenario in a distributed system.

6.2.1 Sources of failure

Let's examine a microservice to understand where failure might arise, using one of SimpleBank's services as an example. You can assume a few things about the market-data service:

- The service will run on hardware—likely a virtualized host—that ultimately depends on a physical data center.
- Other upstream services depend on the capabilities of this service.
- This service stores data in some store—for example, a SQL database.
- It retrieves data from third-party data sources through APIs and file uploads.
- It may call other downstream SimpleBank microservices.

Figure 6.3 illustrates the service and its relationship to other components.

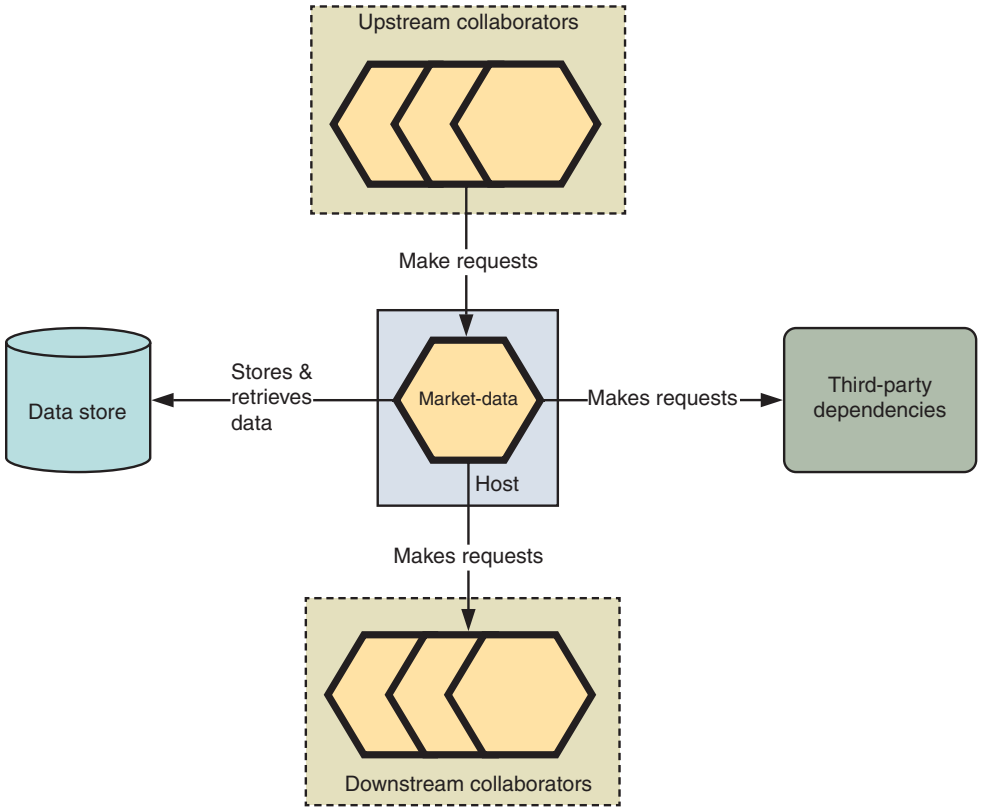


Figure 6.3 Relationships between the market-data microservice and other components of the application

Every point of interaction between your service and another component indicates a possible area of failure. Failures could occur in four major areas:

- *Hardware*—The underlying physical and virtual infrastructure on which a service operates
- *Communication*—Collaboration between different services and/or third parties
- *Dependencies*—Failure within dependencies of a service
- *Internal*—Errors within the code of the service itself, such as defects introduced by engineers

Let's explore each category in turn.

HARDWARE

Regardless of whether you run your services in a public cloud, on-premise, or using a PaaS, the reliability of the services will ultimately depend on the physical and virtual infrastructure that underpins them, whether that's server racks, virtual machines, operating systems, or physical networks. Table 6.1 illustrates some of the causes of failure within the hardware layer of a microservice application.

Table 6.1 Sources of failure within the hardware layer of a microservice application

Source of failure	Frequency	Description
Host	Often	Individual hosts (physical or virtual) may fail.
Data center	Rare	Data centers or components within them may fail.
Host configuration	Occasionally	Hosts may be misconfigured— for example, through errors in provisioning tools.
Physical network	Rare	Physical networking (within or between data centers) may fail.
Operating system and resource isolation	Occasionally	The OS or the isolation system—for example, Docker—may fail to operate correctly.

The range of possible failures at this layer of your application are diverse and unfortunately, often the most catastrophic because hardware component failure may affect the operation of multiple services within an organization.

Typically, you can mitigate the impact of most hardware failures by designing appropriate levels of redundancy into a system. For example, if you're deploying an application in a public cloud, such as AWS, you'd typically spread replicas of a service across multiple zones—geographically distinct data centers within a wider region—to reduce the impact of failure within a single center.

It's important to note that hardware redundancy can incur additional operational cost. Some solutions may be complex to architect and run—or just plain expensive. Choosing the right level of redundancy for an application requires careful consideration of the frequency and impact of failure versus the cost of mitigating against potentially rare events.

Communication

Communication between services can fail: network, DNS, messaging, and firewalls are all possible sources of failure. Table 6.2 details possible communication failures.

Table 6.2 Sources of communication failure within a microservice application

Source of failure	Description
Network	Network connectivity may not be possible.
Firewall	Configuration management can set security rules inappropriately.
DNS errors	Hostnames may not be correctly propagated or resolved across an application.
Messaging	Messaging systems—for example, RPC—can fail.
Inadequate health checks	Health checks may not adequately represent instance state, causing requests to be routed to broken instances.

Communication failures can affect both internal and external network calls. For example, connectivity between the market-data service and the external APIs it relies on could degrade, leading to failure.

Network and DNS failures are reasonably common, whether caused by changes in firewall rules, IP address assignment, or DNS hostname propagation in a system. Network issues can be challenging to mitigate, but because they're often caused by human intervention (whether through service releases or configuration changes), the best way to avoid many of them is to ensure that you test configuration changes robustly, and that they're easy to roll back if issues occur.

Dependencies

Failure can occur in other services that a microservice depends on, or within that microservice's internal dependencies (such as databases). For example, the database that market-data relies on to save and retrieve data might fail because of underlying hardware failure or hitting scalability limits—it wouldn't be unheard of for a database to run out of disk space!

As we outlined earlier, such failures have a drastic effect on overall system availability. Table 6.3 outlines possible sources of failure.

Table 6.3 Sources of dependency-related failure

Source of failure	Description
Timeouts	Requests to services may time out, resulting in erroneous behavior.
Decommissioned or nonbackwards-compatible functionality	Design doesn't take service dependencies into account, unexpectedly changing or removing functionality.
Internal component failures	Problems with databases or caches prevent services from working correctly.
External dependencies	Services may have dependencies outside of the application that don't work correctly or as expected—for example, third-party APIs.

In addition to operational sources of failure, such as timeouts and service outages, dependencies are prone to errors caused by design and build failures. For example, a service may rely on an endpoint in another service that's changed in a nonbackwards-compatible way or, even worse, removed completely without appropriate decommissioning.

SERVICE PRACTICES

Lastly, inadequate or limited engineering practices when developing and deploying services may lead to failure in production. Services may be poorly designed, inadequately tested, or deployed incorrectly. You may not catch errors in testing, or a team may not adequately monitor the behavior of their service in production. A service might scale ineffectively: hitting memory, disk, or CPU limits on its provisioned hardware such that performance is degraded—or the service becomes completely unresponsive.

Because each service contributes to the effectiveness of the whole system, one poor quality service can have a detrimental effect on the availability of swathes of functionality. Hopefully the practices we outline throughout this book will help you avoid this—unfortunately common—source of failure!

6.2.2 Cascading failures

You should now understand how failure in different areas can affect a single microservice. But the impact of failure doesn't stop there. Because your applications are composed of multiple microservices that continually interact with each other, failure in one service can spread across an entire system.

Cascading failures are a common mode of failure in distributed applications. A cascading failure is an example of *positive feedback*: an event disturbs a system, leading to some effect, which in turn increases the magnitude of the initial disturbance. In this case, positive means that the effect increases—not that the outcome is beneficial.

You can observe this phenomenon in several real-world domains, such as financial markets, biological processes, or nuclear power stations. Consider a stampede in a herd of animals: panic causes an animal to run, which in turn spreads panic to other animals, which causes them to run, and so on. In a microservice application, overload can cause a domino effect: failure in one service increases failure in upstream services, and in turn their upstream services. At worst, the result is widespread unavailability.

Let's work through an example to illustrate how overload can result in a cascading failure. Imagine that SimpleBank built a UI to show a user their current financial holdings (or positions) in an account. That might look something like figure 6.4.

Each financial position is the sum of the transactions—purchases and sales of a stock—made to date, multiplied by the current price. Retrieving these values relies on collaboration between three services:

- *Market-data*—A service responsible for retrieving and processing price and market information for financial instruments, such as stocks
- *Transactions*—A service responsible for representing transactions occurring within an account
- *Holdings*—A service responsible for aggregating transactions and market-data to report financial positions

Holdings *as at 2017-07-23*

	Quantity	Value
BHP Billiton Ltd BHP	1000	$91,720
Google GOOGL	103	$14,023
ABC Company ABC	24	$1.20

Figure 6.4 A user interface that reports financial holdings in an account

Figure 6.5 outlines the production configuration of these services. For each service, load is balanced across multiple replicas.

Suppose that holdings are being retrieved 1,000 times per second (QPS). If you have two replicas of your holdings service, each replica will receive 500 QPS (figure 6.6).

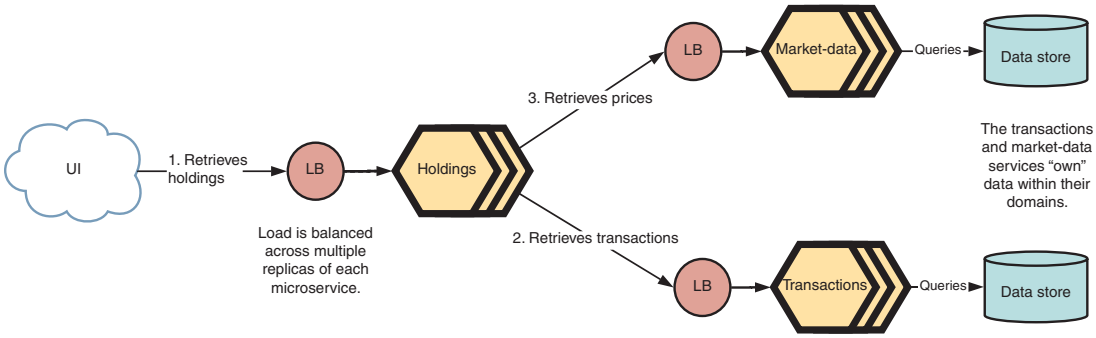


Figure 6.5 Production configuration and collaboration between services to populate the "current financial holdings" user interface

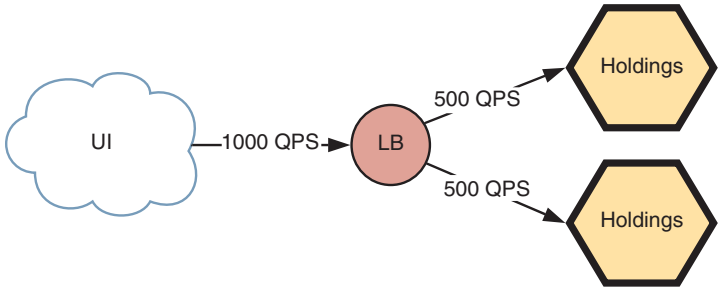


Figure 6.6 Queries made to a service are split across multiple replicas.

Your holdings service subsequently queries transactions and market-data to construct the response. Each call to holdings will generate two calls: one to transactions and one to market-data.

Now, let's say a failure occurs that takes down one of your transactions replicas. Your load balancer reroutes that load to the remaining replica, which now needs to service 1,000 QPS (figure 6.7).

But that reduced capacity is unable to handle the level of demand to your service. Depending on how you've deployed your service—the characteristics of your web server—the change in load might first lead to increased latency as requests are queued. In turn, increased latency might start exceeding the maximum wait time that the holdings service expects for that query. Alternatively, the transactions service may begin dropping requests.

It's not unreasonable for a service to retry a request to a collaborator when it fails. Now, imagine that the holdings service will retry any request to transactions that times out or fails. This will further increase the load on your remaining transactions resource, which now needs to handle both the regular request volume and the heightened retry volume (figure 6.8). In turn, the holdings service takes longer to respond while it waits on its collaborator.

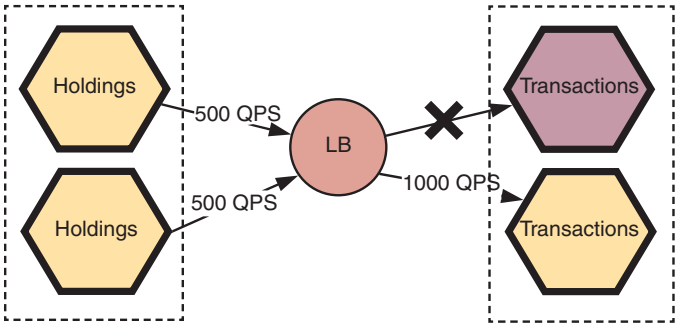


Figure 6.7 One replica of a collaborating service fails, sending all load to the remaining instance.

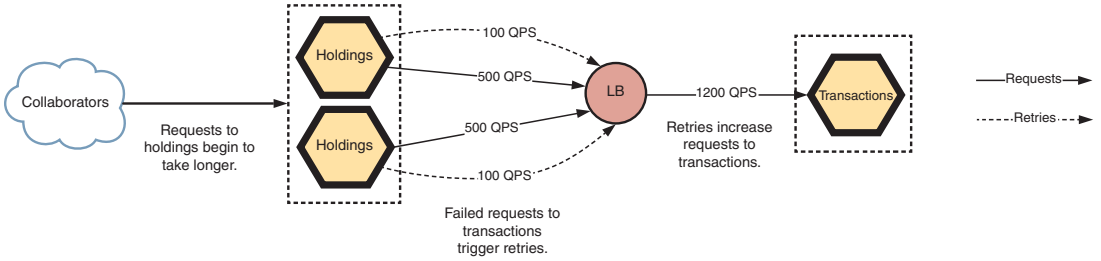


Figure 6.8 Overload on transactions causes some requests to fail, in turn causing holdings to retry those requests, which starts to degrade holdings' response time.

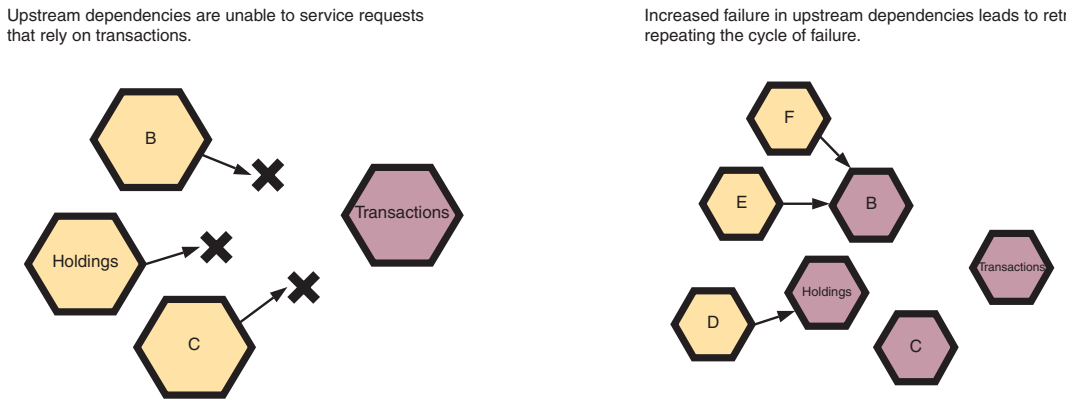


Figure 6.9 Overload in a service leads to complete failure. Unhealthy retry behavior is repeated across dependency chains as service performance progressively degrades, leading to further overloads.

This feedback loop—failed requests lead to a higher volume of subsequent requests, leading to a higher rate of failure—continues to escalate. Your whole system is unable to complete work, as other services that rely on transactions or holdings begin to fail. Your initial failure in a single service causes a domino effect, worsening response times and availability across several services. At worst, the cumulative impact on the transactions service causes it to fail completely. Figure 6.9 illustrates this final stage of a cascading failure.

Cascading failures aren't only caused by overload—although this is one of the most common root causes. In general, increased error rates or slower response times can lead to unhealthy service behavior, increasing the chance of failure across multiple services that depend on each other.

You can use several approaches to limit the occurrence of cascading failures in microservice applications: circuit breakers; fallbacks; load testing and capacity planning; back-off and retries; and appropriate timeouts. We'll explore these approaches in the next section.

6.3 Designing reliable communication

Earlier, we emphasized the importance of collaboration in a microservice application. Dependency chains of multiple microservices will achieve most useful capabilities in an application. When one microservice fails, how does that impact its collaborators and ultimately, the application's end customers?

If failure is inevitable, you need to design and build your services to maximize availability, correct operation, and rapid recovery when failure does occur. This is fundamental to achieving resiliency. In this section, we'll explore several techniques for ensuring that services behave appropriately—maximizing correct operation—when their collaborators are unavailable:

- Retries
- Fallbacks, caching, and graceful degradation
- Timeouts and deadlines

- Circuit breakers
- Communication brokers

Before we start, let's get a simple service running that we can use to illustrate the concepts in this section. You can find these examples in the book's Github repository (http://mng.bz/7eN9). Clone the repository to your computer and open the chapter-6 directory. This directory contains some basic services—holdings and market-data—which you'll run inside Docker containers (figure 6.10). The holdings service exposes a `GET /holdings` endpoint, which makes a JSON API request to retrieve price information from market-data.

To run these, you'll need `docker-compose` installed (directions online: https://docs.docker.com/compose/install/). If you're ready to go, type the following at the command line:

```
$ docker-compose up
```

This will build Docker images for each service and start them as isolated containers on your machine. Now let's dive in!

6.3.1 Retries

In this section, we'll explore how to use retries when failed requests occur. To understand these techniques, let's start by examining communication from the perspective of your upstream service, holdings.

Imagine that a call from the holdings service to retrieve prices fails, returning an error. From the perspective of the calling service, it's not clear yet whether this failure is isolated—repeating that call is likely to succeed, or systemic—the next call has a high likelihood of failing. You expect calls to retrieve data to be *idempotent*—to have no effect on the state of the target system and therefore be repeatable.[2]

As a result, your first instinct might be to retry the request. In Python, you can use an open source library—`tenacity`—to decorate the appropriate method of your API client (the `MarketDataClient` class in holdings/clients.py) and perform retries if the method throws an exception. The following listing shows the class with retry code added.

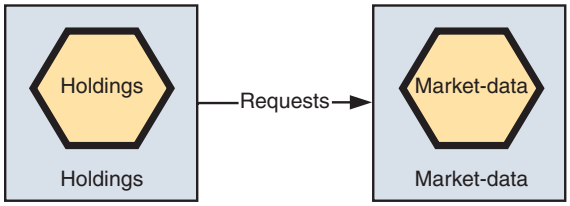


Figure 6.10 Docker containers for working with microservice requests

[2] Requests that effect some system change aren't typically idempotent. One strategy for guaranteeing "exactly once" semantics is to implement idempotency keys. See Brandur Leach, "Designing robust and predictable APIs with idempotency," February 22, 2017, https://stripe.com/blog/idempotency.

Listing 6.1 Adding a retry to a service call

```
import requests
import logging
from tenacity import retry, stop, before

class MarketDataClient(object):

    logger = logging.getLogger(__name__)
    base_url = 'http://market-data:8000'

    def _make_request(self, url):
        response = requests.get(f"{self.base_url}/{url}",
                                headers={'content-type': 'application/json'})
        return response.json()

    @retry(stop=stop_after_attempt(3),
           before=before_log(logger, logging.DEBUG))
    def all_prices(self):
        return self._make_request("prices")
```

Imports relevant functions from the library

Retries the query up to three times

Logs each retry before execution

Let's call the holdings service to see how it behaves. In another terminal window, make the following request:

```
curl -I http://{DOCKER_HOST}/holdings
```

This will return a 500 error, but if you follow the logs from the market-data service, you can see a request being made to `GET /prices` three times, before the holdings service gives up.

If you read the previous section, you should be wary at this point. Failure might be isolated or persistent, but the holdings service can't know which one is the case based on one call.

If the failure is isolated and transient, then a retry is a reasonable option. This helps to minimize direct impact to end users—and explicit intervention from operational staff—when abnormal behavior occurs. It's important to consider your budget for retries: if each retry takes a certain number of milliseconds, then the consuming service can only absorb so many retries before it surpasses a reasonable response time.

But if the failure is persistent—for example, if the capacity of market-data is reduced—then subsequent calls may worsen the issue and further destabilize the system. Suppose you retry each failed request to market-data five times. Every failed request you make to this service potentially results in another five requests; the volume of retries continues to grow. The entire service is doing less useful work as it attempts to service a high volume of retries. At worst, retries suffocate your market-data service, magnifying your original failure. Figure 6.11 illustrates this growth of requests.

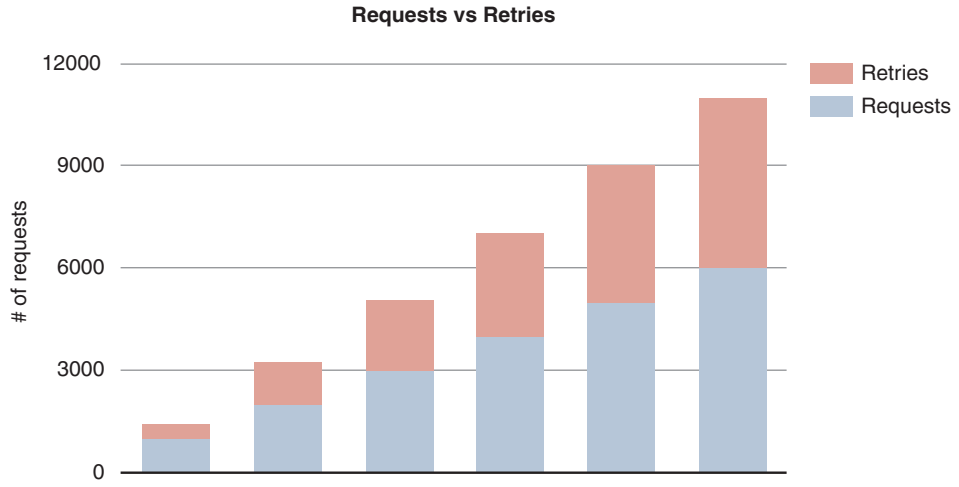


Figure 6.11 Growth of load on your unstable market-data service resulting from failed requests being retried

How can you use retries to improve your resiliency in the face of intermittent failures without contributing to wider system failure if persistent failures occur? First, you could use a variable time between successive retries to try to spread them out evenly and reduce the frequency of retry-based load. This is known as an *exponential back-off* strategy and is intended to give a system under load time to recover. You can change the retry strategy you used earlier, as shown in the following listing. Afterwards, by curling the `/holdings` endpoint, you can observe the retry behavior of the service.

Listing 6.2 Changing your retry strategy to exponential back-off

```
@retry(wait=wait_exponential(multiplier=1, max=5),
       stop=stop_after_delay(5))
def all_prices(self):
    return self._make_request("prices")
```

Waits 2 ^ x * 1 second between each retry

Stops after five seconds

Unfortunately, exponential back-off can lead to another instance of curious emergent behavior. Imagine that a momentary failure interrupts several calls to market-data, leading to retries. Exponential back-off can cause the service to schedule those retries together so they further amplify themselves, like the ripples from throwing a stone in a pond.

Instead, back-off should include a random element—jitter—to spread out retries to a more constant rate and avoid thundering herds of synchronized retries.[3] The following listing shows how to adjust your strategy again.

[3] A great article by Marc Brooker about exponential back-off and the importance of jitter is available on the AWS Architecture Blog, March 4, 2015, http://mng.bz/TRk5.

Listing 6.3 Adding jitter to an exponential back-off

```
@retry(wait=wait_exponential(multiplier=1, max=5) + wait_random(0, 1),
       stop=stop_after_delay(5))
def all_prices(self):
    return self._make_request("prices")
```

Stops after five seconds

Exponentially backs off, adding a random wait between zero and one second

This strategy will ensure that retries are less likely to happen in synchronization across multiple waiting clients.

Retries are an effective strategy for tolerating intermittent dependency faults, but you need to use them carefully to avoid exacerbating the underlying issue or consuming unnecessary resources:

- Always limit the total number of retries.
- Use exponential back-off with jitter to smoothly distribute retry requests and avoid compounding load.
- Consider which error conditions should trigger a retry and, therefore, which retries are unlikely to, or will never, succeed.

When your service meets retry limits or can't retry a request, you can either accept failure or find an alternative way to serve the request. In the next section, we'll explore fallbacks.

6.3.2 Fallbacks

If a service's dependencies fail, you can explore four fallback options:

- Graceful degradation
- Caching
- Functional redundancy
- Stubbed data

Graceful Degradation

Let's return to the problem with the holdings service: if market-data fails, the application may not be able to provide valuations to end customers. To resolve this issue, you might be able to design an acceptable degradation of service. For example, you could show holding quantities without valuations. This limits the richness of your UI but is better than showing nothing—or an error. You can see techniques like this in other domains. For example, an e-commerce site could still allow purchases to be made, even if the site's order dispatch isn't functioning correctly.

Caching

Alternatively, you could cache the results of past queries for prices, reducing the need to query the market-data service at all. Say a price is valid for five minutes. If so, the holdings service could cache pricing data for up to five minutes, either locally or in

a dedicated cache (for example, Memcached or Redis). This solution would both improve performance and provide contingency in the event of a temporary outage.

Let's try out this technique. You'll use a library called `cachetools`, which provides an implementation of a time-to-live cache. As you did earlier with retries, you'll decorate your client method, as shown in the following listing.

Listing 6.4 Adding in-process caching to a client call

```
import requests
import logging
from cachetools import cached, TTLCache

class MarketDataClient(object):

    logger = logging.getLogger(__name__)
    cache = TTLCache(maxsize=10, ttl=5*60)      <-- Instantiates a cache
    base_url = 'http://market-data:8000'

    def _make_request(self, url):
        response = requests.get(f"{self.base_url}/{url}",
                                headers={'content-type': 'application/json'})
        return response.json()

    @cached(cache)      <-- Decorates your method to store results using your cache
    def all_prices(self):
        logger.debug("Making request to get all_prices")
        return self._make_request("prices")
```

Subsequent calls made to `GET /holdings` should retrieve price information from the cache, rather than by making calls to market-data. If you used an external cache instead, multiple instances could use the cache, further reducing load on market-data and providing greater resiliency for all holdings replicas, albeit at the cost of maintaining an additional infrastructural component.

FUNCTIONAL REDUNDANCY

Similarly, you might be able to fall back to other services to achieve the same functionality. Imagine that you could purchase market data from multiple sources, each covering a different set of instruments at a different cost. If source A failed, you could instead make requests to source B (figure 6.12).

Functional redundancy within a system has many drivers: external integrations; algorithms for producing similar results with varying performance characteristics; and even older features that remain operational but have been superseded. In a globally distributed deployment, you could even fall back on services hosted in another region.[4]

Only some failure scenarios would allow the use of an alternative service. If the cause of failure was a code defect or resource overload in your original service, then rerouting to another service would make sense. But a general network failure could affect multiple services, including ones you might try rerouting to.

[4] At the ultimate end of this scale, Netflix can serve a given customer from any of their global data centers, conveying an impressive degree of resilience.

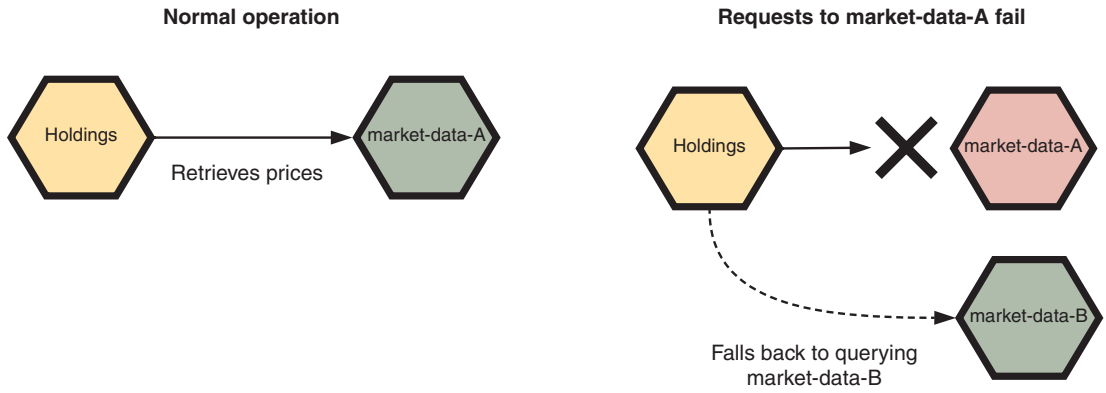


Figure 6.12 If service failure occurs, you may be able to serve the same capability with other services.

STUBBED DATA

Lastly, although it wouldn't be appropriate in this specific scenario, you could use stubbed data for fallbacks. Picture the "recommended to you" section on Amazon: if the backend was unable for some reason to retrieve those personalized recommendations, it'd be more graceful to fall back to a nonpersonalized data set than to show a blank section on the UI.

6.3.3 Timeouts

When the holdings service sends a request to market-data, that service consumes resources waiting for a reply. Setting an appropriate deadline for that interaction limits the time those resources are consumed.

You can set a timeout within your HTTP request function. For HTTP calls, you want to timeout if you haven't received any response, but not if the response itself is slow to download. Try the following listing to add a timeout.

Listing 6.5 Adding a timeout to an HTTP call

```
def _make_request(self, url):
    response = requests.get(f"{self.base_url}/{url}",
                            headers={'content-type': 'application/json'},
                            timeout=5)
    return response.json()
```

Sets a timeout of five seconds before receiving data from market-data

In a computational sense, network communication is slow, so the speed of failures is important. In a distributed system, some errors might happen almost instantly. For example, a dependent service may rapidly fail in the event of an internal bug. But many failures are slow. For example, a service that's overloaded by requests may respond sluggishly, in turn consuming the resources of the calling service while it waits for a response that may never come.

Slow failures illustrate the importance of setting appropriate deadlines—timing out in a reasonable timeframe—for communication between microservices. If you don't set upper bounds, it's easy for unresponsiveness to spread through entire microservice dependency chains. In fact, lack of deadlines can extend the impact of issues because a server consumes resources while it waits forever for an issue to be resolved.

Picking a deadline can be difficult. If they're too long, they can consume unnecessary resources for a calling service if a service is unresponsive. If they're too short, they can cause higher levels of failure for expensive requests. Figure 6.13 illustrates these constraints.

For many microservice applications, you set deadlines at the level of individual interactions; for example, a call from holdings to market-data may always have a deadline of 10 seconds. A more elegant approach is to apply an absolute deadline across an entire operation and propagate the remaining time across collaborators.

Without propagating deadlines, it can be difficult to make them consistent across a request. For example, holdings could waste resources waiting for market-data far beyond the overall deadline imposed by a higher level of the stack, such as an API gateway.

Imagine a chain of dependencies between multiple services. Each service takes a certain amount of time to do its work and expects its collaborators to take some time. If any of those times vary, static expectations may no longer be correct (figure 6.14).

If your service interactions are over HTTP, you could propagate deadlines using a custom HTTP header, such as `X-Deadline: 1000`, passing that value to set read timeout values on subsequent HTTP client calls. Many RPC frameworks, such as gRPC, explicitly implement mechanisms for propagating deadlines within a request context.

6.3.4 Circuit breakers

You can combine some of the techniques we've discussed so far. You can consider an interaction between holdings and market-data as analogous to an electrical circuit. In electrical wiring, circuit breakers perform a protective role—preventing spikes in current from damaging a wider system. Similarly, a circuit breaker is a pattern for pausing requests made to a failing service to prevent cascading failures.

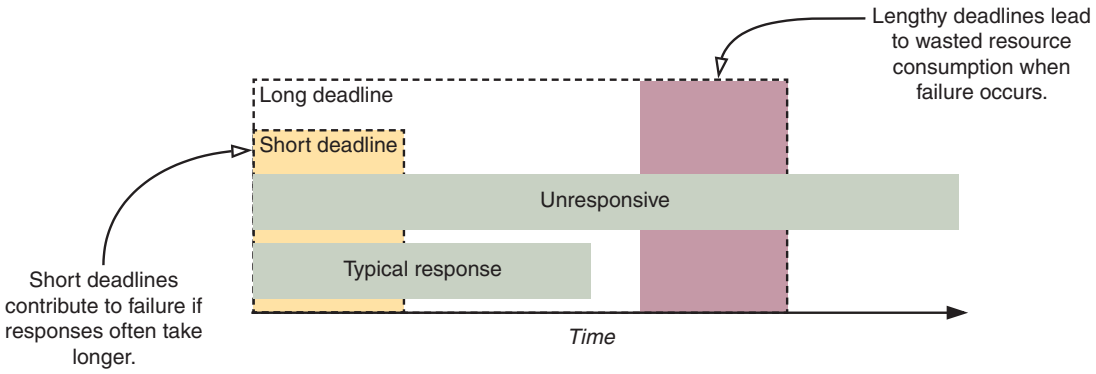


Figure 6.13 Choosing the right deadline requires balancing time constraints to maximize the window of successful requests.

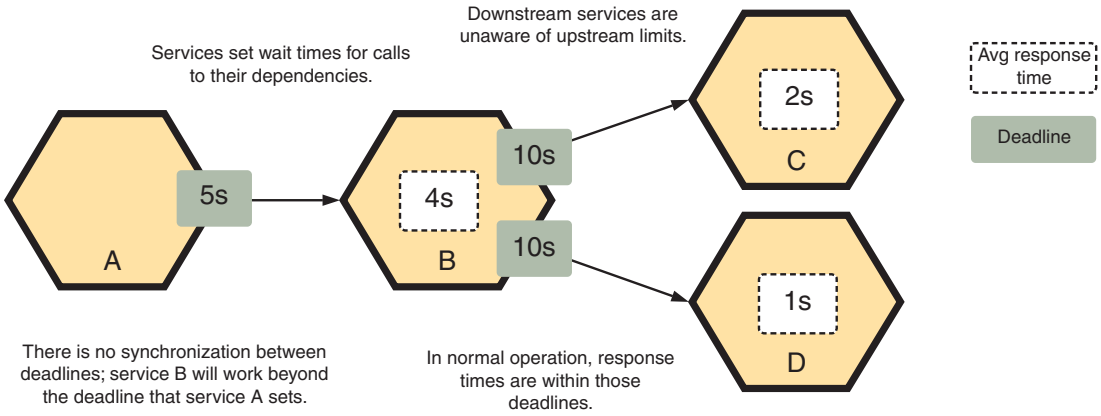


Figure 6.14 Services may set expectations about how long they expect calls to collaborators to take; varying widely because of failure or latency can exacerbate the impact of those failures.

How does it work? Two principles, both of which we touched on in the previous section, inform the design of a circuit breaker:

1. Remote communication should fail quickly in the event of an issue, rather than wasting resources waiting for a response that might never come.
2. If a dependency is failing consistently, it's better to stop making further requests until that dependency recovers.

When making a request to a service, you can track the number of times that request succeeds or fails. You might track this number within each running instance of a service or share that state (using an external cache) across multiple services. In this normal operation, we consider the circuit to be closed.

If the number of failures seen or the rate of failures within a certain time window passes a threshold, then the circuit is opened. Rather than attempting to send requests to your collaborating service, you should short-circuit requests and, where possible, perform an appropriate fallback—returning a stubbed message, routing to a different service, or returning a cached response. Figure 6.15 illustrates the lifecycle of a request using a circuit breaker.

Setting the time window and threshold requires careful consideration of both the expected reliability of the target service and the volume of interactions between services. If requests are relatively sparse, then a circuit breaker may not be effective, because a large time window might be required to obtain a representative sample of requests. For service interactions with contrasting busy and quiet periods, you may want to introduce a minimum throughput to ensure a circuit only reacts when load is statistically significant.

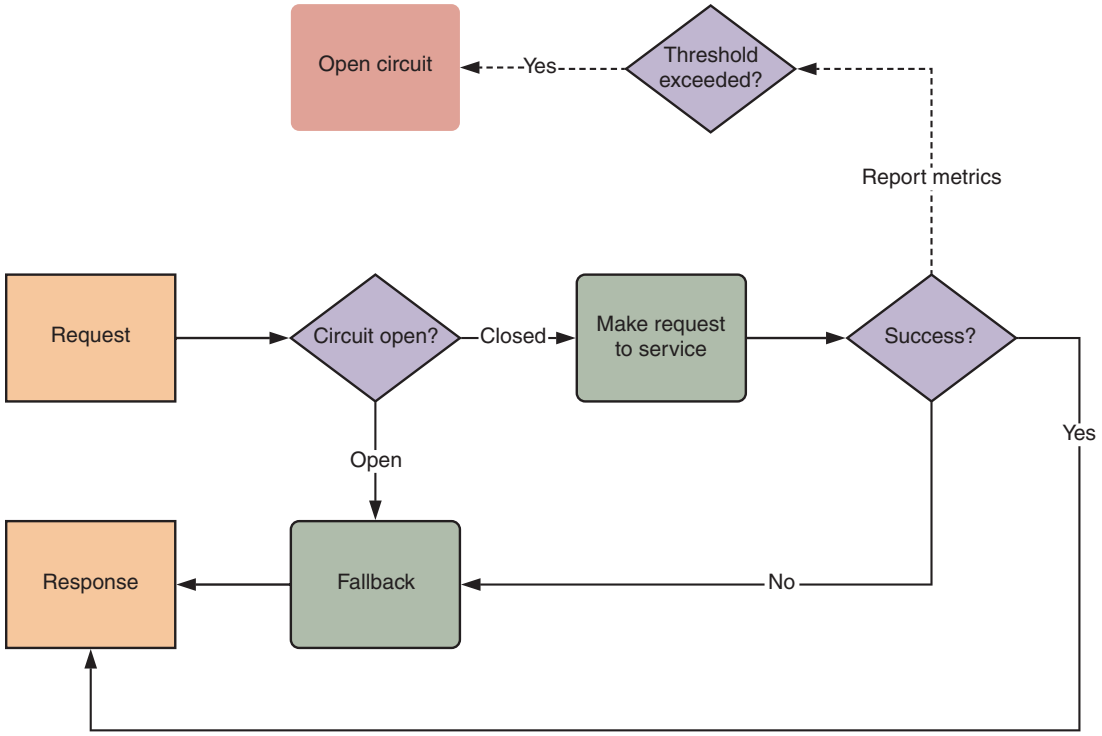


Figure 6.15 A circuit breaker controls the flow of requests between two services and opens when the number of failed requests surpasses a threshold.

> **NOTE** You should monitor when circuits are opened and closed, as well as potentially alerting the team responsible, especially if the circuit is frequently opened. We'll discuss this further in part 4.

Once the circuit has opened, you probably don't want to leave it that way. When availability returns to normal, the circuit should be closed. The circuit breaker needs to send a trial request to determine whether the connection has returned to a healthy state. In this trial state, the circuit breaker is *half open*: if the call succeeds, the circuit will be closed; otherwise, it will remain open. As with other retries, you should schedule these attempts with an exponential back-off with jitter. Figure 6.16 shows the three distinct states of a circuit breaker.

Several libraries are available that provide implementations of the circuit breaker pattern in different languages, such as Hystrix (Java), CB2 (Ruby), or Polly (.NET).

> **TIP** Don't forget that closed is the good state for a circuit breaker! The use of open and closed to represent, respectively, negative and positive states may seem counterintuitive but reflects the real-world behavior of an electrical circuit.

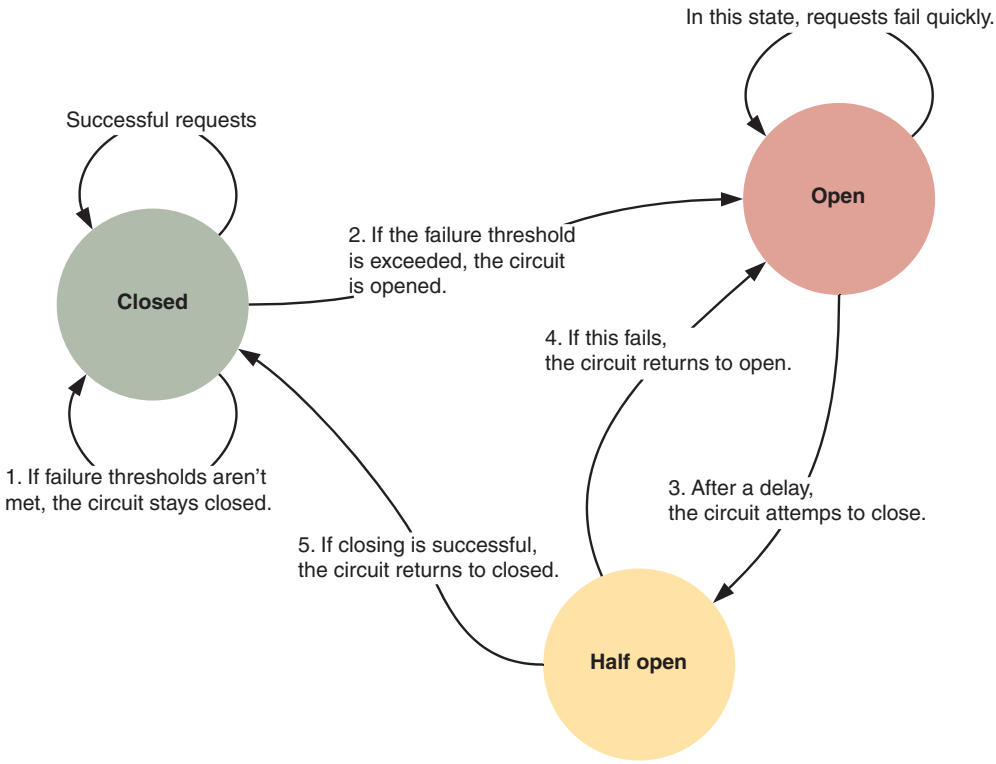


Figure 6.16 A circuit breaker transitions between three stages: open, closed, and half open.

6.3.5 Asynchronous communication

So far, we've focused on failure in synchronous, point-to-point communication between services. As we outlined in the first section, the more services in a chain, the lower overall availability you can guarantee for that path.

Designing asynchronous service interactions, using a communication broker like a message queue, is another technique you can use to maximize reliability. Figure 6.17 illustrates this approach.

Where you don't need immediate, consistent responses, you can use this technique to reduce the number of direct service interactions, in turn increasing overall availability—albeit at the expense of making business logic more complex. As we mentioned elsewhere in this book, a communication broker becomes a single point of failure that will require careful attention for you to scale, monitor, and operate effectively.

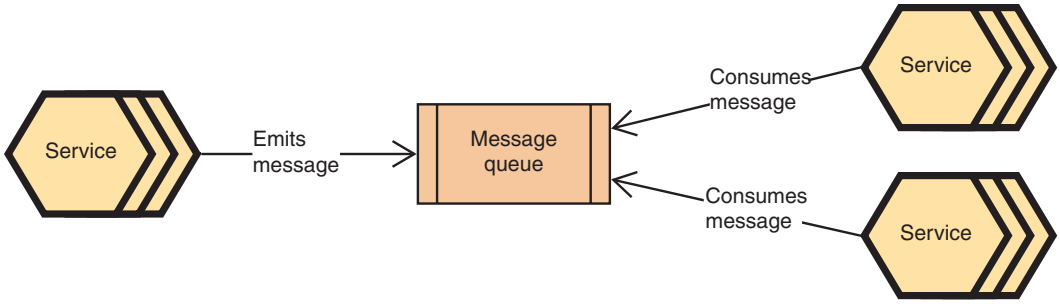


Figure 6.17 Using a message queue to decouple services from direct interaction

6.4 Maximizing service reliability

In the previous sections, we explored techniques to ensure a service can tolerate faults in interactions with its collaborators. Now, let's consider how you can maximize availability and fault tolerance within an individual service. In this section, we'll explore two techniques—health checks and rate limits—as well as methods for validating the resilience of services.

6.4.1 Load balancing and service health

In production, you deploy multiple instances of your market-data service to ensure redundancy and horizontal scalability. A load balancer will distribute requests from other services between these instances. In this scenario, the load balancer plays two roles:

1. Identifying which underlying instances are healthy and able to serve requests
2. Routing requests to different underlying instances of the service

A load balancer is responsible for executing or consuming the results of *health checks*. In the previous section, you could ascertain the health of a dependency at the point of interaction—when requests were being made. But that's not entirely adequate. You should have some way of understanding the application's readiness to serve requests at any time, rather than when it's actively being queried.

Every service you design and deploy should implement an appropriate health check. If a service instance becomes unhealthy, that instance should no longer receive traffic from other services. For synchronous RPC-facing services, a load balancer will typically query each instance's health check endpoint on an interval basis. Similarly, asynchronous services may use a heartbeat mechanism to test connectivity between the queue and consumers.

> **TIP** It's often desirable for repeated or systematic instance failures, as detected by health checks, to trigger alerts to an operations team—a little human intervention can be helpful. We'll explore that further in part 4 of this book.

You can classify health checks based on two criteria: liveness and readiness. A liveness check is typically a simple check that the application has started and is running correctly. For example, an HTTP service should expose an endpoint—commonly `/health`, `/ping`, or `/heartbeat`—that returns a 200 OK response once the service is running (figure 6.18). If an instance is unresponsive, or returns an error message, the load balancer will no longer deliver requests there.

In contrast, a readiness check indicates whether a service is ready to serve traffic, because being alive may still not indicate that requests will be successful. A service might have many dependencies—databases, third-party services, configuration, caches—so you can use a readiness check to see if these constituent components are in the correct state to serve requests. Both of the example services implement a simple HTTP liveness check, as shown in the following listing.

Listing 6.6 Flask handler for an HTTP liveness check

```
@app.route('/ping', methods=["GET"])
def ping():
    return 'OK'
```

Health checks are binary: either an instance is available or it isn't. This works well with typical round-robin load balancing, where requests are distributed to each replica in turn. But in some circumstances the functioning of a service may be degraded and exhibit increased latency or error rates without a health check reflecting this status. As such, it can be beneficial to use load balancers that are aware of latency and able to route requests to instances that are performing better, or those that are under less load, to achieve more consistent service behavior. This is a typical feature of a microservice proxy, which we'll touch on later in this chapter.

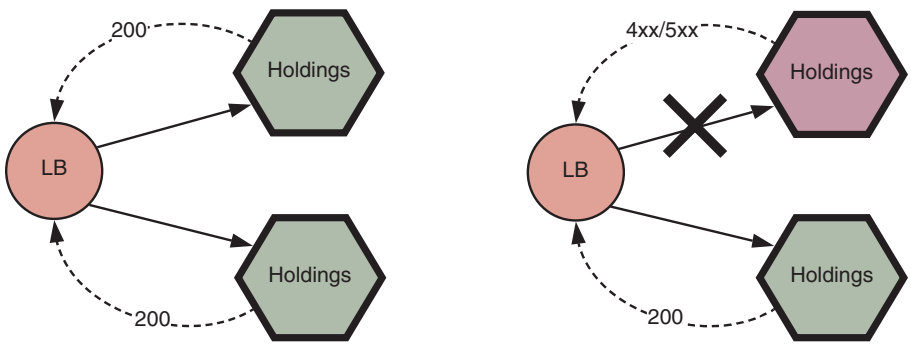


Figure 6.18 Load balancers continuously query service instances to check their health. If an instance is unhealthy, the load balancer will no longer route requests to that instance until it recovers.

6.4.2 Rate limits

Unhealthy service usage patterns can sometimes arise in large microservice applications. Upstream dependencies might make several calls, where a single batch call would be more appropriate, or available resources may not be distributed equitably among all callers. Similarly, a service with third-party dependencies could be limited by restrictions that those dependencies impose.

An appropriate solution is to explicitly limit the rate of requests or total requests available to collaborating services in a timeframe. This helps to ensure that a service—particularly when it has many collaborators—isn't overloaded. The limiting might be indiscriminate (drop all requests above a certain volume) or more sophisticated (drop requests from infrequent service clients, prioritize requests for critical endpoints, and drop low-priority requests). Table 6.4 outlines different rate-limiting strategies.

Table 6.4 Common rate-limiting strategies

Strategy	Description
Drop requests above volume	Drop consumer requests above a specified volume
Prioritize critical traffic	Drop requests to low-priority endpoints to prioritize resources for critical traffic
Drop uncommon clients	Prioritize frequent consumers of the service over infrequent users
Limit concurrent requests	Limit the overall number of requests an upstream service can make over a time period

Rate limits can be shared with a service's clients at design time or, better, at runtime. A service might return a header to a consumer that indicates the remaining volume of requests available. On receipt, the upstream collaborator should take this into account and adjust its rate of outbound requests. This technique is known as back pressure.

6.4.3 Validating reliability and fault tolerance

Applying the tactics and patterns we've covered will put you on a good path toward maximizing availability. But it's not enough to plan and design for resiliency: you need to validate that your services can tolerate faults and recover gracefully.

Thorough testing provides assurance that your chosen design is effective when both predicted and unpredictable failures occur. Testing requires the application of *load testing* and *chaos testing*. Although it's likely you're familiar with code testing—such as unit and acceptance testing to validate implementation, usually in a controlled environment—you might not know that load and chaos testing are intended to validate service limits by closely replicating the turbulence of production operation. Although testing isn't the primary focus of this book, it's useful to understand how these different testing techniques can help you build a robust microservice application.

LOAD TESTING

As a service developer, you can usually be confident that the number of requests made to your service will increase over time. When developing a service, you should

1. Model the expected growth and shape of service traffic to ensure that you understand the likely usage of your service
2. Estimate the capacity required to service that traffic
3. Validate the deployed capacity of the service by load testing against those limits
4. Use business and service metrics as appropriate to re-estimate capacity

Imagine you're considering how much capacity the market-data service requires. First, what do you know about the service's usage patterns? You know that holdings queries the service, but it may be called from elsewhere too—pricing data is used throughout SimpleBank's product.

Let's assume that queries to market-data grow roughly in line with the number of active users on the platform, but you may experience spikes (for example, when the market opens in the morning). You can plan capacity based on predictions of your business growth. Table 6.5 outlines a simple estimation of the QPS that you can expect this service to receive over a three-month period.

Table 6.5 Estimate of calls to a service per second based on growth in average active users over a three-month period

			Jun	*Jul*	*Aug*
Total Users			4000	5600	7840
Expected Growth			40%	40%	40%
Active Users	*Average*	20%	800	1120	1568
	Peak	70%	2800	3920	5488
Service Calls	*Average*				
	Per User/Minute	30	24000	33600	47040
	Per User/Second	0.5	400	560	784
	Peak				
	Per User/Minute	30	84000	117600	164640
	Per User/Second	0.5	1400	1960	2744

Identifying the qualitative factors that drive growth in service utilization is vital to good design and optimizing capacity. Once you've done that, you can determine how much

capacity to deploy. For example, the table suggests you need to be able to service 400 requests per second in normal operation, growing by 40% month on month, with spikes in peak usage to 1,400 requests per second.

> **TIP** An in-depth review of capacity and scale planning techniques is outside the scope of this book, but a great overview is available in Abbott and Fisher's *The Art of Scalability* (Addison-Wesley Professional, 2015) (ISBN 978-0134032801).

Once you've established a baseline capacity for your service, you can then iteratively test that capacity against expected traffic patterns. Along with validating the traffic limits of a microservice configuration, load testing can identify potential bottlenecks or design flaws that aren't apparent at lower levels of load. Load testing can provide you with highly effective insight into the limitations of your services.

At the level of individual services, you should automate the load testing of each service as part of its delivery pipeline—something we'll explore in part 3 of this book. Along with this systematic load testing, you should perform exploratory load testing to identify limits and test your assumptions about the load that services can handle.

You also should load test services together. This can aid in identifying unusual load patterns and bottlenecks based on service interaction. For example, you could write a load test that exercises all the services in the `GET /holdings` example.

CHAOS TESTING

Many failures in a microservice application don't arise from within the microservices themselves. Networks fail, virtual machines fail, databases become unresponsive—failure is everywhere! To test for these types of failure scenarios, you need to apply chaos testing.

Chaos testing pushes your microservice application to fail in production. By introducing instability and failure, it accurately mimics real system failures, as well as training an engineering team to be able to react to those failures. This should ultimately build your confidence in the system's capability to withstand real chaos because you'll be gradually improving the resiliency of your system and reducing the possible number of events that would cause operational impact.

As explained on the "Principles of Chaos Engineering" website (https://principlesofchaos.org/), you can think of chaos testing as "the facilitation of experiments to uncover systemic weaknesses." The website lays out this approach:

1. Define a measurable steady state of normal system operation.
2. Hypothesize that behavior in an experimental and control group will remain steady; the system will be resilient to the failure introduced.
3. Introduce variables that reflect real-world failure events—for example, removing servers, severing network connections, or introducing higher levels of latency.
4. Attempt to disprove the hypothesis you defined in (2).

Recall how the holdings, transactions, and market-data services were deployed in figure 6.5. In this case, you expect steady operation to return holdings data within a reasonable response time. A chaos test could introduce several variables:

1. Killing nodes running market-data or transactions, either partially or completely
2. Reducing capacity by killing holdings instances at random
3. Severing the network connection—for example, between holdings and downstream services or between services and their data stores

Figure 6.19 illustrates these options.

Companies with mature chaos testing practices might even perform testing on both a systematic and random basis against live production environments. This might sound terrifying; real outages can be stressful enough, let alone actively working to make them happen. But without taking this approach, it's incredibly difficult to know that your system is truly resilient in the ways that you expect. In any organization, you should start small, by introducing a limited set of possible failures, or only running scheduled, rather than random, tests. Although you can also perform chaos tests in a staging environment, you'll need to carefully consider whether that environment is truly representative of or equivalent to your production configuration.

TIP Chaos Toolkit (http://chaostoolkit.org/) is a great tool to start with if you'd like to practice chaos engineering techniques.

Ultimately, by regularly and systematically validating your system against chaotic events and resolving the issues you encounter, you and your team will be able to achieve a significant level of confidence in your application's resilience to failure.

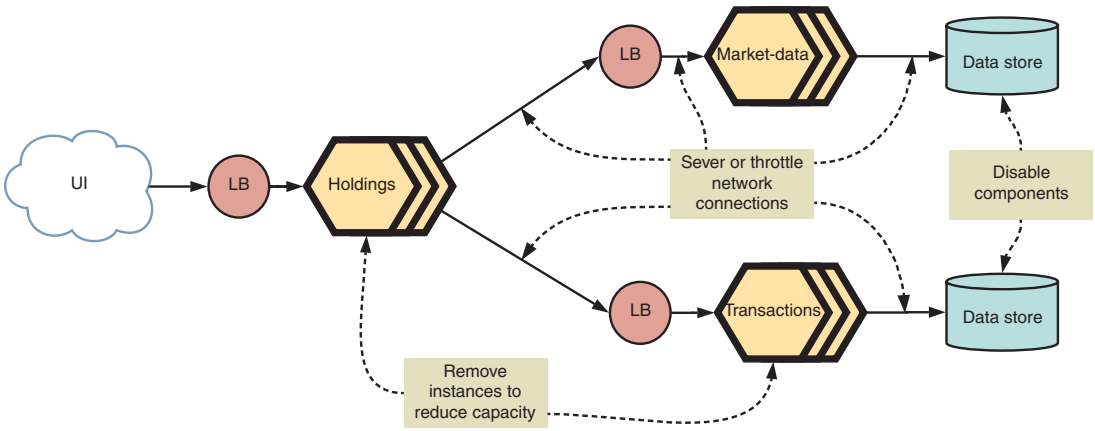


Figure 6.19 Potential variables to introduce in a chaos test to reflect real-world failure events

6.5 Safety by default

Critical paths in your microservice application will only be as resilient and available as their weakest link. Given the impact that individual services can have overall availability, it's imperative to avoid emergencies where introducing new services or changes in a service dependency chain significantly degrade that measure. Likewise, you don't want to find out that crucial functionality can't tolerate faults *when that fault happens*.

When applications are technically heterogeneous, or distinct teams deliver underlying services, it can be exceptionally difficult to maintain consistent approaches to reliable interaction. We touched on this back in chapter 2 when we discussed isolation and technical divergence. Teams are under different delivery pressures and different services have different needs—at worst, developers might forget to follow good resiliency practices.

Any change in service topology can have a negative impact. Figure 6.20 illustrates two examples: adding a new collaborator downstream from market-data might decrease market-data's availability, whereas adding a new consumer might reduce the overall capacity of the market-data service, reducing service for existing consumers.

Frameworks and proxies are two different technical approaches to applying communication standards across multiple services that make it easy for engineers to fall into doing the right thing by ensuring services communicate resiliently and safely by default.

6.5.1 Frameworks

A common approach for ensuring services always communicate appropriately is to mandate the use of specific libraries implementing common interaction patterns like circuit breakers, retries, and fallbacks. Standardizing these interactions across all services using a library has the following advantages:

1. Increases the overall reliability of your application by avoiding roll-your-own approaches to service interaction
2. Simplifies the process of rolling out improvements or optimizations to communication across any number of services
3. Clearly and consistently distinguishes network calls from local calls within code
4. Can be extended to provide supporting functionality, such as collecting metrics on service interactions

This approach tends to be more effective when a company uses one language (or few languages) for writing code; for example, Hystrix, which we mentioned earlier, was intended to provide a standardized way—across all Java-based services in Netflix's organization—of controlling interactions between distributed services.

> **NOTE** Standardizing communication is a crucial element of building a microservice chassis, which we'll explore in the next chapter.

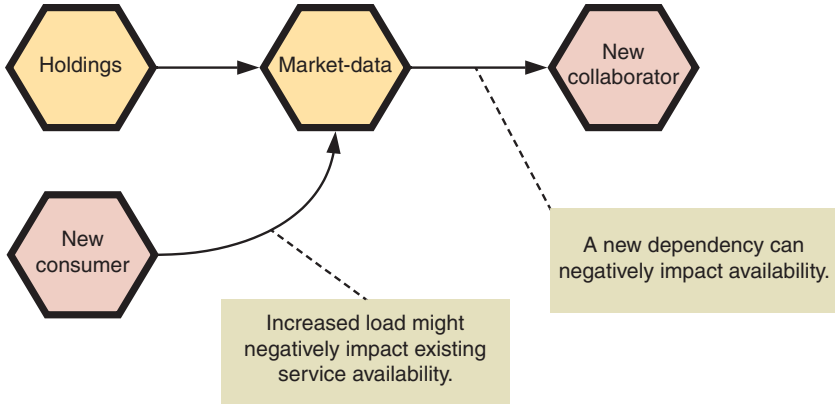


Figure 6.20 Availability impact of new services in a dependency chain

6.5.2 Service mesh

Alternatively, you could introduce a *service mesh*, such as Linkerd (https://linkerd.io) or Envoy (www.envoyproxy.io), between your services to control retries, fallbacks, and circuit breakers, rather than making this behavior part of each individual service. A service mesh acts as a proxy. Figure 6.21 illustrates how a service mesh handles communication between services.

Instead of services communicating directly with other services, service communication passes through the service mesh application, typically deployed as a separate process on the same host as the service. You then can configure the proxy to manage that traffic appropriately—retrying requests, managing timeouts, or balancing load across different services. From the caller's perspective, the mesh doesn't exist—it makes HTTP or RPC calls to another service as normal.

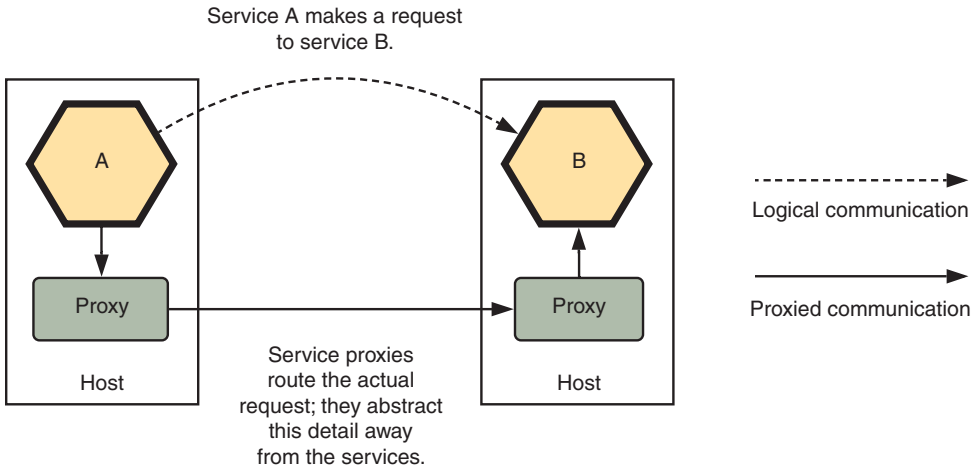


Figure 6.21 Communication between services using a service mesh

Although this may make the treatment of service interaction less explicit to an engineer working on a service, it can simplify defensive communication in applications that are heterogeneous. Otherwise, consistent communication can require significant time investment to achieve across different languages, because ecosystems and libraries may have unequal capabilities or support for resiliency features.

Summary

- Failure is inevitable in complex distributed systems—you have to consider fault tolerance when you're designing them.
- The availability of individual services affects the availability of the wider application.
- Choosing the right level of risk mitigation for an application requires careful consideration of the frequency and impact of failure versus the cost of mitigating against potentially rare events.
- Most failures occur in one of four areas: hardware, communication, dependencies, or internally.
- Cascading failures result from positive feedback and are a common failure mode in a microservice application. They're most commonly caused by server overload.
- You can use retries and deadlines to mitigate against faults in service interactions. You need to apply retries carefully to avoid exacerbating failure in other services.
- You can use fallbacks—such as caching, alternative services, and default results—to return successful responses, even when service dependencies fail.
- You should propagate deadlines between services to ensure they're consistent across a system and to minimize wasted work.
- Circuit breakers between services protect against cascading failures by failing quickly when a high threshold of errors is encountered.
- Services can use rate limits to protect themselves from spikes in load beyond their capacity to service.
- Individual services should expose health checks for load balancers and monitoring to be able to use.
- You can effectively validate resiliency by practicing both load and chaos testing.
- You can apply standards—whether through proxies or frameworks—to help engineers "fall into the pit of success" and build services that tolerate faults by default.

7 Building a reusable microservice framework

This chapter covers

- Building a microservice chassis
- Advantages of enforcing uniform practices across teams
- Abstracting common concerns in a reusable framework

Once an organization fully embraces microservices and teams grow in number, it's quite likely that each of those teams will start specializing in a given set of programming languages and tools. Sometimes, even when using the same programming language, different teams will choose a different combination of tools to achieve the same purpose. Although nothing is wrong with this, it may lead to an increased challenge for engineers moving between different teams. The ritual to set up new services, as well as the code structure, may be quite different. Even if teams eventually end up solving the same challenges in different ways, we believe this potential duplication is better than having to add a synchronization layer.

Having strict rules on the tools and languages that teams can use and enforcing a canonical way of setting up new services across all teams may harm speed and innovation and will eventually lead to the use of the same tools for every problem. Fortunately, you can enforce some common practices while keeping things rather free for teams to choose the programming language for specific services. You can encapsulate a set of tools for each adopted language while making sure that engineers have access to resources that'll make it easy to abide by the practices across all teams. If team A decides to go with Elixir to create a service for managing notifications and team B decides to use Python for an image analysis service, they should both have the tools that allow those two services to emit metrics to the common metrics collection infrastructure.

You should centralize logs in the same place and with the same format, and things like circuit breakers, feature flags, or the ability to share the same event bus should be available. That way, teams can make choices but also have the tools to become aligned with the infrastructure available to run their services. These tools form a *chassis*, a foundation, that you can build new services on without much up-front investigation and ceremony. Let's consider how to build a chassis for your services—one that abstracts common concerns and architectural choices while at the same time enables teams to quickly bootstrap new services.

7.1 A microservice chassis

Imagine an organization has eight different engineering teams and four engineers on each team. Now imagine one engineer on each team is responsible for bootstrapping a new service in Python, Java, or C#. Those languages, like most mainstream languages, have a lot of options in the form of available libraries. From http clients to logging libraries, the choice is plentiful. What would be the odds of two teams selecting the same language ending up with the same combination of libraries? I'd say pretty narrow! This issue isn't exclusive to microservice applications; for a monolithic application I worked on, different programmers were using three distinct http client libraries!

In figures 7.1 to 7.3, you can see the choices a team may face while choosing components to use in a new project.

As you can see in figures 7.1–7.3, the choice isn't easy! No matter which language you choose, options are plentiful, so the time you take to select components can increase, along with the risk of picking up a less than ideal library. An organization most likely will settle with two or three languages as the widely adopted ones, depending on the problems they need to solve. As a result, teams using the same language will coexist. Once one team gains some experience with a set of libraries, why wouldn't you use that experience to the benefit of other teams? You can provide a set of libraries and tools already used in production that people bootstrapping new projects could choose from without the burden of having to dig deeply into each library to weigh the pros and cons.

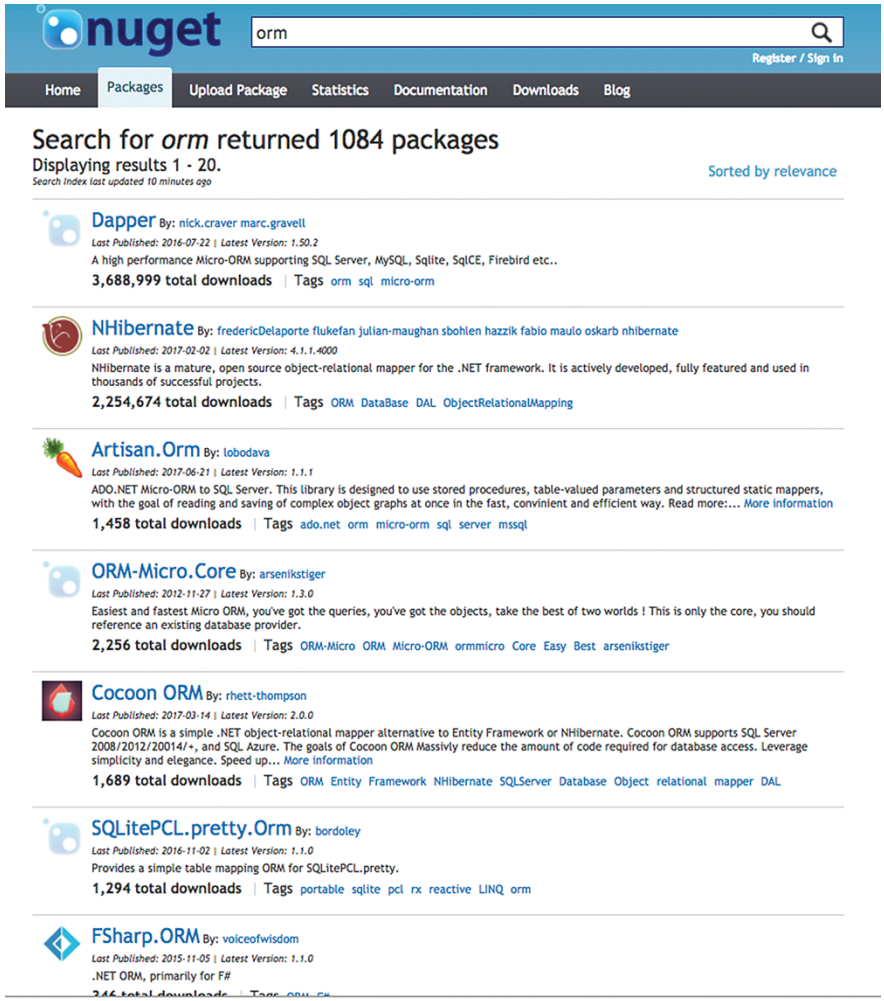
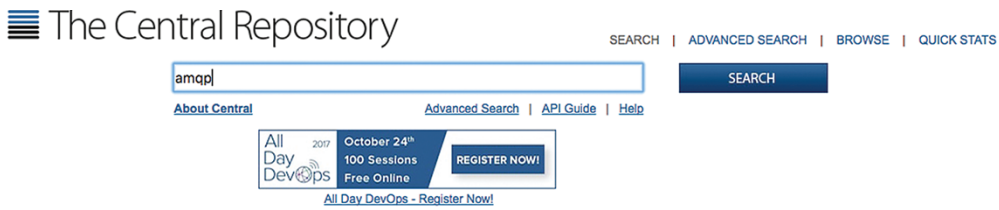


Figure 7.1 Search results for object-relational mapping (ORM) libraries for the .NET ecosystem

To make the job easier for your teams to create new services, it's worth your while to provide basic structure and a set of vetted tools for each of the languages your organization uses to build and operate services. You also should make sure that structure abides with your standards regarding observability and the abstraction of infrastructure-related code and it reflects your architectural choices regarding communication between services. An example of this, if the organization favors asynchronous communication between services, would be providing the needed libraries for using an event bus infrastructure that's already in place.

Not only would you be able to soft-enforce some practices, you also could make it easier to spawn new services quickly and allow fast prototyping. After all, it wouldn't make sense to take longer to bootstrap a service than to write the business logic that powers it.



Search Results

< 1 2 3 … 6 7 > displaying 1 to 20 of 136

GroupId	ArtifactId	Latest Version	Updated	Download
org.apache.activemq.examples.amqp	amqp	2.2.0 all (13)	25-Jul-2017	pom
de.dentrassi.kura.addons	amqp	0.3.0 all (5)	02-Jun-2017	pom
org.webjars.npm	amqp	0.2.4	08-Oct-2015	pom jar javadoc.jar sources.jar
com.nummulus.amqp.driver	amqp-parent_2.10	0.1.0	26-Oct-2014	pom
org.springframework.amqp	spring-amqp-parent	1.1.1.RELEASE all (3)	31-May-2012	pom
de.dentrassi.kura.amqp	de.dentrassi.kura.amqp-parent	0.0.3 all (2)	04-Oct-2016	pom
com.nummulus.amqp.driver	amqp-driver-test_2.10	0.1.0	26-Oct-2014	pom jar
org.wildfly.swarm	camel-amqp	2017.8.1 all (6)	02-Aug-2017	pom jar
org.apache.nifi	nifi-amqp-bundle	1.3.0 all (17)	05-Jun-2017	pom
org.carewebframework	org.carewebframework.amqp-parent	4.0.7 all (12)	10-Jan-2017	pom
org.kaazing	java.client.amqp.common	5.1.0.4	28-May-2015	pom
org.objectweb.joram	mom-amqp	1.0 all (2)	13-Jul-2011	pom jar
org.springframework.amqp	spring-amqp	1.7.3.RELEASE all (48)	07-Jun-2017	pom jar javadoc.jar sources.jar
org.springframework.amqp	spring-amqp-dist	1.6.1.RELEASE	10-Nov-2016	dist.zip docs.zip schema.zip
de.dentrassi.kura.amqp	de.dentrassi.kura.camel.amqp.runtime	0.0.3 all (2)	04-Oct-2016	pom dp
com.nummulus.amqp.driver	amqp-driver-blackbox_2.10	0.1.0	26-Oct-2014	pom jar javadoc.jar sources.jar
com.nummulus.amqp.driver	amqp-driver_2.10	0.1.0	26-Oct-2014	pom jar javadoc.jar sources.jar
com.espertech	esperio-amqp	7.0.0-beta1 all (16)	03-Aug-2017	pom jar javadoc.jar sources.jar
net.liftmodules	amqp_3.1_2.11	1.5.0	30-Jul-2017	pom jar javadoc.jar sources.jar
net.liftmodules	amqp_3.1_2.12	1.5.0	30-Jul-2017	pom jar javadoc.jar sources.jar

< 1 2 3 … 6 7 > displaying 1 to 20 of 136

Figure 7.2 Search for Advanced Message Queuing Protocol (AMQP) libraries for the Java ecosystem

Package	Weight*	Description
circuitbreaker 1.0.1	14	Python Circuit Breaker pattern implementation
breakers 0.1.0	10	Usable Circuit Breaker pattern implementation
gevent-breaker 1.0.0	10	Circuitbreaker pattern for gevent apps
protector 0.6.0	10	A circuit breaker for Time series databases like InfluxDB that prevents expensive queries
pybreaker 0.4.0	10	Python implementation of the Circuit Breaker pattern
python-circuit 0.1.9	10	Simple implementation of the Circuit Breaker pattern
circuit-breaker 0.1	8	Timeout decorator for functions with backup
requests-circuit 0.1.0	8	A circuit breaker for Python requests
datawire-mdk 2.0.37	2	The Microservices Development Kit (MDK)
dynamic-dynamodb 2.4.0	2	Automatic provisioning for AWS DynamoDB tables
dyno 0.2	2	Dependency injection framework based of Netflix's Hystrix
lxc_ssh_controller 0.1.3	2	Simple wrapper over LXC via SSH (paramiko).
oe_daemonutils 0.9.0	2	Daemon Utility Library
qcache-client 0.5.0	2	Python client library for QCache
waiter 0.4	2	Delayed iteration for polling and retries.

*: occurrence of search term weighted by field (name, summary, keywords, description, author, maintainer)

Figure 7.3. Search for circuit breaker libraries for Python

The chassis structure allows teams to select a tech stack (language + libraries) and quickly set up a service. You might ask yourself: how hard is it to bootstrap a service without this so-called chassis? It can be easy, if you don't have concerns like

- Enabling deployments in the container scheduler from day one (CI/CD)
- Setting up log aggregation
- Collecting metrics
- Having a mechanism for synchronous and asynchronous communication
- Error reporting

At SimpleBank, no matter what programming language or tech stack a team chooses, services should be providing all the functionality described in the list above. This type of setup isn't trivial to achieve, and, depending on the stack you chose, it can take more than a day to set up. Also, the combination of libraries two teams would choose for the same purpose could be quite different. You mitigate any issues related to that difference by providing a microservice chassis, so each team can focus on delivering features that SimpleBank customers will be using.

7.2 What's the purpose of a microservice chassis?

The purpose of a microservice chassis is to allow you to make services easier to create while ensuring you have a set of standards that all services abide by, no matter which team owns a service. Let's look into some of the advantages of having a microservices chassis in place:

- Making it easier to onboard team members
- Getting a good understanding of the code structure and concerns regarding the tech stack that an engineering team uses
- Limiting the scope of experimentation for production systems as the team builds common knowledge, even if not always in the same tech stack
- Helping to adhere to best practices

Having a predictable code structure and commonly used libraries will make it easier for team members to quickly understand a service's implementation. They'll only need to bother with the business logic implementation, because any other code will be pretty much common throughout all services. For example, common code will include code to deal with or configure

- Logging
- Configuration fetching
- Metrics collection
- Data store setup
- Health checks
- Service registry and discovery
- The chosen transport-related boilerplate (AMQP, HTTP)

If common code has already taken care of those concerns when someone is creating a new service, the need to write boilerplate is reduced or eliminated, and developers will less likely have to reinvent the wheel. Good practices within the organization will also be easier to enforce.

From a knowledge sharing perspective, having a microservice chassis will also enable easy code reviews by members of different teams. If they're using the same chassis, they'll be familiar with the code structure and how things are done. This will increase visibility and allow you to gather opinions from engineers from other teams. It's always desirable to have a different view on the problems a specific team is working on solving.

7.2.1 Reduced risk

By providing a microservice chassis, you reduce the risk you face, because you'll have less of a chance of picking a combination of language and libraries that won't work for a particular need. Imagine a service you're creating needs to fully communicate asynchronously with other services using an already existing event bus. If your chassis already covers that use case, you're not likely to end up with a setup that you need to tweak and eventually won't work well. You can cover that asynchronous communication use case as well as the synchronous one so you don't need to expend further effort to find a working solution.

The chassis can be constantly evolving to incorporate the findings of different teams, allowing you to be always up to date with the organization's practices and experience dealing with multiple use cases. All in all, there will be less chance for a team to face a challenge that other teams haven't solved before. And in case no one has solved that type of challenge yet, only one team needs to solve it; then you can incorporate the solution into the chassis, reducing the risks other teams have to take in the future.

Having a microservice chassis that already selects a set of libraries for use will limit the management of dependencies an engineering team will have to deal with. Referring to figures 7.1 to 7.3, if you have available one ORM, one AMQP, and one circuit breaker library, those will eventually be well known across multiple teams, and if someone finds a vulnerability in any of those libraries, you'll be able to update them with ease.

7.2.2 Faster bootstrapping

It makes little sense to spend one or two days bootstrapping a service when it could take far less time to implement the business logic. Also, wiring the needed components that form a service is a repetitive task that can be error prone. Why make people have to go and set up components all over again every time they create a new service? Using, maintaining, and updating a microservice chassis will lead to a setup that's sound, tested, and reusable. This will allow for faster service bootstrapping. Then you could use the extra time you gained by not having to write boilerplate code to develop, test, and deploy your features.

Having a sound foundation that teams use widely and know well allows you to experiment a lot more without worrying too much about the initial effort. If you can quickly turn a concept into a deployable service, you can easily validate it and decide to proceed with it or abandon it altogether. The key notion here is to be fast and to have it as easy as possible to create new functionality. Having a chassis in place also can significantly lower the entry barrier for new team members, because it'll be quicker for them to jump into any project once they learn the structure that's common to all services in each language.

7.3 Designing a chassis

At SimpleBank, the team responsible for implementing the purchasing and selling of stocks decided to create a chassis for the wider engineering team to use—they had faced a couple of challenges and want to share their experiences. We described a feature for selling stocks in chapter 2, figure 2.7. Let's look at a flow diagram to better understand it (figure 7.4).

To sell stocks, a user issues a request via the web or a mobile application. An API gateway will pick up the request and will act as the interface between the user-facing application and all internal services that'll collaborate to provide the functionality.

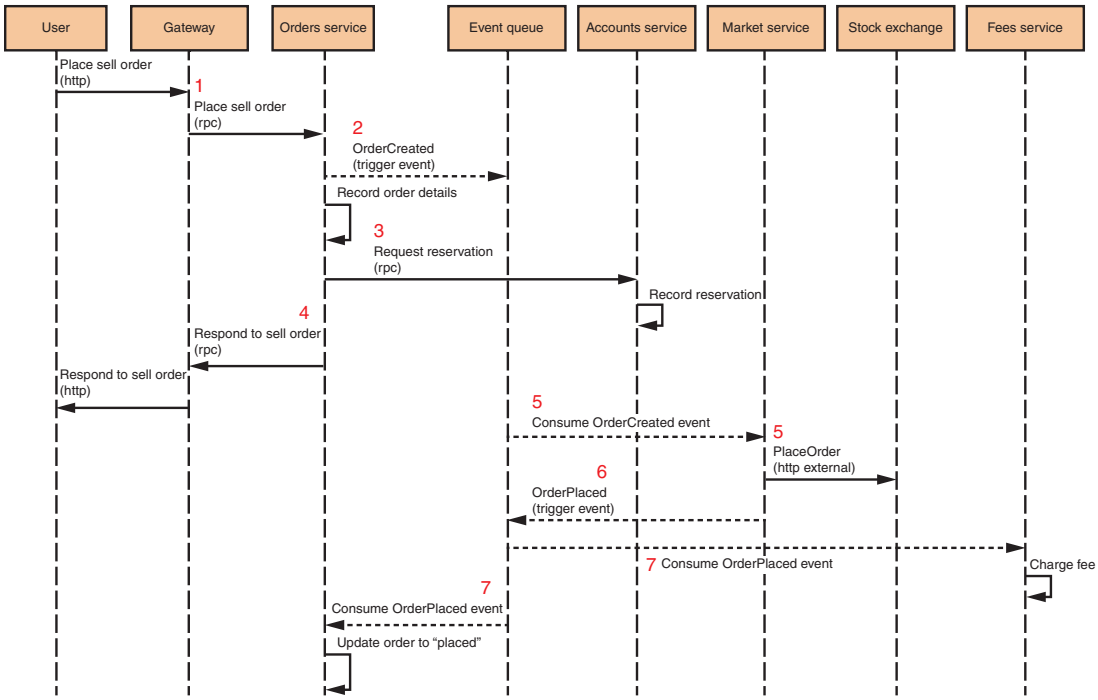


Figure 7.4 The flow for selling stocks involves both synchronous and asynchronous communication between the intervening services.

Given that it can take a while to place the order to the stock exchange, most operations will be asynchronous, and you'll return a message to the client indicating their request will be processed as soon as possible. Let's look into the interactions between services and the type of communication:

1. The gateway passes the user request to the orders service.
2. The orders service sends an `OrderCreated` event to the event queue.
3. The orders service requests the reservation of a stock position to the account transaction service.
4. The orders service replies to the initial call from the gateway, then the gateway informs the user that the order is being processed.
5. The market service consumes the `OrderCreated` event and places the order to the stock exchange.
6. The market service emits an `OrderPlaced` event to the event queue.
7. Both the fees service and the orders service consume the `OrderPlaced` event; they then charge the fees for the operation and update the status of the order to "placed," respectively.

For this feature, you have four internal services collaborating, interactions with an external entity (stock exchange), and communication that's a mix of synchronous and asynchronous. The use of the event queue allows other systems to react to changes; for instance, a service responsible for emailing or real-time notifications to clients can easily consume the `OrderPlaced` event, allowing it to send notifications of the placed order.

Given that the team owning this feature was comfortable with using Python, they created the initial prototype using the nameko framework (https://github.com/nameko/nameko). This framework offers, out of the box, a few things:

- AMQP RPC and events (pub-sub)
- HTTP GET, POST, and websockets
- CLI for easy and rapid development
- Utilities for unit and integration testing

But a few things were missing, like circuit breakers, error reporting, feature flags, and emitting metrics, so the team decided to create a code repository with libraries to take care of those concerns. They also created a Dockerfile and Docker compose file to allow building and running the feature with minimum effort and to offer a base for other teams to use when developing in Python. The code for the initial Python chassis (http://mng.bz/s4B2) and for the described feature (http://mng.bz/D19l) is available at the book code repository.

We'll now look with more detail at how the built chassis deals with service discovery, observability, transport, and balancing and limiting.

7.3.1 Service discovery

Service discovery for the Python chassis that emerged from implementing the feature we previously described is quite simple. The communication between the services involved occurs either synchronously via RPC calls or asynchronously by publishing events. SimpleBank uses RabbitMQ (www.rabbitmq.com) as the message broker, so this indirectly provides a way of registering services for both the asynchronous and synchronous use case. RabbitMQ allows the use of synchronous request/response communication implementing RPC over queues, and it'll also load balance the consumers using a round-robin algorithm (https://en.wikipedia.org/wiki/Round-robin_scheduling) by default. This allows you to use the messaging infrastructure to register services as well as to automatically distribute load between multiple instances of the same service. Figure 7.5 shows the RPC exchange your different services connect to.

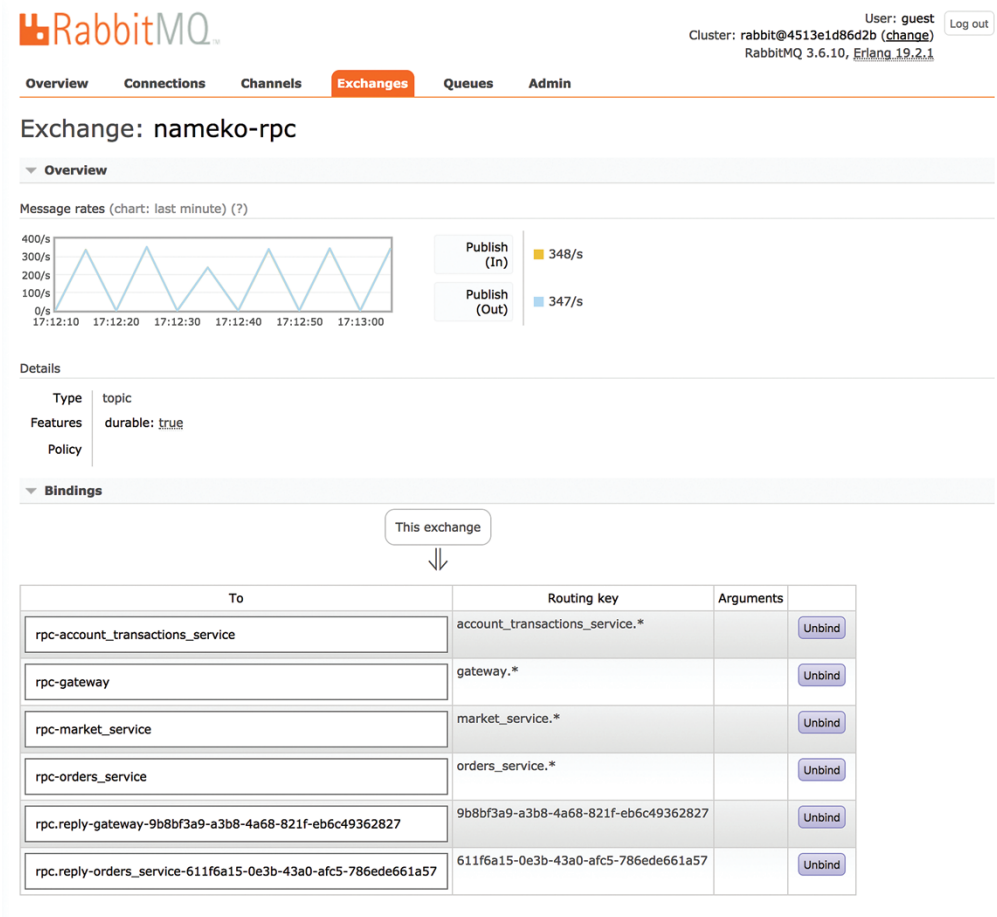


Figure 7.5 Services communicating via RPC register in an exchange. Multiple instances for a given service use the same routing key, and RabbitMQ will route the incoming requests between those instances.

All running services register themselves in this exchange. This will allow for them to communicate seamlessly without the need for each one to know explicitly where any service is located. This is also the case for RPC communication over the AMQP protocol, which allows you to have the same request/response behavior you'd get by using HTTP.

Let's take a look on how easy it is to have this feature available to you by using the capacities that the chassis provides, in this case by using the nameko framework, as shown in the following listing.

Listing 7.1 microservices-in-action/chapter-7/chassis/rpc_demo.py

```
from nameko.rpc import rpc, RpcProxy

class RpcResponderDemoService:
    name = "rpc_responder_demo_service"

    @rpc
    def hello(self, name):
        return "Hello, {}!".format(name)

class RpcCallerDemoService:
    name = "rpc_caller_demo_service"

    remote = RpcProxy("rpc_responder_demo_service")

    @rpc
    def remote_hello(self, value="John Doe"):
        res = u"{}".format(value)
        return self.remote.hello(res)
```

Assigns the service name a variable—This is the name that a particular service registers to allow others to call it.

Allows nameko to set up the RabbitMQ queues necessary to perform a request/response type of call—The rpc call will behave synchronously.

Creates an RPC Proxy for service that'll be invoked via RPC—You pass the name of the remote service.

Calls the remote service via the RpcProxy—This will execute the hello function on the RpcResponderDemoService class.

In this example, we've defined two classes, a responder and a caller. In each class, we also defined a name variable that holds the identifier for the service. Use of the `@rpc` annotation will decorate the function. This decoration will allow you to transform what seems an ordinary function into something that'll make use of the underlying AMQP infrastructure (that RabbitMQ offers) to invoke a method in a service running elsewhere. Calling the `remote_hello` method from the `RpcCallerDemoService` class will result in invoking the `hello` function in the `RpcResponderDemoService`, because that service is registered as remote via a `RpcProxy` that the framework provides.

Once you run this example code, RabbitMQ will display something like figure 7.6. In Figure 7.6, you can observe that once you boot the services that `rpc_demo.py` defines, each one registers in a queue scoped to the service name: `rpc-rpc_caller_demo_service` and `rpc-rpc_responder_demo_service`. Two other queues—`rpc.reply-rpc_caller_demo_service*` and `rpc.reply-standalone_rpc_proxy*`—also appear, and they'll relay back the responses to the caller service. This is a way of implementing blocking synchronous communication in RabbitMQ (http://mng.bz/4blSh).

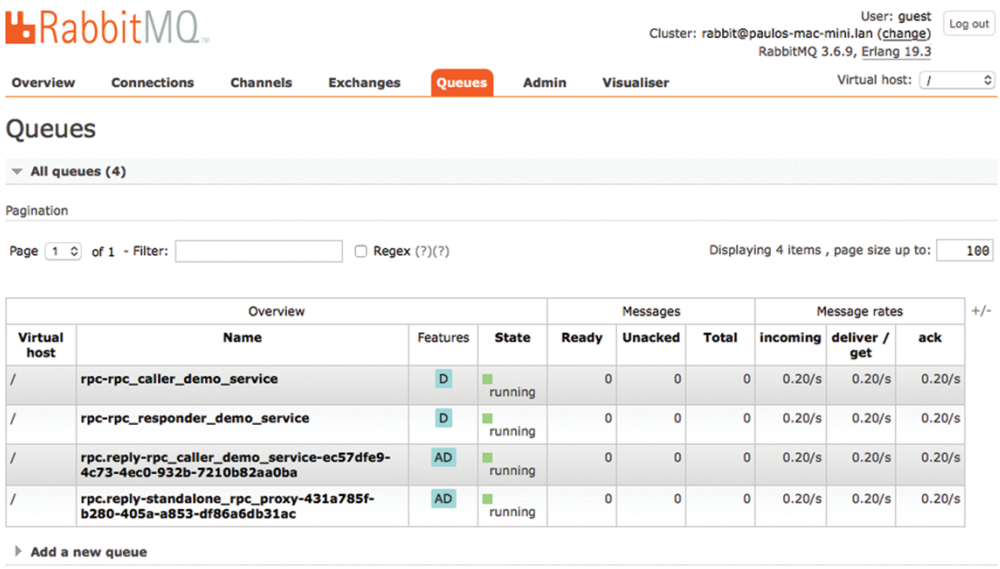


Figure 7.6 Caller and responder demo services registered in RabbitMQ queues

Your chassis makes it super easy to access this functionality so you can use the same infrastructure for both synchronous and asynchronous communication between services. This setup brings you huge speed gains while prototyping solutions, because the team can spend its time developing new features instead of having to build all the functionality from scratch. If you opt for an orchestrated behavior, with blocking calls between services, a choreographed behavior where all communication is asynchronous, or a mix between the two, you can use the same infrastructure and library.

The following listing shows an example on how to use full asynchronous communication between services by using the functionality of the chassis.

Listing 7.2 microservices-in-action/chapter-7/chassis/events_demo.py

```
from nameko.events import EventDispatcher, event_handler
from nameko.rpc import rpc
from nameko.timer import timer

class EventPublisherService:
    name = "publisher_service"

    dispatch = EventDispatcher()

    @rpc
    def publish(self, event_type, payload):
        self.dispatch(event_type, payload)

class AnEventListenerService:
```

Registers the service name, which allows you to refer to it on other services

Allows this service to create events that'll be routed to a queue in RabbitMQ

By using this annotation, ListenBothEventsService will execute the function when the publisher service issues an event. The first argument of the annotation is the name of the service whose events will be listened to, and the second argument of the annotation is the name of the event.

Registers the service name, which allows you to refer to it on other services

```
        name = "an_event_listener_service"

        @event_handler("publisher_service", "an_event")
        def consume_an_event(self, payload):
            print("service {} received:".format(self.name), payload)

    class AnotherEventListenerService:
        name = "another_event_listener_service"

        @event_handler("publisher_service", "another_event")
        def consume_another_event(self, payload):
            print("service {} received:".format(self.name), payload)

    class ListenBothEventsService:
        name = "listen_both_events_service"

        @event_handler("publisher_service", "an_event")
        def consume_an_event(self, payload):
            print("service {} received:".format(self.name), payload)

        @event_handler("publisher_service", "another_event")
        def consume_another_event(self, payload):
            print("service {} received:".format(self.name), payload)
```

As with the previous code example, each service a Python class implements declares a name variable that the framework will use to set up the underlying queues that allow communication. When running the services that each class in this file defines, RabbitMQ will create four queues, one for each service. As you can see in figure 7.7, the publisher service registers an RPC queue, without reply queue setup, contrary to the previous example that figure 7.6 illustrated. The other listener services register a queue per consumed event.

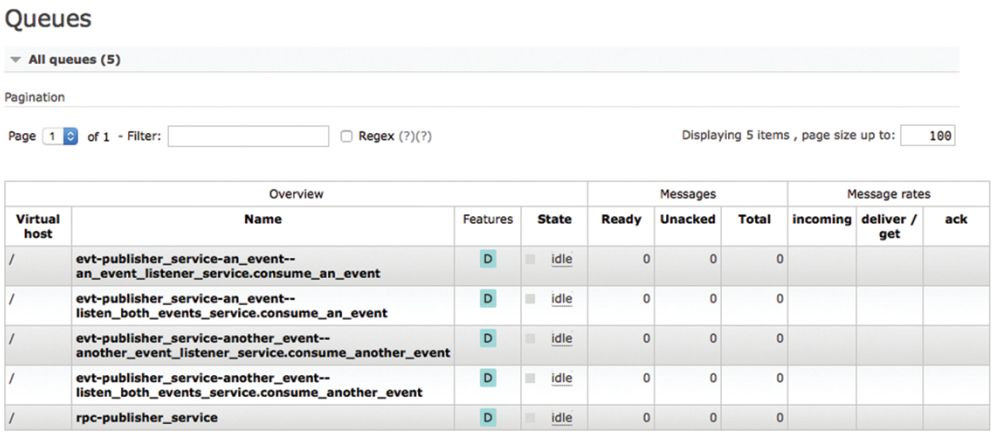


Figure 7.7 The queues that RabbitMQ creates when you run the services defined in events_demo.py

The team chose nameko to be part of the microservice chassis because it makes it easy to abstract from the details of implementing and setting up these two types of communication over the existing message broker. In section 7.3.3, we'll also look into another advantage that comes out of the box, because the message broker also takes care of load balancing.

7.3.2 Observability

To operate and maintain services, you need to be aware of what's going on in production at all times. As a result, you'll want the services to emit metrics to reflect the way they're operating, report errors, and aggregate logs in a usable format. In part 4 of the book, we'll focus on all these topics in more detail. But for now, let's keep in mind that services should address these concerns from day one. Operating and maintaining services is as important as writing them in the first place, and, in most cases, they'll spend a lot more time running than being developed.

Your microservice chassis has the dependencies shown in the following listing.

Listing 7.3 microservices-in-action/chapter-7/chassis/setup.py

```
(...)

    keywords='microservices chassis development',

    packages=find_packages(exclude=['contrib', 'docs', 'tests']),

    install_requires=[
        'nameko>=2.6.0',
        'statsd>=3.2.1',
        'nameko-sentry>=0.0.5',
        'logstash_formatter>=0.5.16',
        'circuitbreaker>=1.0.1',
        'gutter>=0.5.0',
        'request-id>=0.2.1',
    ],

(...)
```

Library to emit metrics in StatsD format

Library to Integrate with Sentry error reporting

Library to format the logs in logstash format

From the seven declared dependencies, you use three of them for observability purposes. These libraries will allow you to collect metrics, report errors, and gather some contextual information around them and to adapt your logging to the format you use in all services deployed at SimpleBank.

METRICS

Let's start with metrics collection and the use of StatsD.[1] Etsy originally developed StatsD as a way to aggregate application metrics. It quickly became so popular that it's now the *de facto* protocol to collect application metrics with clients in multiple programming languages. To be able to use StatsD, you need to instrument your code to capture all metrics you find relevant. Then a client library, in your case `statsd` for Python, will collect those metrics and send them to an agent that listens to UDP traffic from client libraries, aggregates the data, and periodically sends it to a monitoring system. Both commercial and open source solutions are available for the monitoring systems.

In the code repository, you'll be able to find a simple agent that'll be running in its own Docker container to simulate metrics collection. It's a trivial ruby script that listens to port 8125 over UDP and outputs to the console, as follows.

Listing 7.4 microservices-in-action/chapter-7/feature/statsd-agent/statsd-agent.rb

```
#!/usr/bin/env ruby
#
# This script was originally found  in a post by Lee Hambley
# (http://lee.hambley.name)
#
require 'socket'
require 'term/ansicolor'

include Term::ANSIColor

$stdout.sync = true

c = Term::ANSIColor
s = UDPSocket.new
s.bind("0.0.0.0", 8125)
while blob = s.recvfrom(1024)
  metric, value = blob.first.split(':')
  puts "StatsD Metric: #{c.blue(metric)} #{c.green(value)}"
end
```

This simple script allows you to simulate metrics collection while developing your services. Figure 7.8 shows metrics collection for services running when placing a sell order, the feature we use as an example for this chapter.

Using an annotation in the code for each service, you enable them to send metrics for some operations. Even though this is a simple example, because they're only emitting timing metrics, it serves the purpose of showing how you can instrument your code to collect data you find relevant. Let's look into one of the services to see how this is done. Consider the listing 7.5.

[1] See Ian Malpass, "Measure Anything, Measure Everything," *Code as Craft*, Etsy, http://mng.bz/9Tqo.

```
2017-08-06T15:34:03.061181622Z StatsD Metric: simplebank-demo.market.place_order_stock_exchange 72.827101|ms
2017-08-06T15:34:03.066127133Z StatsD Metric: simplebank-demo.market.place_order_stock_exchange 74.574471|ms
2017-08-06T15:34:03.066544049Z StatsD Metric: simplebank-demo.fees.place_order 0.101805|ms
2017-08-06T15:34:03.091995584Z StatsD Metric: simplebank-demo.orders.request_reservation 84.543467|ms
2017-08-06T15:34:03.092277160Z StatsD Metric: simplebank-demo.orders.sell_shares 135.972023|ms
2017-08-06T15:34:03.092536845Z StatsD Metric: simplebank-demo.account-transactions.request_reservation 0.047684|ms
2017-08-06T15:34:03.092615610Z StatsD Metric: simplebank-demo.account-transactions.request_reservation 0.082254|ms
2017-08-06T15:34:03.094587037Z StatsD Metric: simplebank-demo.orders.create_event 63.513279|ms
2017-08-06T15:34:03.095981420Z StatsD Metric: simplebank-demo.orders.create_event 61.930418|ms
2017-08-06T15:34:03.097697761Z StatsD Metric: simplebank-demo.gateway.sell_shares 166.799784|ms
2017-08-06T15:34:03.099692478Z StatsD Metric: simplebank-demo.market.create_event 80.114603|ms
2017-08-06T15:34:03.100135682Z StatsD Metric: simplebank-demo.market.request_reservation 157.628059|ms
2017-08-06T15:34:03.102362996Z StatsD Metric: simplebank-demo.orders.create_event 66.290379|ms
2017-08-06T15:34:03.103641730Z StatsD Metric: simplebank-demo.orders.create_event 68.701506|ms
2017-08-06T15:34:03.105580072Z StatsD Metric: simplebank-demo.orders.place_order 0.069857|ms
2017-08-06T15:34:03.106025175Z StatsD Metric: simplebank-demo.orders.place_order 0.064850|ms
2017-08-06T15:34:03.110052692Z StatsD Metric: simplebank-demo.fees.place_order 0.042200|ms
2017-08-06T15:34:03.110966388Z StatsD Metric: simplebank-demo.fees.place_order 0.092030|ms
2017-08-06T15:34:03.114130988Z StatsD Metric: simplebank-demo.market.create_event 75.125694|ms
2017-08-06T15:34:03.114507121Z StatsD Metric: simplebank-demo.market.request_reservation 170.120478|ms
2017-08-06T15:34:03.119891138Z StatsD Metric: simplebank-demo.market.place_order_stock_exchange 86.928844|ms
2017-08-06T15:34:03.127947173Z StatsD Metric: simplebank-demo.market.place_order_stock_exchange 82.599401|ms
2017-08-06T15:34:03.132498358Z StatsD Metric: simplebank-demo.orders.request_reservation 85.164547|ms
2017-08-06T15:34:03.132530844Z StatsD Metric: simplebank-demo.market.place_order_stock_exchange 97.762585|ms
2017-08-06T15:34:03.133046016Z StatsD Metric: simplebank-demo.orders.sell_shares 178.215742|ms
2017-08-06T15:34:03.133705024Z StatsD Metric: simplebank-demo.account-transactions.request_reservation 0.060797|ms
2017-08-06T15:34:03.135355994Z StatsD Metric: simplebank-demo.account-transactions.request_reservation 0.036240|ms
2017-08-06T15:34:03.135917545Z StatsD Metric: simplebank-demo.orders.request_reservation 90.982914|ms
2017-08-06T15:34:03.136233106Z StatsD Metric: simplebank-demo.orders.sell_shares 151.567221|ms
2017-08-06T15:34:03.139329135Z StatsD Metric: simplebank-demo.gateway.sell_shares 205.009937|ms
2017-08-06T15:34:03.140012433Z StatsD Metric: simplebank-demo.gateway.sell_shares 172.596216|ms
2017-08-06T15:34:03.140286212Z StatsD Metric: simplebank-demo.account-transactions.request_reservation 0.033617|ms
2017-08-06T15:34:03.141460892Z StatsD Metric: simplebank-demo.account-transactions.request_reservation 0.032902|ms
2017-08-06T15:34:03.142613482Z StatsD Metric: simplebank-demo.orders.place_order 4.956722|ms
2017-08-06T15:34:03.143533674Z StatsD Metric: simplebank-demo.orders.place_order 0.093222|ms
2017-08-06T15:34:03.146497263Z StatsD Metric: simplebank-demo.market.place_order_stock_exchange 98.298788|ms
2017-08-06T15:34:03.152099483Z StatsD Metric: simplebank-demo.fees.place_order 0.044823|ms
2017-08-06T15:34:03.152432636Z StatsD Metric: simplebank-demo.fees.place_order 0.035286|ms
```

Figure 7.8 StatsD agent collecting metrics that services collaborating in a place sell order operation have emitted

Listing 7.5 microservices-in-action/chapter-7/feature/fees/app.py

```
import json
import datetime
from nameko.events import EventDispatcher, event_handler
from statsd import StatsClient

class FeesService:
    name = "fees_service"
    statsd = StatsClient('statsd-agent', 8125,
                         prefix='simplebank-demo.fees')

    @event_handler("market_service", "order_placed")
    @statsd.timer('charge_fee')
    def charge_fee(self, payload):
        print("[{}] {} received order_placed event ... charging fee".format(
            payload, self.name))
```

Imports the StatsD client so you can use it in the module

Configures the StatsD client by passing the host, the port, and the prefix you'll use for all the emitted merics

Using this annotation enables you to collect the time it takes for the 'charge_fee' function to run. The StatsD library uses the value passed as an argument for the annotation as the metric name. In this case, the charge_fee function will emit the metric named 'simplebank-demo.fees.charge_fee'; the prefix you configured before will be prepended to the metric name passed to the annotation.

To collect metrics using the StatsD client library, you need to initialize the client by passing the hostname, in this case `statsd-agent`, the port, and an optional prefix for metrics collected in this service scope. If you annotate the `charge_fee` method with `@statsd.timer('charge_fee')`, the library will wrap the execution of that method in a timer and will collect the value from the timer and send it to the agent. You can collect these metrics and feed them to monitoring systems that'll allow you to observe your system behavior and set up alerts or even autoscale your services.

For example, imagine the fees service becomes too busy, and the execution time that StatsD reports increases over a threshold you set. You can automatically be alerted about that and immediately investigate to understand if the service is throwing errors or if you need to increase its capacity by adding more instances. Figure 7.9 shows an example of a dashboard displaying metrics that StatsD collected.

ERROR REPORTING

Metrics allow you to observe how the system is behaving on an ongoing basis, but, unfortunately, they aren't the only thing you need to care about. Sometimes errors happen, and you need to be alerted about them and, if possible, gather some information about the context in which the error occurred. For example, you might get a stack trace so you can diagnose and try to replicate and solve the error. Several services provide alerting and aggregation of errors. It's easy to integrate error reporting in your services, as shown in the following listing.

Listing 7.6 microservices-in-action/chapter-7/chassis/http_demo.py

```
import json
from nameko.web.handlers import http
from werkzeug.wrappers import Response
from nameko_sentry import SentryReporter          <-- Imports the error reporting module

class HttpDemoService:
    name = "http_demo_service"
    sentry = SentryReporter()                     <-- Initializes the error reporting service

    @http("GET", "/broken")
    def broken(self, request):
        raise ConnectionRefusedError()            <-- Raises an exception so you can test the error reporting service

    @http('GET', '/books/<string:uuid>')
    def demo_get(self, request, uuid):
        data = {'id': uuid, 'title': 'The unbearable lightness of being',
                'author': 'Milan Kundera'}
        return Response(json.dumps({'book': data}),
                        mimetype='application/json')

    @http('POST', '/books')
    def demo_post(self, request):
        return Response(json.dumps({'book': request.data.decode()}),
                        mimetype='application/json')
```

Figure 7.9 Example of a dashboard displaying metrics that StatsD collected from an application

Setting up error reporting in the chassis you assembled is simple. You initialize the error reporter, and it'll take care of capturing any exceptions and sending them over to the error reporting service backend. It's common for the error reporter to send along some context with the errors, like a stack trace. Figure 7.10 shows the dashboard with the error you get if you access the `/broken` endpoint in the demo service.

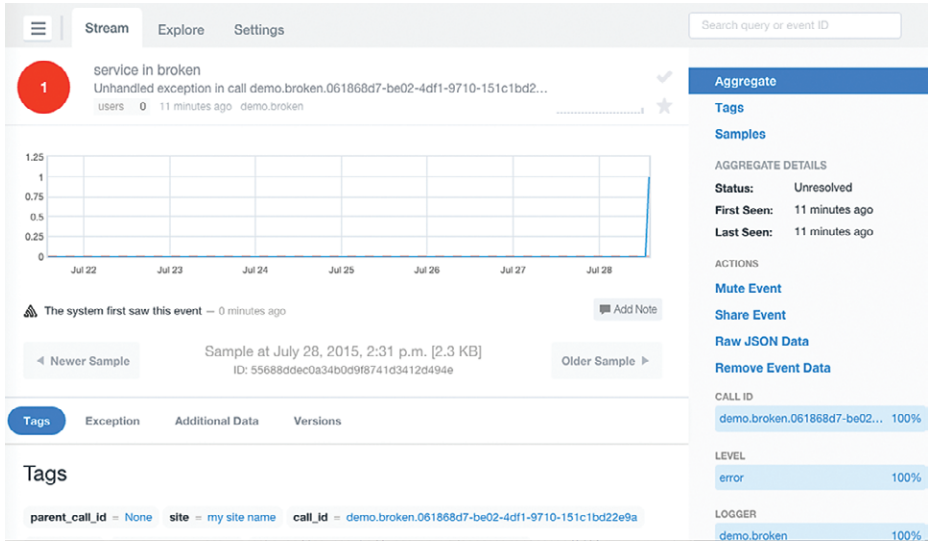


Figure 7.10 Dashboard for an error reporting service (Sentry) after accessing the `/broken` endpoint

LOGGING

Your services output information either to log files or to the standard output. These files can record a given interaction, such as the result and timing of an http call or any other information developers find useful to record. Having multiple services running this recording means you potentially have multiple services logging information across the organization. In a microservice architecture, where interactions happen between multiple services, you need to make sure you can trace those interactions and have access to them in a consistent way.

Logging is a concern for all teams and plays an important role in any organization. This is the case either for compliance reasons, when you may need to keep track of specific operations, or for allowing you to understand the flow of execution between different systems. The importance of logging is a sound reason for making sure that teams, no matter what language they're using to develop their services, keep logs in a consistent way and, preferably, aggregate them in a common place.

At SimpleBank, the log aggregation system allows complex searches in logs, so you agree to send logs to the same place and in the same format. You use logstash format for logging, so the Python chassis includes a library to emit logs in logstash format.

Logstash is an open source data processing pipeline that allows ingestion of data from multiple sources. The logstash format became quite popular and is widely used because it's a json message with some default fields, such as the ones you can find in the following listing.

Listing 7.7 Logstash json formatted message

```
{
  "message"    => "hello world",
  "@version"   => "1",
  "@timestamp" => "2017-08-01T23:03:14.111Z",
  "type"       => "stdin",
  "host"       => "hello.local"
}
```

Figure 7.11 shows the log output that the gateway service generates when receiving a place sell order request from a client. In such cases, it generates two messages. They both contain a wealth of information, like the filename, module, and line executing code, as well as the time it took for the operation to complete. The only information you passed explicitly to the logger was what appears in the message fields. The library you're using inserts all the other information.

By sending this information to a log aggregation tool, you can correlate data in many interesting ways. In this case, here are some example queries:

- Group by module and function name
- Select all entries for operations that took longer than *x* miliseconds
- Group by host

Figure 7.11 Logstash formatted log messages that the gateway service generated

The most interesting thing is that the host, type, version, and timestamp fields will appear in all the messages that the services using the chassis generate, so you can correlate messages from different services.

In your Python chassis, the following listing shows the code responsible for generating the log entries you can see in figure 7.11.

Listing 7.8 Logstash logger configuration in the Python chassis

```
import logging
from logstash_formatter import LogstashFormatterV1

logger = logging.getLogger(__name__)
handler = logging.StreamHandler()
formatter = LogstashFormatterV1()
handler.setFormatter(formatter)
logger.addHandler(handler)

(…)

# to log a message …
logger.info("this is a sample message")
```

This code is responsible for initializing the logging and adding the handler that'll format the output in the logstash json format.

By using the microservice chassis, you create a standard way of accessing the tools to achieve the goal of running observable services. By choosing certain libraries, you're able to enforce having all teams use the same underlying infrastructure without forcing any team to choose a particular language.

7.3.3 Balancing and limiting

We mentioned in section 7.3.1 on service discovery that the message broker provided not only a way for services to discover each other implicitly but also a load balancing capability.

While benchmarking the place sell order feature, say you realize you have a bottleneck in your processing. The market service has to interact with an external actor, the stock exchange, and will only do that after a successful response creates the `OrderPlaced` event

that both the fees service and the orders service will consume. Requests are accumulating because the HTTP call to the external service is slower than the rest of the processing in the system. For this reason, you decide to increase the number of instances running the market service. You deploy three instances to compensate for the extra time that the order placement onto the stock exchange takes. This change is seamless, because once you add the new instances, they're registered with the `rpc-market_service` queue in RabbitMQ. Figure 7.12 shows the three instances of the service connected.

As you can see, three instances are connected to the queue, each of them set to prefetch 10 messages from the queue as soon as they arrive. Now that you have multiple instances consuming from the same queue, you need to make sure only one of those instances processes each request. Once again, RabbitMQ makes your life easier because it deals with load balancing. By default, it'll use a round-robin algorithm to schedule the delivery of messages between the service instances. This means it'll deliver the first 10 messages to instance 1, then the next 10 to instance 2, and finally 10 to instance 3. It'll keep repeating this over and over. This is a naïve approach to scheduling work, because one instance may take longer than another one, but it generally works quite well and is easy to understand.

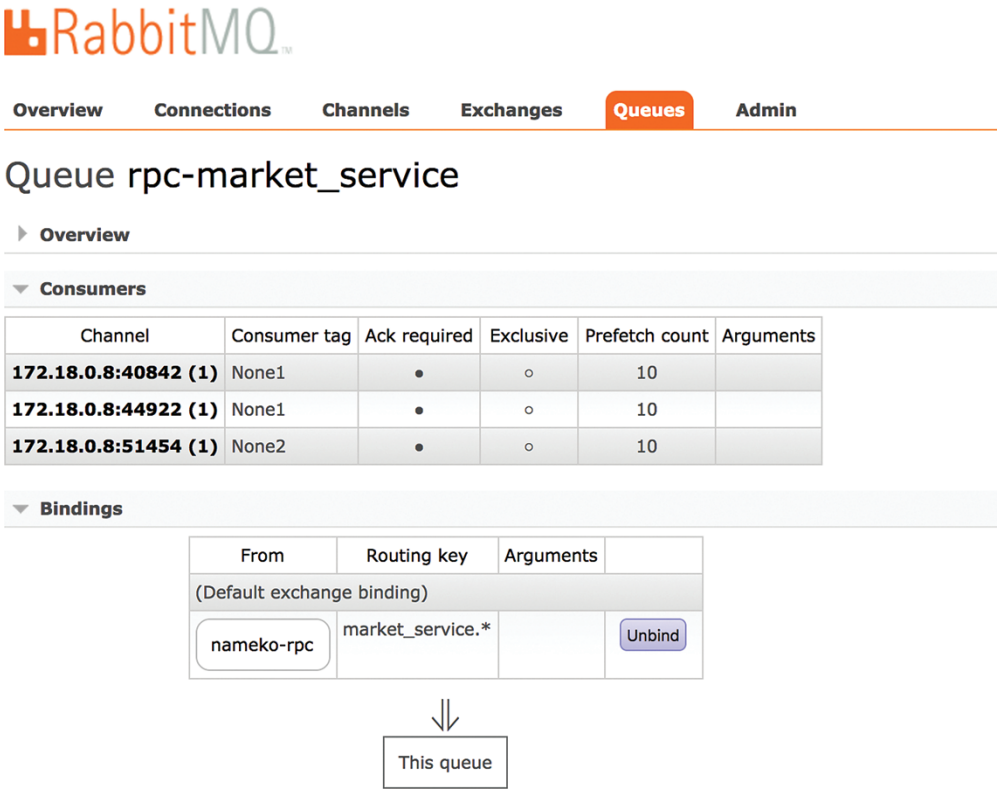


Figure 7.12 Multiple instances of the market service registered in the RPC queue

The only thing you need to be careful about is checking if the connected instances are healthy so they don't start accumulating messages. You can do so by making use of metrics, using StatsD, to monitor the number of messages that each instance is processing and if they're accumulating. In your code, you also can implement health checks so that any instance not responding to those health check requests can be flagged and restarted. RabbitMQ also will work as a limiting buffer, storing messages until the service instances can process them. According to the configuration shown in figure 7.12, each instance will receive ten messages to process at a time and will only be assigned new messages after it has finished processing previous ones.

It's worth mentioning that in the particular case of the market service as it interacts with a third-party system, you also implement a circuit breaking mechanism. Let's look at the service code where the call to the stock exchange is implemented, as follows.

Listing 7.9 microservices-in-action/chapter-7/feature/market/app.py

```
import json
import requests
(…)
from statsd import StatsClient
from circuitbreaker import circuit

class MarketService:
    name = "market_service"
    statsd = StatsClient('statsd-agent', 8125,
                         prefix='simplebank-demo.market')

    (…)

    @statsd.timer('place_order_stock_exchange')
    @circuit(failure_threshold=5, expected_exception=
➥ConnectionError)
    def __place_order_exchange(self, request):
        print("[{}] {} placing order to stock exchange".format(
            request, self.name))
        response = requests.get('https://jsonplaceholder.typicode.com/
 posts/1')
        return json.dumps({'code': response.status_code, 'body': response.
 text})
```

Imports the circuit breaker functionality to use in the module

Allows you to configure how many exceptions you tolerate before opening the circuit and the type of exceptions that'll count as a failure

You make use of the circuit breaker library to configure the number of consecutive failures to connect that you'll tolerate. In the example shown, if you have five consecutive failing calls with the `ConnectionError` exception, you'll open the circuit, and no call will be made for 30 seconds. After those 30 seconds, you'll enter the recovery stage, allowing one test call. If the call is successful, it'll close the circuit again, resuming normal operation and allowing calls to the external service; otherwise, it'll prevent calls for another 30 seconds.

> **NOTE** Because 30 seconds is the default value the circuit breaker library sets for the `recovery_timeout` parameter, you don't see it in listing 7.9. If you want to adjust this value, you can do so by passing it explicitly.

You could use this technique not only for external calls but also for calls between internal components, because it will allow you to degrade the service. In the case of the market service, using this technique would mean messages that services retrieved from the queue wouldn't be acknowledged and would accumulate in the broker. Once the external service connectivity was resumed, you'd be able to start processing messages from the queue. You could complete the call to the stock exchange and create the `OrderPlaced` event that allows both the fees service and the orders service to complete the execution of a place sell order request.

7.4 Exploring the feature implemented using the chassis

In the previous section, you saw code examples for the implementation of the place sell order feature. Let's briefly look into the resulting feature prototype that you'd implement using the chassis. Based on the chassis code that you can find in the code repository under chapter7/chassis, say you've created five services:

- Gateway
- Orders service
- Market service
- Account transactions service
- Fees service

Figure 7.13 shows the project structure and a Docker Compose file that allows you to locally start the five components and the StatsD agent we mentioned previously. The Docker Compose file will allow booting the services as well as the needed infrastructure components: RabbitMQ, Redis, and the local StatsD agent, which will simulate metrics collection.

We won't go deep on Docker or Docker Compose right now, because we'll cover it in the upcoming chapters. But if you do have Docker and Docker Compose available, you can boot the services by entering the feature directory and running `docker-compose up –build`. This will build a Docker container for each service and boot everything up.

Figure 7.14 shows all services running and processing a `POST` request to the `shares/sell` gateway endpoint.

Even though the feature makes use of both synchronous and asynchronous communication between the different components, the chassis you have in place allows you to quickly prototype it and run initial benchmarks using a tool that allows you to simulate concurrent requests, with results such as the following: (Please note that these benchmarks ran locally on a development machine and are merely indicative.)

```
$ siege -c20 -t300S -H 'Content-Type: application/json'
     'http://192.168.64.3:5001/shares/sell POST'

    (benchmark running for 5 minutes …)

Lifting the server siege...
```

```
Transactions:                  12663 hits
Availability:                  100.00 %
Elapsed time:                  299.78 secs
Data transferred:              0.77 MB
Response time:                 0.21 secs
Transaction rate:                     42.24 trans/sec
Throughput:                    0.00 MB/sec
Concurrency:                   9.04
Successful transactions:              12663
Failed transactions:           0
Longest transaction:           0.52
Shortest transaction:          0.08
```

These numbers look good, but it's worth mentioning that once the benchmark stopped, the market service still needed to consume 3000 messages—almost a quarter of the total requests that the gateway processed. This benchmark allows you to identify the bottleneck happening in the market service that we mentioned in section 7.3.3. Referring to figure 7.4, you can see that the gateway receives a response from the orders service, but asynchronous processing still happens after that.

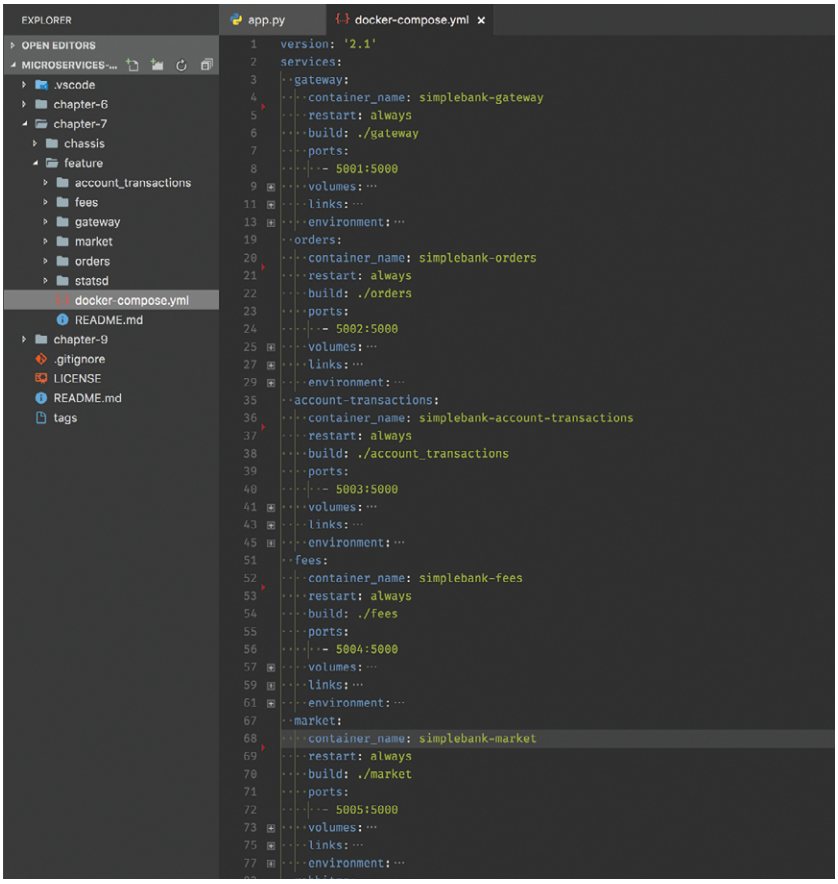


Figure 7.13 Project structure for the place sell order feature and the Docker Compose file that allows booting the services and the needed infrastructure components

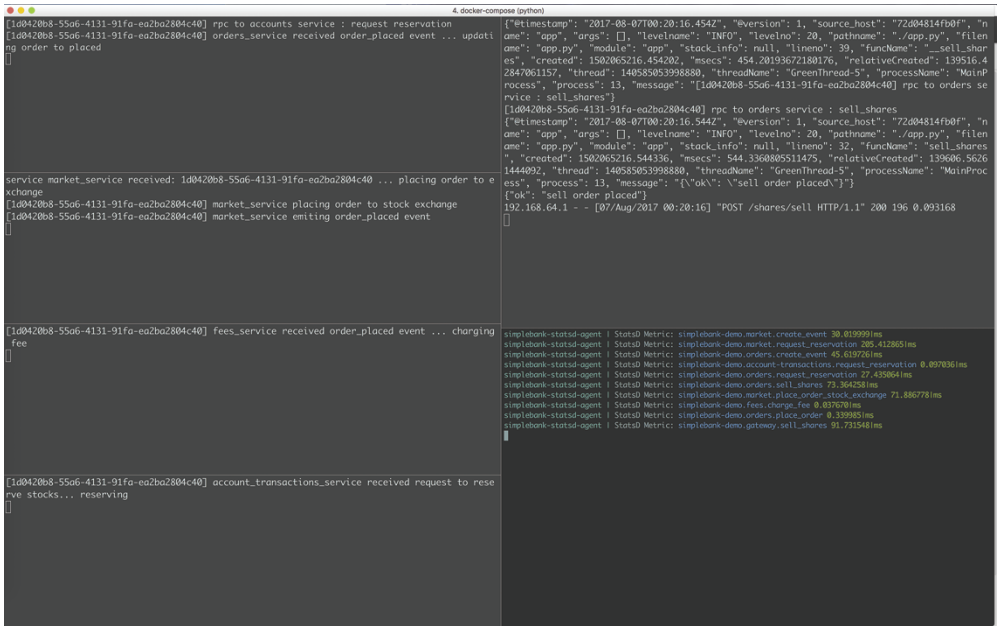


Figure 7.14 Services used in the place sell order running locally

The engineering team at SimpleBank certainly will continue to improve the Python chassis so it reflects continuous team learnings. For now though, it's already usable to implement nontrivial functionality.

7.5 Wasn't heterogeneity one of the promises of microservices?

In the previous sections, we covered building and using a chassis for Python applications at SimpleBank. You can apply the principles to any language used within your organization though. At SimpleBank, teams also use Java, Ruby, and Elixir for building services. Would you go and build a chassis for each of these languages and stacks? If the language is widely adopted within the organization and different teams bootstrap more than a couple of services, I'd say sure! But it's not imperative that you create a chassis. The only thing to keep in mind is that with or without a chassis, you need to maintain principles like observability.

One of the advantages of a microservice architecture is enabling heterogeneity of languages, paradigms, and tooling. In the end, it'll enable teams to choose the right tool for the job. Although in theory the choices are limitless, the fact is, teams will specialize in a couple of technology stacks for their day-to-day development. They'll naturally develop a deeper knowledge around one or two different languages and their supporting ecosystems. A supporting ecosystem is also important. Independent teams, such as the ones you need to have in place to successfully run a microservice architecture, will also focus

on operations and will know about the platforms running their apps. Some examples are the Java virtual machine (JVM) or the Erlang virtual machine (BEAM). Knowing about the infrastructure will help with delivering better and more efficient apps.

Netflix is a good example because they have a deep knowledge of the JVM. This enables them to be a proficient contributor of open source tools, allowing the community to benefit from the same tools they use to run their service. The fact that they have so many tools written targeting the JVM will make that ecosystem the first choice for their engineering teams. In some sense, it feels like: "You're free to choose whatever you want, as long as it abides with our given set of rules and implements some interfaces..., or you can use this chassis that takes care of all of that!"

Having existing chassis for some of the languages and stacks an organization has adopted may help direct teams' choices toward those languages and stacks. Not only will services be easier and faster to bootstrap, they'll also become more maintainable from a risk standpoint. A chassis is a great way to indirectly enforce key concerns and practices of an engineering team.

TIP DRY (don't repeat yourself) isn't mandatory. A chassis shouldn't be a sort of shared library or dependency to be included in services and updated in a centralized way. You should use the chassis to bootstrap new services, but not necessarily to update all running services with a given feature. It's preferable to repeat yourself a little than to bring in shared libraries that increase coupling. Do repeat yourself if that results in keeping systems decoupled and independently maintained and managed.

Summary

- A microservice chassis allows for quick bootstrapping of new services, enabling greater experimentation and reducing risk.
- The use of a chassis allows you to abstract the implementation of certain infrastructure-related code.
- Service discovery, observability, and different communication protocols are concerns of a microservice chassis, and it should provide them.
- You can quickly prototype a complex feature like the place sell order example, if the proper tooling exists.
- Although the microservice architecture is often associated with the possibility of building systems in any language, those systems, when in production, need to offer some guarantees and have mechanisms to allow their operation and maintenance to be manageable.
- A microservice chassis is a way to provide those guarantees while allowing fast bootstrap and quick development for you to test ideas and, if proven, deploy them to production.

Part 3

Deployment

An application is only useful if you can deploy it to your users. This part of the book will introduce you to deployment practices for microservices. We'll explore deployment techniques, such as continuous delivery and packaging, and deployment platforms, including Google Cloud Platform and Kubernetes. Throughout the next few chapters, you'll learn how to build a deployment pipeline to take microservice code changes safely and rapidly to production.

8 Deploying microservices

This chapter covers

- Why it's crucial to get deployment right in a microservice application
- The fundamental components of a microservice production environment
- Deploying a service to a public cloud
- Packaging a service as an immutable artifact

Mature deployment practices are crucial to building reliable and stable microservices. Unlike a monolithic application, where you can optimize deployment for a single use case, microservice deployment practices need to scale to multiple services, written in different languages, each with their own dependencies. You need to be able to trust your deployment process to push out new features—and new services—without harming overall availability or introducing critical defects.

As a microservice application evolves at the level of deployable units, the cost of deploying new services must be negligible to enable engineers to rapidly innovate and deliver value to users. The added development speed you gain from microservices will be wasted if you can't get them to production rapidly and reliably. Automated deployments are essential to developing microservices at scale.

In this chapter, we'll explore the components of a microservice production environment. Following that, we'll look at some deployment building blocks—such as artifacts and rolling updates—and how they apply to microservices. Throughout the chapter, we'll work with a simple service—market-data—to try out different approaches to packaging and deployment using a well-known cloud service, Google Cloud Platform. You can find a starting point for this service in the book's repository on Github (https://github.com/morganjbruce/microservices-in-action).

8.1 Why is deployment important?

Deployment is the riskiest moment in the lifecycle of a software system. The closest real-world equivalent would be changing a tire—except the car is still moving at 100 miles an hour. No company is immune to this risk: for example, Google's site reliability team identified that roughly 70% of outages are due to changes in a live system (https://landing.google.com/sre/book/chapters/introduction.html).

Microservices drastically increase the number of moving parts in a system, which increases the complexity of deployment. You'll face four challenges when deploying microservices (figure 8.1):

- Maintaining stability when facing a high volume of releases and component changes
- Avoiding tight coupling between components leading to build- or release-time dependencies
- Releasing breaking changes to the API of a service, which may negatively impact that service's clients
- Retiring services

When you do them well, deployments are based on simplicity and predictability. A consistent build pipeline produces predictable artifacts, which you can apply atomically to a production environment.

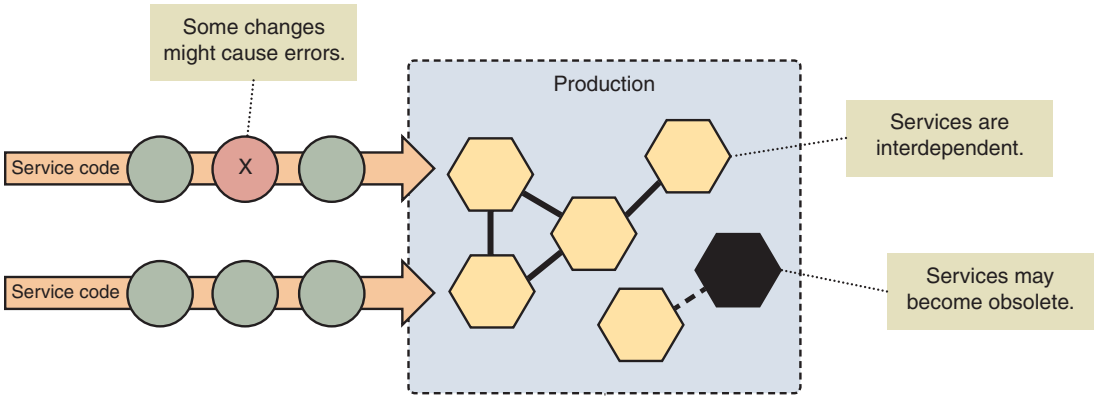


Figure 8.1 A high-level view of production deployment

8.1.1 Stability and availability

In an ideal world, deployment is "boring:" not unexciting, but incident-free. We've seen too many teams—both monolithic and microservice—that experience deploying software as incredibly stressful. But if working with microservices means you're releasing more components more frequently, doesn't that mean you're introducing more risk and instability into a system?

MANUAL CHANGE MANAGEMENT IS COSTLY

Traditional change management methodologies attempt to reduce deployment risk by introducing governance and ceremony. Changes must go through numerous quality gates and formal approvals, usually human-driven. Although this is intended to ensure that only working code reaches production, this approach is costly to apply and doesn't scale well to multiple services.

SMALL RELEASES REDUCE RISK AND INCREASE PREDICTABILITY

The larger a release, the higher the risk of introducing defects. Naturally, microservice releases are smaller because the codebases are smaller. And that's the trick—by releasing smaller changes more often, you reduce the total impact of any single change. Rather than stopping everything for a deployment, you can design your services and deployment approaches with the expectation that they'll face continuous change. Reducing the surface area of possible change leads to releases that are quicker, easier to monitor, and less disruptive to the smooth functioning of an application.

AUTOMATION DRIVES DEPLOYMENT PACE AND CONSISTENCY

Even if your releases are smaller, you still need to make sure your change sets are as free from defects as possible. You can achieve this by automating the process of commit validation—unit tests, integration tests, linting, and so on—and the process of rollout—applying those changes in the production environment. This helps you to build systematic confidence in the code changes you're making and apply consistent practices across multiple services.

> **TIP** Building for anti-fragility, or resilience during failure, is also an important element of overall application stability—don't forget to read chapter 6!

8.2 A microservice production environment

Deployment is a combination of process and architecture:

- The process of taking code, making it work, and keeping it working
- The architecture of the environment in which the software is operated

Production environments for running microservices vary widely, as do monolith production environments. What's appropriate for your application may depend on your organization's existing infrastructure, technical capabilities, and attitude toward risk, as well as regulatory requirements.

8.2.1 Features of a microservice production environment

The production environment for a microservice application needs to provide several capabilities to support the smooth operation of multiple services. Figure 8.2 gives a high-level view of the capabilities of the production environment.

A microservice production environment has six fundamental capabilities:

1. A *deployment target*, or *runtime platform*, where services are run, such as virtual machines (Ideally, engineers can use an API to configure, deploy, and update service configuration. You also could call this API the control pane, as shown in the figure.)
2. *Runtime management*, such as autohealing and autoscaling, that allows the service environment to respond dynamically to failure or changes in load without human intervention (For example, if a service instance fails, it should automatically be replaced.)
3. *Logging and monitoring* to observe service operation and provide insight for engineers into how services are behaving
4. Support for *secure operation*, such as network controls, secret management, and application hardening
5. Load balancers, DNS, and other *routing components* to route requests from users and between microservices
6. A *deployment pipeline* that delivers services from code, safely into operational usage in the production environment

These components are part of the *platform* layer of the microservice architecture stack.

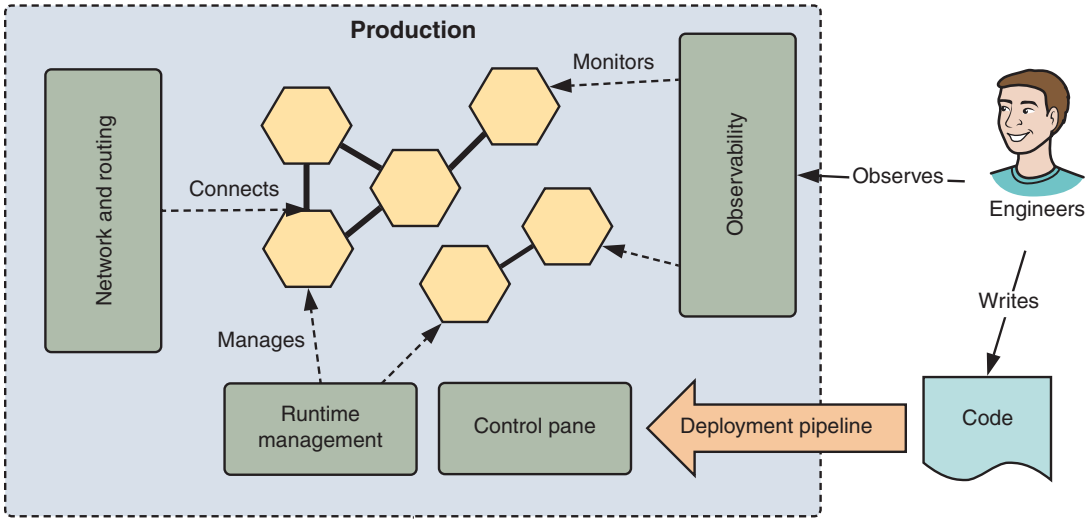


Figure 8.2 A microservice production environment

8.2.2 Automation and speed

Along with the six fundamental features, two factors are key in assessing the suitability of a deployment platform for a microservice application:

- *Automation*—The bulk of infrastructural management and configuration, such as spinning up a new host, should be highly amenable to automation, ideally by the team developing services themselves.
- *Speed*—If a significant cost is associated with every new deploy—whether obtaining infrastructure resources or setting up a new deployment—then a microservice approach will be significantly hampered.

Although you may not always have the luxury of choosing your deployment environment, it's important to appreciate how different platforms might affect these characteristics and how you develop your microservice application. I once worked for a company that took six weeks to provision each new server. Suffice it to say that taking new services into production was an exhausting endeavor!

It's not coincidental that the popularity of microservice architecture coincides with the wider adoption of DevOps practices, such as *infrastructure as code*, and the increasing use of cloud providers to run applications. These practices enable rapid iteration and deployment of services, which in turn makes a microservice architecture a scalable and feasible approach.

When possible you should aim to use a public infrastructure as a service (IaaS) cloud, such as Google Cloud Platform (GCP), AWS, or Microsoft Azure, for deploying any nontrivial microservice application. These cloud services offer a wide range of features and tools that ease the development of a robust microservice platform at a lower level of abstraction than a higher level deployment solution (such as Heroku). As such, they offer more flexibility. In the next section, we'll show you how to use GCP to deploy, access, and scale a microservice.

8.3 Deploying a service, the quick way

It's time to get your hands dirty and deploy a service. You need to take your code, get it running on a virtual machine, and make it accessible from the outside world—as figure 8.3 illustrates.

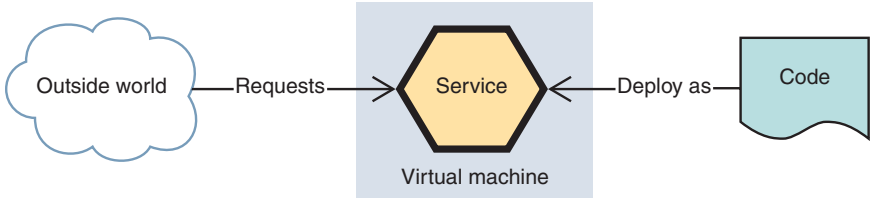


Figure 8.3 A simple microservice deployment

You'll use Google Compute Engine (GCE) as a production environment. This is a service on GCP that you can use to run virtual machines. You can sign up for a free trial GCP subscription, which will have enough credit for this chapter's examples. Although the operations you'll perform are specific to this platform, all major cloud providers, such as AWS and Azure, provide similar abstractions.

WARNING This example *isn't* a robust production deployment solution!

To interact with GCE, you'll use the `gcloud` command-line tool. This tool interacts with the GCE API to perform operations on your cloud account. You can find install instructions in the GCP documentation (https://cloud.google.com/sdk/docs/quickstarts). It's not the only option—you could use third-party tools like Ansible or Terraform instead.

Assuming you've followed the install instructions and logged in with `gcloud init`, you can create a new project:

```
gcloud projects create <project-id> --set-as
➥-default --enable-cloud-apis
```

Replace <project-id> with the name of your choice.

This project will contain the resources that'll run your service.

TIP Don't forget to tear down your project when you're done. Running `gcloud projects delete <project-id>` will do the trick.

8.3.1 Service startup

To run your service, you'll use a startup script, which will be executed at startup time when Google Cloud provisions your machine. We've written this for you already—you can find it at chapter-8/market-data/startup-script.sh.

Take your time to read through the script, which performs four key tasks:

- Installs binary dependencies required to run a Python application
- Downloads your service code from Github
- Installs that code's dependencies, such as the flask library
- Configures a supervisor to run the Python service using the Gunicorn web server

Now, let's try it out.

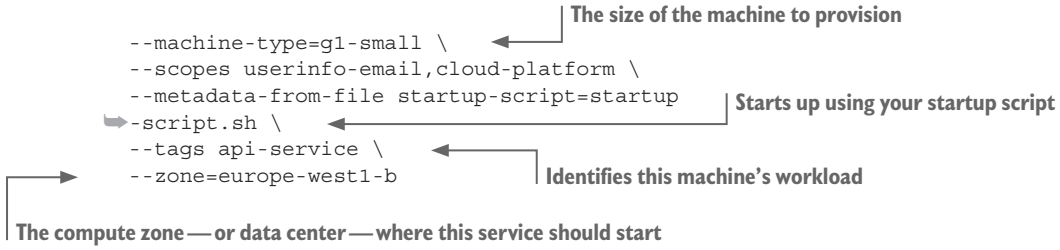
8.3.2 Provisioning a virtual machine

You can provision a virtual machine from the command line. Change to the chapter-8/market-data directory and run the following command:

The name of your machine

```
gcloud compute instances create market-data-service \
  --image-family=debian-9 \
  --image-project=debian-cloud \
```

The base image you'll use for the machine



This will create a machine and return the machine's external IP address—something like figure 8.4.

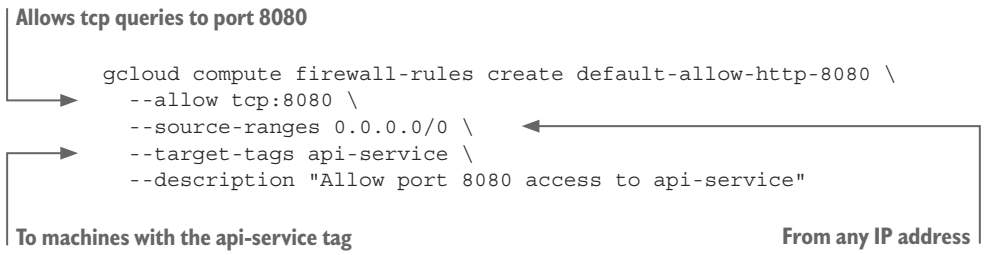
This approach to startup does take a while. If you want to watch the progress of the startup process, you can tail the output of the virtual machine's serial port:

```
gcloud compute instances tail-serial-port-output market-data-service
```

Once the startup process has completed, you should see a message in the log, similar to this example:

```
Mar 16 12:17:14 market-data-service-1 systemd[1]: Startup finished in
➥ 1.880s (kernel) + 1min 52.486s (userspace) = 1min 54.367s.
```

Great! You've got a running service—although you can't call it yet. You'll need to open the firewall to make an external call to this service. Running the following command will open up public access to port 8080 for all services with the tag `api-service`:



You can test your service by curling the external IP of the virtual machine. The external IP was returned when you created the instance (figure 8.4). If you didn't note it, you can retrieve all instances by running `gcloud compute instances list`. Here's the `curl`:

```
curl -R http://<EXTERNAL-IP>:8080/ping
```

Replace EXTERNAL-IP with the IP address of your service.

If all is going well, the response you get will be the name of the virtual machine—`market-data-service`.

```
Created [https://www.googleapis.com/compute/v1/projects/market-data-1/zones/europe-west1-b/instances/market-data-service].
NAME                ZONE            MACHINE_TYPE  PREEMPTIBLE  INTERNAL_IP  EXTERNAL_IP     STATUS
market-data-service  europe-west1-b  g1-small                   10.132.0.2   35.187.126.221  RUNNING
```

Figure 8.4 Information about a newly created virtual machine

8.3.3 Run multiple instances of your service

It's unlikely you'll ever run a single instance of a microservice:

- You'll want to scale horizontally (the X-axis of scalability) by deploying multiple clones of the same service, each handling a proportion of requests. Although you *could* serve more requests with progressively larger machines, it's ultimately possible to scale further using more machines.
- It's important to deploy with redundancy to ensure that failures are isolated. A single instance of a service won't maximize resiliency when failures occur.

Figure 8.5 illustrates a service group. Requests made to the logical service, market-data, are load balanced to underlying market-data instances. This is a typical production configuration for a stateless microservice.

> **NOTE** Services that consume from an event queue or message bus are also horizontally scalable—you distribute message load by running multiple message consumers.

You can try this out. On GCE, a group of virtual machines is called an *instance group* (or on AWS, it's an auto-scaling group). To create a group, you first need to create an `instance template`:

```
gcloud compute instance-templates create market-data-service-template \
  --machine-type g1-small \
  --image-family debian-9 \
  --image-project debian-cloud \
  --metadata-from-file startup-script=startup-script.sh \
  --tags api-service \
  --scopes userinfo-email,cloud-platform
```

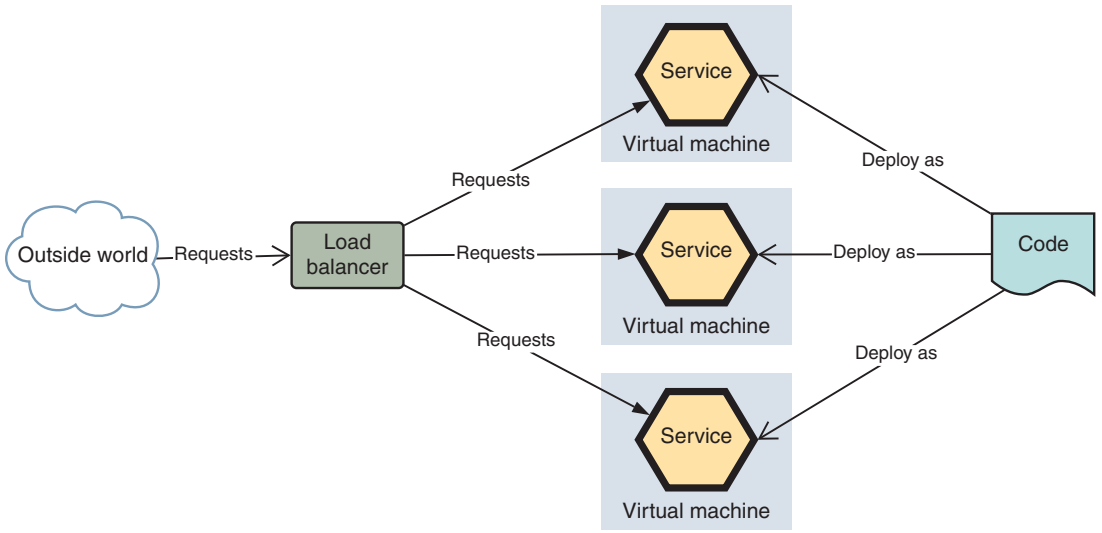


Figure 8.5 A service group and load balancer

Running this code will create a template to build multiple market-data-service instances like the one you built earlier. Once the template has been set up, create a group:

```
gcloud compute instance-groups managed create market-data-service-group \
  --base-instance-name market-data-service \
  --size 3 \
  --template market-data-service-template \
  --region europe-west1
```

The name prefix of each new instance

The number of instances in the group

The template to use

The region to start these instances in

This will spin up three instances of your market-data service. If you open the Google Cloud console and navigate to Compute Engine > Instance Groups, you should see a list like the one in figure 8.6.

Using an instance template to build a group gives you some interesting capabilities out of the box: failure zones and self-healing. These two features are crucial to operating a resilient microservice.

Failure zones

First, note the zone column in figure 8.6. It lists three distinct values: `europe-west1-d`, `europe-west1-c`, and `europe-west1-b`. Each of these zones represents a distinct data center. If one of those data centers fails, that failure will be isolated and will only affect 33% of your service capacity.

Self-healing

If you select one of those instances, you'll see the option to delete that instance (figure 8.7). Give it a shot!

Name	Creation time	Template	Zone	Internal IP	External IP	Connect
market-data-service-86w0	16 Mar 2018, 13:25:09	market-data-service-template	europe-west1-d	10.132.0.2	35.205.158.180	SSH
market-data-service-l1js	16 Mar 2018, 13:25:10	market-data-service-template	europe-west1-c	10.132.0.5	130.211.86.76	SSH
market-data-service-nrrh	16 Mar 2018, 13:25:10	market-data-service-template	europe-west1-b	10.132.0.4	35.195.39.131	SSH

Figure 8.6 Instances within an instance group

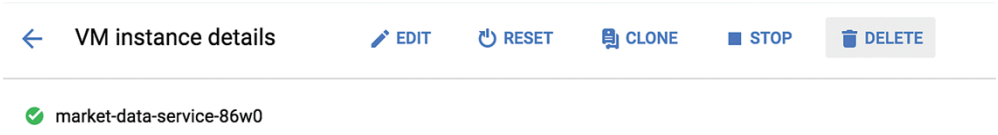


Figure 8.7 Deleting a VM instance

Deleting an instance will cause the instance group to spin up a replacement instance, ensuring that capacity is maintained. If you look at the operation history of the project (Compute Engine > Operations), you'll see that the delete operation results in GCE automatically recreating the instance (figure 8.8).

The instance group will attempt to self-heal in response to any event that results in an instance falling out of service, such as underlying machine failure. You can improve this by adding a health check that also targets your application:

```
gcloud compute health-checks create http api-health-check \
  --port=8080 \
  --request-path="/ping"

gcloud beta compute instance-groups managed set
➥-autohealing \
  market-data-service-group \
  --region=europe-west1 \
  --http-health-check=api-health-check
```

The health check will make HTTP calls to :8080/ping.

Associates the check with your existing instance group

Now, with the addition of the health check, whenever the application fails to reply to it, the virtual machine will be recycled.

ADDING CAPACITY

As your service is now deployed from a template, it's trivial to add more capacity. You can resize the group from the command line:

```
gcloud compute instance-groups managed resize market-data-service-group \
--size=6 \
--region=europe-west1
```

Your new target group size

You also can add autoscaling rules to automatically add more capacity if metrics you observe from your group, such as average CPU utilization, pass a given threshold.

8.3.4 Adding a load balancer

In all that excitement, you forgot to expose your service group to the wild! In this case, GCE will provide your load balancer, which consists of a few interconnected components, as outlined in figure 8.9. The load balancer uses these routing rules, proxies, and maps to forward requests from the outside world to a set of healthy service instances.

✓ Create an instance	market-data-service-86w0		Start: 16 Mar 2018, 13:55:03 End: 16 Mar 2018, 13:55:08
✓ Recreate an instance	market-data-service-86w0		Start: 16 Mar 2018, 13:54:20 End: 16 Mar 2018, 13:54:20
✓ Delete an instance	market-data-service-86w0	morganjbruce@gmail.com	Start: 16 Mar 2018, 13:54:20 End: 16 Mar 2018, 13:54:58

Figure 8.8 Deleting an instance in a group results in the instance being recreated to maintain target capacity (from bottom to top)

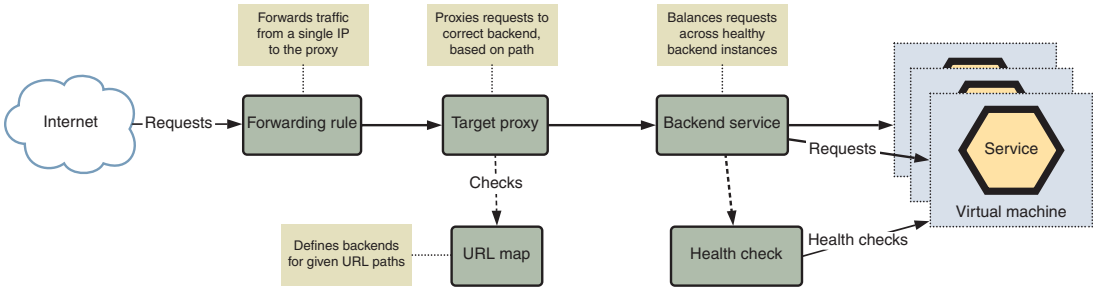


Figure 8.9 Request lifecycle for GCE load balancing

> **NOTE** Managed load balancers are a key feature for all major cloud providers. Outside of these environments, you may come across other software load balancers, such as HAProxy.

First, you'll want to add a backend service, which is the most important component of your load balancer because it's responsible for directing traffic optimally to underlying instances:

```
gcloud compute instance-groups managed set-named-ports \
  market-data-service-group \
  --named-ports http:8080 \
  --region europe-west1

gcloud compute backend-services create \
➥market-data-service \
  --protocol HTTP \
  --health-checks api-health-check \
  --global
```

The name of your backend service

The health check you created earlier

This code creates two entities: a named part, identifying the port your service exposes, and a backend service, which uses the http health check you created earlier to test the health of your service.

Next, you need a URL map and a proxy:

```
gcloud compute url-maps create api-map \
  --default-service market-data-service

gcloud compute target-http-proxies create api-proxy \
  --url-map api-map
```

Creates a URL map for your backend service

Creates a proxy that uses the new URL map

If you had more than one service, you could use the map to route different subdomains to different backends. In this case, the URL map will direct all requests, regardless of URL, to the market-data-service you created earlier.

Finally, you need to create a static IP address for your service and a forwarding rule that connects that IP to the HTTP proxy you've created:

```
gcloud compute addresses create market-data-service-ip \
  --ip-version=IPV4 \
  --global

export IP=`gcloud compute addresses describe market
➥-data-service-ip --global --format json | jq -raw
➥-output '.address'`

gcloud compute forwarding-rules create \
➥api-forwarding-rule \
  --address $IP \
  --global \
  --target-http-proxy api-proxy \
  --ports 80

printenv IP
```

Retrieves the IP address

Adds a forwarding rule that forwards from the IP address to the HTTP proxy

Outputs the IP address you created

This code creates a public IP address and configures requests to that IP to be forwarded to your HTTP proxy and on to your backend service. Once run, these rules take several minutes to propagate. After a wait, try to curl the service—`curl "http://$IP/ping?[1-100]"`. That will start you with 100 requests. If you see the names of different market-data nodes being output to your terminal—terrific—you've deployed a load-balanced microservice!

> **NOTE** In the real world, you'd be unlikely to expose microservices directly to the outside world. You're only doing it here because it makes testing much easier. GCE also supports internal load balancing (https://cloud.google.com/load-balancing/docs/internal/) and Cloud Endpoints, a managed API gateway (https://cloud.google.com/endpoints/).

8.3.5 What have you learned?

In these examples, you've built some of the key elements of a microservice deployment process:

- Using an instance template established a primitive deployment operation, making it simple to add and remove capacity for a given service.
- Combining instance groups, load balancers, and health checks allowed you to autoscale and autoheal your microservice deployment.
- Deploying into independent zones helped you build bulwarks to limit the impact of failures.

But a few things are missing. Your releases weren't predictable, because you pulled your latest code and compiled it on the machine. A new code commit could cause different service instances to be running inconsistent versions of the code (figure 8.10). Without any explicit versioning or packaging, there would be no easy way to roll your code forward or back.

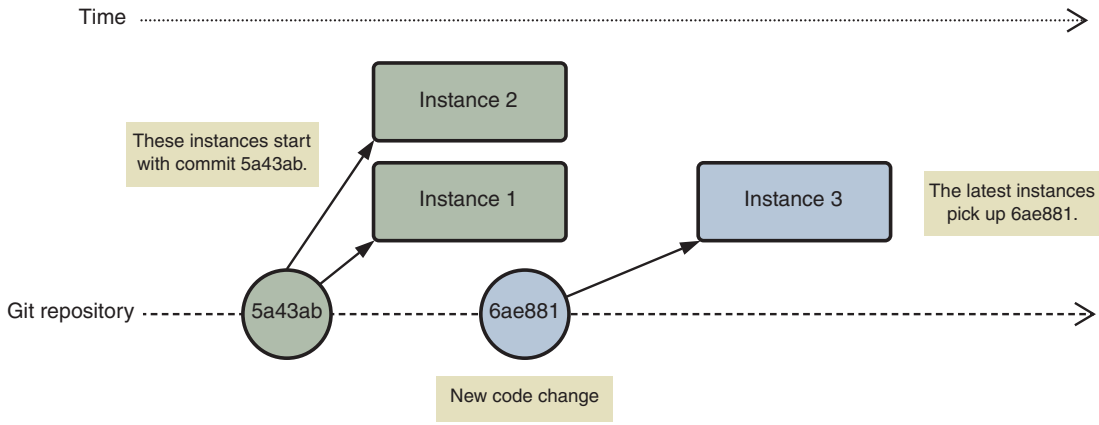


Figure 8.10 Releasing without packaged versions results in deploying inconsistent code.

The process of starting machines was slow because you made pulling dependencies part of startup, rather than baking them into your instance template. This arrangement also meant that the dependencies could become inconsistent across different instances.

Lastly, you didn't automate anything. Not only will a manual process not scale to multiple microservices, but it's likely to be error prone. Over the next few sections and chapters, you can make this much better.

8.4 Building service artifacts

In the earlier deployment example, you didn't package your code for deployment. The startup script that you ran on each node pulled code from a Git repository, installed some dependencies, and started your application. That worked, but it was flawed:

- Starting up the application was slow, as each node performed the same pull and build steps in parallel.
- There was no guarantee that each node was running the same version of your service.

This made your deployment unpredictable—and fragile. To get the benefits you want, you need to build a *service artifact*. A service artifact is an immutable and deterministic package for your service. If you run the build process again for the same commit, it should result in an equivalent artifact.

Most technology stacks offer some sort of deployment artifact (for example, JAR files in Java, DLLs in .NET, gems in Ruby, and packages in Python). The runtime characteristics of these artifacts might differ. For instance, you need to run .NET web services using an IIS server whereas JARs may be self-executable, embedding a server process like Tomcat.

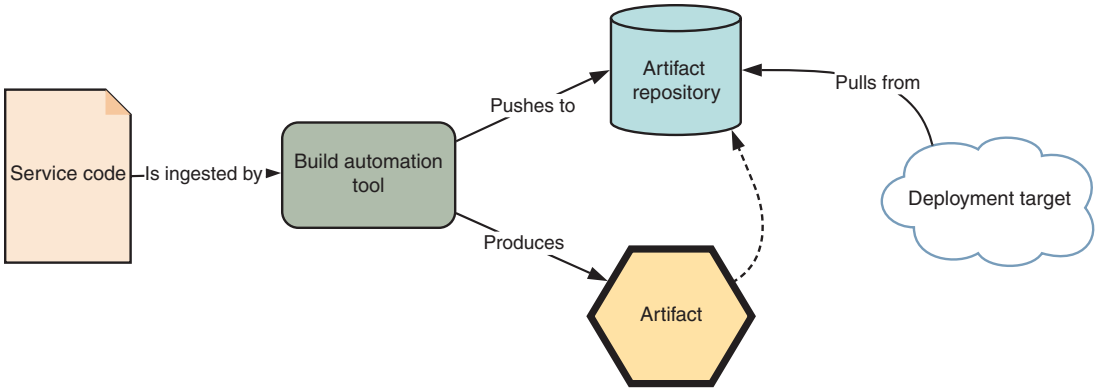


Figure 8.11 An artifact repository stores service artifacts that a build automation tool constructs and you can pull later for deployment.

Figure 8.11 illustrates the artifact construction, storage, and deployment process. Typically, a build automation tool (such as Jenkins or CircleCI) builds a service artifact and pushes it to an artifact repository. An artifact repository might be a dedicated tool—for example, Docker provides a registry for storing images—or a generic file storage tool, such as Amazon S3.

8.4.1 What's in an artifact?

A microservice isn't only code; it'll have many constituent parts:

- Your application code, compiled or not (depending on programming language)
- Application libraries
- Binary dependencies (for example, ImageMagick or libssl) that are installed on the operating system
- Supporting processes, such as logging or cron
- External dependencies, such as data stores, load balancers, or other services

Some of these dependencies, such as application libraries, are explicitly defined. Others may be implicit; for example, language-specific package managers are often ignorant of binary dependencies. Figure 8.12 illustrates these different parts.

An ideal deployment artifact for a microservice would allow you to package up a specific version of your compiled code, specifying any binary dependencies, and provide a standard operational abstraction for starting and stopping that service. This should be environment-agnostic: you should be able to run the same artifact locally, in test, and in production. By abstracting out differences between languages at runtime, you both reduce cognitive load and provide common abstractions for managing those services.

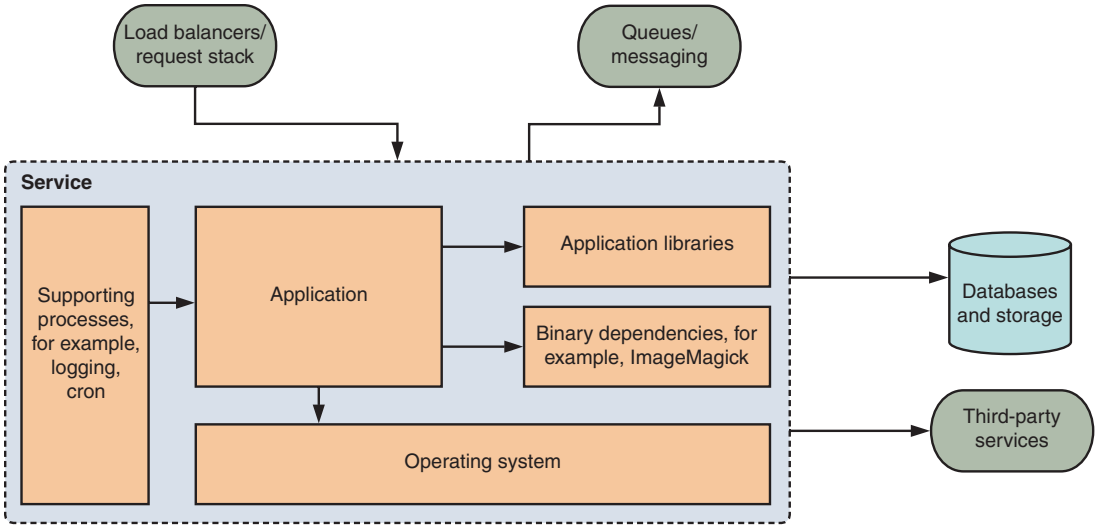


Figure 8.12 A service with internal and external dependencies

8.4.2 Immutability

We've touched on immutability a few times so far—let's take a moment to look at why it matters. An immutable artifact, encapsulating as many dependencies of your service as feasible, gives you the highest possible confidence that the package you tested throughout your deployment pipeline will be the same as what is deployed in production. Immutability also allows you to treat your service instances as disposable—if a service develops a problem, you can easily replace it with a new instance of the last known good state. On GCE, this autohealing process was automated by the instance group you created.

If a build of the same code can result in a different artifact being created—for example, pulling different versions of dependencies—you increase the risk in deployment and the fragility of your code because unintentional changes can be included in a release. Immutability increases the predictability of your system, as it's easier to reason through a system's state and recreate a historic state of your application—crucial for rollback.

Immutability and server management

Immutability isn't only for service artifacts: it's also an important principle for effective virtual server management.

One approach to managing the state of hosts is to apply cumulative changes over time—installing patches, upgrading software, changing configuration. This often means that the ideal current state of a server isn't defined anywhere—there's no known good state that you can use to build new servers. This approach also encourages applying live

(continued)

fixes to servers which, counterintuitively, increases the risk of failure. These servers suffer from *configuration drift*.

This approach might make sense if individual hosts are a scarce resource. In a cloud environment, where individual hosts are cheap to run and replace, immutability is a better option. Instead of managing hosts, you should build them using a base template that itself is version controlled. Rather than updating older hosts, you replace them with hosts you build from a new version of a base template.

8.4.3 Types of service artifacts

Many languages have their own packaging mechanism, and this heterogeneity makes deployment more complex when working with services written in different languages. Your deployment tools need to treat differently the interface that each deployment package provides to get it running on a server (or to stop it).

Better tooling can reduce these differences, but technology-specific artifacts tend to work at too low an abstraction level. They primarily focus on packaging code, rather than the broader nature of application requirements:

- They lack a runtime environment. As you saw earlier, you needed to separately install other dependencies to run your service.
- They don't provide any form of resource management or isolation, which makes it challenging to adequately run multiple services on a single host.

Luckily, you've got a few options: operating system packages, server images, or containers (figure 8.13).

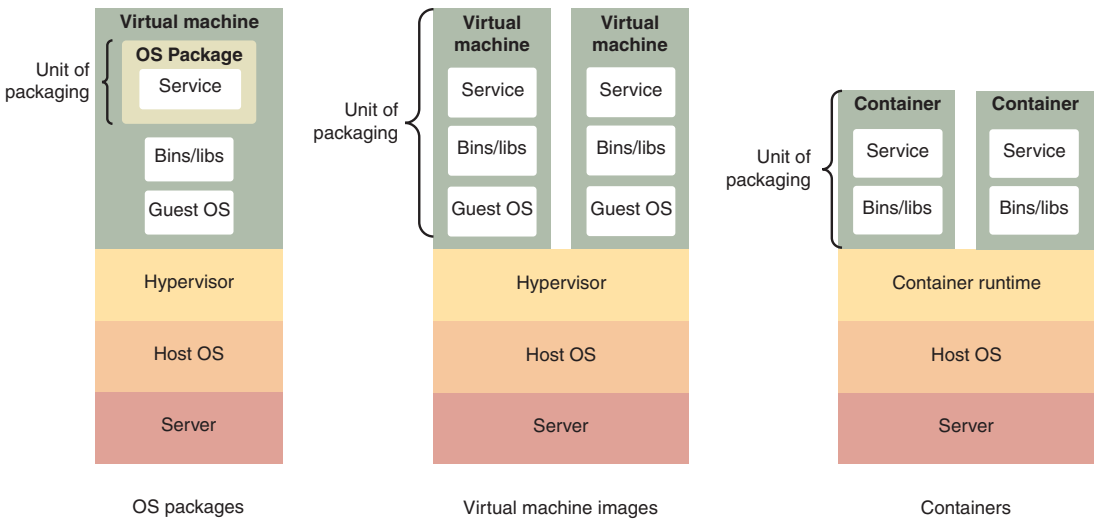


Figure 8.13 The structure of different service artifact types

Operating system packages

You could use the packaging format of your target operating system, such as `apt` or `yum` in Linux. This approach standardizes the installation of an artifact, regardless of contents, as you can use standard operating system tools to automate the installation process. When you start a new host, you can pull the appropriate version of your service package. In addition, packages can specify dependencies on other packages—for example, a Rails application might specify dependencies on common Linux packages, such as libxml, libmagic, or libssl.

> **NOTE** If you're interested in exploring this approach further, you could try to build a deb package using py2deb (github.com/paylogic/py2deb) and the example service.

The OS package approach has three weaknesses:

- It adds a different infrastructure requirement: you'll need to host and manage a package repository.
- These packages are often tightly coupled to a particular operating system, reducing your flexibility in using different deployment targets.
- The packages aren't at quite the right level of abstraction, as you still need to execute them in a host environment.

Server images

In typical virtualized environments, each server you run is built from an image, or template. The instance template you built in section 8.3 is an example of a server image.

You can use this image itself as a deployment artifact. Rather than pulling a package onto a generic machine, you could instead bake a new image for each version of your service that you want to deploy. A typical bake process has four steps:

1. Select a template image as the basis for the new image.
2. Start a VM based on the template image.
3. Provision the new VM to the desired state.
4. Take a snapshot of the new VM and save it as a new image template.

You can try that out using Packer.

> **TIP** You'll need to set up Packer to authenticate with GCE. You can find directions for that in the Packer documentation: https://www.packer.io/docs/builders/googlecompute.html.

First, save the following configuration file as instance-template.json.

Listing 8.1 The instance-template.json file

```
{
  "variables": {          <-- User-provided variables
    "commit": "{{env `COMMIT`}}"
```

```
  },
  "builders":                                  Defines how an image will be built
  [
    {
      "type": "googlecompute",
      "project_id": "market-data-1",
      "source_image_family": "debian-9",
      "zone": "europe-west1-b",
      "image_name": "market-data-service-{{user `commit`}}",
      "image_description": "image built for market-data
➥-service {{user `commit`}}",
      "instance_name": "market-data-service-{{uuid}}",
      "machine_type": "n1-standard-1",
      "disk_type": "pd-ssd",
      "ssh_username": "debian",
      "startup_script_file": "startup-script.sh"
    }
  ]
}
```

Now, run the `packer build` command from within the chapter-8/market-data directory:

```
packer build \
-var "commit=`git rev-parse head`" \          Gets the latest commit hash
instance-template.json                       Uses the instance template defined in listing 8.1
```

If you watch the console output, it'll reflect the four steps I outlined above: using the GCE API, Packer will start an instance, run the startup script, and save the instance as a new template image, tagged with the source Git commit. You can use the Git commit to explicitly distinguish different versions of your code.

> **NOTE** In this case, you still pulled code directly from Git to your machine image. In complied languages such as Java, compilation into an executable should be a separate step that a build automation tool executes.

This approach builds an immutable, predictable, and self-contained artifact. This immutable server pattern, combined with a configuration tool like Packer, allows you to store a reproducible base state as code.

It has a few limitations:

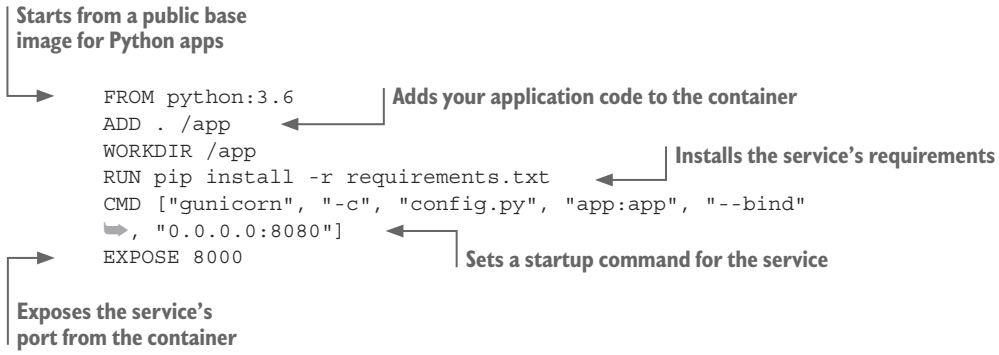
- Images are locked to one cloud provider, making them nontransferable to other providers as well as to developers who want to recreate the deployed artifact on their machines.
- Image builds are often slow because of the lengthy time it takes to spin up a machine and take a snapshot.
- It's not easy for you to use for a multiple service-per-host model.

CONTAINERS

Instead of distributing entire machines, containerization tools, such as Docker or rkt, provide a more lightweight approach to encapsulating an application and its dependencies. You can run multiple containers on one machine, isolated from each other but with lower resource overhead than a virtual machine because they share the kernel of one operating system. They avoid the overhead of virtualizing the disk and guest operating system of each virtual machine.

Try a quick example using Docker. (You can find instructions for installing Docker on the Docker website: https://docs.docker.com/install/.) You build a Docker image from a Dockerfile. Add the following file to the chapter-8/market-data folder.

Listing 8.2 Dockerfile for market-data service



Then, use the `docker` command-line tool to build the container:

```
$ docker build -t market-data:`git rev-parse head` .
Sending build context to Docker daemon 71.17 kB
Step 1/3 : FROM python:3.6
 ---> 74145628c331
Step 2/3 : ADD . /app
 ---> bb3608d5143f
Removing intermediate container 74c250f83f8c
Step 3/3 : WORKDIR /app
 ---> 7a595179cc39
Removing intermediate container 19d3bffa4d2a
Successfully built 7a595179cc39
```

This will build a container image and tag it with the name market-data:<commit ID>.

Now that you've built an image for the application, you can run it locally. Try it out:

```
$ docker run -d -p 8080:8080 market-data:`git rev-parse head`
```

You'll see startup logs from gunicorn in your terminal. If you like, try to curl the service on port 8000. You probably noticed that startup and build time for the container was significantly faster than the virtual machines on GCE. This is one of the key benefits of using containers.

In a few short steps, you can run this container image on GCE. First, you need to push the image to a container registry. Luckily, GCE already provides one:

```
TAG="market-data:$(git rev-parse head)"
PROJECT_ID=<your-project-id>
```

Replace with your GCE project ID

```
docker tag $TAG eu.gcr.io/$PROJECT_ID/$TAG
```

Renames the Docker image you created

```
gcloud docker -- push eu.gcr.io/$PROJECT_ID/$TAG
```

Pushes the Docker image to GCE

This registry acts as an artifact repository where you can store your Docker images for later use. After the push has completed, start an instance running this container:

```
gcloud beta compute instances create-with-container \
  market-data-service-c \
  --container-image eu.gcr.io/$PROJECT_ID/$TAG
  --tags api-service
```

You set the project ID and tag variables in the previous example.

Success! You've deployed a container, and you've seen firsthand that it provides a more flexible—and easy-to-use—abstraction than a VM image.

As well as acting as a packaging mechanism, a container provides a runtime environment that isolates execution, effectively easing the operation of diverse containers on a single machine. This is compelling because it provides sane abstractions above individual hosts.

Unlike virtual machine images, container images are portable; you can run the same container on any infrastructure that supports the container runtime. This eases deployment in scenarios where multiple deployment targets are required, such as companies that run workloads in both cloud and on-premise environments. It also simplifies local development; running multiple containers on a typical developer machine is much more manageable than building and managing multiple virtual machines.

8.4.4 Configuration

The service's configuration is likely to differ based on deployment environment (staging, dev, production, and so on). For that and other reasons, you can't represent all elements of a service within an artifact:

- You can't distribute secrets or sensitive configuration data, such as database passwords, in clear text or source control. You may want to retain the ability to change them independently of a service deployment (for example, as part of automated credential rotation, or worse, in the event of a security breach).
- Environment-specific configuration data, such as database URLs, log levels, or third-party service endpoints, will vary.

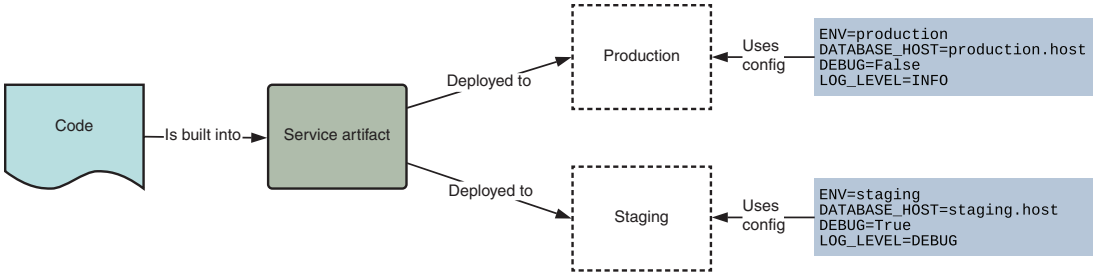


Figure 8.14 Service configuration that differ by environment

The third principle of *The Twelve-Factor App* manifesto (12factor.net) states that you should strictly separate deployment configuration from code and provide it as environment variables (figure 8.14). In practice, the deployment mechanism you choose will define how you store and provide environment-specific configuration. We recommend storing configuration in two places:

- In source control, version-controlled alongside the service, for nonsensitive configuration (These are commonly stored in .env files.)
- A separate, access-restricted "vault" for secret information (such as HashiCorp's perfectly named www.vaultproject.io)

The process that starts a service artifact should pull this configuration and inject it into the application's environment.

Unfortunately, managing configuration separately can increase risk, as people may make changes to production outside of your immutable artifacts, affecting the predictability of your deployments. You should err on the side of restraint and attempt to include as much configuration as possible within your artifacts and rely on the speed and robustness of your deployment pipeline for rapidly changing configuration.

8.5 Service to host models

In this section, we'll review three common models for deploying services to underlying hosts: single service to host, multiple services to host, and container scheduling.

8.5.1 Single service to host

In earlier examples, we've used a one-to-one relationship between service and underlying host. This approach is easy to understand and provides a clear and explicit isolation between the resource needs and runtime of multiple services. Figure 8.15 illustrates this approach. Although the analogy is somewhat cruel, using this model lets you treat servers as cattle: indistinguishable units that you can start, stop, and destroy on command.

This model isn't perfect. Sizing virtual instances appropriately for the needs of each service requires ongoing effort and evaluation. If you're not running in the cloud, you may run into the limits of your data center or virtualization solution. And as we touched on earlier, virtual machine startup time is comparatively slow, often taking several minutes.

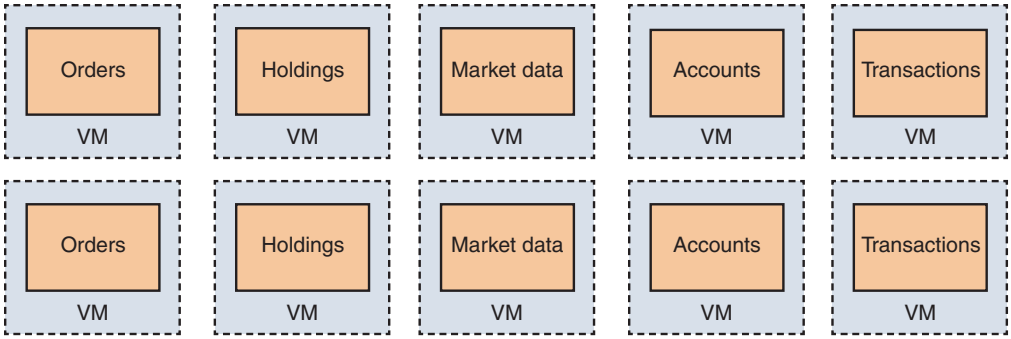


Figure 8.15 A single service to host model

8.5.2 Multiple static services per host

It's possible to run multiple services per host (figure 8.16). In the static variant of this model, the allocation of services to hosts is manual and static; the service owner makes a conscious choice, predeployment, about where each service should be run.

At first glance, this approach might seem desirable. If obtaining new hosts is costly or hosts are scarce, then the easiest route to production would be to maximize usage of your existing, limited number of hosts.

But this approach has several weaknesses. It increases coupling between services: deploying multiple services to a host leads to coupling between services, eliminating your desire to release services independently. It also increases the complexity of dependency management: if one service needs package v1.1, but another needs v2.0, the difference is difficult to reconcile. It becomes unclear which service owns the deployment environment—and therefore which team has responsibility for managing that configuration.

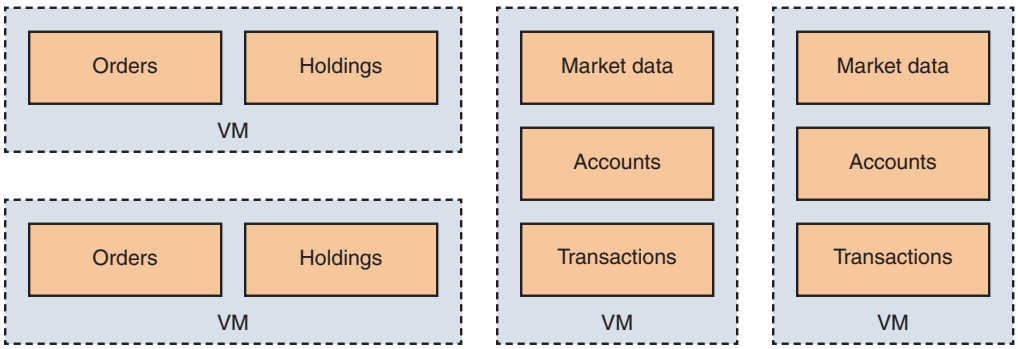


Figure 8.16 A single virtual machine can potentially run multiple services.

This approach also leads to challenges in monitoring and scaling services independently. One noisy service on a box might adversely impact other services, and it can be difficult to monitor the resource usage (CPU, memory) of services independently.

8.5.3 Multiple scheduled services per host

It'd be even simpler if you could avoid thinking about the underlying hosts that run your services altogether and focus entirely on the unique runtime environment of each application. This was the initial promise of platform as a service (Paas) solutions, such as Heroku. A PaaS provides tools for deploying and running services with minimal operational configuration or exposure to underlying infrastructural resources. Although these platforms are easy to use, they often strike a difficult balance between automation and control—simplifying deployment but removing customization from the developer's hands—as well as being highly vendor specific.

Containers provide a more elegant abstraction:

- An engineer can define and distribute a holistic application artifact.
- A virtual machine can run multiple individual containers, isolating them from each other.
- Containers provide an operational API that you can automate using higher level tooling.

These three facets enable scheduling, or orchestration, of containers. A container scheduler is a software tool that abstracts away from underlying hosts by managing the execution of atomic, containerized applications across a shared pool of resources. Typically, a scheduler consists of a master node that distributes application workloads to a cluster of worker nodes. Developers, or a deployment automation tool, send instructions to this master node to perform container deployments. Figure 8.17 illustrates this setup.

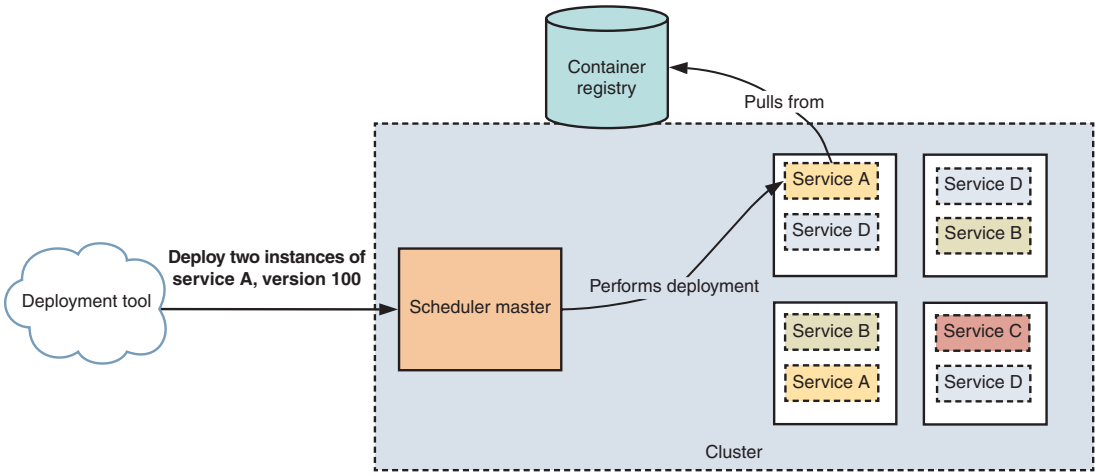


Figure 8.17 A container scheduler executes containers across a cluster of nodes, balancing the resource needs of those nodes.

ADVANTAGES OF A SCHEDULING MODEL

Unlike the multiple static services per host model, the allocation of services in a scheduler model is dynamic and depends on the resources (CPU, disk, or memory needs) defined for each application. This avoids the pitfalls of the static model, as the scheduler aims to continually optimize resource usage within the cluster of nodes, while the container model preserves service independence.

By using a scheduler as a deployment platform, a service developer can focus on the environment of their service in isolation from the underlying needs of machine configuration. Operations engineers can focus on running the underlying scheduler platform and defining common operational standards for running services.

CONTAINER SCHEDULERS ARE COMPLEX

Container schedulers such as Kubernetes are complex pieces of software and require significant expertise to operate, especially because the tools themselves are relatively new. We strongly recommend them as the ideal deployment platform for microservices, but only if you can use a managed scheduler (such as Google's Kubernetes Engine) or have the operational resources to run it in-house. If not, the single service per host model, combined with container artifacts, is a great and flexible fallback.

8.6 Deploying services without downtime

So far, you've only deployed market-data once. But in a real application, you'll be deploying services often. You need to be able to deploy new versions without downtime to maintain overall application stability. Every service will rely on others to be up and running, so you also need to maximize the availability of every service.

Three common deployment patterns are available for zero-downtime deployments:

- *Rolling deploy*—You progressively take old instances (version N) out of service while you bring up new instances (version N+1), ensuring that you maintain a minimum percentage of capacity during deployment.
- *Canaries*—You add a single new instance[1] into service to test the reliability of version N+1 before continuing with a full rollout. This pattern provides an added measure of safety beyond a normal rolling deploy.
- *Blue-green deploys*—You create a parallel group of services (the green set), running the new version of the code; you progressively shift requests away from the old version (the blue set). This can work better than canaries in scenarios where service consumers are highly sensitive to error rates and can't accept the risk of an unhealthy canary.

All of these patterns are built on a single primitive operation. You're taking an instance, moving it to a running state in an environment, and directing traffic toward it.

[1] In larger service groups, for example, >50 instances, you may need more than one canary to get representative feedback.

8.6.1 Canaries and rolling deploys on GCE

It's always better when you can see things in action. You can deploy a new version of market-data to GCE. First, you'll want to create a new instance template. You can use the container you built and pushed in section 8.4.3:

```
gcloud beta compute instance-templates create-with-container \
  market-data-service-template-2 \
  --container-image eu.gcr.io/$PROJECT_ID/$TAG
  --tags=api-service
```

Then, initiate a canary update:

```
gcloud beta compute instance-groups managed rolling-action start-update \
  market-data-service-group \
  --version template=market-data-service-template \
  --canary-version template=market-data-service
➥-template-2,target-size=1 \
  --region europe-west1
```

Rolls out one instance of your new template

GCE will add the canary instance to the group and the backend service to begin receiving requests (figure 8.18). It'll take a few minutes to come up. You also can see this on the GCE console (figure 8.19; Compute Engine > Instance Groups).

If you're happy, you can proceed with the rolling update:

```
gcloud beta compute instance-groups managed rolling-action start-update \
  market-data-service-group \
  --version template=market-data-service-template-2 \
  --region europe-west1
```

The speed at which this update occurs depends on how much capacity you want to maintain during the rollout. You also can elect to surge beyond your current capacity during rollout to ensure the target number of instances is always maintained. Figure 8.20 illustrates the stages of a rollout across three instances.

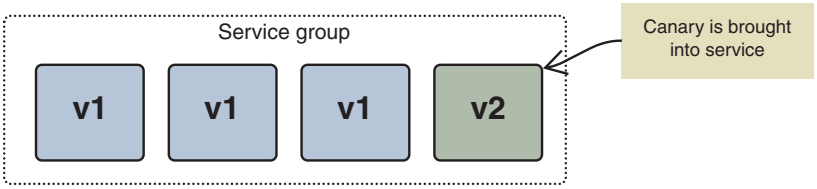


Figure 8.18 You add a new canary to the group.

Name	Creation time	Template	Zone	Internal IP	External IP	Connect
market-data-service-fwqr	17 Mar 2018, 15:18:06	market-data-service-template	europe-west1-d	10.132.0.4	35.205.51.221	SSH
market-data-service-gfp9	18 Mar 2018, 09:45:57	market-data-service-template-2	europe-west1-b	10.132.0.3	35.205.138.120	SSH
market-data-service-rp98	16 Mar 2018, 14:19:24	market-data-service-template	europe-west1-c	10.132.0.8	104.199.107.141	SSH
market-data-service-t8pv	16 Mar 2018, 14:19:24	market-data-service-template	europe-west1-b	10.132.0.7	35.195.188.36	SSH

Figure 8.19 Your instance group contains your original instances plus a canary instance of a new version.

If you were unhappy, you could roll back the canary:

```
gcloud beta compute instance-groups managed rolling-action start-update \
  market-data-service-group \
  --version template=market-data-service-template \
  --region europe-west1
```

← The original version

The command for a rollback is identical to a rollout, but it goes to a previous version. In the real world, rollback may not be atomic. For example, the incorrect operation of new instances may have left data in an inconsistent state, requiring manual intervention and reconciliation. Releasing small change sets and actively monitoring release behavior will limit the occurrence and extent of these scenarios.

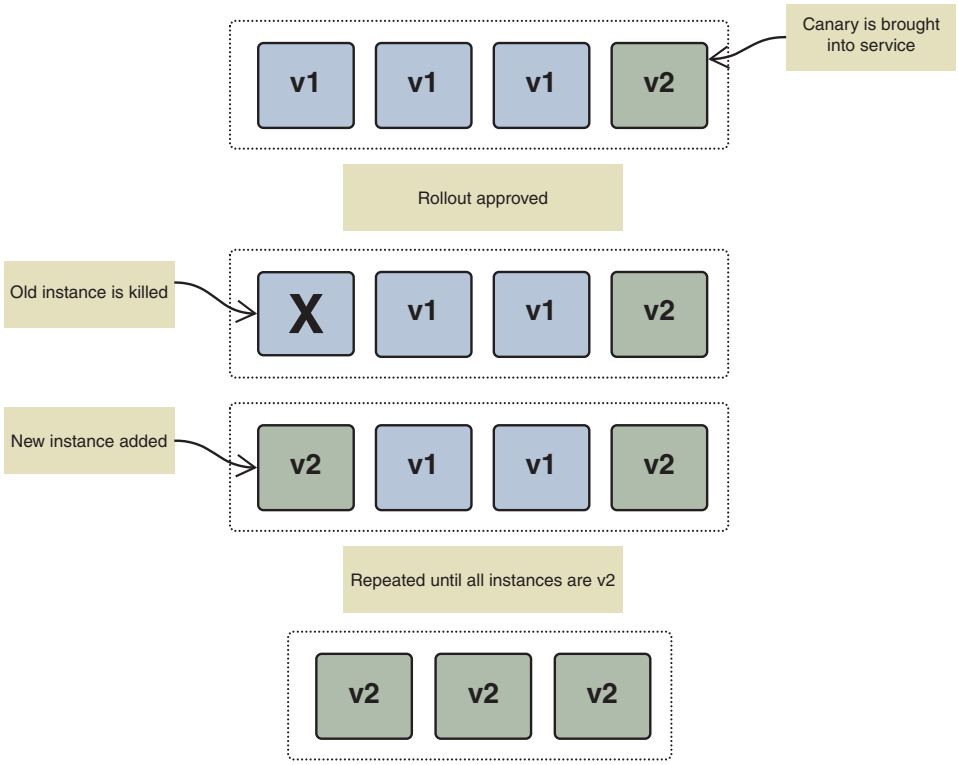


Figure 8.20 Stages of a rolling deploy, beginning with a canary instance

We've covered a lot of ground in this chapter: you've deployed manually to a cloud provider, packaged a service as a container and a virtual machine, and practiced safe rollout patterns. By building immutable service artifacts and performing safe, downtime-free deployments, you're well on your way to building a deployment process that works reliably across multiple services. Ultimately, the more stable, reliable, and seamless your deployment process, the easier it is to standardize services, release new services more rapidly, and deliver valuable new features without friction or risk.

Summary

- Deploying new applications and changes must be standardized and straightforward to avoid friction in microservice development.
- Microservices can run anywhere, but ideal deployment platforms need to support a range of features, including security, configuration management, service discovery, and redundancy.
- You deploy a typical service as a group of identical instances, connected by a load balancer.
- Instance groups, load balancers, and health checks enable autohealing and autoscaling of deployed services.
- Service artifacts must be immutable and predictable to minimize risk, reduce cognitive load, and simplify deployment abstractions.
- You can package services as language-specific packages, OS packages, virtual machine templates, or container images.
- Being able to add/remove a single instance of a microservice is a fundamental primitive operation that you can use to compose higher level deployment.
- You can use canaries or blue-green deployments to reduce the impact of unexpected defects on availability.

9 Deployment with containers and schedulers

This chapter covers

- Using containers to package a microservice into a deployable artifact
- How to run a microservice on Kubernetes, a container scheduler
- Core Kubernetes concepts, including pods, services, and replica sets
- Performing canary deployments and rollbacks on Kubernetes

Containers are an elegant abstraction for deploying and running microservices, offering consistent cross-language packaging, application-level isolation, and rapid startup time.

In turn, container schedulers provide a higher level deployment platform for containers by orchestrating and managing the execution of different workloads across a pool of underlying infrastructure resources. Schedulers also provide (or tightly integrate with) other tools—such as networking, service discovery, load balancing, and configuration management—to deliver a holistic environment for running service-based applications.

Containers aren't a requirement for working with microservices. You can deploy services using many methods such as using the single service per VM model we outlined in the previous chapter. But together with a scheduler, containers provide a particularly elegant and flexible approach that meets our two deployment goals: speed and automation.

Docker is the most commonly used container tool, although other container runtimes are available, such as CoreOS's rkt. An active group—the Open Container Initiative—is also working to standardize container specifications.

Some of the popular container schedulers available are Docker Swarm, Kubernetes, and Apache Mesos; different tools and distributions are built on top of those platforms. Of these, Kubernetes, Google's open source container scheduler, has the widest mindshare and has garnered significant implementation support from other organizations, such as Microsoft, and the open source community. Because of this popularity and the ease of setting up a local installation, we'll use Kubernetes in this book.

We significantly increased deployment velocity at our own company using Kubernetes. Whereas our previous approach could take several days to get a new service deployment working smoothly, with Kubernetes, any engineer can now deploy a new service in a few hours.

In this chapter, you'll get your hands dirty with Docker and Kubernetes. You'll use Docker to build, store, and run a container for a new service at SimpleBank. And you'll take that service to production using Kubernetes. Along with these examples, we'll illustrate how a scheduler executes and manages different types of workloads and how familiar production concepts map to a scheduler platform. We'll also examine the high-level architecture of Kubernetes.

9.1 Containerizing a service

Let's jump right in! Over the course of this chapter, your goal will be to take one of SimpleBank's Python services—market-data—and get it running in production. You can find a starting point for this service in the book's repository on Github (http://mng.bz/7eN9). Figure 9.1 illustrates the process that will occur. Docker packages service code into a container image, which is stored in a repository. You'll use deploy instructions to tell a scheduler to deploy and operate the packaged service on a cluster of underlying hosts.

As you know, a successful deployment is about more than running a single instance. For each new version, you want to build an artifact that you can deploy multiple times for redundancy, reliability, and horizontal scaling. In this section, you'll learn how to do the following:

- Build an image for a service
- Run multiple instances—or containers—of your image
- Push your image to a shared repository, or registry

First things first: if you're going to ship this, you need to figure out how to put it in a box. For this section, you'll need to have Docker installed. You can find up-to-date instructions online at https://docs.docker.com/install.

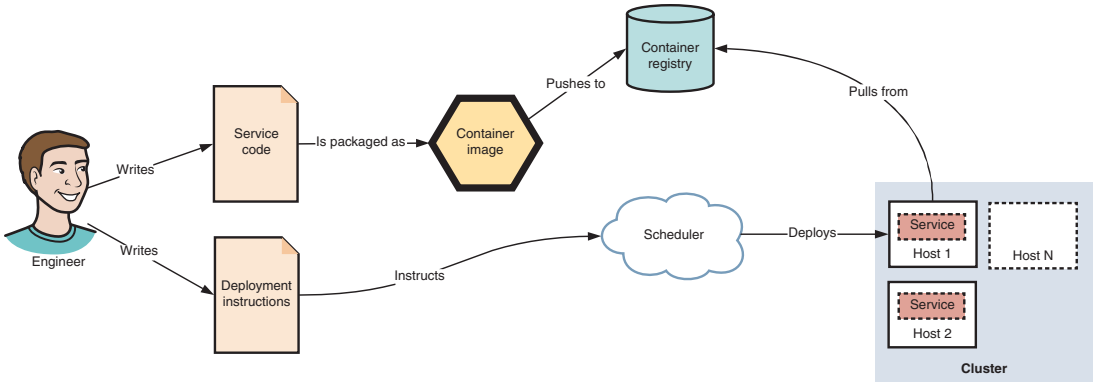


Figure 9.1 The process of deploying service code to a cluster scheduler

9.1.1 Working with images

To package an application into a container, you need to build an image. The image will include the file system that your application needs to run—code and dependencies—and other metadata, such as the command that starts your application. When you run your application, you'll start multiple instances of this image.

Most powerfully, images can inherit from other images. That means your application images can inherit from public, canonical images for different technology stacks, or you can build your own base images to encapsulate standards and tools you use across multiple services.

To get a feel for working with images, fire up the command line and try to pull a publicly available Docker image:

```
$ docker pull python:3.6
3.6: Pulling from library/python
ef0380f84d05: Pull complete
24c170465c65: Pull complete
4f38f9d5c3c0: Pull complete
4125326b53d8: Pull complete
35de80d77198: Pull complete
ea2eeab506f8: Pull complete
1c7da8f3172e: Pull complete
e30a226be67a: Pull complete
Digest:
     sha256:210d29a06581e5cd9da346e99ee53419910ec8071d166ad499a909c49705ba9b
Status: Downloaded newer image for python:3.6
```

Pulling an image downloads it to your local machine, ready for you to run. In this case, you pulled a Python image from Docker Hub, the default public registry (or repository) for Docker images. Running the following command will start an instance of that image, placing you at a Python interactive shell inside your new container:

```
$ docker run --interactive --tty python:3.6
Python 3.6.1 (default, Jun 17 2017, 06:29:46)
```

```
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You should note a few things here. The `--interactive` (or `-i`) flag indicates that the container should be interactive, accepting input from STDIN, whereas the `--tty` (or `-t`) flag connects a terminal for user input to the Docker container. When you started the container, it executed the default command set within the image. You can check what that is by inspecting the image metadata:

```
$ docker image inspect python:3.6 --format="{{.Config
➥.Cmd}}"
[python3]
```

The Docker image configuration is output as JSON, which you can parse using Go text templates.

You can instruct Docker to execute other commands inside your container; for example, to enter the container at an OS shell, rather than Python, you could suffix the command you used to start the image instance with `bash`.

When you watched the output of your earlier pull command, you might've noticed that Docker downloaded multiple items, each identified by a hash—these are layers. An image is a union of multiple layers; when you build an image, each command you run (`apt-get update`, `pip install`, `apt-get install -y`, and so on) creates a new layer. You can list the commands that went into building the python:3.6 image:

```
$ docker image history python:3.6
```

Each line that this script returns represents a different command used to construct the python:3.6 image. In turn, some of those layers were inherited from another base image. Commands defined in a Dockerfile specify the layers in an image using a lightweight domain-specific language (DSL). If you look at the Dockerfile for this image—you can find it on Github (http://mng.bz/JxDj)—you'll notice the first line:

```
FROM buildpack-deps:jessie
```

This specifies that the image should inherit from the `buildpack-deps:jessie` image. If you follow that thread on Docker Hub, you can see that your Python container has a deep inheritance hierarchy that installs common binary dependencies and the underlying Debian operating system. This is detailed in figure 9.2.

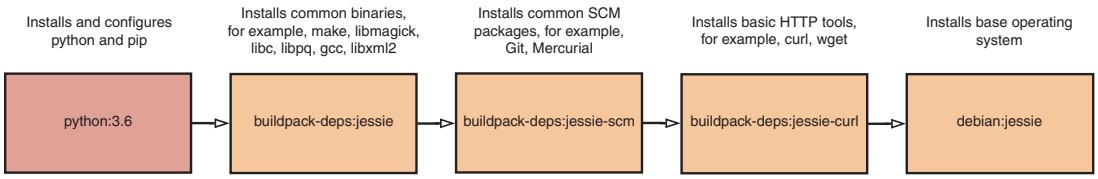


Figure 9.2 The inheritance hierarchy of images used to construct the python:3.6 container on Docker Hub

Other container ecosystems use different mechanisms—for example, rkt uses the `acbuild` command-line tool—but the end outcome is similar.

As well as enabling reusability, image layers optimize launch times for containers. If a parent layer is shared between two derivative images on one machine, you only need to pull it from a registry once, not twice.

9.1.2 Building your image

This Python image is a good starting point for you to build your own application image. Let's take a quick look at the dependencies of the market-data service:

1. It needs to run on an operating system—any distribution of Linux should do.
2. It relies on Python 3.6.x.
3. It installs several open source dependencies from PyPI using pip, a Python package manager.

In fact, this list maps quite closely to the structure of the image that you're going to build. Figure 9.3 illustrates the relationship between your image and the Python base image you've worked with so far.

To build this image, first you need to create a Dockerfile in the root of the market-data service directory. This should do the trick:

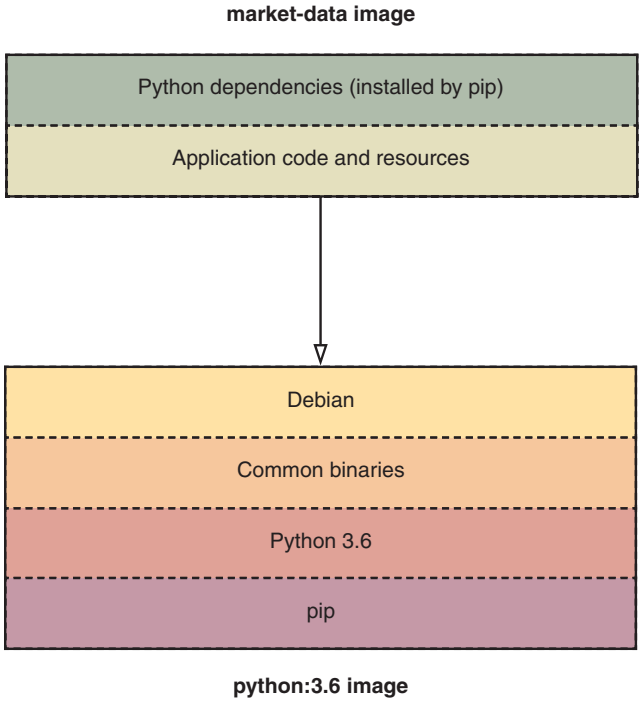
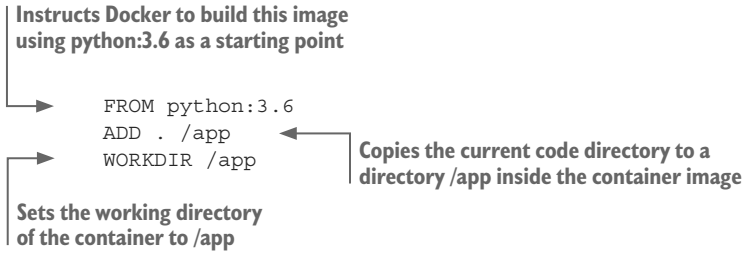


Figure 9.3 The structure of your market-data container image and its relationship to the python:3.6 base image

Listing 9.1 Dockerfile for application container



That's not quite the whole picture, but try building this image and see what it looks like. You can use the `docker build` command to create an image from a Dockerfile:

```
$ docker build -t market-data:first-build .
Sending build context to Docker daemon 71.17 kB
Step 1/3 : FROM python:3.6
 ---> 74145628c331
Step 2/3 : ADD . /app
 ---> bb3608d5143f
Removing intermediate container 74c250f83f8c
Step 3/3 : WORKDIR /app
 ---> 7a595179cc39
Removing intermediate container 19d3bffa4d2a
Successfully built 7a595179cc39
```

This builds an image with the name `market-data` and the tag `first-build`. We'll make more use of tagging later in this chapter. Check that you can start the container and it contains the files you expect:

```
$ docker run market-data:first-build bash -c 'ls'
Dockerfile
app.py
config.py
requirements.txt
```

The output of this command should match the contents of the market-data directory. If it did for you, that's great! You've built a new container and added some files—only a few more steps until you have it running an application.

Although you've added your application code, you still need to pull down dependencies and start the application up. First, you can use a `RUN` command within your Dockerfile to execute an arbitrary shell script:

```
RUN pip install -r requirements.txt
```

If you recall, the pip tool itself was installed as part of the python base image. If you were working with Ruby or Node, at this point you might call `bundle install` or `npm install`; if you were working with a compiled language, you might use a tool like `make` to produce compiled artifacts.

> **NOTE** For more complex applications, especially for compiled languages, you may want to use the builder pattern or multistage builds to separate your development and runtime Docker images: http://mng.bz/LMFr.

Next, you need to set the command that'll be used to start your application. Add another line to your Dockerfile:

```
CMD ["gunicorn", "-c ", "config.py", "app:app"]
```

And a final touch: you need to instruct Docker to expose a port to your app. In this case, your Flask app expects traffic on port 8000. Putting that together, you get your final Dockerfile, as shown in the following listing. You should build the image again, this time tagging it as `latest`.

Listing 9.2 Complete Dockerfile for the market-data service

```
FROM python:3.6
ADD . /app
WORKDIR /app
RUN pip install -r requirements.txt
CMD ["gunicorn", "-c ", "config.py", "app:app"]
EXPOSE 8000
```

Public images and security

The `python:3.6` image we've used so far is derived from `debian:jessie`, which has a reputation for being well maintained and rapidly releasing patches to disclosed vulnerabilities.

But, as when working with any software, it's important to be aware that using public Docker images potentially increases your security risk. Many images, particularly those that aren't officially maintained, aren't regularly patched or updated, which can increase the threat surface of your system.

If in doubt, security scanning tools, such as Clair (https://github.com/coreos/clair), exist for analyzing the security stance of Docker containers. You can use these on an ad-hoc basis or integrate them into your continuous integration pipeline.

Maintaining your own base images is also an option but does involve an extra time investment. Deciding to take this route requires careful consideration of your team's capabilities and security expertise.

9.1.3 *Running containers*

Now that you've built an image for the application, you can run it. Try it out:

```
$ docker run -d -p 8000:8000 --name market-data market-data:latest
```

This command should return a long hash to the terminal. That's the ID of your container—you've started it in detached mode, rather than in the foreground. You've also used the `-p` flag to map the container port so it's accessible from the Docker host. If

you try and call the service—it has a health-check endpoint at /ping—you should get a successful response:

```
$ curl -I http://{DOCKER_HOST}:8000/ping
HTTP/1.0 200 OK
Content-Type: text/plain
Server: Werkzeug/0.12.2 Python/3.6.1
```

DOCKER_HOST will depend on how you've installed Docker in your environment.

You could easily run multiple instances and balance between them. Try a basic example, using NGINX as a load balancer. Luckily you can pull an NGINX container from the public registry—no hard work to get that running. Figure 9.4 illustrates the containers you're going to run.

First, start up three instances of the market-data service. Run the code below in your terminal:

```
$ docker network create market-data
```

Creates a container network named market-data

```
$ for i in {1..3}
  do
    docker run -d \
      --name market-data-$i \
      --network market-data \
      market-data:latest
  done
```

Runs three containers based on the market-data:latest image you created earlier

If you run `docker ps -a`, you'll see three instances of the market-data service up and running.

TIP Instead of working on the command line, you could use Docker Compose to define a set of containers declaratively—in a YAML file—and run them. But in this case, it's better to start at a lower level so you can see what's happening.

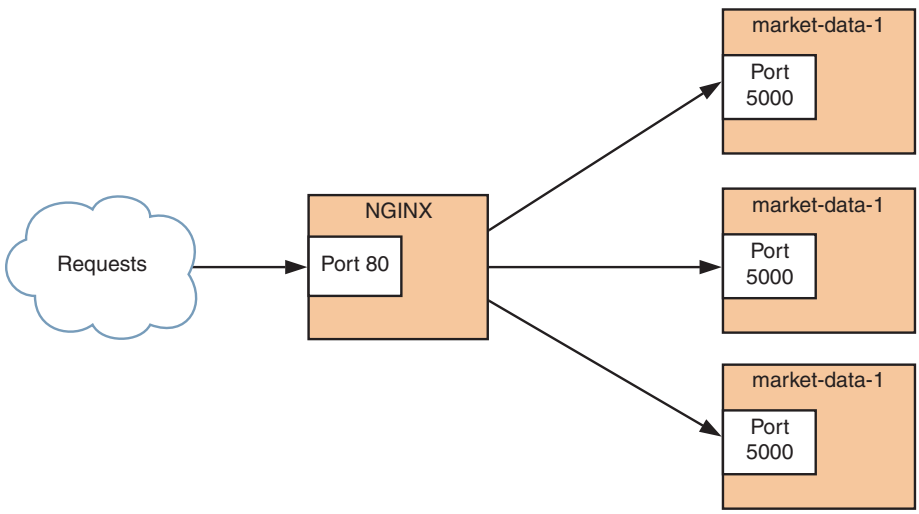


Figure 9.4 NGINX load-balances requests made to it between three market-data containers

Unlike earlier, you didn't map each container's port to the host machine. That's because you'll only access these containers through NGINX. Instead, you created a network. Running on the same network will allow the NGINX container to easily discover your market-data instances, using the container name as a host name.

Now, you can set up NGINX. Unlike before, you're not going to build your own image; instead, you'll pull the official NGINX image from the public Docker registry. First, you'll need to configure NGINX to load balance between three instances. Create a file called nginx.conf using the following code.

Listing 9.3 nginx.conf

```
upstream app {
    server market-data-1:8000;
    server market-data-2:8000;
    server market-data-3:8000;
}

server {
    listen 80;

    location / {
        proxy_pass http://app;
    }
}
```

You configure the upstream application using the container name and port.

The NGINX server proxies requests received on port 80 to the upstream application.

Then you can start an NGINX container. You'll use the `volume` flag (or `-v`) to mount your new nginx.conf file into the container, sharing it with the local filesystem. This is useful for sharing secrets and configurations that aren't—or shouldn't be—built into a container image, such as encryption keys, SSL certificates, and environment-specific configuration files. In this case, you avoid having to build a separate container to include a single new configuration file. Start the container by entering the following:

```
$ docker run -d --name=nginx \
--network market-data \
--volume `pwd`/nginx.conf:/etc/nginx/conf.d/
➥default.conf \
-p 80:80 \
nginx
```

Runs on the same network as your market-data containers

Maps the container port to port 80 on your host machine

Mounts your configuration into an appropriate location inside the container

And that should do the trick. Curling `http://localhost/ping` should return the hostname—by default, the container ID—of the container instance responding to that request. NGINX will round-robin requests across the three nodes to (naively) balance load across your instances.

9.1.4 Storing an image

Good work so far—you've built an image and you've seen that it's easy to run multiple independent instances of an application. Unfortunately, that image isn't much use in the long run if it's only on your machine. When it comes to deploying this image, you'll pull it from a Docker registry. This might be Docker Hub, which you've already encountered; a managed registry, such as AWS ECR or Google Container Registry; or self-hosted—for example, using the Docker distribution open source project (https://github.com/docker/distribution). When you build a continuous delivery pipeline, that pipeline will push to your registry on every valid commit.

> **NOTE** It's also possible to save Docker images as a tarball, using the `docker save` command, although this isn't commonly used in image distribution. In contrast, rkt natively uses tarballs for container distribution. This means you can store images in standard file stores, for example, S3, rather than using a custom registry.

For now, you can push your image to https://hub.docker.com. First, you'll need to create an account and choose a Docker ID. This will be the namespace you'll use to store your containers. Once you've logged in, you'll need to create a new repository—a store for multiple versions of the same image—using the web UI (figure 9.5).

To push to this repository, you need to tag your market-data image with an appropriate name. Docker image names follow the format `<registry>/<repository>:<tag>`. Once that's done, a simple docker push will upload your image to the registry. Try it out:

```
$ docker tag market-data:latest <docker id>/market-data:latest
$ docker login
$ docker push <docker id>/market-data:latest
```

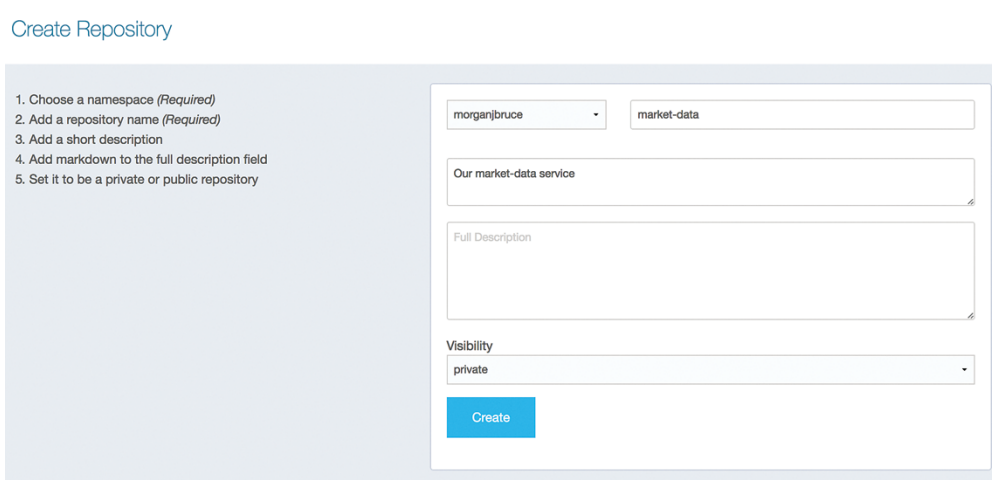


Figure 9.5 Using the Create Repository page on Docker Hub to create a repository for market-data images

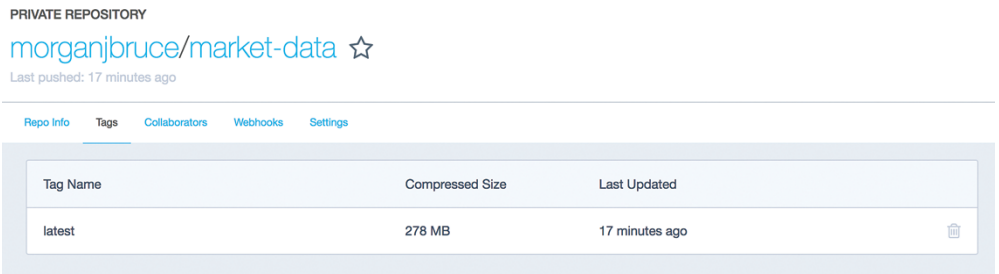


Figure 9.6 The private repository page on Docker Hub shows a record of the tagged image you pushed.

That's it! You've successfully pushed your image to a public repository. You can double-check that through the web UI (figure 9.6) by logging into https://hub.docker.com. Other engineers (if your repository is private, you'll need to grant them access) can pull your image using `docker pull [image name]`.

Let's take stock for a moment:

- You've learned how to package a simple application into a lightweight, cross-platform artifact—a container image.
- We've explored how Docker images are built from multiple layers to support inheritance from common base containers and increase startup speed.
- You've run multiple isolated instances of an application container.
- You've pushed the image you built to a Docker registry.

Using these techniques in a build pipeline will ensure greater consistency and predictability across a fleet of services, regardless of underlying programming language, as well as helping to simplify local development. Next, we'll explore how a container scheduler works by taking your containerized application and deploying it with Kubernetes.

9.2 Deploying to a cluster

A container scheduler is a software tool that abstracts away from underlying hosts by managing the execution of atomic, containerized applications across a shared pool of resources. This is possible because containers provide strong isolation of resources and a consistent API.

Using a scheduler is a compelling deployment platform for microservices because it eases the management of scaling, health checks, and releases across, in theory, any number of independent services. And it does so while ensuring efficient utilization of underlying infrastructure. At a high level, a container scheduler workflow looks something like this:

- Developers write declarative instructions to specify which applications they want to run. These workloads might vary: you might want to run a stateless, long-running service; a one-off job; or a stateful application, like a database.
- Those instructions go to a master node.

- The master node executes those instructions, distributing the workloads to a cluster of underlying worker nodes.
- Worker nodes pull containers from an appropriate registry and run those applications as specified.

Figure 9.7 illustrates this scheduler architecture. To an engineer, where and how an application is executed is ultimately unimportant: the scheduler takes care of it. In addition to running containers, Kubernetes provides other functionality to support running applications, such as service discovery and secret management.

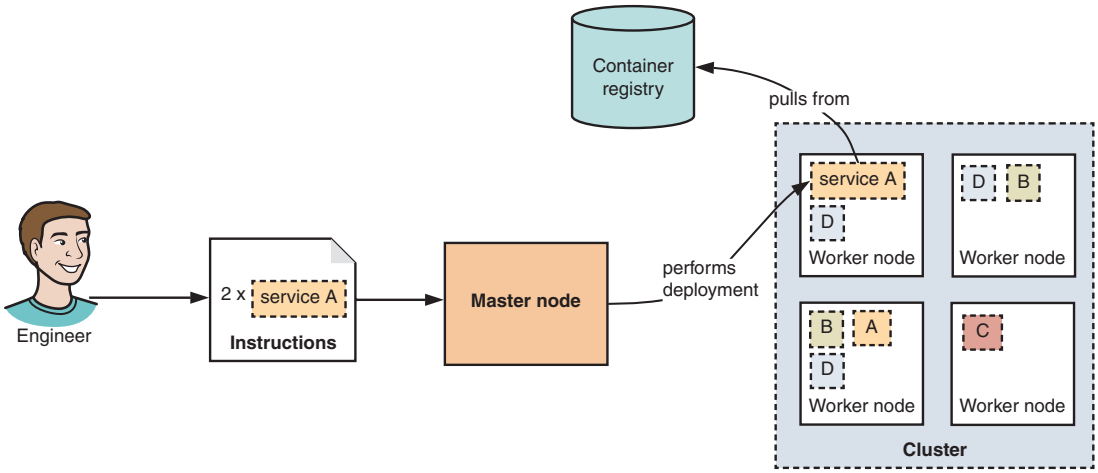


Figure 9.7 High-level scheduler architecture and deployment process

Many well-known cluster management tools are available, but in your case, you're going to use Kubernetes, an open-source project that evolved from Google's internal work on Borg and Omega (https://research.google.com/pubs/pub41684.html). It's possible to run Kubernetes pretty much anywhere—public cloud, private data center, or as a managed service (such as Google Kubernetes Engine (GKE)).

In the next few sections, we're going to cover a lot of ground. You'll do the following:

- Learn about the unit of deployment used on Kubernetes—pods
- Define and deploy multiple replicas of a pod for the market-data microservice
- Route requests to your pods using services
- Deploy a new version of the market-data microservice
- Learn how to communicate between microservices on Kubernetes

We'll start by using Minikube, which will run in a virtual machine on your local host. In a real deployment environment, the master and worker nodes would be separate virtual machines, but locally, the same machine will fulfill both roles. You can find an

installation guide for Minikube on the project's Github page (https://github.com/kubernetes/minikube).

> **TIP** If you used a private repository in section 9.1.4, you'll need to configure Minikube so it can access that repository by running `minikube addons configure registry-creds` and following the instructions onscreen.

9.2.1 Designing and running pods

The basic building block in Kubernetes is a pod: a single container or a tightly coupled group of containers that are scheduled together on the same machine. A pod is the unit of deployment and represents a single instance of a service. Because it's the unit of deployment, it's also the unit of horizontal scalability (or replication). When you scale capacity up or down, you add or remove pods.

> **TIP** Sometimes a service is deployed as more than one container—a composite container. For example, Flask services running on a Gunicorn web server are typically served behind NGINX. Using Kubernetes, a single pod would contain both the service and the NGINX container. Other examples of composite container patterns are discussed on the Kubernetes blog at (http://mng.bz/tOyC).

You can define a set of pods for your market-data service. Create a file called market-data-replica-set.yml in your app directory. Don't worry if it doesn't make much sense yet. Include the following code in your file.

Listing 9.4 market-data-replica-set.yml

```
---
kind: ReplicaSet                          <- Defines a set of pods
apiVersion: extensions/v1beta1
metadata:
  name: market-data
spec:                                     <- Should contain three replicas of your market-data pod
  replicas: 3
  template:                               <- Creates each pod using this template
    metadata:
      labels:                             <- Identifies pods within Kubernetes by label
        app: market-data
        tier: backend
        track: stable
    spec:
      containers:
      - name: market-data                 <- Contains a single container, pulled from your Docker registry
        image: <docker id>/market-data:latest
        ports:
        - containerPort: 8000
```

In Kubernetes, you typically declare instructions to the scheduler in YAML files (or JSON, but YAML's easier on the eyes). These instructions define Kubernetes objects,

and a pod is one kind of object. These configuration files represent the desired state of your cluster. When you apply this configuration to Kubernetes, the scheduler will continually work to maintain that ideal state. In this file, you've defined a `ReplicaSet`, which is a Kubernetes object that manages a group of pods.

> **NOTE** We'll occasionally use dot-notation to refer to paths within a *.yml file. For example, in listing 9.4, the path to the market-data container definition would be spec.template.spec.containers[0].

To apply this to your local cluster, you can use the `kubectl` command-line tool. When you started Minikube, it should have automatically configured `kubectl` to operate on your cluster. This tool interacts with an API exposed by the cluster's master node. Give it a try:

```
$ kubectl apply -f market-data-replica-set.yml
replicaset "market-data" configured
```

Kubernetes will asynchronously create the objects you've defined. You can observe the status of this operation using `kubectl`. Running `kubectl get pods` (or `kubectl get pods -l app=market-data`) will show you the pods that your command has created (figure 9.8). They'll take a few minutes to start up for the first time as the node downloads your Docker image.

You saw earlier that you didn't create individual pods. It's unusual to create or destroy pods directly; instead, pods are managed by controllers. A controller is responsible for taking some desired state—say, always running three instances of the market-data pod—and performing actions to reach that state. This observe-diff-act loop happens continually.

You've just encountered the most common type of controller: the `ReplicaSet`. If you've ever encountered instance groups on AWS or GCP, you might find their behavior similar. A replica set aims to ensure a specific number of pods are running at any one time. For example, let's say a pod dies—maybe a node in the cluster failed—the replica set will observe that the state of the cluster no longer matches the desired state and will attempt to schedule a replacement elsewhere in the cluster.

You can see this in action. Delete one of the pods you've just created (pods are identified by name):

```
$ kubectl delete pod <pod name>
```

The replica set will schedule a new pod to replace the one you destroyed (figure 9.9).

```
NAME                READY     STATUS    RESTARTS   AGE
market-data-dv023   1/1       Running   0          16m
market-data-fwnlt   1/1       Running   0          16m
market-data-gdkms   1/1       Running   0          16m
```

Figure 9.8 The results of the kubectl get pods command after creating a new replica set

```
NAME                READY     STATUS        RESTARTS   AGE
market-data-1p0z3   1/1       Running       0          7s
market-data-dv023   0/1       Terminating   0          41m
market-data-fwnlt   1/1       Running       0          41m
market-data-gdkms   1/1       Running       0          41m
```

Figure 9.9 The state of running pods after one member of the replica set is deleted

This matches the ideal we laid out in chapter 8: that deploying microservice instances should be built on a single primitive operation. By combining controllers and immutable containers, you can treat pods like cattle and rely on automation to maintain capacity, even when the underlying infrastructure is unreliable.

> **WARNING** A cluster alone isn't a complete redundancy solution; your infrastructure design also determines this. For example, if you run a cluster in a single data center—or one availability zone in AWS—you won't have redundancy if that entire data center goes down. It's important, where possible, to run your cluster(s) across multiple isolated zones of failure.

9.2.2 Load balancing

Right, so you're running a microservice on Kubernetes. That was pretty quick. The bad news is, you can't access those pods yet. Like you did earlier with NGINX, you need to link them to a load balancer to route requests and expose their capabilities to other collaborators, either inside or outside your cluster.

In Kubernetes, a service defines a set of pods and provides a method for reaching them, either by other applications in the cluster or from outside the cluster. The networking magic that achieves this feat is outside the scope of this book, but figure 9.10 illustrates how a service would connect to your existing pods.

Now, you're currently running a replica set containing three market-data pods. If you recall from listing 9.4, your market-data pods have the labels `app: market-data` and `tier: backend`. That's important, because a service forms a group of pods based on their labels.

To create a service, you need another YAML file, as shown in the following listing. This time, call it market-data-service.yml (great naming convention).

Listing 9.5 market-data-service.yml

```
---
apiVersion: v1
kind: Service
metadata:
  name: market-data
spec:
  type: NodePort
```

```
selector:
  app: market-data
  tier: backend
ports:
  - protocol: TCP
    port: 8000
    nodePort: 30623
```

Defines which pods this service will access

The service will be exposed as a specified port on the cluster. Excluding this line will assign a random port in range 30000-32767.

The service will route to this port on the specified pods.

Apply this configuration using the same `$ kubectl apply -f` command you used to create the replica set before, substituting the name of your new YAML file. This will create a service accessible on port 30623 of your cluster, which routes requests to your market-data pods on port 8000.

You should be able to `curl` your service and send requests to your pods. Doing so will return the name of each pod that serves the request:

```
$ curl http://`minikube ip`:30623/ping
```

`minikube ip` returns the IP address of your local cluster.

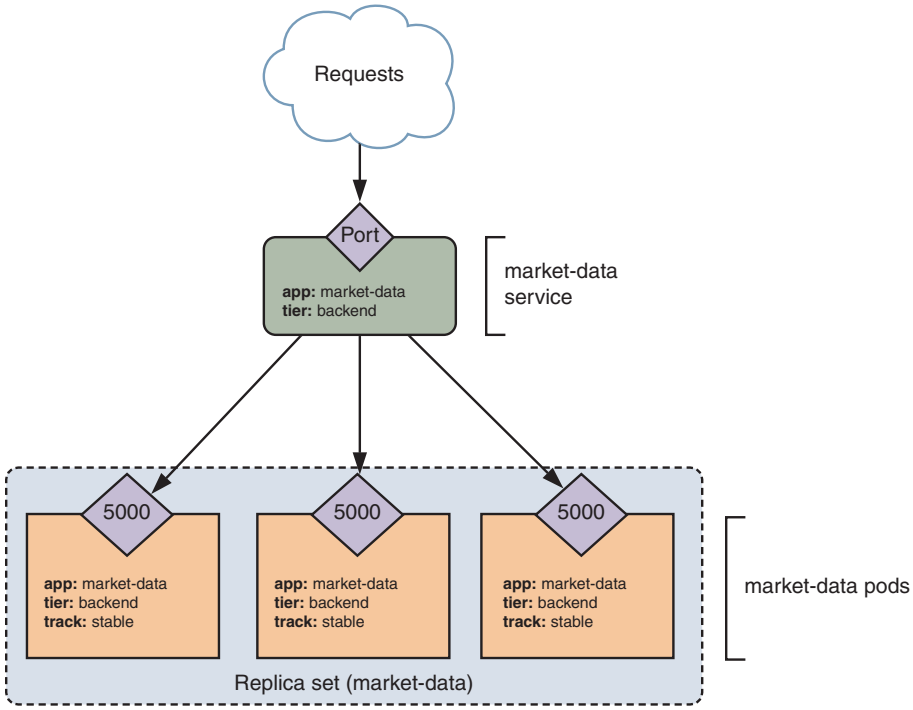


Figure 9.10 Requests made to a service are forwarded to pods that match the label selector of the service.

Several types of services are available, and they're outlined in table 9.1. In this case, you used a `NodePort` service to map your service to an externally available port on your cluster, but if only other cluster services access your microservice, it usually makes more sense to use `ClusterIP` to keep access local to the cluster.

Table 9.1 Types of service on Kubernetes

Service type	Behavior
`ClusterIP`	Exposes the service on an IP address local to the cluster
`NodePort`	Exposes the service on a static port accessible at the cluster's IP address
`LoadBalancer`	Exposes the service by provisioning an external cloud service load balancer (If you're using AWS, this creates an ELB.)

The service listens for events across the cluster and will be dynamically updated if the group of pods changes. For example, if you kill a pod, it will be removed from the group, and the service will route requests to any new pod created by the replica set.

9.2.3 A quick look under the hood

So far, this has been seamless: you send an instruction, and Kubernetes executes it! Let's take a moment to learn how Kubernetes runs your pods.

If you drill down a level, you can see that the master and worker nodes on Kubernetes run several specialized components. Figure 9.11 illustrates these components.

COMPONENTS OF THE MASTER NODE

The master node consists of four components:

- *The API server* — When you ran commands on `kubectl`, this is what it communicated with to perform operations. The API server exposes an API for both external users and other components within the cluster.
- *The scheduler* — This is responsible for selecting an appropriate node where a pod will run, given priority, resource needs, and other constraints.
- *The controller manager* — This is responsible for executing control loops: the continual observe-diff-act operation that underpins the operation of Kubernetes.
- *A distributed key-value data store, etcd* — This stores the underlying state of the cluster and thereby makes sure it persists when nodes fail or restarts are required.

Together, these components act as a control plane for the cluster. Picture this as something like the cockpit of an airplane. Together, these components provide the API and backend required to orchestrate operations across a cluster of nodes.

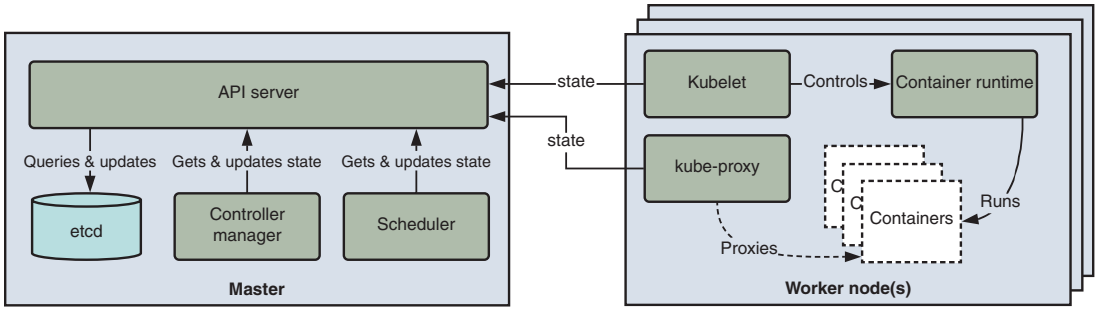


Figure 9.11 Components of the master and worker nodes in a Kubernetes cluster

COMPONENTS OF A WORKER NODE

Each worker node uses the following components to run and monitor applications:

- *A container runtime* — In your case, this is Docker.
- *The kubelet* — This interacts with the Kubernetes master to start, stop, and monitor containers on the node.
- *The kube-proxy* — This provides a network proxy to direct requests to and between different pods across the cluster.

These components are relatively small and loosely coupled. A key design principle of Kubernetes is to separate concerns and ensure components can operate autonomously—a little like microservices!

WATCHES FOR STATE CHANGES

The API server is responsible for recording the state of the cluster—and receiving instructions from clients—but it doesn't explicitly tell other components what to do. Instead, each component works independently to orchestrate cluster behavior when some event or change occurs. To learn about state changes, each component watches the API server: a component requests to be notified by the API server when something interesting happens, so it can perform appropriate actions to attempt to match the desired state.

For example, the scheduler needs to know when it should assign new pods to nodes. Therefore, it connects to the API server to receive a continuous stream of events that relate to the pod resource. When it receives a notification about a newly created pod, it finds an appropriate node for that pod. Figure 9.12 shows this process.

In turn, your kubelets watch the API server to learn when a pod has been assigned to its node and then they start the pod appropriately. Each component watches resources and events that interest it; for example, the controller manager watches replica sets and services (among other things).

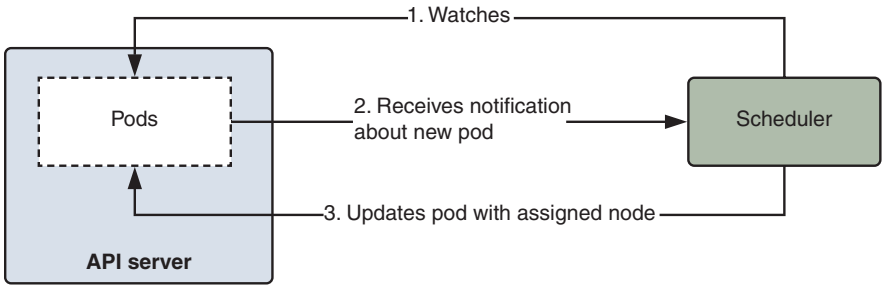


Figure 9.12 The scheduler watches the API server for newly created pods and determines which node they should run on.

UNDERSTANDING HOW PODS ARE RUN

What happens when you create a replica set? You saw earlier that this results in the expected number of pods being run—from your perspective, it looked simple! But in reality, creating your replica set through `kubectl` triggers a complex chain of events across multiple components. This chain is illustrated in figure 9.13.

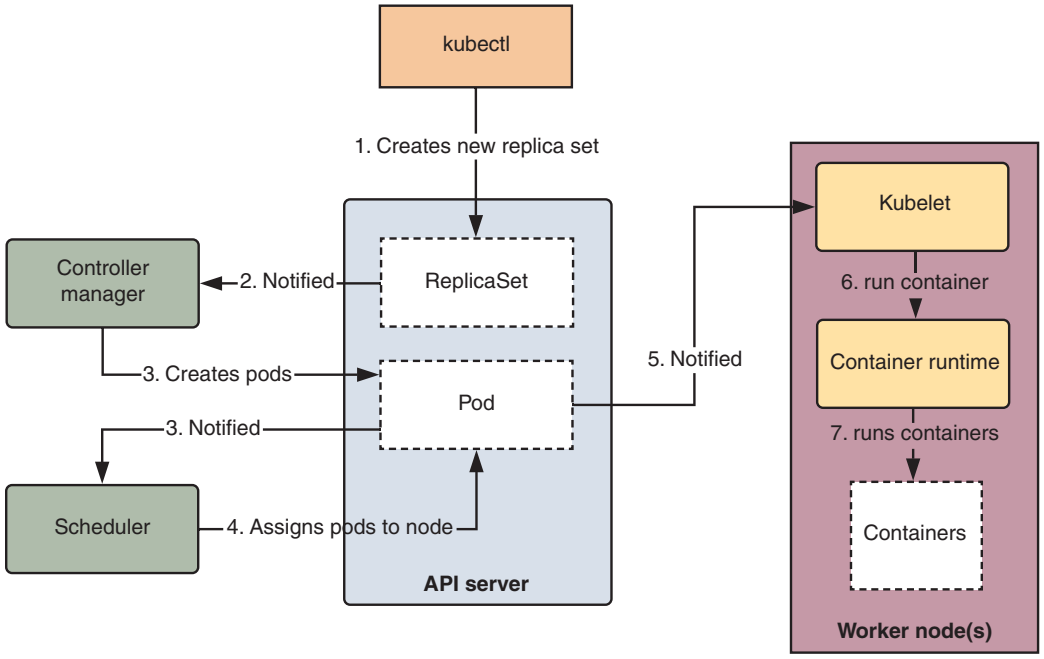


Figure 9.13 The series of events from creating a replica set to running pods on Kubernetes

Let's walk through each step:

1. You instructed the API server to create a new replica set, using `kubectl`. The API server stores this new resource in etcd.
2. The controller manager watches for creation and modification of replica sets. It receives a notification about the new set you created.
3. The controller manager compares the current state of the cluster to the new state, determining that it needs to create new pods. It creates these pod resources through the API server, based on the template you provided through `kubectl`.
4. The scheduler receives a notification about a new pod and assigns it an appropriate node, again updating the pod's definition through the API server. At this point, you haven't run any real application—the controllers and scheduler have only updated the state that the API server is storing.
5. Once the pod is assigned to a node, the API server notifies the appropriate kubelet, and the kubelet instructs Docker to run containers. Images are downloaded, containers are started, and the kubelet begins to monitor their operation. At this point, your pods are running!

As you can see, each component acts independently, but together, they orchestrate a complex deployment action. Hopefully this has given you a useful glance under the cover. Now, back to running your microservices.

9.2.4 Health checks

You're missing something. Unlike a typical cloud load balancer, a Kubernetes service doesn't itself execute health checks on your underlying application. Instead, the service checks the shared state of the cluster to determine if a pod is ready to receive requests. But how do you know if a pod is ready?

In chapter 6, we introduced two types of health check:

- *Liveness*—Whether an application has started correctly
- *Readiness*—Whether an application is ready to serve requests

These health checks are crucial to the resiliency of your service. They ensure that traffic is routed to healthy instances of your microservice and away from instances that are performing poorly (or not at all).

By default, Kubernetes executes lightweight, process-based liveness checks for every pod you run. If one of your market-data containers fails a liveness check, Kubernetes will attempt to restart that container (as long as the container's restart policy isn't set to `Never`). The kubelet process on each worker node carries out this health check. This process continually queries the container runtime (in your case, the Docker daemon) to establish whether it needs to restart a container.

TIP Kubernetes performs restarts on an exponential back-off schedule; if a pod isn't live after five minutes, it'll be marked for deletion. If a replica set manages the pod, the controller will attempt to schedule a new pod to maintain the desired service capacity.

This alone isn't adequate, as your microservice may run into failure scenarios that don't cause the container itself to fail: whether deadlocks due to request saturation, timeouts of underlying resources, or a plain old coding error. If the scheduler can't identify this scenario, performance can deteriorate as a service routes requests to unresponsive pods, potentially leading to cascading failures.

To avoid this situation, you need the scheduler to continually check the state of the application inside your container, ensuring it's both live and ready. With Kubernetes, you can configure probes to achieve this, which you can define as part of your pod template. Figure 9.14 illustrates how these checks, and the previous process check, will be run.

Adding probes is straightforward, although you do need to add some configuration see the next listing to the container specification in market-data-replica-set.yml. Probes can be `HTTP GET` requests, scripts executed inside a container, or TCP socket checks. In this case, you'll use a `GET` request, as shown in the following listing.

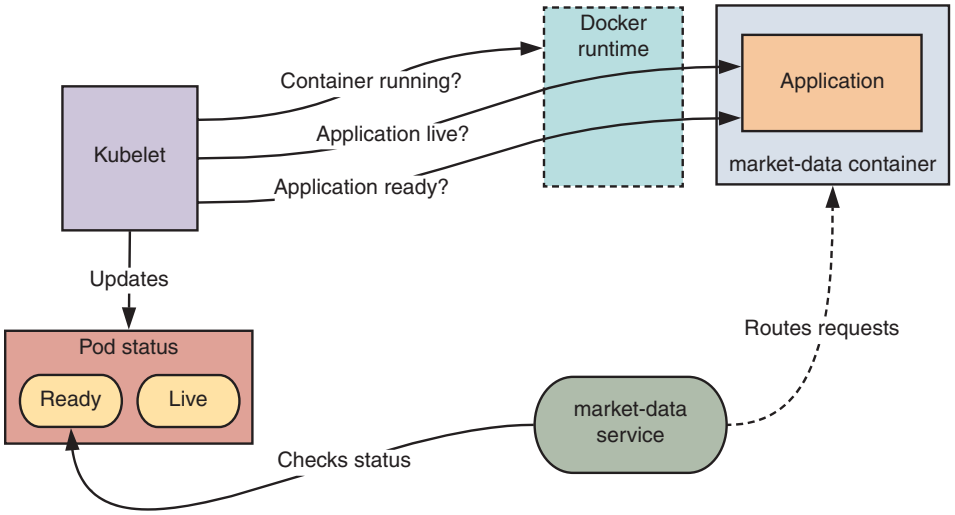


Figure 9.14 The kubelet process on each worker node runs health checks, or probes, in Kubernetes. Readiness probe results control routing by services.

Listing 9.6 Liveness probe in market-data-replica-set.yml

```
livenessProbe:
  httpGet:
    path: /ping
    port: 8000
  initialDelaySeconds: 10
  timeoutSeconds: 15
readinessProbe:
    path: /ping
    port: 8000
  initialDelaySeconds: 10
  timeoutSeconds: 15
```

Configures a liveness probe to query /ping on port 8000

Configures a readiness probe to query /ping on port 8000

Reapply this configuration using `kubectl` to update the state of the replica set. Kubernetes will, to the best of its ability, use these probes to help ensure instances of your microservice are alive and kicking. In this example, both liveness and readiness check the same endpoint, but if your microservice has external dependencies, such as a queueing service, it makes sense to make readiness dependent on connectivity from your application to those dependencies.

9.2.5 Deploying a new version

You should now understand how you use replica sets, pods, and services to run stateless microservices on Kubernetes. On top of these concepts, you can build a stable, seamless deployment process for each of your microservices. In chapter 8, you learned about canary deployments; in this section, you'll try out the technique with Kubernetes.

DEPLOYMENTS

Before we get started, we should quickly introduce deployments. Kubernetes provides a higher level abstraction, the `Deployment` object, for orchestrating the deployment of new replica sets. Each time you update a deployment, the scheduler will orchestrate a rolling update of instances in a replica set, ensuring they're deployed seamlessly.

You can change the original approach to use a deployment instead. First, delete your original replica set:

```
$ kubectl delete replicaset market-data
```

After that, create a new file, market-data-deployment.yml. This should be similar to the replica set you created earlier, except that the type of object should be `Deployment`, rather than `ReplicaSet`, as shown in the following listing.

Listing 9.7 market-data-deployment.yml

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: market-data
spec:
  replicas: 3
```

Defines a Kubernetes deployment object

The desired number of pods to deploy

```
template:                          <-- The template to use for creating each pod
  metadata:
    labels:
      app: market-data
      tier: backend
      track: stable
  spec:
    containers:
    - name: market-data
      image: <docker id>/market-data:latest
      resources:
        requests:
          cpu: 100m
          memory: 100Mi
      ports:
      - containerPort: 8000
      livenessProbe:
        httpGet:
          path: /ping
          port: 8000
        initialDelaySeconds: 10
        timeoutSeconds: 15
      readinessProbe:
        httpGet:
          path: /ping
          port: 8000
        initialDelaySeconds: 10
        timeoutSeconds: 15
```

Use `kubectl` to apply this file to the cluster. This will create a deployment, which will create a replica set and three instances of the market-data pod.

CANARIES

In a canary deploy, you deploy a single instance of a microservice to ensure that a new build is stable when it faces real production traffic. This instance should run alongside existing production instances. A canary release has four steps:

1. You release a single instance of a new version alongside the previous version.
2. You route some proportion of traffic to the new instance.
3. You assess the health of the new version by, for example, monitoring error rates or observing behavior.
4. If the new version is healthy, you commence a full rollout to replace other instances. If not, you remove the canary instance, halting the release.

On Kubernetes, you can use labels to identify a canary pod. In the first example, you specified a label `track: stable` on each pod in your replica set. To deploy a canary, you'll need to deploy a new pod that's distinguished with `track: canary`. The service you created earlier only selects on two labels (`app` and `tier`), so it'll route requests to both stable and canary pods. This is illustrated in figure 9.15.

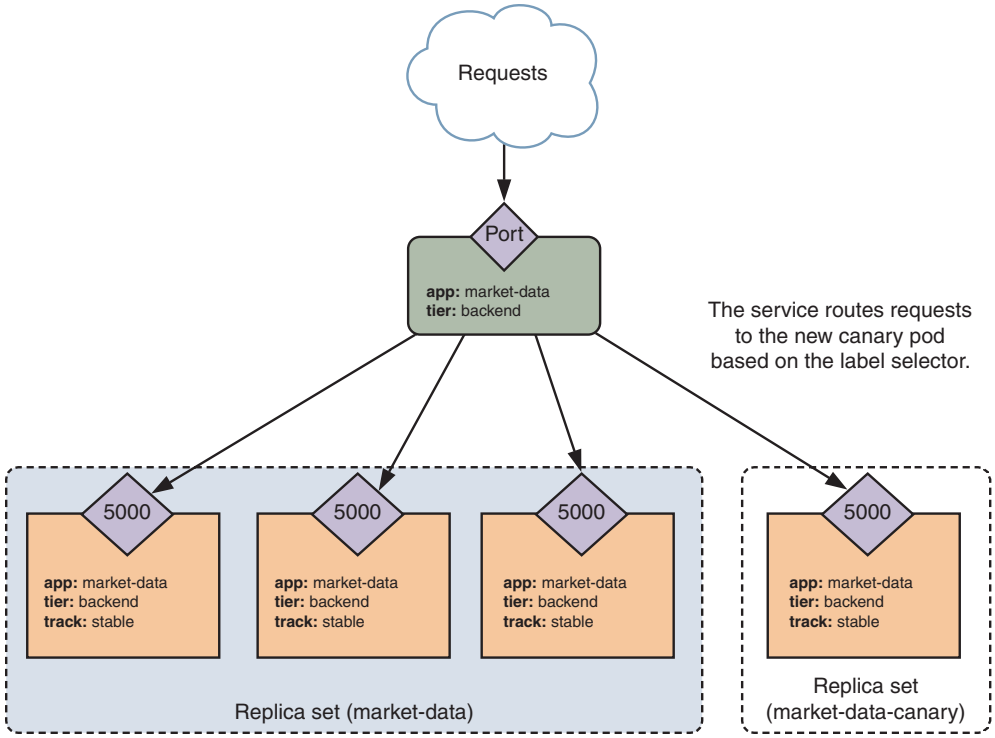


Figure 9.15 The service forwards requests to your new canary pod based on the service's label selector, which doesn't restrict on `track`.

First, you should build a new container for your new release. You'll use tags to identify the new version, and don't forget to substitute your own Docker ID:

```
$ docker build -t <docker id>/market-data:v2 .
$ docker push <docker id>/market-data:v2
```

This version is tagged as v2, although in practice it may not be appropriate to apply a numeric versioning scheme to your services. We've found tagging them with the commit ID also works well. (For Git repositories, we use `git rev-parse --short HEAD`.)

Once you've pushed that new image, create a yml file specifying your canary deployment:

- It should create one replica, instead of three.
- It should release the `v2` tag of the container, rather than `latest`.
- It should look like the following listing.

Listing 9.8 market-data-canary-deployment.yml

```
---
apiVersion: extensions/v1beta1
kind: Deployment
```

```
metadata:
  name: market-data-canary
spec:
  replicas: 1                          <-- You want to create one canary.
  template:
    metadata:
      labels:
        app: market-data               <-- The canary deployment has
        tier: backend                      a distinct set of labels.
        track: canary
    spec:
      containers:
      - name: market-data
        image: <docker id>/market-data:v2   <-- This deployment will release
        resources:                              market-data:v2.
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 8000
        livenessProbe:
          httpGet:
            path: /ping
            port: 8000
          initialDelaySeconds: 10
          timeoutSeconds: 15
        readinessProbe:
          httpGet:
            path: /ping
            port: 8000
          initialDelaySeconds: 10
          timeoutSeconds: 15
```

Use `kubectl` to apply this to your cluster. The deployment will create a new replica set containing a single canary pod for v2.

Let's take a closer look at the state of your cluster. If you run `minikube dashboard` on the command line, it'll open the dashboard for your cluster in a browser window (figure 9.16). In the dashboard—under Workloads—you should be able to see:

- The canary deployment you've just created, as well as your original deployment
- Four pods: the original three, plus a canary pod
- Two replica sets: one each for the stable and canary tracks

So far so good! At this stage, for a real microservice, you might run some automated tests, or check the monitoring output of your service to ensure it's processing work as expected. For now, you can safely assume your canary is healthy and performing as expected, which means you can safely roll out the new version, replacing all your old instances.

Edit the market-data-deployment.yml file and make two changes:

- Change the container used to `market-data:v2`.
- Add a strategy field to specify how pods will be updated.

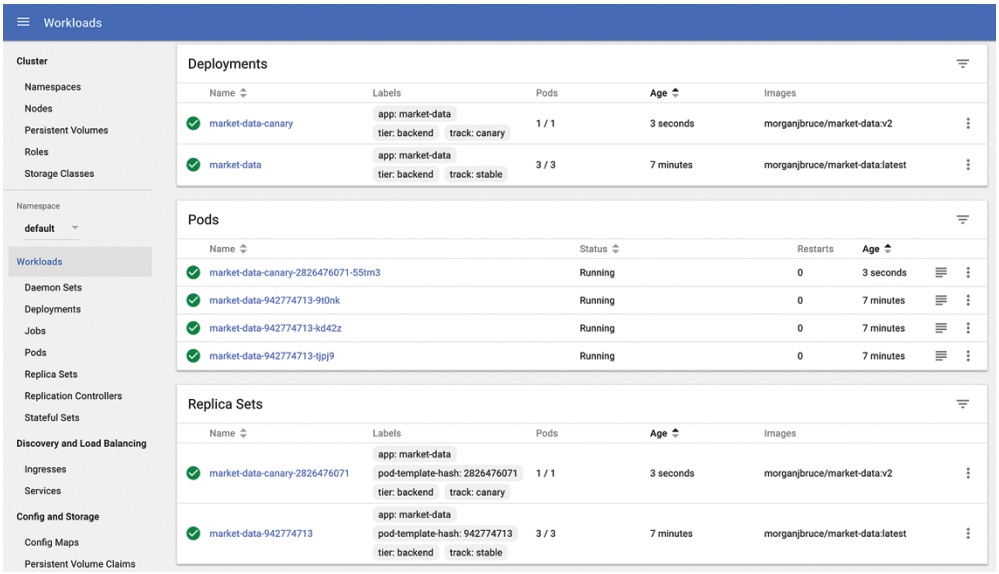


Figure 9.16 The Kubernetes dashboard after multiple deploys—stable and canary—of the market-data microservice

Your updated deployment file should look like the following listing.

Listing 9.9 Updated market-data-deployment.yml

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: market-data
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 50%
      maxSurge: 50%
  template:
    metadata:
      labels:
        app: market-data
        tier: backend
        track: stable
    spec:
      containers:
      - name: market-data
        image: morganjbruce/market-data:v2
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
```

The strategy field describes how Kubernetes will execute the deployment of new pods.

```
      ports:
      - containerPort: 8000
```

Applying this configuration will create a new replica set, starting instances one by one while removing them from the original set. This process is illustrated in figure 9.17.

You can also observe this in the event history of the controller by running `kubectl describe deployment/market-data` (figure 9.18).

From this history, you can see how Kubernetes allows you to build higher level deployment operations on top of simple operations. In this case, the scheduler used your desired state of the world and a set of constraints to determine an appropriate path of deployment, but you could use replica sets and pods to build any deployment pattern that was appropriate for your service.

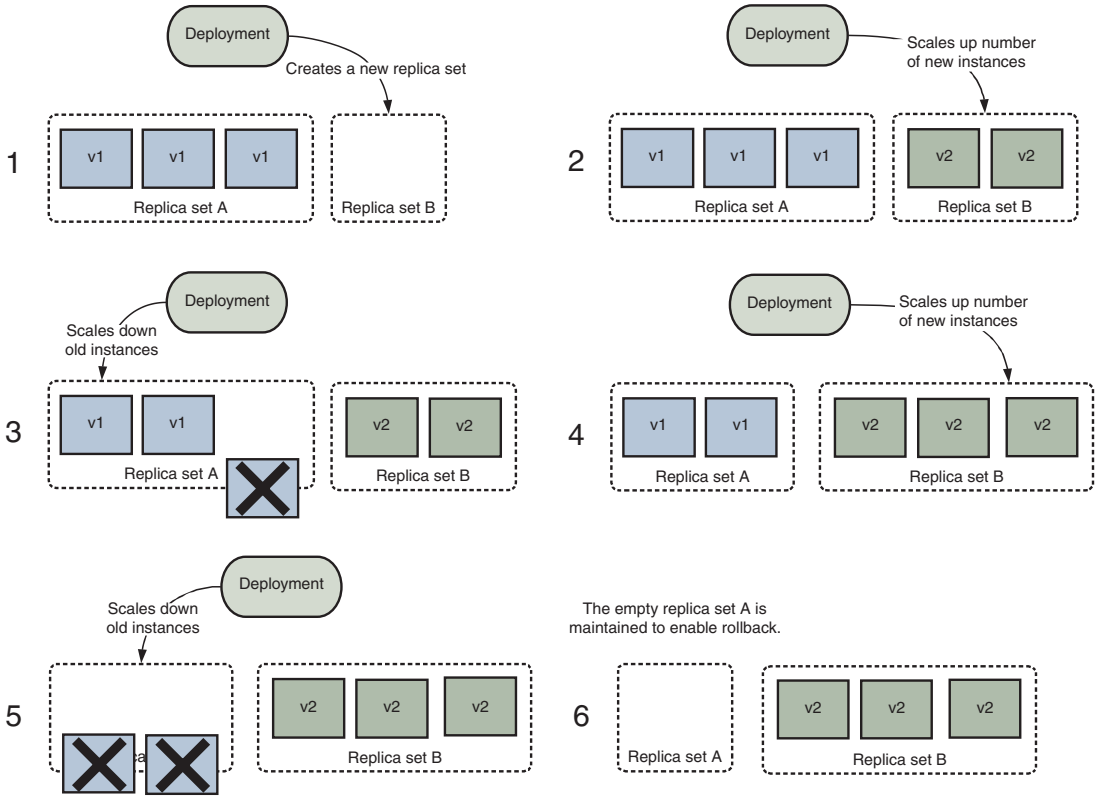


Figure 9.17 A new deployment creates a new replica set and progressively rolls instances between the old and new set.

```
17m   17m   1   deployment-controller   Normal   ScalingReplicaSet   Scaled up replica set market-data-2045614036 to 2
17m   17m   1   deployment-controller   Normal   ScalingReplicaSet   Scaled down replica set market-data-942774713 to 2
17m   17m   1   deployment-controller   Normal   ScalingReplicaSet   Scaled up replica set market-data-2045614036 to 3
16m   16m   1   deployment-controller   Normal   ScalingReplicaSet   Scaled down replica set market-data-942774713 to 1
16m   16m   1   deployment-controller   Normal   ScalingReplicaSet   Scaled down replica set market-data-942774713 to 0
```

Figure 9.18 Events Kubernetes emitted during a rolling deployment

9.2.6 Rolling back

Well done! You've smoothly deployed a new version of your microservice. If something went wrong, you can also use the deployment object to undo all your hard work. First, check the rollout history:

```
$ kubectl rollout history deployment/market-data
```

This should return two revisions: your original deployment and your v2 deployment. To roll back, specify the target revision:

```
$ kubectl rollout undo deployment/market-data --to-revision=1
```

This will perform the reverse of the previous rolling update to return the underlying replica set to its original state.

9.2.7 Connecting multiple services

Lastly, your microservice isn't going to be much use by itself, and several of SimpleBank's services depend on the capabilities that the market-data service provides. It'd be pretty much insane to hardcode a port number or an IP address into each service to refer to the underlying endpoint of each collaborator; you shouldn't tightly couple any service to another's internal network location. Instead, you need some way of accessing a collaborator by a known name.

Kubernetes integrates a local DNS service to achieve this, and it runs as a pod on the Kubernetes master. When new service is created, the DNS service assigns a name in the format `{my-svc}.{my-namespace}.svc.cluster.local`; for example, you should be able to resolve your market-data service from any other pod using the name `market-data.default.svc.cluster.local`.

Give it a shot. You can use `kubectl` to run an arbitrary container in your cluster—try busybox, which is a great little image containing several common Linux utilities, such as nslookup. Run the following command to open a command prompt inside a container running on Minikube:

```
$ kubectl run -i --tty lookup --image=busybox /bin/sh
```

Then you can try an `nslookup`:

```
/ # nslookup market-data.default.svc.cluster.local
```

You should get output that looks something like this:

```
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      market-data.default.svc.cluster.local
Address 1: 10.0.0.156 market-data.default.svc.cluster.local
```

The IP address in the last entry should match the cluster IP assigned to your service. (If you don't believe me, you can double-check by calling `kubectl get services`.) If so, success! You've covered a lot of ground: building and storing an image for a microservice, running it on Kubernetes, load-balancing multiple instances, deploying a new version (and rolling back), and connecting microservices together.

Summary

- Packaging microservices as immutable, executable artifacts allows you to orchestrate deployment through a primitive operation—adding or removing a container.
- Schedulers and containers abstract away underlying machine management for service development and deployment.
- Schedulers work by trying to match the resource needs of an application to the resource usage of a cluster of machines, while health-checking running services to ensure they're operating correctly.
- Kubernetes provides ideal features of a microservice deployment platform, including secret management, service discovery, and horizontal scalability.
- A Kubernetes user defines the desired state (or specification) of their cluster services, and Kubernetes figures out how to achieve that state, executing a continual loop of observe-diff-act.
- The logical application unit on Kubernetes is a pod: one or more containers that execute together.
- Replica sets manage the lifecycle of groups of pods, starting new pods if existing ones fail.
- Deployments on Kubernetes are designed to maintain service availability by executing rolling updates of pods across replica sets.
- You can use service objects to group underlying pods and make them available to other applications inside and outside of the cluster.

10 Building a delivery pipeline for microservices

This chapter covers

- Designing a continuous delivery pipeline for a microservice
- Using Jenkins and Kubernetes to automate deployment tasks
- Managing staging and production environments
- Using feature flags and dark launches to distinguish between deployment and release

Rapidly and reliably releasing new microservices and new features to production is crucial to successfully maintaining a microservice application. Unlike a monolithic application, where you can optimize deployment for a single use case, microservice deployment practices need to scale to multiple services, written in different languages, and each with their own dependencies. Investing in consistent and robust deployment tooling and infrastructure will go a long way toward making a success of any microservice project.

You can achieve reliable microservice releases by applying the principles of continuous delivery. The fundamental building block of continuous delivery is a deployment pipeline. Picture a factory production line: a conveyer belt takes your software

from code commits to deployable artifact to running software, while continually assessing the quality of the output at each stage. Doing this leads to frequent, small deployments, rather than big-bang changes, to production.

So far, you've built and deployed a service using Docker, Kubernetes, and command-line scripts. In this chapter, you'll combine those steps into an end-to-end build pipeline, using Jenkins, a widely used open source build automation tool. Along the way, we'll examine how this approach minimizes risk and increases the stability of your overall application. After that, we'll examine the difference between deploying new code and releasing new features.

10.1 Making deploys boring

Deploying software should be boring. You should be able to roll out changes and new features without peering through your fingers or obsessively watching error dashboards.

Unfortunately, as we mentioned in chapter 8, human error causes most issues in production, and microservice deployments leave plenty of room for that! Consider the big picture: teams are developing and deploying tens—if not hundreds—of independent services on their own schedule, without explicit coordination or collaboration between teams. Any bad change to a service might have a wide-ranging impact on the performance of other services and the wider application.

An ideal microservice deployment process should meet two goals:

- *Safety at pace*—The faster you can deploy new services and changes, the quicker you can iterate and deliver value to your end users. Deployment should maximize safety: you should validate, as much as feasible, that a given change won't negatively impact the stability of a service.
- *Consistency*—Consistency of deployment process across different services, regardless of underlying tech stack, helps alleviate technical isolation and makes operations more predictable and scalable.

It's not easy to maintain the fine balance between safety and pace. You could move quickly without safety by deploying code changes directly to production, but that'd be crazy. Likewise, you could achieve stability by investing in a time-consuming change-control and approval process, but that wouldn't scale well to the high volume of change in a large, complex microservice application.

10.1.1 A deployment pipeline

Continuous delivery strikes an ideal balance between reducing risk and increasing speed:

- Releasing smaller sets of commits increases safety by reducing the amount of change happening at any one time. Smaller changesets are also easier to reason through.
- An automated pipeline of commit validation increases the probability that a given changeset is free from defects.

Releasing small changesets and systematically verifying their quality gives teams the confidence to release features rapidly. Smaller, more atomic releases are less risky. The continuous delivery approach empowers teams to ship services rapidly and independently.

One of the weaknesses of monolithic development is that releases often become large, coupling together disparate features at release time. Likewise, even small changes in a large application can have an unintentionally broad impact, particularly when made to cross-cutting concerns. At worst, commits in monolithic development become stale while waiting for a deployment; they're no longer relevant to the needs of the application or business by the time they reach customers.

> **NOTE** Continuous delivery isn't quite the same as continuous deployment. In the latter, every validated change is automatically deployed to production; in the former, you can deploy every change to production, but whether you deploy it or not is up to the engineering team and business needs.

Let's look at an example in figure 10.1. Most of the steps in this pipeline should look familiar:

1. First, an engineer commits some code to a microservice repository.
2. Next, a build automation server builds the code.
3. If the build is successful, the automation server runs unit tests to validate that code.
4. If these tests pass, the automation server packages the service for deployment and stores this package in an artifact repository.
5. The automation server deploys code to a staging environment, where you can test the service against other live collaborators.
6. If this is successful, the automation server will deploy the code to a production environment.

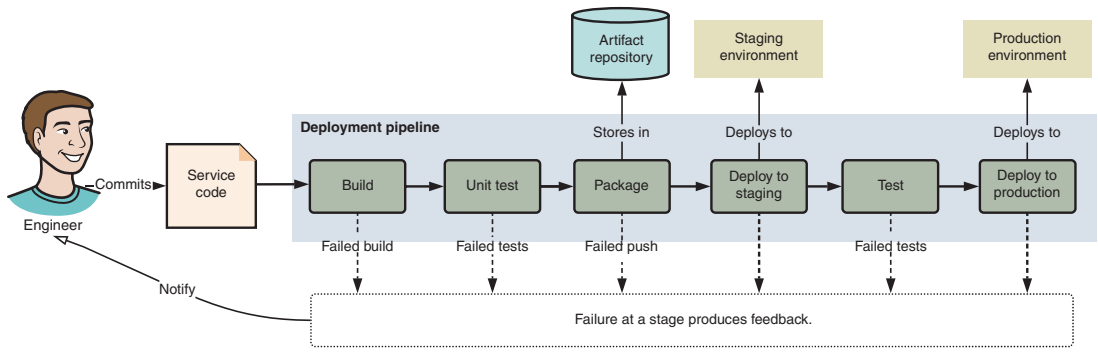


Figure 10.1 An example deployment pipeline builds, validates, and deploys a commit to production, providing feedback to engineers.

Each step in this pipeline provides feedback to the engineering team on the correctness of their code. For example, if step 3 fails, you'll receive a list of failed test assertions to correct.

Implementing this pipeline should make the process of deployment highly visible and transparent—crucial for an audit trail, or if something goes wrong. Regardless of the underlying language or technology, every service you deploy should be able to follow a similar process.

10.2 Building a pipeline with Jenkins

In the previous chapter, you ran command-line scripts to perform steps in deployment: building containers, publishing artifacts, and deploying code. Now you'll use Jenkins—a build automation tool—to connect those steps together into a coherent, reusable, and extensible deployment pipeline. We've picked Jenkins because it's open source, is easy to get running, supports scriptable build jobs, and is widely used.

Unfortunately, no perfect out-of-the-box solution for deployment is available: any pipeline is usually a combination of multiple tools, depending on both the service's tech stack and the target deployment platform. In your case, you'll be using Jenkins to assemble tools you've (mostly) already used. Figure 10.2 illustrates the components of your deployment pipeline.

In the next few sections, we're going to cover a lot of ground:

- Using Jenkins to script complex deployment pipelines
- Building a pipeline that builds, tests, and deploys your service to different environments
- Managing staging environments for microservices
- Reusing your deployment pipeline across multiple services

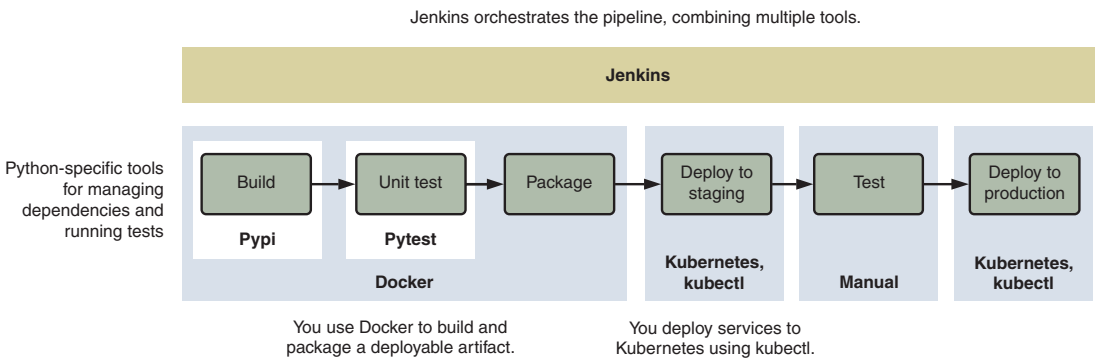


Figure 10.2 The deployment pipeline you'll use, which combines multiple tools dependent on the tech stack and target deployment platform you're using

You'll need to have access to a running Jenkins instance to run the examples in this chapter. The appendix walks you through Jenkins setup on a local Minikube cluster—we'll assume you're using that approach in the following sections.

10.2.1 Configuring a build pipeline

The Jenkins application consists of a master node and, optionally, any number of agents. Running a Jenkins *job* executes scripts (using common tools, such as make or Maven) across these agent nodes to perform deployment activities. A job operates within a *workspace*—a local copy of your code repository. Figure 10.3 illustrates this architecture.

To write your build pipeline, you're going to use a feature called Scripted Pipeline. In Scripted Pipeline, you can express a build pipeline using a general-purpose domain-specific language (DSL) written in Groovy. This DSL defines common methods for writing build jobs, such as `sh` (for executing shell scripts) and `stage` (for identifying different parts of a build pipeline). The Scripted Pipeline approach is more powerful than you might think—by the end of the chapter, you'll use it to build your own higher level, declarative DSL.

> **NOTE** At the time of writing, Jenkins Pipeline only supports Groovy as a scripting language. Don't worry—if you're comfortable with Java, Python, or Ruby, understanding Groovy won't be too taxing.

Jenkins will execute build jobs by executing a pipeline script defined in a Jenkinsfile. Try it yourself! First, copy the contents of chapter-10/market-data into a new directory and push that to a Git repo. It's easiest if you push it to somewhere public, like GitHub. This is the service you'll be deploying in this chapter.

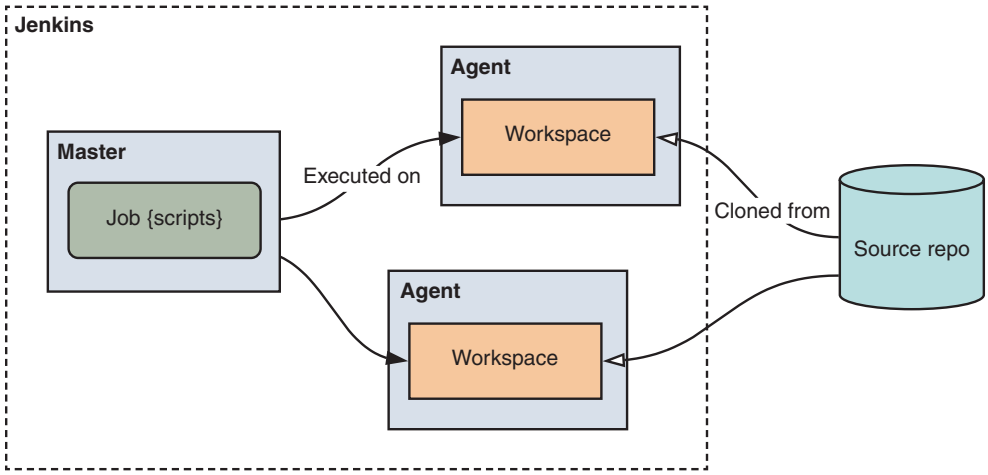


Figure 10.3 A Jenkins deployment consists of a master node, which manages execution, and agents that perform build tasks within a workspace—a clone of the repository being built.

Next, you'll want to create a Jenkinsfile in the root of your repository, and it should look like the following listing.

Listing 10.1 A basic Jenkinsfile

Identifies a distinct phase of your pipeline

Takes a closure (or function) as a parameter, instructing Jenkins to execute this code on a build node

Checks out some code from source control

```
stage("Build Info") {
  node {
    def commit = checkout scm
    echo "Latest commit id: ${commit.GIT_COMMIT}"
  }
}
```

When Jenkins runs this script, the script will check out a code repository as a workspace and write the latest commit ID to the console.

You can try this out by setting up a pipeline job for a service. Commit the Jenkinsfile you just created, then push your changes to origin. Now, open the Jenkins UI. (Remember, you can do this with `minikube service jenkins`.) Follow these steps to create a multibranch pipeline job:

1. Navigate to the Create New Jobs page.
2. Enter an item name, market-data; select Multibranch Pipeline as the job type; and click OK.
3. On the following page (see figure 10.4), select a Branch Source of Git and add your repository's clone URL to the Project Repository field. If you're using a private Git repository, you'll also need to configure your credentials.
4. Elect to periodically scan the pipeline every minute. This will trigger builds if changes are detected.
5. Save your changes.

Once you've saved your changes, Jenkins will scan your repository for branches containing a Jenkinsfile. The multibranch pipeline job type will generate a unique build for each branch in your repository—later, this will let you treat feature branches differently from the master branch.

> **TIP** Instead of clicking through the UI, you can use the Jenkins Job DSL to generate pipeline jobs. This is (another) Groovy DSL that generates jobs in Jenkins' underlying XML format. You can find examples in the project documentation (https://github.com/jenkinsci/job-dsl-plugin/wiki).

Once the indexing is complete, Jenkins will run a build for your master branch. Clicking on the name of the branch will take you the build history for that branch (figure 10.5).

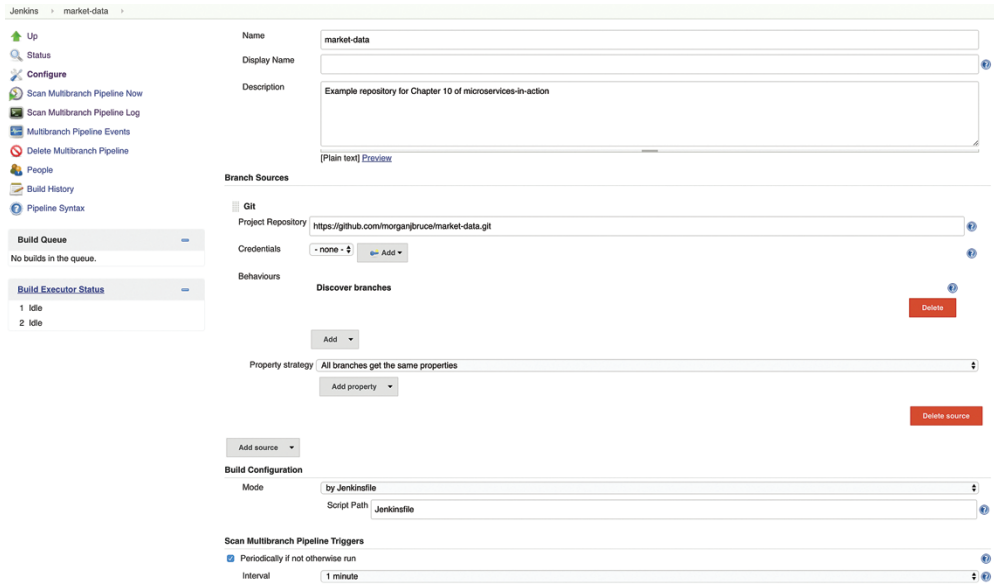


Figure 10.4 New project configuration screen, showing Branch Sources options

Click the build number and then Console Output. This traces the output of the build. Within that output, you should be able to see how your Jenkinsfile has been executed:

```
Agent default-q3ccc is provisioned from template Kubernetes Pod Template
Agent specification [Kubernetes Pod Template] (jenkins-jenkins-slave):
* [jnlp] jenkins/jnlp-slave:3.10-1(resourceRequestCpu: 200m, resourceRequest
➥Memory: 256Mi, resourceLimitCpu: 200m, resourceLimitMemory: 256Mi)

Running on default-q3ccc in /home/jenkins/workspace/market-data_master
➥-27MDVADAYDBX5WJSRWQIFEL3T7GD4LWPU5CXCZNTJ4CKBDLP3LVA
[Pipeline] {
[Pipeline] checkout
Cloning the remote Git repository
Cloning with configured refspecs honoured and without tags
Cloning repository https://github.com/morganjbruce/market-data.git
 > git init /home/jenkins/workspace/market-data_master
[CA}-27MDVADAYDBX5WJSRWQIFEL3T7GD4LWPU5CXCZNTJ4CKBDLP3LVA # timeout=10
Fetching upstream changes from https://github.com/morganjbruce/
➥market-data.git
 > git --version # timeout=10
 > git fetch --no-tags --progress https://github.com/morganjbruce/
➥market-data.git +refs/heads/*:refs/remotes/origin/*
 > git config remote.origin.url https://github.com/morganjbruce/
➥market-data.git # timeout=10
 > git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/
➥* # timeout=10
 > git config remote.origin.url https://github.com/morganjbruce/
➥market-data.git # timeout=10
Fetching without tags
```

```
Fetching upstream changes from https://github.com/morganjbruce/
➥market-data.git
 > git fetch --no-tags --progress https://github.com/morganjbruce/
➥market-data.git +refs/heads/*:refs/remotes/origin/*
Checking out Revision 80bfb7bdc4fa0b92dcf360393e5d49e0f348b43b (master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 80bfb7bdc4fa0b92dcf360393e5d49e0f348b43b
Commit message: "working through ch10"
First time build. Skipping changelog.
[Pipeline] echo
Latest commit id: 80bfb7bdc4fa0b92dcf360393e5d49e0f348b43b
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // stage
[Pipeline] End of Pipeline
Finished: SUCCESS
```

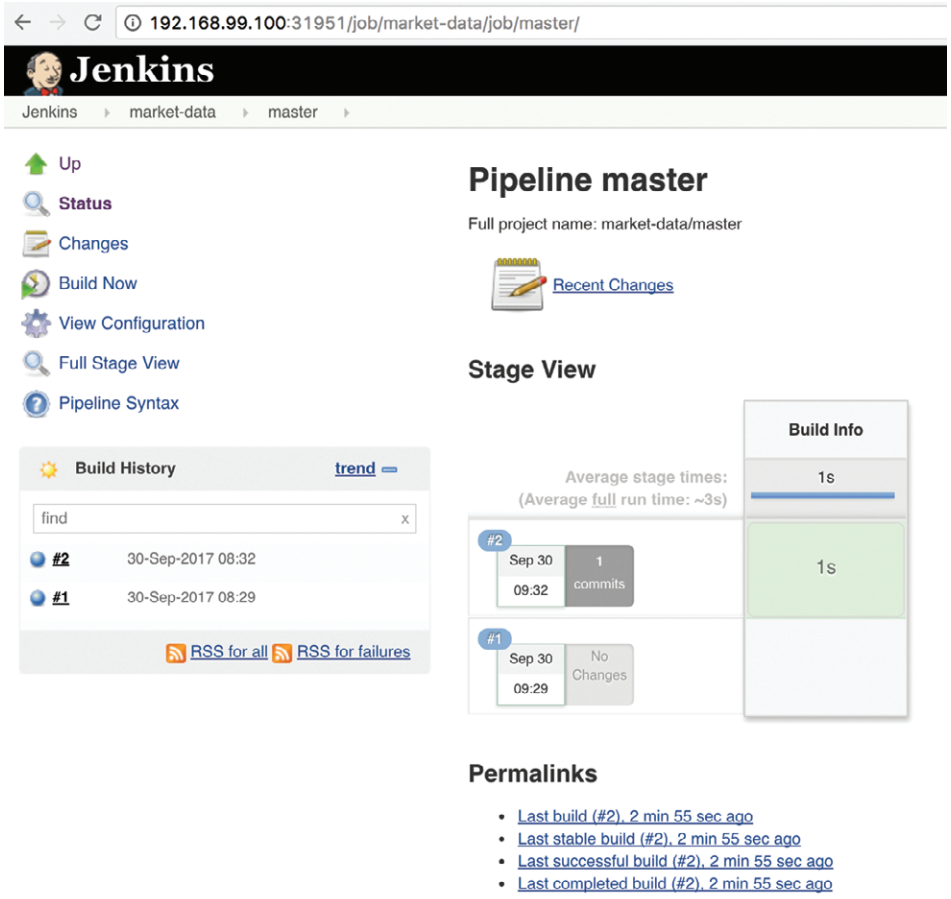


Figure 10.5 Build history for the master branch of your repository

Each `[Pipeline]` step traces the execution of your code. Awesome—you've deployed a build automation tool, configured it against a service repository, and run your first build pipeline! Next, let's look at the first stage of your pipeline: build.

10.2.2 Building your image

You're going to use Docker to build and package your images. First, let's change your Jenkinsfile, as shown in the following listing.

Listing 10.2 Jenkinsfile for build step

```
def withPod(body) {
  podTemplate(label: 'pod', serviceAccount: 'jenkins', containers: [
      containerTemplate(name: 'docker', image: 'docker', command: 'cat',
➥ttyEnabled: true),
      containerTemplate(name: 'kubectl', image: 'morganjbruce/kubectl',
➥command: 'cat', ttyEnabled: true)
    ],
    volumes: [
      hostPathVolume(mountPath: '/var/run/docker.sock', hostPath:
➥'/var/run/docker.sock'),
    ]
 ) { body() }
}

withPod {
  node('pod') {
    def tag = "${env.BRANCH_NAME}.${env.BUILD_NUMBER}"
    def service = "market-data:${tag}"

    checkout scm

    container('docker') {
      stage('Build') {
        sh("docker build -t ${service} .")
      }
    }
  }
}
```

Defines a pod template to use to run your job

Requests an instance of your pod template

Checks out the latest code from Git

Enters the Docker container of your pod

Starts a new pipeline stage

Runs a docker command to build your service image

This script will build your service and tag the resulting Docker container with the current build number. It's definitely more complex than the earlier version, so let's take a quick walk through what you're doing:

1. You define a pod template for your build, which Jenkins will use to create pods on Kubernetes for a build agent. This pod contains two containers—Docker and kubectl.

2. Within that pod, you check out the latest version of your code from Git.
3. You then start a new pipeline stage, which you've called `Build`.
4. Within that stage, you enter the Docker container and run a `docker` command to build your service image.

TIP Jenkins also provides a Groovy DSL for Docker instead of the shell commands you've used. For example, you could use `docker.build(imageName)` in place of the `sh` call in listing 10.5.

Commit this new Jenkinsfile to your Git repo and navigate to the build job on Jenkins. Wait for a rerun—or trigger the job manually—and in the console output, you should see your container image being built successfully.

10.2.3 Running tests

Next, you should run some tests. This should be like any other continuous integration job: if the tests are green, deployment can proceed; if not, you halt the pipeline. At this stage, you aim to provide rapid and accurate feedback on the quality of a changeset. Fast test suites help engineers iterate quickly.

Building your code and performing unit tests are only two of the possible activities you might perform during this commit stage of the build pipeline. Table 10.1 outlines other possibilities.

Table 10.1 Possible activities in the commit stage of a deployment pipeline

Activity	Description
Unit tests	Code-level tests
Compilation	Compiling the artifact into an executable artifact
Dependency resolution	Resolving external dependencies—for example, open source packages
Static analysis	Evaluating code against metrics
Linting	Checking syntax and stylistic principles of code

For now, you should get your unit tests running. Add a `Test` stage to your Jenkinsfile, immediately after the `Build` stage as shown in the next listing.

Listing 10.3 Test stage

```
stage('Test') {
  sh("docker run --rm ${service} python setup.py test")
}
```

Commit your Jenkinsfile and run the build. This will add a new stage to your pipeline, which executes the test cases defined in /tests (figure 10.6).

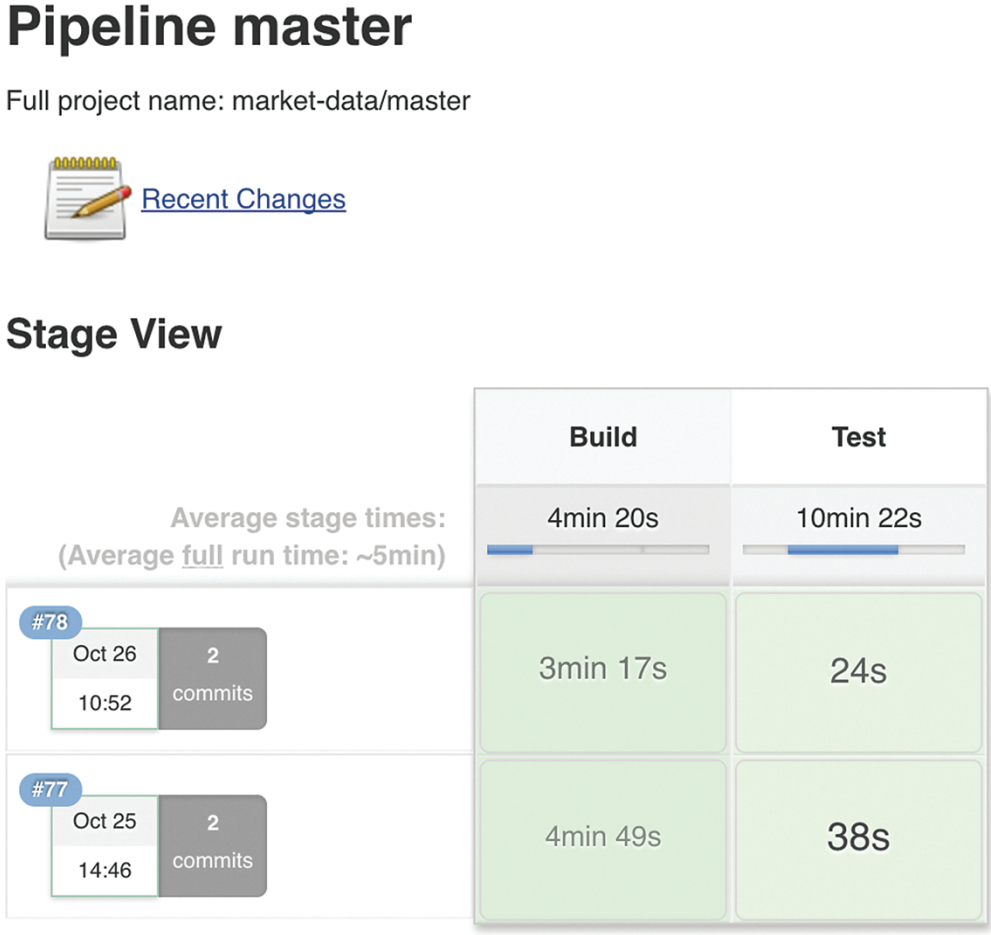


Figure 10.6 Your pipeline so far with `Build` and `Test` stages

Unfortunately, this code alone won't make the results visible in the build. Only success or failure will do that. You can archive the XML results you're generating by adding the following to your Jenskinsfile.

Listing 10.4 Archiving results from test stage

```
stage('Test') {
  try {
    sh("docker run -v `pwd`:/workspace --rm ${service}
➥python setup.py test")
  } finally {
    step([$class: 'JUnitResultArchiver', testResults:
➥'results.xml'])
  }
}
```

Mounts the current workspace as a volume

Archives the results that the test job generates

This code mounts the current workspace as a volume within the Docker container. The `python` test process will write output to that volume as /workspace/result.xml, and you can access those results even after Docker has stopped and removed the service container. You use the `try–finally` statement to ensure you achieve results regardless of pass or failure.

> **TIP** A good build pipeline directs feedback to the responsible engineering team. For example, our deployment pipelines at Onfido notify commit authors through Slack and email if pipeline stages fail. We also emit events from our pipeline for monitoring tools such as PagerDuty to consume. For more on sending notifications, see the Jenkins documentation (http://mng.bz/C5X3).

Committing your changed Jenkinsfile and running a fresh build will store test results in Jenkins. You can view them on the build page. Great—you've validated your underlying code. Now you're almost ready to deploy.

10.2.4 Publishing artifacts

You need to publish an artifact—in this case, our Docker container image—to be able to deploy it. If you used a private Docker registry in chapter 9, you'll need to configure your Docker credentials within Jenkins:

1. Navigate to Credentials > System > Global Credentials > Add Credentials.
2. Add username and password credentials, using your credentials to https://hub.docker.com.
3. Set the ID as dockerhub and click OK to save these credentials.

If you're using a public registry, you can skip this step. Either way, when you're ready, add a third step to your Jenkinsfile, as follows.

Listing 10.5 Publishing artifacts

```
def tagToDeploy = "[your-account]/${service}"

stage('Publish') {
  withDockerRegistry(registry: [credentialsId:
➥'dockerhub']) {
    sh("docker tag ${service} ${tagToDeploy}")
    sh("docker push ${tagToDeploy}")
  }
}
```

The target public image tag—replace with your account name

Logs in to the Docker registry using stored credentials

Tags the image with your Docker account name

When you have that ready, commit and run your build. Jenkins will publish your container to the public Docker registry.

10.2.5 Deploying to staging

At this point, you've tested the service internally but in complete isolation; you haven't interacted with any of the service's upstream or downstream collaborators. You could deploy directly to production and hope for the best, but you probably shouldn't. Instead, you can deploy to a staging environment where you can run further automated and manual tests against real collaborators.

You're going to use Kubernetes namespaces to logically segregate your staging and production environments. To deploy your service, you'll use `kubectl`, using an approach similar to the one you took in chapter 9. Rather than installing the tool on Jenkins, you can use Docker to wrap this command-line tool. This is quite a useful technique.

> **WARNING** Logical segregation isn't always appropriate in a real-world environment. Compliance and security standards, such as PCI DSS, often mandate network-level isolation between production and development workloads, which Kubernetes namespaces wouldn't currently satisfy. In addition, completely separating staging and production infrastructure reduces the risk of a "noisy neighbor" in staging, such as a resource-hungry service, affecting production reliability.

First, let's look at your deployment and service definition. You should save the following listing to deploy/staging/market-data.yml within your market-data repo.

Listing 10.6 Deployment specification for market-data

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: market-data
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 50%
      maxSurge: 50%
  template:
    metadata:
      labels:
        app: market-data
        tier: backend
        track: stable
    spec:
      containers:
      - name: market-data
```

```
image: BUILD_TAG          <-- A placeholder for the image you want to deploy
resources:
  requests:
    cpu: 100m
    memory: 100Mi
ports:
- containerPort: 8000
livenessProbe:
  httpGet:
    path: /ping
    port: 8000
  initialDelaySeconds: 10
  timeoutSeconds: 15
readinessProbe:
  httpGet:
    path: /ping
    port: 8000
  initialDelaySeconds: 10
  timeoutSeconds: 15
```

If you saw this in chapter 9, you'll notice one key difference: you don't set a specific image tag to deploy, only a placeholder of `BUILD_TAG`. You'll replace this in your pipeline with the version you're deploying. This is a little unsophisticated—as you build more complex deployments, you might want to explore higher level templating tools, such as ksonnet (https://ksonnet.io).

You'll also want to add market-data-service.yml, as shown in the following listing, to the same location.

Listing 10.7 market-data service definition

```
---
apiVersion: v1
kind: Service
metadata:
  name: market-data
spec:
  type: NodePort
  selector:
    app: market-data
    tier: backend
  ports:
    - protocol: TCP
      port: 8000
      nodePort: 30623
```

Before you deploy, create distinct namespaces to segregate your workloads, using kubectl:

```
kubectl create namespace staging
kubectl create namespace canary
kubectl create namespace production
```

Now, add a deploy stage to your pipeline, as follows.

Listing 10.8 Deployment to staging (Jenkinsfile)

```
stage('Deploy') {
  sh("sed -i.bak 's#BUILD_TAG#${tagToDeploy}#' ./
➥deploy/staging/*.yml")

  container('kubectl') {
    sh("kubectl --namespace=staging apply -f deploy/
➥staging/")
  }
}
```

Uses sed to replace BUILD_TAG with the name of your new Docker image

Applies all configuration files in deploy/staging to your local cluster, using the staging namespace

Again, commit and run the build. This time, a Kubernetes deploy should be triggered! You can check the status of this deployment using `kubectl rollout status`:

```
$ kubectl rollout status -n staging deployment/market-data
Waiting for rollout to finish: 2 of 3 updated replicas are available...
deployment "market-data" successfully rolled out
```

As you can see, although your build was marked as complete, the deployment itself takes some time to roll out. This is because `kubectl apply` works asynchronously and doesn't wait for the cluster to finish updating to reflect the new state. If you like, you can add a call to the above `kubectl rollout status` method within the Jenkinsfile so that Jenkins waits for rollouts to complete before proceeding.

Either way, once the rollout is complete, you can access this service:

```
$ curl `minikube service --namespace staging --url market-data`/ping
HTTP/1.0 200 OK
Content-Type: text/plain
Server: Werkzeug/0.12.2 Python/3.6.1
```

This example service doesn't do much. For your own services, you might trigger further automated testing or perform further exploratory testing of the service and code changes you've just deployed. Table 10.2 outlines some of the activities you might perform at this stage of a deployment pipeline. For now, great work—you've automated your first microservice deployment!

Table 10.2 Possible activities to perform to validate a staging release of a microservice

Acceptance testing	Automated tests	Running automated tests to check expectations, either regression or acceptance
	Manual tests	Some services may require manual validation or exploratory testing.
Nonfunctional testing	Security tests	Testing the security posture of the service
	Load/capacity tests	Validating expectations about capacity and load on a service

10.2.6 Staging environments

Let's take a break for a moment to discuss staging environments. You should make any new release of a service to staging first. Microservices need to be tested together, and production isn't the first place where that should happen.

The infrastructure configuration of your staging environment should be an exact copy of production, albeit with less real traffic. It doesn't need to run at the same scale. The volume and type of testing you'll use to put your services through their paces can determine the necessary size. As well as conducting various types of automated testing, you might manually validate services in staging to ensure they meet acceptance criteria.

Along with shared staging environments, you might also run isolated staging environments for individual or small sets of closely related services. Unlike full staging, these environments might be ephemeral and spun up on-demand for the duration of testing. This is useful for testing a feature in relative isolation, with tighter control of the state of the environment. Figure 10.7 compares these different approaches to staging environments.

Although staging environments are crucial, they can be hard to manage in a microservice application, as well as the source of significant contention between teams. A microservice might have many dependencies, all of which should be present and stable in full staging. Although a service in staging will have passed testing, code review, and other quality gates, it's still possible that services in staging will be less stable than their production equivalents, and that can cause chaos. Any engineer deploying to a shared environment needs to act as a good neighbor to ensure that issues with services they own don't substantially impact another team's ability to smoothly test (and therefore deliver) other services.

> **TIP** To further reduce friction in staging, consider building your deployment pipeline to allow any engineer to easily roll back the last deployment, regardless of whether they own that service or not.

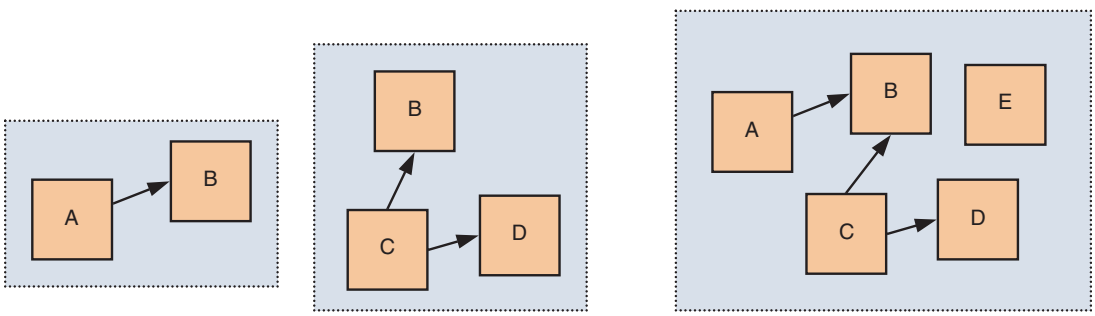


Figure 10.7 Isolated versus full staging environments

10.2.7 Deploying to production

You can use what you've learned so far to take this service to production. Table 10.3 outlines some of the different actions you might perform at this stage of your pipeline.

Table 10.3 Possible activities to perform in deployment

Code deployment	Deploying code to a runtime environment
Rollback	Rolling back code to a previous version, if errors or unexpected behavior occurs
Smoke tests	Validating the behavior of a system using light-touch tests

In this case, if a deployment to staging is successful, here's what should happen next:

1. Your pipeline should wait for human approval to proceed to production.
2. When you have approval, you'll release a canary instance first. This helps you validate that your new build is stable when it faces real production traffic.
3. If you're happy with the performance of the canary instance, the pipeline can proceed to deploy the remaining instances to production.
4. If you're not happy, you can roll back your canary instance.

First, you should add an approval stage. In continuous delivery—unlike continuous deployment—you don't necessarily want to push every commit immediately to production. Add the following to your Jenkinsfile.

Listing 10.9 Approving a production release

```
stage('Approve release?') {
  input message: "Release ${tagToDeploy} to production?"
}
```

Running this code in Jenkins will show a dialog box in the build pipeline view, with two options: Proceed or Abort. Clicking Abort will cancel the build; clicking Proceed will, for now, cause the build to finish successfully—you haven't added a deploy step!

First, try a production deploy without a canary instance. Copy the YAML files you created earlier, from listings 10.6 and 10.7, to a new deploy/production directory. Feel free to increase the number of replicas you'll deploy.

Next, add the code in the next listing to your Jenkinsfile, after the approval stage. This is similar to the code you used in staging. Don't worry about the code duplication for now—you can work on that in a moment.

Listing 10.10 Production release stage

```
stage('Deploy to production') {
  sh("sed -i.bak 's#BUILD_TAG#${tagToDeploy}#' ./deploy/production/*.yml")

  container('kubectl') {
```

```
    sh("kubectl --namespace=production apply -f deploy/production/")
  }
}
```

As always, commit and run the build in Jenkins. If successful, you've released to production! Let's take this a few steps further and add some code to release a canary instance. But before you add a new stage, let's DRY up your code a little bit. You can move your release-related code into a separate file called deploy.groovy, as shown in the following listing.

Listing 10.11 deploy.groovy

Works for any namespace and deployment

```
def toKubernetes(tagToDeploy, namespace,
➥deploymentName) {
  sh("sed -i.bak 's#BUILD_TAG#${tagToDeploy}#' ./deploy/${namespace}/*.yml")

  container('kubectl') {
    kubectl("apply -f deploy/${namespace}/")
  }
}

def kubectl(namespace, command) {
  sh("kubectl --namespace=${namespace} ${command}")
}

def rollback(deploymentName) {
  kubectl("rollout undo deployment/${deploymentName}")
}

return this;
```

Performs any operations on Kubernetes

Then you can load it in your Jenkinsfile, as shown in the following listing.

Listing 10.12 Using deploy.groovy in your Jenkinsfile

```
def deploy = load('deploy.groovy')

stage('Deploy to staging') {
  deploy.toKubernetes(tagToDeploy, 'staging', 'market-data')
}

stage('Approve release?') {
  input "Release ${tagToDeploy} to production?"
}

stage('Deploy to production') {
  deploy.toKubernetes(tagToDeploy, 'production', 'market-data')
}
```

That's much cleaner. This isn't the only way to reuse pipeline code—we'll discuss a better approach in section 10.3.

Next, create a canary deployment file. If you've read through chapter 9, you'll remember that you use a distinct deployment with unique labels to identify this instance. In deploy/canary, create a deployment YAML file like the one you used for production but with three changes:

1. Add a label `track: canary` to the pod specification.
2. Reduce the number of replicas to 1.
3. Change the name of the deployment to market-data-canary.

After you've added that file, add a new stage to your deployment, as shown in the following listing, before releasing to production.

Listing 10.13 Canary release stage (Jenkinsfile)

```
stage('Deploy canary') {
  deploy.toKubernetes(tagToDeploy, 'canary', 'market-data-canary')

  try {
    input message: "Continue releasing ${tagToDeploy} to
➥production?"
  } catch (Exception e) {
    deploy.rollback('market-data-canary')
  }
}
```

Asks for human input to proceed

Rolls back your canary if rollout is aborted

In this example, we're assuming human approval for moving from canary to production. In the real world, this might be an automated decision; for example, you could write code to monitor key metrics, such as error rate, for some time after a canary deploy.

Once you've committed this code, you should be able to run the whole pipeline. Figure 10.8 illustrates the full journey of your code to production.

Let's take a breather so you can reflect on what you've learned:

- You've automated the delivery of code from commit to production by using Jenkins to build a structured deployment pipeline.
- You've built different stages to validate the quality of that code and provide appropriate feedback to an engineering team.
- You've learned about the importance of—and challenges in operating—a staging environment when developing microservices.

These techniques provide a consistent and reliable foundation for delivering code safely and rapidly to production. This helps ensure overall stability and robustness in a microservice application. But it's far from ideal if every microservice copies and pastes the same deployment code or reinvents the wheel for every new service. In the next section, we'll discuss patterns for making deployment approaches reusable across a fleet of services.

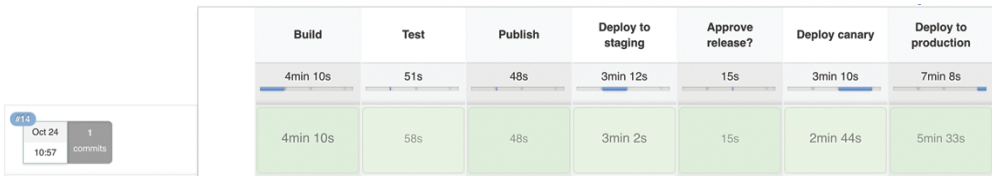


Figure 10.8 Successful deployment pipeline from commit to production release

10.3 Building reusable pipeline steps

Microservices enable independence and technical homogeneity, but these advantages come at a cost:

- It's harder for developers to move between teams, as the tech stack can vastly differ.
- It's more complex for engineers to reason through the behavior of different services.
- You have to invest more time in different implementations of the same concerns, such as deployment, logging, and monitoring.
- People may make technical decisions in isolation, risking local, rather than global, optimization.

To balance these risks while maintaining technical freedom and flexibility, you should aggressively standardize the platform and tooling that services operate on. Doing so will ensure that, even if the technology stack changes, common abstractions remain as close as possible across different services. Figure 10.9 illustrates this approach.

NOTE We introduced the platform layer of microservice architecture in chapter 3.

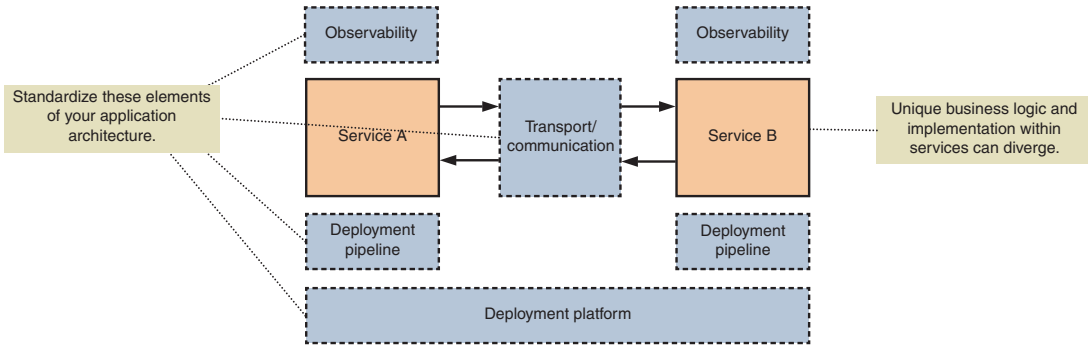


Figure 10.9 You can standardize many elements of a microservice application to reduce complexity, increase reuse, and reduce ongoing operational cost.

Over the past few chapters, you've applied this thinking in a few areas:

- Using a microservice chassis to abstract common, nonbusiness logic functionality, such as monitoring and service discovery
- Using containers—with Docker—as a standardized service artifact for deployment
- Using a container scheduler—Kubernetes—as a common deployment platform

You also can apply this approach to your deployment pipelines.

10.3.1 Procedural versus declarative build pipelines

The pipeline scripts you've written so far have three weaknesses:

1. *Specific*—They're tied to a single repository, and another repository can't share them.
2. *Procedural*—They explicitly describe how you want the build to be carried out.
3. *Don't abstract internals*—They assume a lot of knowledge about Jenkins itself, such as how you start nodes, run commands, and use command-line tools.

Ideally, a service deployment pipeline should be declarative: an engineer describes what they expect to happen (test my service, release it, and so on), and your framework decides how to execute those steps. This approach also abstracts away changes to how those steps happen: if you want to tweak how a step works, you can change the underlying framework implementation. Abstracting these implementation decisions away from individual services leads to greater consistency across the microservice application.

Compare the following script to the Jenkinsfile you wrote earlier in the chapter.

Listing 10.14 Example declarative build pipeline

```
service {
  name('market-data')

  stages {
    build()
    test(command: 'python setup.py test', results: 'results.xml')
    publish()
    deploy()
  }
}
```

This script defines some common configuration (service name) and a series of steps (build, test, publish, deploy) but hides the complexity of executing those steps from a service developer. This allows any engineer to quickly follow best practice to reliably and rapidly take a new service to production.

With Jenkins Pipeline, you can implement declarative pipelines using shared libraries. We won't go into detail in this chapter—not enough pages left!—but this book's

Github repository includes an example pipeline library (http://mng.bz/P7hD). In addition, the Jenkins documentation (http://mng.bz/p3wz) provides a detailed reference on using shared libraries.

> **NOTE** In other build tools, such as Travis CI or DroneCI, you declare build configuration using YAML files. These approaches are great, especially if your needs are relatively straightforward. Conversely, building a DSL with a dynamic language can offer an extra degree of flexibility and extensibility.

10.4 Techniques for low-impact deployment and feature release

Throughout the past few chapters, we've used the terms *deployment* and *release* interchangeably. But in a microservice application, it's important to distinguish between the technical activity of deployment—updating the software version running in a production environment—and the decision to release a new feature to customers or consuming services.

You can use two techniques—dark launches and feature flags—to complement your continuous delivery pipeline. These techniques will allow you to deploy new features without impacting customers and provide a flexible mechanism for rollback.

10.4.1 Dark launches

Dark launching is the practice of deploying a service to a production environment significantly prior to making it available to consumers. At our company, we practice this regularly and try to deploy within the first few days of building a new service, regardless of whether it's feature-complete. Doing this allows us to perform exploratory testing from an early stage, which helps us understand how a service behaves and makes a new service visible to our internal collaborators.

In addition, dark launching to a production environment allows you to test your services against real production traffic. Let's say that SimpleBank wants to offer a new financial prediction algorithm as a service. By passing production traffic in parallel with the existing service, they can easily benchmark the new algorithm and understand how it performs in the real world, rather than against limited and artificial test scenarios (figure 10.10).

Whether you validate this output manually or automatically depends on the nature of the feature and the volume and distribution of requests required to adequately exhaust possible scenarios. The dark launch approach is also useful for testing that refactoring doesn't regress sensitive functionality.[1]

[1] Similarly, the Ruby Scientist gem (https://github.com/github/scientist) was originally designed to help Github validate whether refactoring of user permissions caused authorization issues, such as users having incorrect access to repositories.

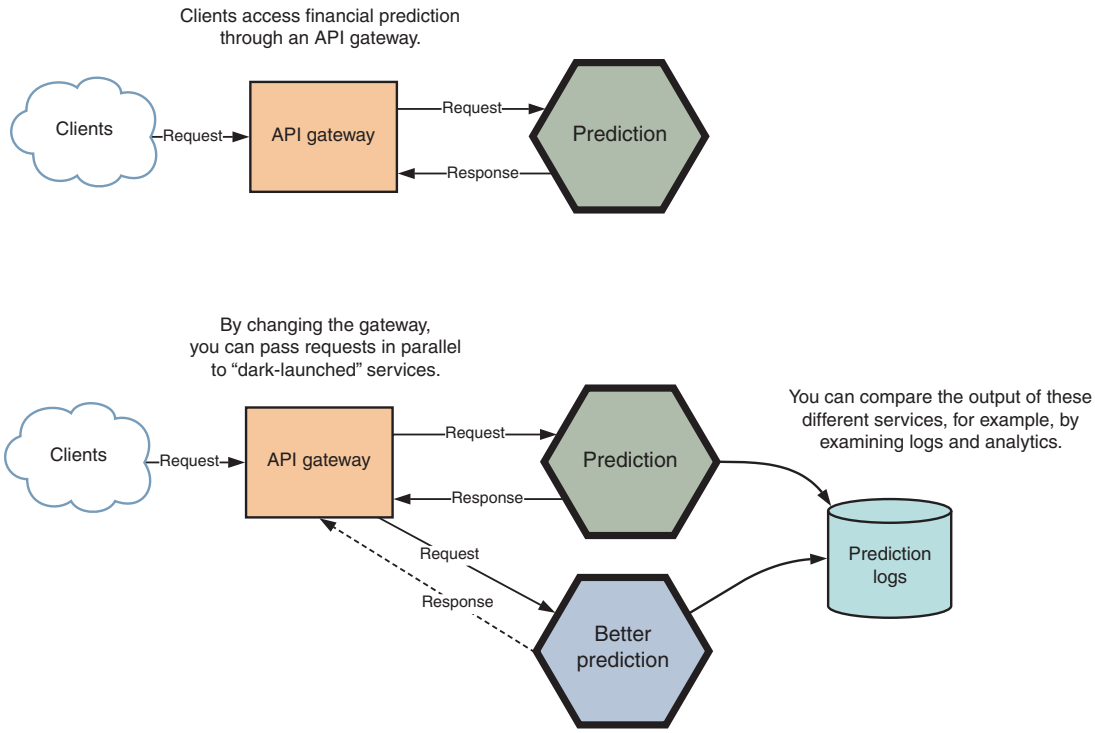


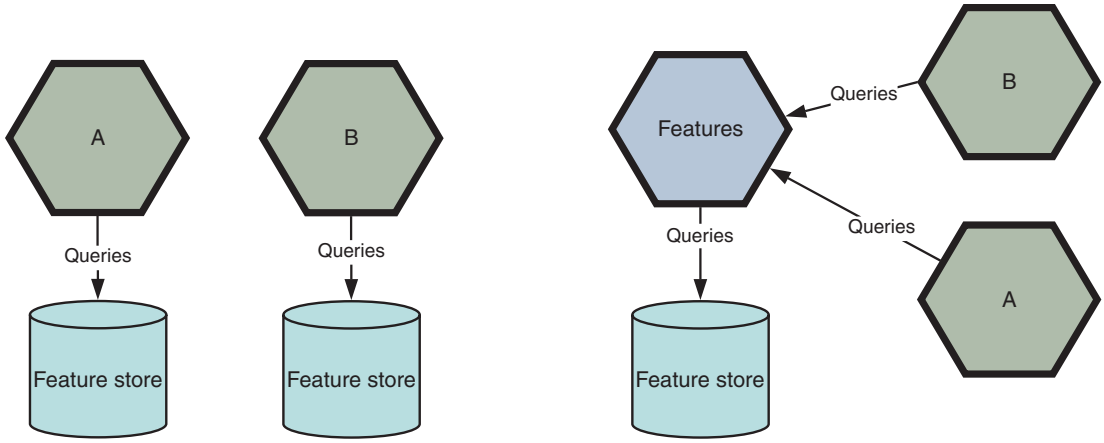
Figure 10.10 Dark launches enable validation of new service behavior against real production traffic without exposing features to customers.

10.4.2 Feature flags

Feature flags control the availability of features to customers. Unlike dark launches, you can use them at any point in the lifecycle of a service, such as a feature release. A feature flag (or toggle) wraps a feature in conditional logic, only enabling it for a certain set of users. Many companies will use them to control rollout; for example, only releasing a feature for internal staff first, or progressively increasing the number of users who can access a feature over time.

Several libraries are available to implement feature flags, such as Flipper (http://github.com/jnunemaker/flipper) or Togglz (http://github.com/togglz/togglz). These libraries typically use a persistent backing store, like Redis, to maintain the state of feature flags for an application. In a larger microservice application, you may find it desirable to have a single feature store to synchronize the rollout of features that involve the interaction of multiple services, rather than independently managing features per service. Figure 10.11 illustrates these different approaches.

Managing features per service is likely to be easier in a small microservice system than a larger one. As your system becomes larger, centralizing feature configuration in a single service reduces coordination overhead if you encounter situations where feature rollouts necessitate changes in multiple microservices.



Approach 1: Each service owns and maintains a separate feature store.

Approach 2: A features service owns all feature configuration, and other services call it.

Figure 10.11 You can store feature flags centrally—owned by one service—or maintain them in separate applications.

By controlling which users see a change, feature flags can aid in minimizing the potential impact of any change to a system, as you have partial control over code execution and feature availability. If errors occur, feature flags often allow for more rapid recovery than typical rollback. For microservices, they can enable safer release of new functionality without adversely affecting service consumers.

Summary

- A microservice deployment process should meet two goals: safety at pace and consistency.
- The time it takes to deploy a new service is often a barrier in microservice applications.
- Continuous delivery is an ideal deployment practice for microservices, reducing risk through the rapid delivery of small, validated changesets.
- A good continuous delivery pipeline ensures visibility, correctness, and rich feedback to an engineering team.
- Jenkins is a popular build automation tool that uses a scripting language to tie multiple tools together into a delivery pipeline.
- Staging environments are invaluable but can be challenging to maintain when they face a high volume of independent change.
- You can reuse declarative pipeline steps across multiple services; aggressive standardization makes deployment predictable across teams.
- To provide fine-grained control over rollout and rollback, you should manage the technical activity of deployment separately from the business activity of releasing a feature.

Part 4

Observability and ownership

Once you've deployed your services, you need to know what they're actually doing. In this part, you'll build a monitoring system—using metrics, tracing, and logging—to give you rich visibility into your microservice application. After that, we'll conclude our microservice journey by exploring how this architectural approach impacts how developers work together and discussing good day-to-day practices for developing microservice applications.

11 Building a monitoring system

This chapter covers

- Understanding what signals to gather from running applications
- Building a monitoring system to collect metrics
- Learning how to use the collected signals to set up alerts
- Observing the behavior of individual services and their interactions as a system

You've now set up an infrastructure to run your services and have deployed multiple components that you can combine to provide functionality to your users. In this chapter and the next, we'll consider how you can make sure you'll always be able to know how those components are interacting and how the infrastructure is behaving. It's fundamental to know as early as possible when something isn't behaving as expected. In this chapter, we'll focus on building a monitoring system so you can collect relevant metrics, observe the system behavior, and set up relevant alerts to allow you to keep your systems running smoothly by taking actions preemptively.

When you can't be preemptive, you'll at least be able to quickly pinpoint the areas that need your attention so you can address any issues. It's also worth mentioning that you should instrument as much as possible. The collected data you may not use today may turn out to be useful someday.

11.1 A robust monitoring stack

A robust monitoring stack will allow you to start gathering metrics from your services and infrastructure and use those metrics to gather insights from the operation of a system. It should provide a way to collect data and store, display, and analyze it.

You should start by emitting metrics from your services, even if you have no monitoring infrastructure in place. If you have those metrics stored, at any time you'll be able to access, display, and interpret them. Observability is a continuous effort, and monitoring is a key element in that effort. Monitoring allows you to know whether a system is working, whereas observability lets you ask why it's not working.

In this chapter, we'll be focusing on monitoring, metrics, and alerts. We'll explain logs and traces in chapter 12, and they'll constitute the observability component.

Monitoring doesn't only allow you to anticipate or react to issues, you can also use collected metrics from monitoring to predict system behavior or to provide data for business analytic purposes.

Multiple open source and commercial options are available for setting up a monitoring solution. Depending on the team size and resources available, you may find that a commercial solution may be easier or more convenient to use. Nonetheless, in this chapter you'll be using open source tools to build your own monitoring system. Your stack will be made up of a metrics collector and a display and alerting component. Logs and traces are also essential to achieve system observability. Figure 11.1 gives an overview of all the components you need to be able to understand your system behavior and achieve observability.

In figure 11.1, we display the components of a monitoring stack:

- Metrics
- Logs
- Traces

Each of these components feeds into its own dashboards as an aggregation of data from multiple services. This allows you to set up automated alerts and look into all the collected data to investigate any issues or better understand system behavior. Metrics will enable monitoring, whereas logs and traces will enable observability.

11.1.1 Good monitoring is layered

In chapter 3, we discussed the architecture tiers: client, boundary, services, and platform. You should implement monitoring in all of these layers, because you can't determine the behavior of a given component in total isolation. A network issue will most likely affect a service. If you collect metrics at the service level, the only thing you'll be

able to know is that the service itself isn't serving requests. That alone tells you nothing about the cause of the issue. If you also collect metrics at the infrastructure level, you can understand problems that'll most likely affect multiple other components.

In figure 11.2, you can see the services that work together to allow a client to place an order for selling or buying shares. Multiple services are involved. Some communication between services is synchronous, either via RPC or HTTP, and some is asynchronous, using an event queue. To be able to understand how services are performing, you need to be able to collect multiple data points to monitor and either diagnose issues or prevent them before they even arise.

Monitoring individual services will be of little to no use because services provide isolation but don't exist isolated from the outside world. Services often depend on each other and on the underlying infrastructure (for example, the network, databases, cache stores, and event queues). You can get a lot of valuable information by monitoring services, but you need more. You need to understand what's going on in all your layers.

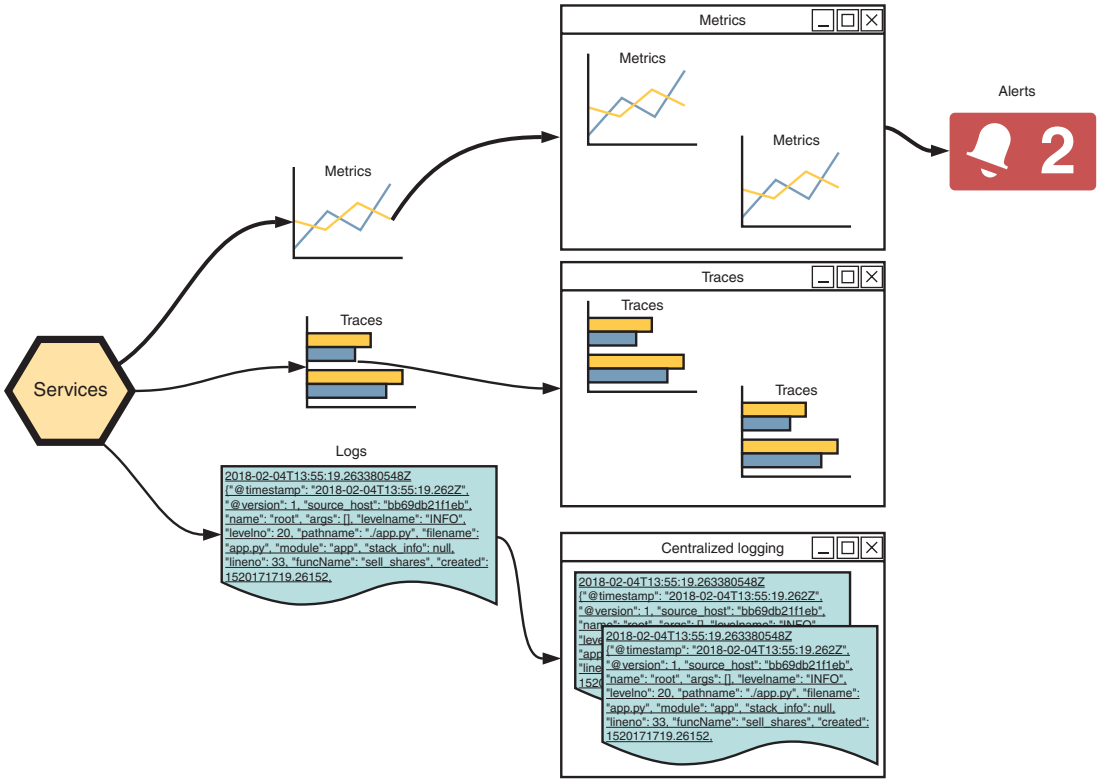


Figure 11.1 Components of a monitoring stack—metrics, traces, and logs—each aggregated in their own dashboards

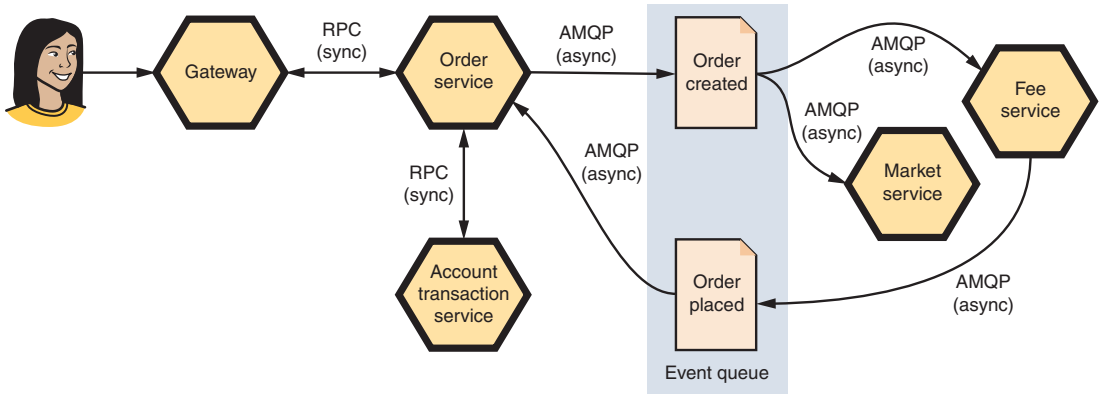


Figure 11.2 Services involved in placing orders and their communication protocols

Your monitoring solution should allow you to know what is broken or degrading and why. You'll be able to quickly reveal any symptoms and use the available monitors to determine causes.

Referring to figure 11.2, it's worth mentioning that symptoms and causes vary depending on the observation point. If the market service might be having issues communicating with the stock exchange, you can diagnose that by measuring response times or HTTP status codes for that interaction. In that situation, you'll be almost sure that the place order feature won't be working as expected.

But what if you have an issue with connectivity from services to the event queue? Services won't be publishing messages, so downstream services won't be consuming them. In that situation, no service is failing because no service is performing any work. If you have proper monitoring in place, it can alert you to the abnormal decrease in throughput. You can set your monitoring solution to send you automated notifications when the number of messages in a given queue goes below a certain threshold.

Lack of messages isn't the only thing that can indicate issues, though. What if you have messages accumulating in a given queue? Such accumulation may indicate the services that consume messages from the queue are either not working properly or are having trouble keeping up with increased demand. Monitoring allows you to identify issues or even predict load increases and act accordingly to maintain service quality. Let's take some time for you to learn a bit more about the signals you should collect.

11.1.2 Golden signals

You should focus on four golden signals while collecting metrics from any user-facing system: latency, errors, traffic, and saturation.

LATENCY

Latency measures how much time passes between when you make a request to a given service and when the service completes the request. You can determine a lot from this signal. For example, you can infer that the service is degrading if it shows increasing

latency. You need to take extra care, though, in correlating this signal with errors. Imagine you're serving a request and the application responds quickly but with an error? Latency has a low value in this case, but the outcome isn't the desired one. It's important to keep the latency of requests that result in errors out of this equation, because it can be misleading.

Errors

This signal determines the number of requests that don't result in a successful outcome. The errors may be explicit or implicit—for example, having an HTTP 500 error versus having an HTTP 200 but with the wrong content. The latter isn't trivial to monitor for because you can't rely solely on the HTTP codes, and you may only be able to determine the error by finding wrong content in other components. You generally catch these errors with end-to-end or contract tests.

Traffic

This signal measures the demand placed on a system. It can vary depending on the type of system being observed, the number of requests per second, network I/O, and so on.

Saturation

At a given point, this measures the capacity of the service. It mainly applies to resources that tend to be more constrained, like CPU, memory, and network.

11.1.3 Types of metrics

While collecting metrics, you need to determine the type that's best suited for a given resource you're aiming to monitor.

Counters

Counters are a cumulative metric representing a single numerical value that'll always increase. Examples of metrics using counters are:

- Number of requests
- Number of errors
- Number of each HTTP code received
- Bytes transmitted

You shouldn't use a counter if the metric it represents can also decrease. For that, you should use a gauge instead.

Gauges

Gauges are metrics representing single numerical arbitrary values that can go up or down. Some examples of metrics using gauges are:

- Number of connections to a database
- Memory used
- CPU used
- Load average
- Number of services operating abnormally

HISTOGRAMS

You use histograms to sample observations and categorize them in configurable buckets per type, time, and so on. Examples of metrics represented by histograms are:

- Latency of a request
- I/O latency
- Bytes per response

11.1.4 Recommended practices

As we already mentioned, you should make sure you instrument as much as possible to collect as much data as you can about your services and infrastructure. You can use the collected data at later stages once you devise new ways to correlate and expose it. You can't go back in time to collect data, but you can make data available that you previously collected.

Keep in mind that you should go about representing that data, showing it in dashboards, and setting up alerts in a progression to avoid having too much information at once that will be hard to reason through. There is no point in throwing every single collected metric for a service into one dashboard. You can create several dashboards per service with detailed views, but keep one top-level dashboard with the most important information. This dashboard should allow you, in a glance, to determine if a service is operating properly. It should give a high-level view of the service, and any more in-depth information should appear in more specialized dashboards.

When representing metrics, you should focus on the most important ones, like response times, errors, and traffic. These will be the foundation of your observability capabilities. You also should focus on the right percentiles for each use case: 99th, 95th, 75th, and so on. For a given service, it may be good enough if only 95% of your requests take less than *x* seconds, whereas on another service you may require 99% of the requests to be below that time. There is no fixed rule for which percentile to focus on—that generally depends on the business requirements.

Whenever possible, you should use tags to provide context to your metrics. Examples of tags to associate with metrics are:

- Environment: Production, Staging, QA
- User ID

By tagging metrics, you can group them later on and perhaps come up with some more insights. Take, for example, a response time you've tagged with the User ID; you can group the values by user and determine if all of the user base or only a particular group of users experiences an increase in response times.

Make sure you always abide by some defined standards when you're naming metrics. It's important that you maintain a naming scheme across services. One possible way of

naming metrics is to use the service name, the method, and the type of metric you wish to collect. Here are some examples:

- `orders_service.sell_shares.count`
- `orders_service.sell_shares.success`
- `fees_service.charge_fee.failure`
- `account_transactions_service. request_reservation.max`
- `gateway.sell_shares.avg`
- `market_service.place_order.95percentile`

11.2 Monitoring SimpleBank with Prometheus and Grafana

You need to send the metrics you collect from your services and infrastructure to a system capable of aggregating and displaying them. The system will use those collected metrics to provide alerting capabilities. For that purpose, you'll be using Prometheus to collect metrics and Grafana to display them:

- Prometheus (https://github.com/prometheus) is an open source systems monitoring and alerting toolkit originally built at SoundCloud. It's now a standalone open source project and is maintained independent of any company.
- Grafana (https://grafana.com) is a tool that allows building dashboards on top of multiple metrics data sources, such as Graphite, InfluxDB, and Prometheus.

You'll do all your setup using Docker. In chapter 7, you already added to your services the ability to emit metrics via StatsD. You'll keep those services unchanged and add something to your setup to convert metrics from StatsD format to the format that Prometheus uses. You'll also add a RabbitMQ container that's already set up to send metrics to Prometheus. Figure 11.3 shows the components you'll be adding to set up your monitoring system.

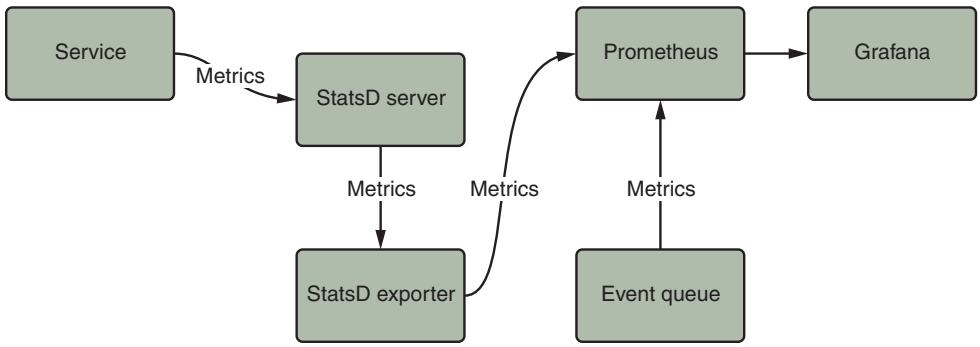


Figure 11.3 The containers you need to build your monitoring system: StatsD server, StatsD exporter, Prometheus, and Grafana

You'll be using both Prometheus and StatsD metrics as a way to show how two types of metrics collection protocols can coexist. StatsD is a push-based tool, whereas Prometheus is a pull-based tool. Systems using StatsD will be pushing data to a collector service, whereas Prometheus will pull that data from the emitting systems.

11.2.1 Setting up your metric collection infrastructure

You'll start by adding the services described in figure 11.2 to the Docker compose file, then you'll focus on configuring both the StatsD exporter and Prometheus. The last step will be to create the dashboards in Grafana and start monitoring the services and the event queue. All the code is available in the book's code repository.

ADDING COMPONENTS TO THE DOCKER COMPOSE FILE

The Docker compose file (see the next listing) will allow you to boot all the services and infrastructure needed for the place order feature. For the sake of brevity, we'll omit the individual services and will only list the infrastructure- and monitoring-related containers.

Listing 11.1 docker-compose.yml file

```
(...)

rabbitmq:
    container_name: simplebank-rabbitmq
    image: deadtrickster/rabbitmq_prometheus
    ports:
      - "5673:5672"
      - "15673:15672"

  redis:
    container_name: simplebank-redis
    image: redis
    ports:
      - "6380:6379"

  statsd_exporter:
    image: prom/statsd-exporter
    command: "-statsd.mapping-config=/tmp/
➥statsd_mapping.conf"
    ports:
      - "9102:9102"
      - "9125:9125/udp"
    volumes:
      - "./metrics/statsd_mapping.conf:/tmp/statsd_mapping.conf"

  prometheus:
    image: prom/prometheus
    command: "--config.file=/tmp/prometheus.yml
➥--web.listen-address '0.0.0.0:9090'"
    ports:
```

You'll use RabbitMQ as the event queue. The image used here is already emitting metrics in the Prometheus format, so you can connect it directly.

Fetches the metrics sent to the StatsD server and converts them to Prometheus format, so Prometheus can fetch them afterwards

Starts statsd_exporter with a custom command that'll load the mapping configuration

Sets up the official Prometheus image

Allows you to start Prometheus, binding it to 0.0.0.0:9000 and reading a custom configuration file that you'll soon see in a bit more detail

```
      - "9090:9090"
    volumes:
      - "./metrics/prometheus.yml:/tmp/prometheus.yml"

  statsd:
    image: dockerana/statsd
    ports:
      - "8125:8125/udp"
      - "8126:8126"
    volumes:
      - "./metrics/statsd_config.js:/src/statsd/
➥config.js"

  grafana:
    image: grafana/grafana
    ports:
      - "3900:3000"
```

Sets the StatsD server that'll collect metrics that the services send

Uses a custom configuration to allow repeating the received metrics to the statsd-exporter container—As with the Prometheus and statsd-exporter containers, the configuration files are located in the metrics folder. This folder will be mounted as a volume so the containers can pick up these configurations at runtime.

Starts Grafana, which will provide a UI for the collected metrics

CONFIGURING STATSD EXPORTER

As we mentioned before, the services involved in the place order feature emit metrics in the StatsD format. In table 11.1, we list all the services and the metrics each one emits. The services will all be emitting timer metrics.

Table 11.1 Timer metrics emitted by the services involved in placing an order

Service	Metrics
Account transactions	request_reservation
Fees	charge_fee
Gateway	health, sell_shares,
Market	request_reservation, place_order_stock_exchange
Orders	sell_shares, request_reservation, place_order

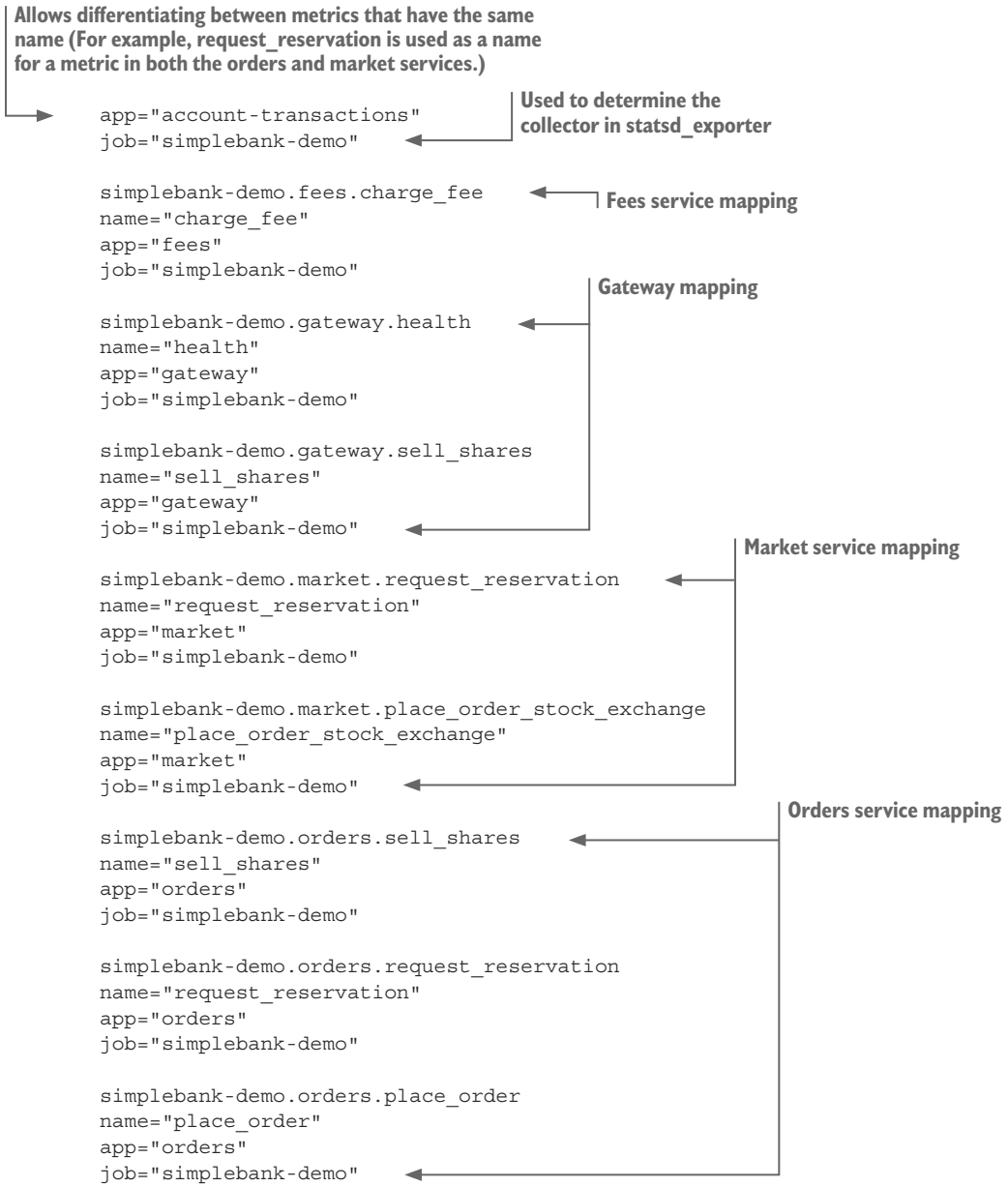
The mapping config file allows you to configure each metric that StatsD collects and add labels to it. The following listing provides the mapping you'll create as a configuration file for the statsd-exporter container.

Listing 11.2 Configuration file to map StatsD metrics to Prometheus

Account transactions service mapping

```
simplebank-demo.account-transactions.request_reservation
name="request_reservation"
```

Sets the name of the metric in Prometheus



If you don't map the above metrics to Prometheus, they'll still get collected, but the way they'll be collected is less convenient. In the figure 11.4 example, you can see the difference between mapped and unmapped metrics fetched from Prometheus from the statsd_exporter service.

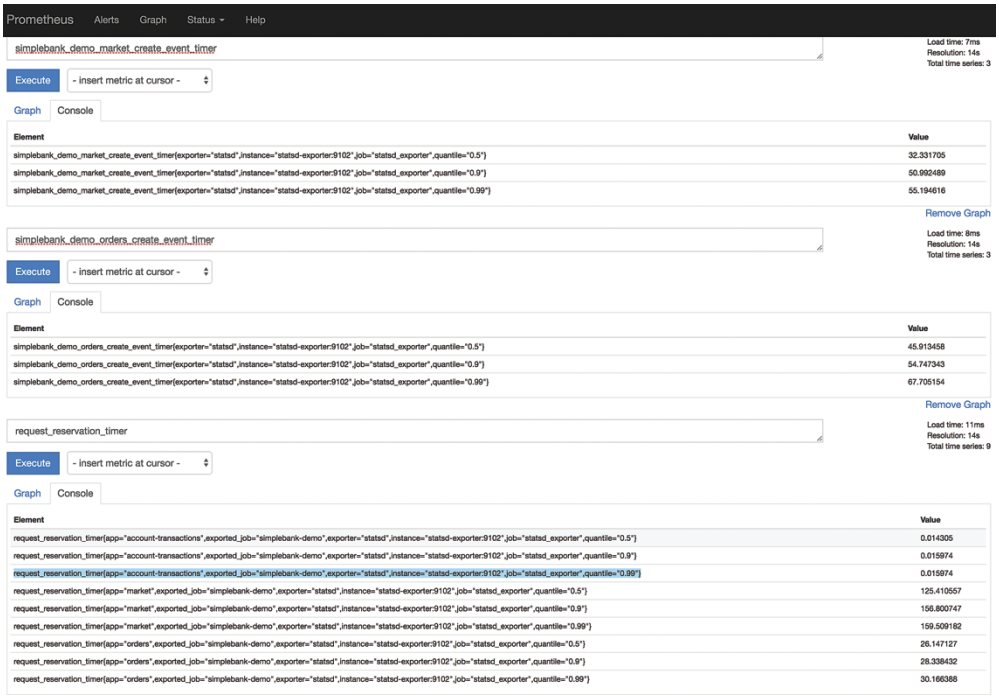


Figure 11.4 Prometheus screenshot with collected SimpleBank metrics—The top two metrics aren't mapped in the statsd_mapping.conf file, whereas the last one is.

As you can observe in figure 11.4, when the unmapped `create_event` metrics that both the market and orders service emit reach Prometheus, they're collected as:

- `simplebank_demo_market_create_event_timer`
- `simplebank_demo_orders_create_event_timer`

For the `request_reservation_timer` metric that the market, orders, and account transactions services emit, there's only one entry, the metric is the same, and the differentiation is in the metadata:

Metric mapped in the statsd_exporter configuration file—The app label takes the values off all apps generating the request_reservation_timer metric.

```
request_reservation_timer{app="*",exported_job="simplebank-demo",e
     xporter="statsd",instance="statsd-exporter:9102",job="statsd_
     exporter",quantile="0.5"}

request_reservation_timer{app="*",exported_job="simplebank-demo",e
     xporter="statsd",instance="statsd-exporter:9102",job="statsd_
     exporter",quantile="0.9"}
```

```
request_reservation_timer{app="*",exported_job="simplebank-demo",e
     xporter="statsd",instance="statsd-exporter:9102",job="statsd_
     exporter",quantile="0.99"}

simplebank_demo_market_create_event_timer{exporter="statsd",instance="statsd-
     exporter:9102",job="statsd_exporter",quantile="0.5"}
```

Metric not mapped in the statsd_exporter configuration file — There's no app and no exported_job labels.

CONFIGURING PROMETHEUS

Now that you've configured the StatsD exporter, it's time to configure Prometheus for it to fetch data from both the StatsD exporter and RabbitMQ, as shown in the following listing. Both of these sources will be available as targets for metrics data fetching.

Listing 11.3 Prometheus configuration file

```
global:
  scrape_interval:     5s
  evaluation_interval: 10s
  external_labels:
      monitor: 'simplebank-demo'

alerting:
  alertmanagers:
  - static_configs:
    - targets:

scrape_configs:
  - job_name: 'statsd_exporter'
    static_configs:
      - targets: ['statsd-exporter:9102']
        labels:
          exporter: 'statsd'
    metrics_path: '/metrics'

  - job_name: 'rabbitmq'
    static_configs:
      - targets: ['rabbitmq:15672']
        labels:
          exporter: 'rabbitmq'
    metrics_path: '/api/metrics'
```

Sets the interval at which Prometheus will scrape the configured targets for metrics

Scrapes config section where each target is configured

Will be added as a label any time series scraped from the config

The target host and metrics path will be concatenated to determine the URL to collect metrics from. Given that in this case the scheme defaults to http, the URL will be 'http://rabbitmq:15672/api/metrics'.

SETTING UP GRAFANA

To receive metrics in Grafana, you need to set up a data source. First, you can boot your applications and infrastructure by using the Docker compose file. This will allow you to access Grafana on port 3900, as follows.

Listing 11.4 Grafana setup in the docker-compose.yml file

```
(...)
grafana:
  image: grafana/grafana
  ports:
    - "3900:3900"
```

Uses the official grafana Docker image with default settings

Grafana uses port 3000 by default. The applications and services you start via the compose file will be able to communicate using the default port. You're mapping it to port 3900 to access it from the host machine.

To start all applications and services using Docker compose, you need to get inside the folder containing the compose file and issue the `up` command:

Stops all SimpleBank running containers

```
chapter-11$ docker stop $(docker ps | grep simplebank |
➥awk '{print $1}')
chapter-11$ docker rm $(docker ps -a | grep simplebank |
➥awk '{print $1}')
chapter-11$ docker-compose up --build --remove-orphans

Starting simplebank-redis ...
Starting chapter11_statsd-exporter_1 ...
Starting chapter11_statsd_1 ...
Starting simplebank-rabbitmq ...
Starting chapter11_prometheus_1 ...
Starting simplebank-rabbitmq ... done
Starting simplebank-gateway ...
Starting simplebank-fees ...
Starting simplebank-orders ...
Starting simplebank-market
Starting simplebank-account-transactions ... done
Attaching to chapter11_prometheus_1, simplebank-redis, chapter11_statsd_1,
     simplebank-rabbitmq, chapter11_statsd-exporter_1, simplebank-gateway,
     simplebank-fees, simplebank-orders, simplebank-market, simplebank-
     account-transactions
(…)
```

Removes all SimpleBank containers so you don't have any name clashes

Starts the containers defined in the docker-compose.yml file — The --build option builds images before starting the containers, and the --remove-orphans option removes any containers that aren't defined in the compose file.

The output of the `docker-compose up` command will allow you to understand when all services and applications are ready. You can reach applications using the URL assigned to Docker or the IP address. By appending port 3900 as configured in the docker-compose.yml file, you can access Grafana's login screen as shown in figure 11.5. You'll be accessing Grafana using the default login credentials: username and password are both *admin*.

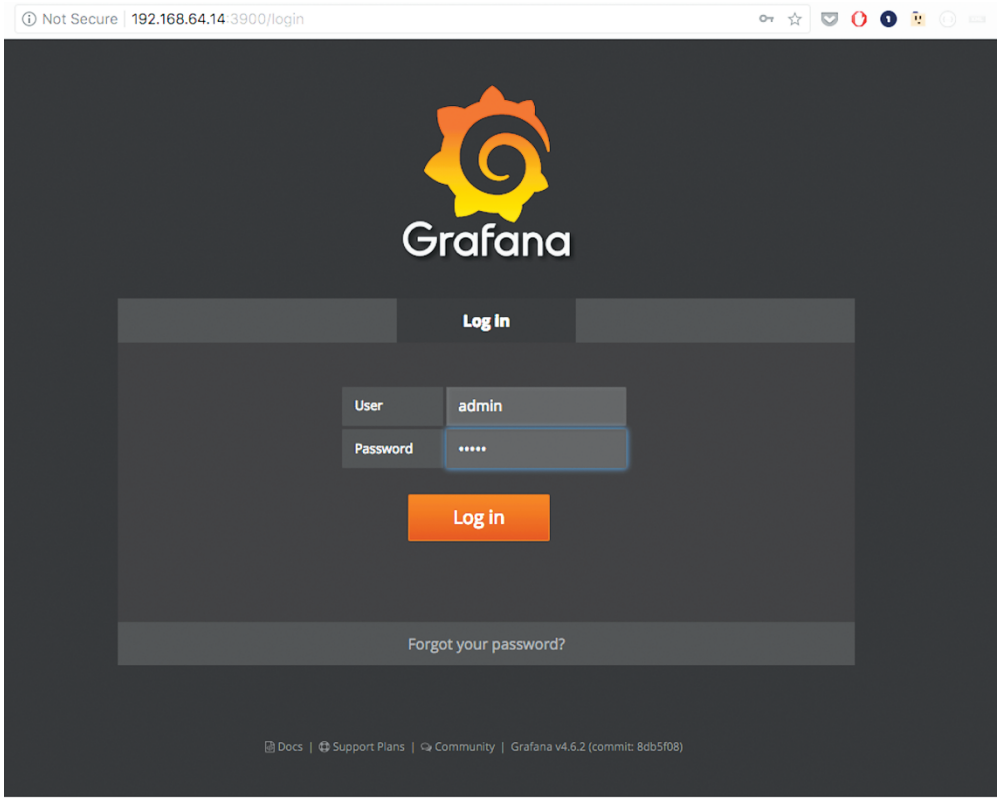


Figure 11.5 Grafana login screen

Once you log in, you'll have an Add Data Source option. Figure 11.6 shows the data source configuration screen, Edit Data Source. To configure a Prometheus data source in Grafana, you need to select Prometheus as the type and insert the URL of the running Prometheus instance, in your case http://prometheus:9090, as configured in the Docker compose file.

The Save & Test button will give you instant feedback on the data source status. Once it's working, you're ready to use Grafana to build dashboards for your collected metrics. In the next few sections, you'll be using it to display metrics both for the services that enable the place orders functionality in SimpleBank and for monitoring a critical piece of the infrastructure, RabbitMQ, the event queue.

11.2.2 Collecting infrastructure metrics—RabbitMQ

To set up the dashboard to monitor RabbitMQ, you'll be using a json configuration file. This is a convenient and easy way to share dashboards. In the source code repository, you'll find a grafana folder. Inside that, a RabbitMQ Metrics.json file holds the

configuration for both the dashboard layout and the metrics you want to collect. You can now import that file to have your RabbitMQ monitoring dashboard up in no time!

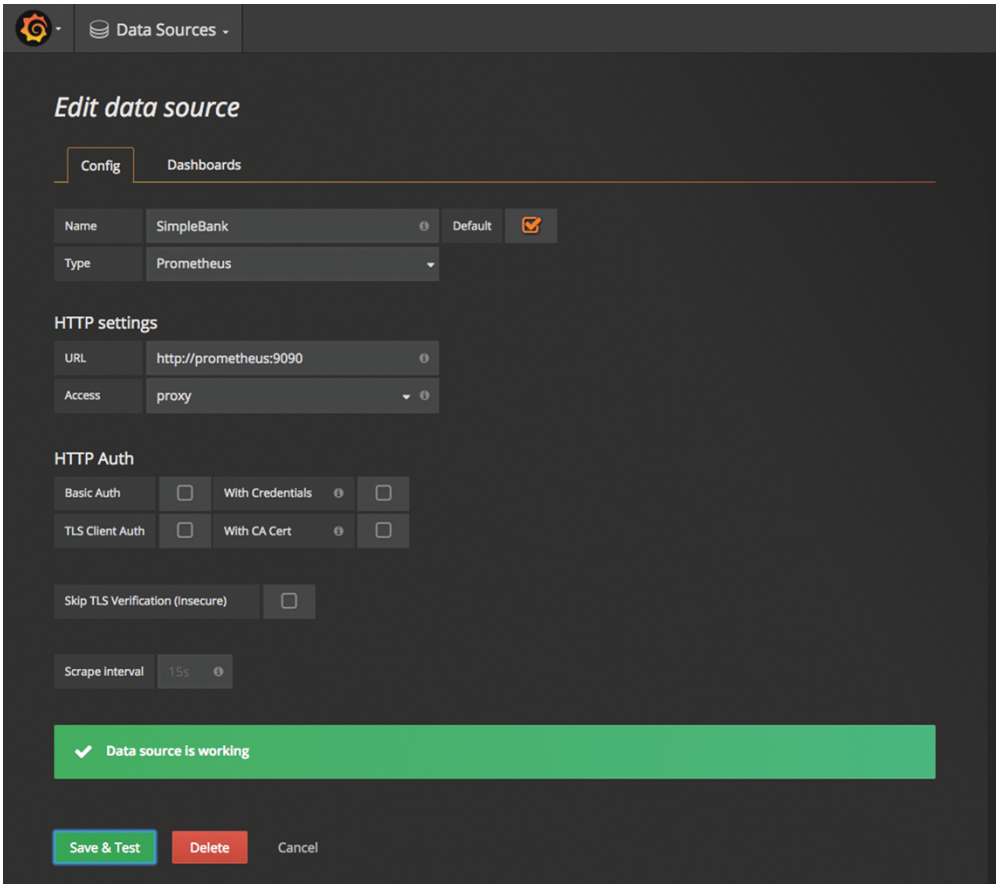


Figure 11.6 Configuring a Prometheus data source in Grafana

Figure 11.7 shows how you can access the import dashboard functionality in Grafana. By clicking Grafana's logo, you bring up a menu; if you hover over Dashboards, the Import option will be available.

The import option will bring up a dialog box that enables you to either paste the json in a text box or upload a file. Before you can use the imported dashboard, you need to configure the data source that'll feed the dashboard. In this case, you'll be using the SimpleBank data source you configured previously.

That's all it takes to have your RabbitMQ dashboard up and running. In figure 11.8 you can see how it looks.

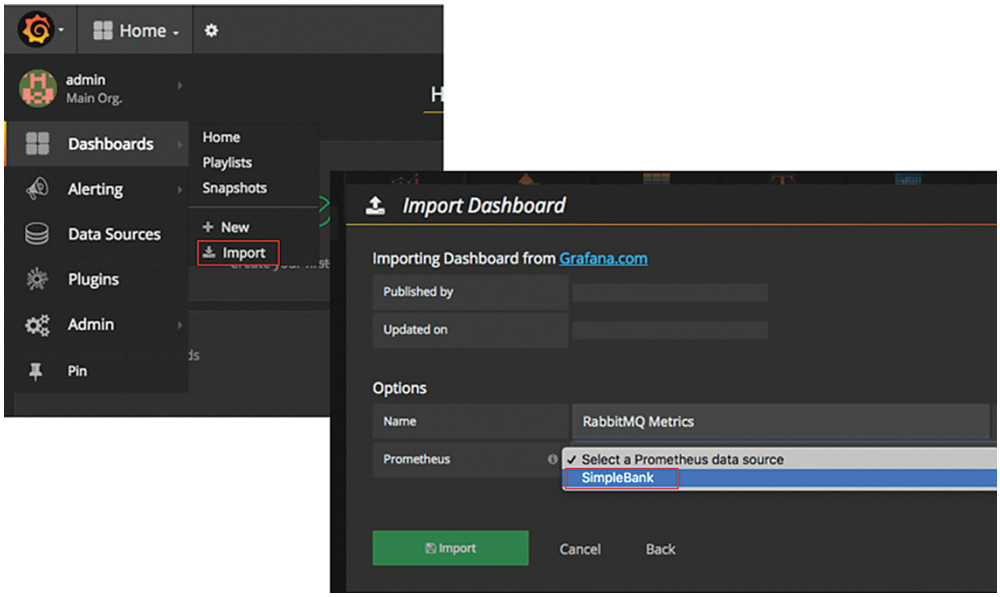


Figure 11.7 Importing a dashboard from a json file

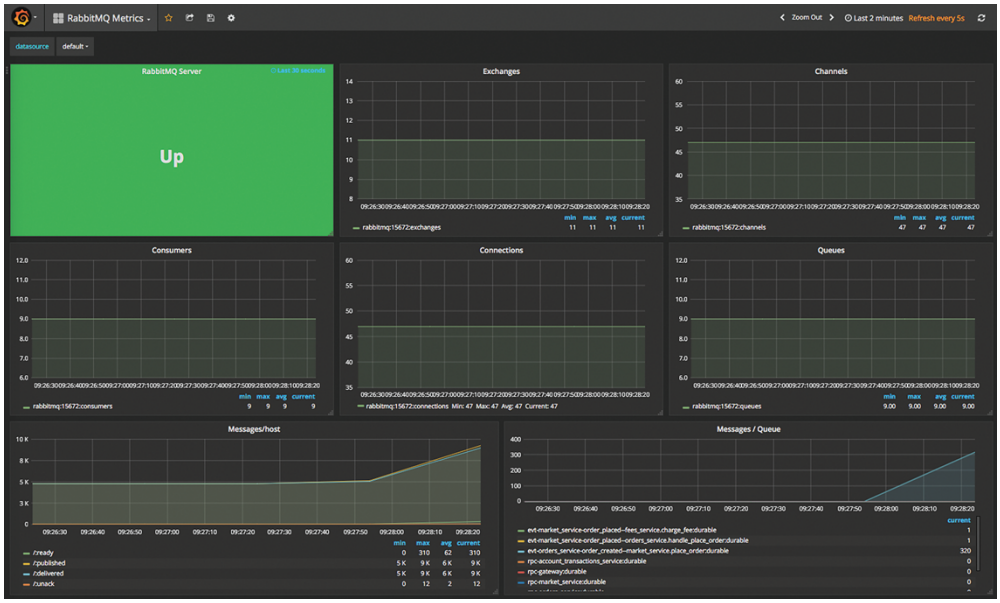


Figure 11.8 RabbitMQ metrics collected via Prometheus and displayed in Grafana

Your RabbitMQ dashboard provides you an overview of the system by displaying a monitor for the server status that shows if it's up or down, along with graphs for Exchanges, Channels, Consumers, Connections, Queues, Messages per Host, and Messages per Queue. You can hover over any graph to display details for metrics at a point in time. Clicking the graph's title will bring up a context menu that allows you to edit, view, duplicate, or delete it.

11.2.3 Instrumenting SimpleBank's place order

Now that you have services up and running, along with the monitoring infrastructure, Prometheus and Grafana, it's time to collect the metrics described in table 11.1. You can start by loading a dashboard exported as json that you can find in the source directory under the grafana folder (Place Order.json). Follow the same instructions as the ones in 11.2.2 for the RabbitMQ dashboard.

Figure 11.9 displays the dashboard collecting metrics for the services involved in the place order feature. By clicking on each of the panel titles, you can view, edit, duplicate, share, and delete each of the panels.

This loaded dashboard collects the time metrics and displays the 0.5, 0.9, and 0.99 quantiles for each metric. In the top right corner, you find the manual refresh button as well as the period for displaying metrics. By clicking the Last 5 Minutes label, you can select another period for displaying metrics, as shown in figure 11.10. You can select one of the Quick Ranges values or create a custom one, and you can display stored metrics in any range you need.

Figure 11.9 Place order dashboard accessible at Grafana's /dashboard/db/place-order endpoint

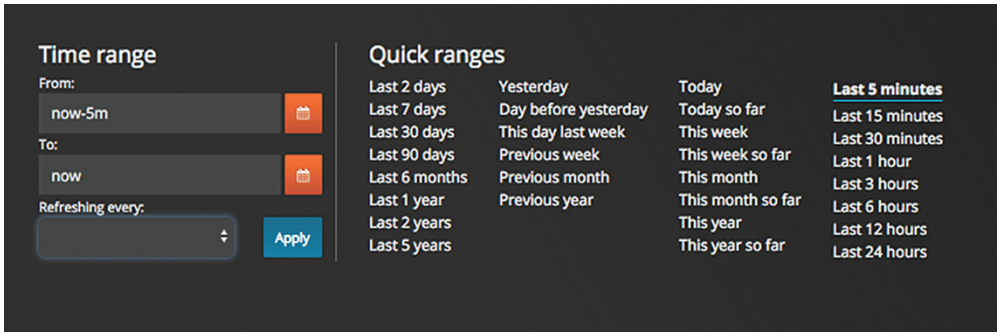


Figure 11.10 Selecting the time range for which you want metrics to be displayed

Let's focus on the Market | Place Order Stock Exchange panel to see in detail how you can configure a specific metric display. To do so, click the panel title and then select the Edit option. Figure 11.11 shows the edit screen for the Market | Place Order Stock Exchange.

The edit screen has a set of tabs (1) you can select to configure different options. The highlighted one is the Metrics tab, where you can add and edit metrics to be displayed. In this particular case, you're only collecting a metric (2), namely the place_order_stock_exchange_timer that gives you the time it took for the market service to place an order into the stock exchange. The default display for a metric contains metadata like the app name, the exported job, and the quantile. To change the way the legend is presented, you set a Legend Format (3). In this case, you set the name and use `{{quantile}}` block that'll be interpolated to display the quantile in both the graph legend and the hovering window next to the vertical red line. (The red line acts as a cursor when you move your mouse across the collected metrics.) In your dashboards, you're displaying the min, max, avg, and current values for each quantile (4).

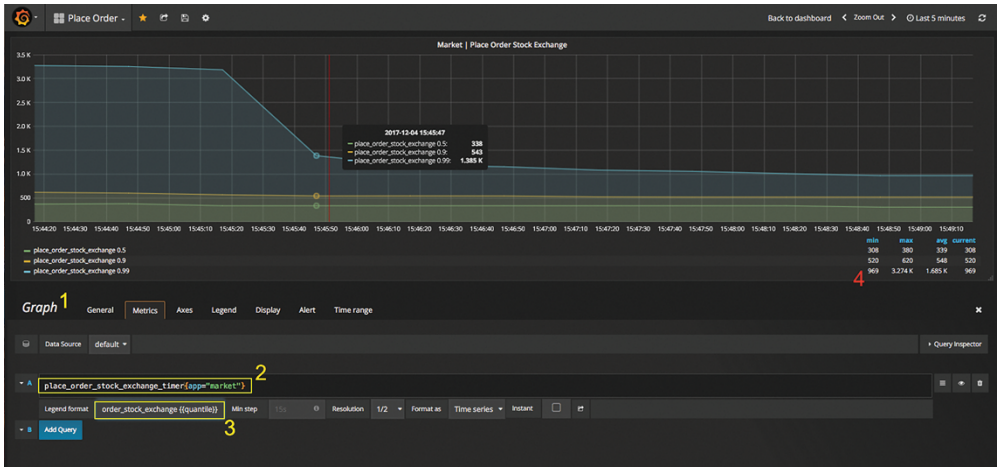


Figure 11.11 Panel edit screen for Market | Place Order Stock Exchange

FURTHER READING To learn more about how to use Prometheus with Grafana and about the Prometheus data model, visit the following documentation pages: http://mng.bz/ui3b and http://mng.bz/PZQ0.

The dashboard you've set up is quite simple, but it allows you to have an overview of how the system is behaving. You're able to collect time-related metrics for several actions that services in your system perform.

11.2.4 Setting up alerts

Now that you're collecting metrics and storing them, you can set up alerts for when values deviate from what you consider as normal for a given metric. This can be an increase in the time taken to process a given request, an increase in the percent of errors, an abnormal variation in a counter, and so on.

In your case, you can consider the market service and set up an alert for knowing when the service needs to be scaled. Once you place a sell order via the gateway service, a lot goes on. Multiple events are fired, and you know the bottleneck tends to be the market service processing the place order event. The good thing is you can set up an alert to send a message whenever messages in the market place order queue go above a certain threshold. You can configure multiple channels for notifications: email, slack, pagerduty, pingdom, webhooks, and so on.

You'll be setting up a webhook notification to receive a message in your alert server every time the number of messages goes above 100 in any message queue. For now, you'll only be receiving it in an alert service made with the purpose of illustrating the feature. But you could easily change this service to trigger an increase in the number of instances of a given service to increase the capacity to process messages from a queue.

The alert service is a simple app that also booted when you started all other apps and services. It'll be listening for incoming `POST` messages, so you can go ahead and configure the alerts in Grafana. Figure 11.12 shows the activity for the market place order event queue with indication of alerts, both when they were triggered and when the alert condition ceased. When you set up alerts, Grafana will indicate as an overlay both the threshold set for alerting (1) and the instants when alerts were triggered (2, 4, 6) and resolved (3, 5, 7).

With the current setup, the alert service sends an alert message as a webhook when the number of messages in any queue goes over 100. The following shows you one of those alert messages:

```
alerts.alert.d26ab4ca-1642-445f-a04c-41adf84145fd:
{
  "evalMatches": [
    {
      "value":158.33333333333334,
      "metric":"evt-orders_service-order_created
➥--market_service.place_order",
      "tags":{
        "__name__":"rabbitmq_queue_messages",
        "exporter":"rabbitmq",
```

Value for the metric at the time of the alert

The name of the metric for which the alert was triggered

```
        "instance":"rabbitmq:15672",
        "job":"rabbitmq",
        "queue":"evt-orders_service-order_created
➥--market_service.place_order",
        "vhost":"/"
      }
    }
  ],
  "message":"Messages accumulating in the queue",
  "ruleId":1,
  "ruleName":"High number of messages in a queue",
  "ruleUrl":"http://localhost:3000/dashboard/db/rabbitmq-metrics?fullscreen\
➥u0026edit\u0026tab=alert\u0026panelId=2\u0026orgId=1",
  "state":"alerting",
  "title":"[Alerting] High number of messages in a queue"
}
```

Shows information about the queue where the alert was triggered

Identifies what rule this alert is related to

Indicates the message type—In this case, "alerting" means an alert was triggered and the number of messages in the queue is above the normal operating threshold.

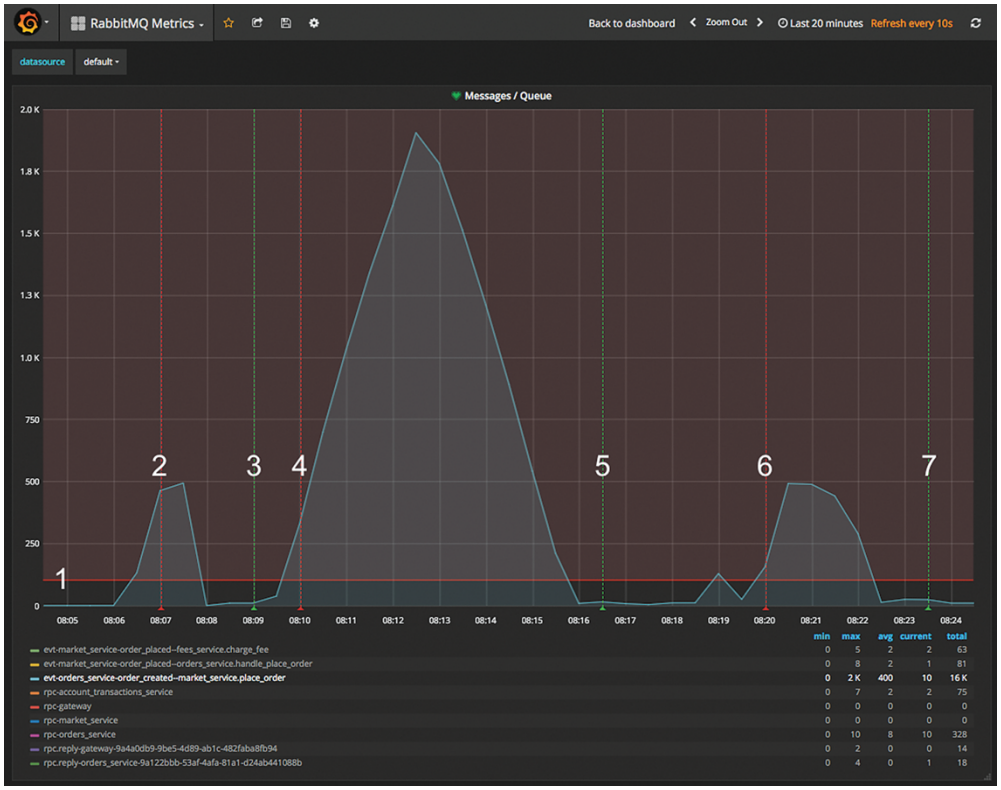


Figure 11.12 The message queue's status showing alert overlays

Likewise, when the number of messages in a queue goes below the value defined as the threshold for alerting, the service also issues a message to notify about it:

```
alerts.alert.209f0d07-b36a-43f4-b97c-2663daa40410:
{
  "evalMatches":[],
  "message":"Messages accumulating in the queue",
  "ruleId":1,
  "ruleName":"High number of messages in a queue",
  "ruleUrl":"http://localhost:3000/dashboard/db/rabbitmq-metrics?fullscreen\
➥u0026edit\u0026tab=alert\u0026panelId=2\u0026orgId=1",
  "state":"ok",
  "title":"[OK] High number of messages in a queue"
}
```

Identifies the rule the alert is related to

The "ok" state means the number of messages in the queues are all back to below the set threshold; the conditions for a previous alert are no longer met.

Let's now see how you can set up this alert for the number of messages in queues. You'll also be using Grafana for setting up the alert, because it offers this capability and the alerts will display on the panels they relate to. You'll be able to both receive notifications and check the panels for previous alerts.

You'll start by adding a notification channel that'll you'll use to propagate alert events. Figure 11.13 shows how to create a new notification channel

To set up a new notification channel in Grafana, follow these steps:

1. Click the Grafana icon on the top left of the screen.
2. Under the Alerting menu, select Notification Channels.

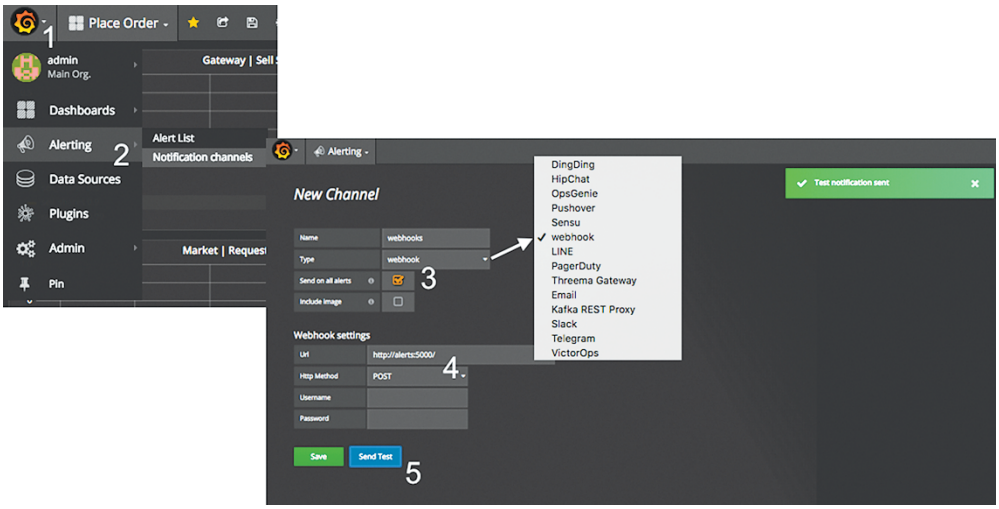


Figure 11.13 Setting up a new notification channel in Grafana

3. Enter the name for the channel and select the type as Webhook, then check the Send on All Alerts option.
4. Enter the URL for the service receiving the alerts. In your case, you'll be using the alerts service and listening for `POST` requests.
5. Click the Send Test button to verify all is working, and if so, click Save to save the changes.

Now that you have an alert channel set up, you can go ahead and create alerts on your panels. You'll be setting an alert on the messages queue panel under the RabbitMQ dashboard you created previously. Clicking the Messages/Queue panel title will bring up a menu where you can select Edit. This allows you to create a new alert under the Alert tab. Figure 11.14 shows how to set up a new alert.

Under the Alert Config screen, start by adding the Name for the alert as well as the frequency at which you want the condition to be evaluated—in this case every 30 seconds. The next step is to set the Conditions for the alert. You'll be setting an alert to notify you whenever the average of the values collected from query A is above 100 in the last minute.

> **TIP** If you click the Metrics tab, you can see query A under it, which will be the rabbitmq_queue_messages metric. You also have the option to test the rules you set by clicking the Test Rule button.

Under the Alert tab, you also can check the history of the configured alerts. Figure 11.15 shows the alert history for the number of messages in queues.

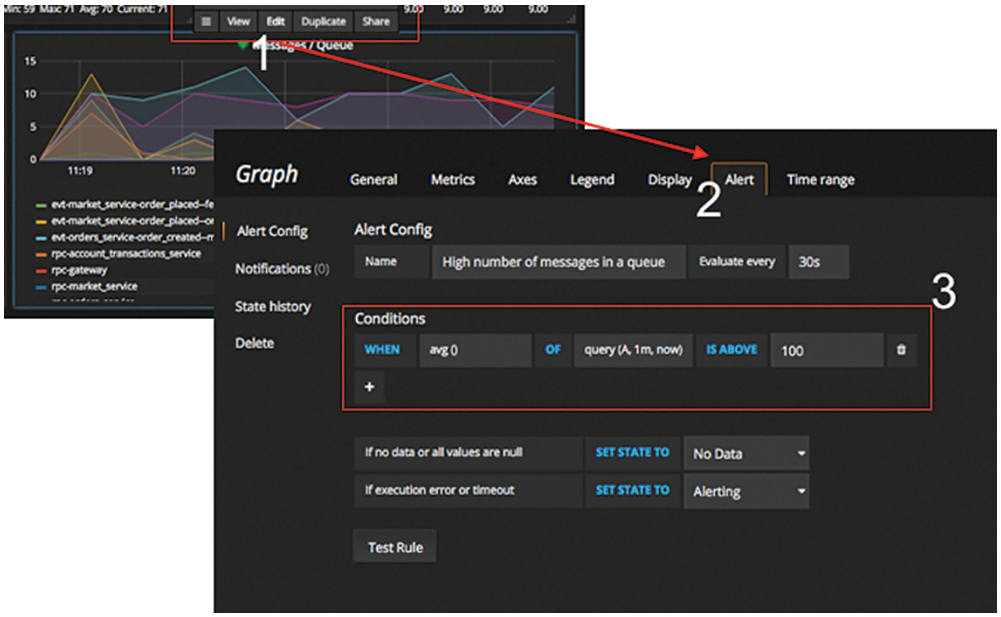


Figure 11.14 Setting up alerts on the Messages/Queue graph on the RabbitMQ Dashboard

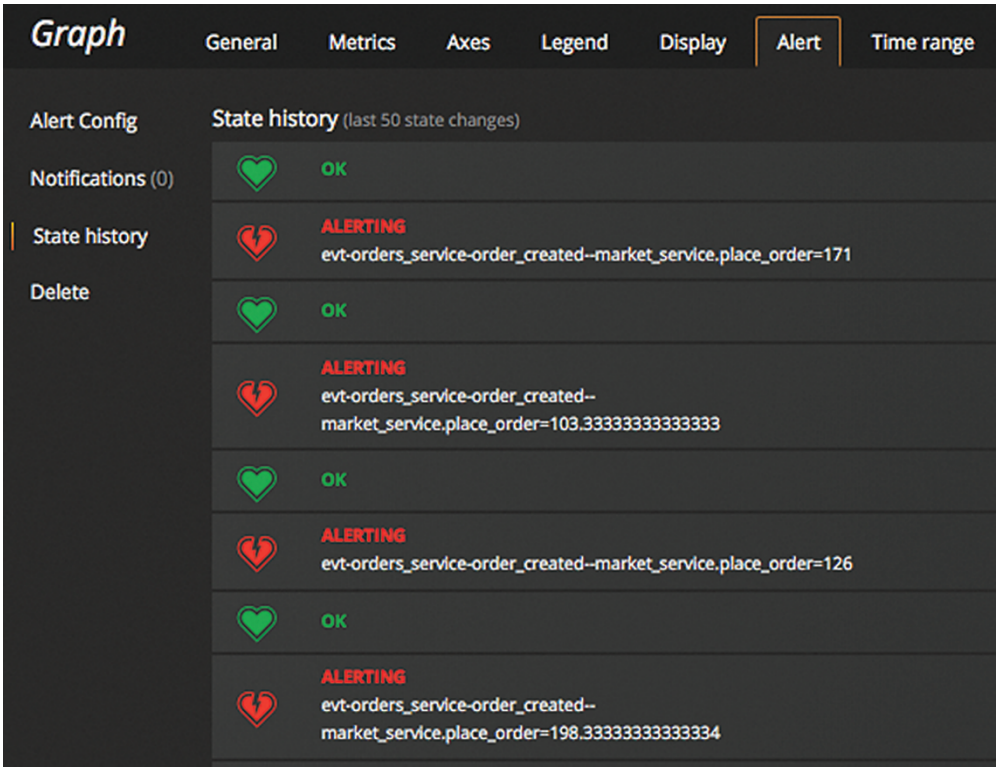


Figure 11.15 Displaying the state history for a given alert

That's it, you're done! You've set up a monitoring infrastructure to collect both metrics that your services already emitted and those that come from a key component that those services use to communicate asynchronously: the event queue. You've also seen how to create alerts to be notified whenever certain conditions in your system are met. Let's now dig a bit deeper into alerts and how to use them.

11.3 Raising sensible and actionable alerts

Having a monitoring infrastructure in place means you can measure system performance and keep a historic record of those measures. It also means you can determine thresholds for your measures and automatically emit notifications when those thresholds are exceeded.

One thing you need to keep in mind, though, is that it's easy to reach a stage where all this information can become overwhelming. Eventually, the overload of information can do more harm than good (for example, if it gets so bad that people start ignoring recurring alerts). You need to make sure that the alerts you raise are actionable—and actioned—and that they're targeting the correct people in the organization.

Although services may consume and take action on some of the alerts automatically; for example, autoscaling a service if messages are accumulating in a queue, humans need to consume and take action on some alerts. You want those alerts to reach the correct people and contain enough information so that diagnosing the cause becomes as easy as possible.

You also need to prioritize alerts, because most likely any issue with your services or infrastructure will trigger multiple alerts. Whoever is dealing with those alerts needs to know immediately the urgency of each one. As a rule, you should direct alerts for services to the teams owning those services. You should map the application into the organization, because this helps with determining the targets for alerts.

11.3.1 Who needs to know when something is wrong?

In day-to-day operation, alerts should target the team who owns the service and originated it. This reflects the "you build it, you run it" mantra that should govern a microservices-oriented engineering team. As teams create and deploy services, it's hard, if not impossible, for everyone to know about every service deployed. People with the most knowledge about a service will be in the best position to interpret and take action in response to alerts that the service generates.

Organizations also may have some on-call rotation or a dedicated team that'll receive and monitor alerts and then escalate if necessary to specialized teams. When setting up alerts and notifications, it's important to keep in mind that other people may consume them, so you should keep those alerts as concise and informative as possible. It's also important that each service have some sort of documentation on common issues and diagnosing recipes so that on-call teams can, when they receive an alert, determine if they can fix the issue or if they need to escalate it.

You also should categorize alerts by levels of urgency. Not every issue will need immediate attention, but some are deal breakers that you need to address as soon as you know about them.

Severe issues should trigger a notification to ensure someone, either an engineer from the team that built the service or an on-call engineer, is notified. Issues that are of moderate severity should generate alerts as notifications in any channels deemed appropriate, so those monitoring them can pick them up. You can think of this type of alert as something generating a queue of tasks that you need to carry out as soon as possible but not immediately—they don't need to interrupt someone's flow or wake someone up in the middle of the night. The lowest priority alerts are those that only generate a record. These alerts aren't strictly for human consumption, because services can receive them and take some kind of action if needed (for example, autoscaling a service when response times increase).

11.3.2 Symptoms, not causes

Symptoms, not causes, should trigger alerts. An example of this is a user-facing error; if users can no longer access a service, that inability should generate an alert. You shouldn't be tempted to trigger alerts for every single parameter that isn't under the *normal*

threshold. With such partial information, you won't be able to know what's going on or what the problem is. In figure 11.2, we illustrated the flow for placing orders in the stock market. Four services cooperate with a gateway that works as the access point for the consumer of the feature. One or more of the services may be exhibiting erroneous behavior or be overloaded. Given the mainly asynchronous nature of the communication between components, it may be hard to pinpoint why a given error may be happening.

Imagine you set an alert that relates the number of requests reaching the gateway and the number of issued notifications of orders placed. It'll be simple to correlate those two metrics over time and determine the ratio between the two. You'll have a symptom: the number of orders placed is greater than the ones completed. You can start from there and then try to understand which component is failing (maybe even multiple components). Is it the event queue or an infrastructure problem? Is the system under high loads and can't cope? The symptom will be the starting point for your investigation, and from there you should follow the leads until you find the cause or causes.

> **TIP** Avoid alert fatigue by keeping alert notifications to a minimum and keeping them actionable. Generating an alert notification for every single deviation from the normal behavior of the system may quickly lead to alerts being disregarded or deemed unimportant. Such minimizing will eventually lead to something important being overlooked.

11.4 Observing the whole application

Correlating metrics can be a precious tool to infer and understand more than a per-service state of the system. Monitoring can also help you understand and reason through the behavior of the system under different conditions, and this can help you to predict and adjust your capacity by using all the collected data. The good thing about collecting per-service metrics is you can iteratively correlate them between different services and have an overall idea of the behavior of the whole application. In figure 11.16, you can see a possible correlation of different service metrics.

Let's look into each of the suggested correlations:

- *A: Creating a new visualization comparing the rate of incoming requests to the gateway and the orders service* — This allows you to understand if there are any issues in processing the incoming requests from your users. You also can use the new correlation to set an alert every time that rate drops from 99%.
- *B: Correlating the number of user requests made to the gateway with the number of order-created messages in the queue* — Given that you know the order service is responsible for publishing those messages, this will, similarly to A, allow you to understand if the system is working correctly and customer requests are being processed.
- *C: Correlating the number of order-placed messages with the number of requests to the order service* — This will allow you to infer if the fee service is working properly.

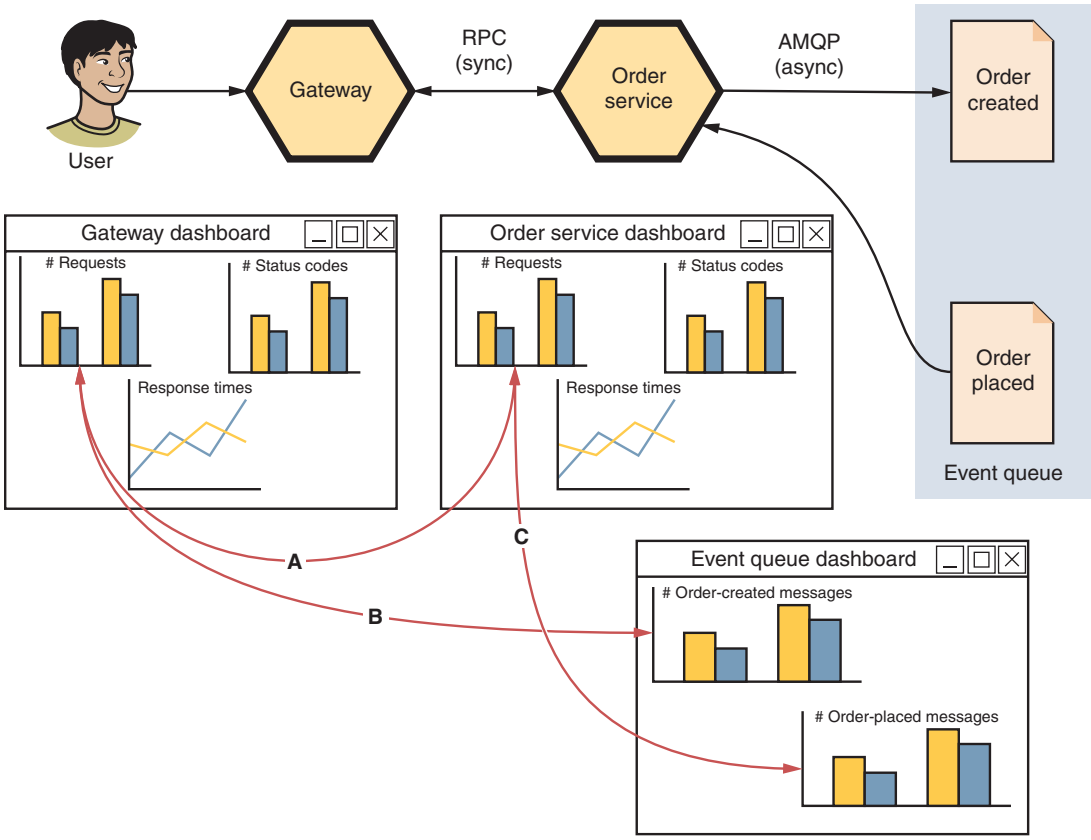


Figure 11.16 Correlation of metrics between different services

Combining different metrics into new dashboards and setting sensible alerts on them allows you to gain insights into the overall application. It's then up to you to determine the desired level of detail, from a high-level view to a detailed one.

So far, we've covered monitoring and alerting. You've set up a monitoring stack to be able to understand *how* things happened. You're now able to understand the status of services, observe the metrics they emit, and determine if they're operating within expected parameters. This is only part of the application observability effort. It's a good starting point, but you do need more!

To be able to fully understand what's going on, you need to invest some more in logging and tracing so you can have both a current view of what's happening and a view of what happened before. In the next chapter, we'll focus on logging and tracing as a complement to monitoring in your journey into observability. Doing so will help you to understand *why* things happened.

Summary

- A robust microservice monitoring stack consists of metrics, traces, and logs.
- Collecting rich data from your microservices will help you identify issues, investigate problems, and understand your overall application behavior.
- When collecting metrics, you should focus on four golden signals: latency, errors, traffic (or throughput), and saturation.
- Prometheus and StatsD are two common, language-independent tools for collecting metrics from microservices.
- You can use Grafana to graph metric data, create human-readable dashboards, and trigger alerts.
- Alerts based on metrics are more durable and maintainable if they indicate the symptoms of incorrect system behavior, rather than the causes.
- Well-defined alerts should have a clear priority, be escalated to the right people, be actionable, and contain concise and useful information.
- Collecting and aggregating data from multiple services will allow you to correlate and compare distinct metrics to gain a rich overall understanding of your system.

12 Using logs and traces to understand behavior

This chapter covers

- Storing logs in a consistent and structured way in a machine-readable format
- Setting up a logging infrastructure
- Using traces and correlation IDs to understand system behavior

In the previous chapter, we focused on emitting metrics from your services and using those metrics to create dashboards and alerts. Metrics and alerts are only one part of what you need to achieve observability in your microservice architecture. In this chapter, we'll focus on collecting logs and making sure you're able to trace the interactions between services. This will allow you to not only have an overview of how the system behaves but also go back in time and retrospectively follow each request. Doing so is important to debug errors and to identify bottlenecks. Logs give you a sort of paper trail that documents the history of each request entering your system, whereas traces provide you a way to establish a timeline for each request, to understand how much time it spent in different services.

By the end of this chapter, you'll have created a basic logging infrastructure and set up the tracing capability. You'll be able to both monitor the operation of your application and have the tools to audit and investigate in case you need to do so for particular requests. In addition, you'll be able to identify performance issues by looking into tracing data.

12.1 Understanding behavior across services

In a microservices-based architecture, multiple services will be involved in providing functionality to users. It gets hard to understand what goes on with every request when you no longer have a central access point to data. Services are distributed across multiple nodes, are ephemeral, and are continuously being deployed and scaled to meet the needs of operation. Let's revisit the sell order use case as you might have implemented it if you'd needed a single application running on a single machine (figure 12.1).

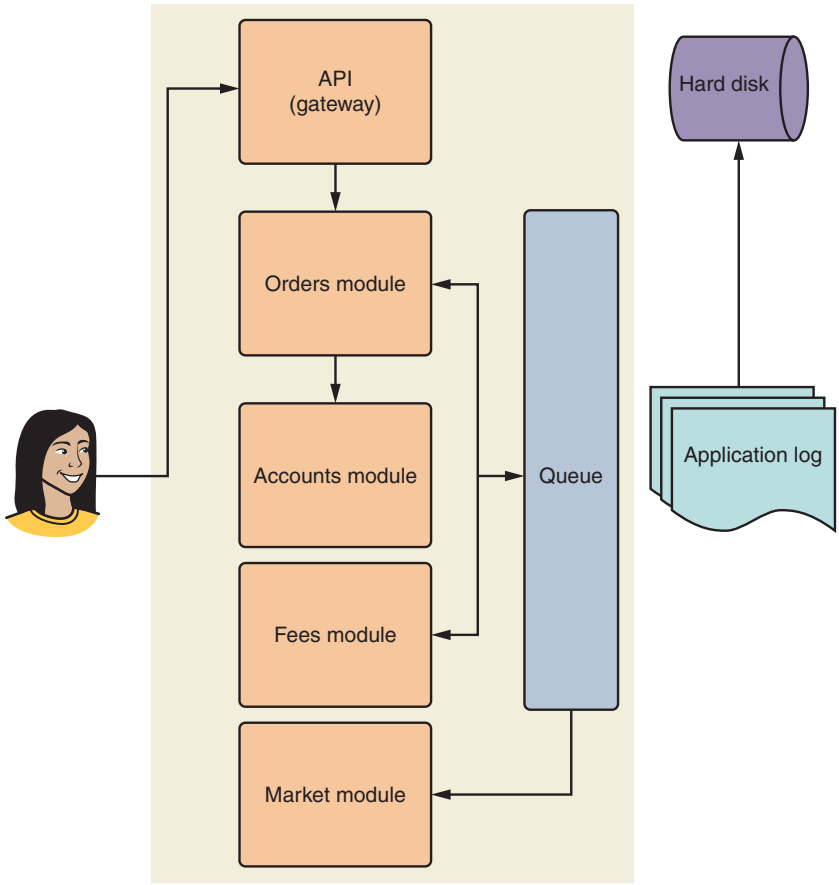


Figure 12.1 The sell order use case implemented in a single application

In figure 12.1, we've represented each of the services that collaborate to allow a client to sell shares as modules in the same application. If you were to inspect a given request lifecycle in the system, you could log in to the machine and inspect log data stored on a hard drive. But you'd most likely have multiple machines running the application, for redundancy and availability, so things wouldn't be as easy as logging in to one machine. Once you identified the request you were interested in observing, you'd have to identify which machine had run the request and then inspect it. Going through the logs from that machine would provide you needed insights.

Maintaining logs in a single machine is by no means easy—a server can also crash and become unavailable. Our aim here isn't to talk about minimizing the complexity of keeping log data (or any data) safely persisted but to point out that having a single point for storing all the data makes it easier and more convenient to consult.

Let's now compare the same scenario in a microservices application. Figure 12.2 illustrates the same use case with multiple services, each with multiple copies of itself, running independently.

As you can see below, you have five services running independently, and each of those services has three instances running. This means potentially none of the pods are executing on the same physical machine. A request coming into the system will most likely flow through multiple pods running on different physical machines, and you have no easy way to track down that request by accessing logs. Could you even do it? What guarantee do you have that any of the pods are still running once you need to access data?

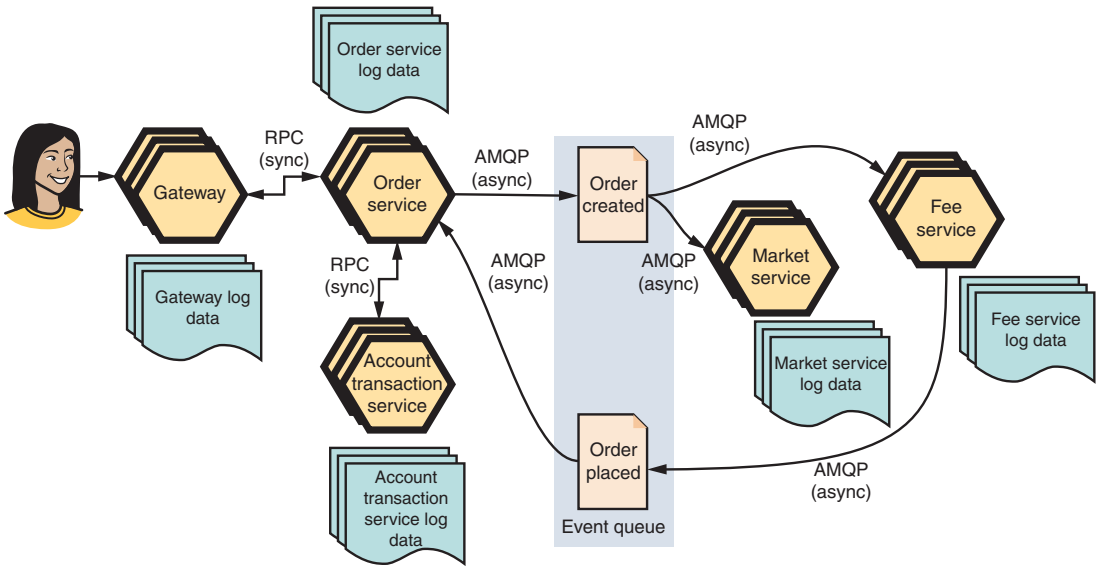


Figure 12.2 The sell order use case in SimpleBank with multiple services running, each with its own log data

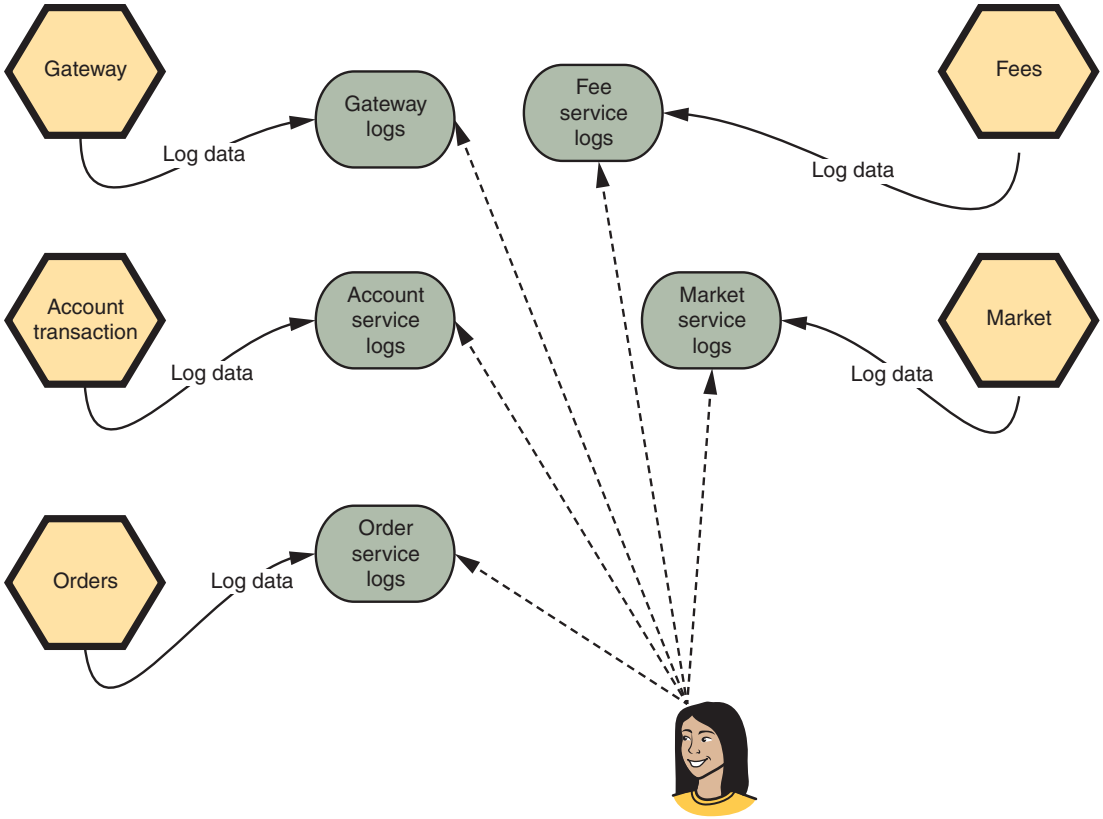


Figure 12.3 Accessing logs for each service in different running instances is challenging.

In figure 12.3, we illustrate the challenges you face when you try to gather data from a distributed system. Even if you had some sort of persistency for the log data that would allow it to survive after a running pod is replaced, it'd be no easy task to track down a request through your system. You need a better way to record what's going on with your system. To be able to fully understand behavior, you need to

- Make sure you persist log data so it survives through service restarts and scaling
- Aggregate all log data from multiple services and instances of those services in a central location
- Make the stored data usable, allowing for easy searches and further processing

Our objective by the end of this chapter will be to have an infrastructure that allows you to collect log data from all your services, aggregating it and allowing you to perform searches in it so you can, at any time, reason through the behavior of the system. You'll be able to use the available data to audit, debug, or even gather new insights by processing it further. One example of the processing you can do to augment the

available information is to collect the IP data stored in the logs and generate a visualization showing the most common geographic areas of your users.

To effectively store and make your log data searchable, you first need to agree on a format the engineering team will use. A consistent format will help to guarantee that you can store and process data effectively.

12.2 Generating consistent, structured, human-readable logs

To be able to achieve observability, you have to collect data from multiple sources; not only from running services but also from your infrastructure. Defining a common format allows you to analyze data more easily and perform searches in that data using existing tools with minimal effort. Examples of the data you may collect and use are:

- Application logs
- Database logs
- Network logs
- Performance data collected from the underlying operating system

For some components, you can't control the format, so you have to cope with their specificities and transform them somehow. But for now, let's focus on what you can control: your services. Making sure the whole engineering team abides by a format and a way of doing things pays off in the long run, because data collection will become simpler and more effective. Let's start by determining what you should store; then we can look at how to store it.

12.2.1 Useful information to include in log entries

For your log data to be useful and effective in helping you to understand behavior in your systems, you need to make sure it includes certain information that will allow you to communicate certain things. Let's look into what you should include as part of each log entry.

TIMESTAMPS

To be able to correlate data and order it appropriately, you need to make sure you attach timestamps to your log entries. Timestamps should be as granular and verbose as possible; for example, use four-digit years and the best resolution available. Each service should render its own timestamps, preferably in microseconds. Timestamps also should include a time zone, and it's advisable that you collect data as GMT/UTC whenever possible.

Having these details allows you to avoid issues with correlating data from different services with different time zones. Ordering data by time of occurrence will become much easier and require less context while analyzing. Getting timestamps right is the first step in making sure you can understand the sequence in which events took place.

IDENTIFIERS

You should use unique identifiers whenever, and as much as, possible in the data you intend to log. Request IDs, user IDs, and other unique identifiers are invaluable when you're cross-referencing data from multiple sources. They allow you to group data from different sources in an effective way.

Most of the time, these IDs already exist in your system because you need to use them to identify resources. It's likely they're already being propagated through different services, so you should make the best use out of them. Unique identifiers used alongside timestamps yield a powerful tool to understand the flow of events in a system.

SOURCE

Identifying the source of a given log entry allows easier debugging when needed. Typical source data you can use includes:

- Host
- Class or module
- Function
- Filename

When adding execution times on a given function call, the information you've collected for the source allows you to infer performance because you can extrapolate execution times, even if not in real time. Although this isn't a replacement for collecting metrics, it can be effective in helping to identify bottlenecks and potential performance issues.

LEVEL OR CATEGORY

Each log entry should contain a category. The category can be either the type of data you're logging or the log level. Typically, the following values are used as log levels: `ERROR`, `DEBUG`, `INFO`, `WARN`.

The category will allow you to group data. Some tools can parse log files searching for messages with the `ERROR` level and communicate them to error reporting systems. This is a perfect example of how you can use the log level or category to automate the process of error reporting without the need for explicit instructions.

12.2.2 Structure and readability

You want to generate log entries in a human-readable format, but at the same time they need to be easily parseable by a machine. What we mean by human readable is avoiding binary encoding of data or any type of encoding that your average human can't understand. An example of this would be storing the binary representation of an image. You should probably use its ID, file size, and other associated data instead.

You also should avoid multiline logs because they can lead to fragmentation while parsing them in log aggregation tools. With such logs, it can be easy to lose some of the information associated with a particular log entry, like the ID, timestamp, or source.

For the examples in this chapter, you'll be using JSON to encode your log entries. Doing so allows you to provide human-readable and machine-parseable data, as well as to automatically include some of the data we mentioned in the previous section.

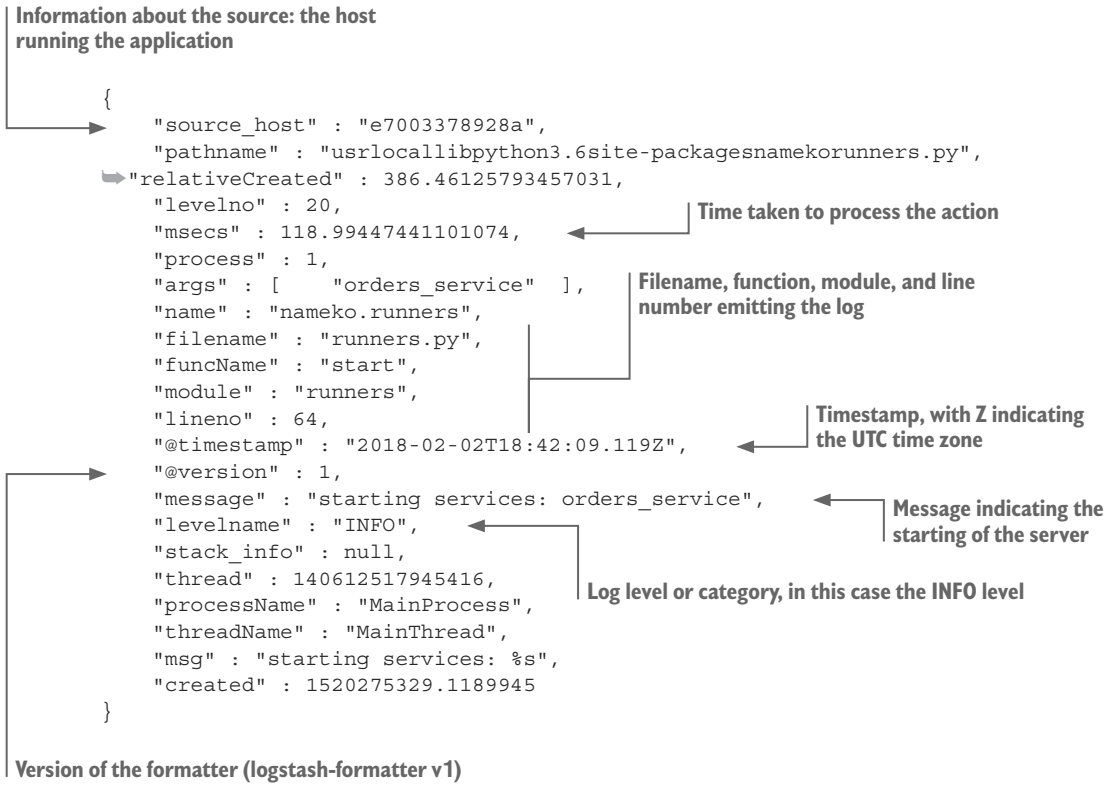
In chapter 7, when we were discussing a microservice chassis, we introduced a Python library that provides log formatting: `logstash-formatter`. Logstash libraries are available for different languages, so you can expect the format to be widespread and easily usable no matter what language you chose to code your services in.

Logstash

Logstash is a tool to collect, process, and forward events and log messages from multiple sources. It provides multiple plugins to configure data collecting.

We're interested in the formatting conventions of Logstash, and you'll be using its V1 format specification in your SimpleBank services.

Let's now look into a log entry collected using the Logstash library for Python. This message is formatted using V1 of the logstash format, and the application generated it automatically when it was booting, without the need for any specific code instruction to log it:



As you can see, the Logstash library inserts relevant information, taking that burden from the developer's shoulders. In the following listing, you'll see how an explicit log call in code renders a log entry.

Listing 12.1 Logstash V1 formatted log message after an explicit log instruction

```
# Python code for generating a log entry
self.logger.info ({"message": "Placing sell order",
➥"uuid": res})
```

The log level, the message field, and the uuid field

```
{
    "@timestamp": "2018-02-02T18:43:08.221Z",
    "@version": 1,
    "source_host": "b0c90723c58f",
    "name": "root",
    "args": [],
    "levelname": "INFO",
    "levelno": 20,
    "pathname": "./app.py",
    "filename": "app.py",
    "module": "app",
    "stack_info": null,
    "lineno": 33,
    "funcName": "sell_shares",
    "created": 1520333830.3000789,
    "msecs": 300.0788688659668,
    "relativeCreated": 15495.944738388062,
    "thread": 140456577504064,
    "threadName": "GreenThread-2",
    "processName": "MainProcess",
    "process": 1,
    "message": "Placing sell order",
    "uuid": "a95d17ac-f2b5-4f2c-8e8e-2a3f07c68cf2"
}
```

The log level determined by the call made to the logger module, in this case, INFO

The uuid field, which identifies and potentially allows correlation between this log entry and another in different services

The message field

In your explicit call to the logger, you've only stated the desired level and the message to log in the form of key-value pairs containing a message and a UUID. Logstash automatically collected and added all the other information present in the log entry without you having to declare it explicitly.

12.3 Setting up a logging infrastructure for SimpleBank

Now that you've set a format for collecting and presenting info, you can move on to creating a basic logging infrastructure. In this section, you'll be setting up the infrastructure that'll allow you to collect logs from all the running services and aggregate them. It also will provide you with search and correlation capabilities. The purpose is to have a central access point to all the log data like you already have for metrics. In figure 12.4, we illustrate what you want to achieve after setting up a logging infrastructure.

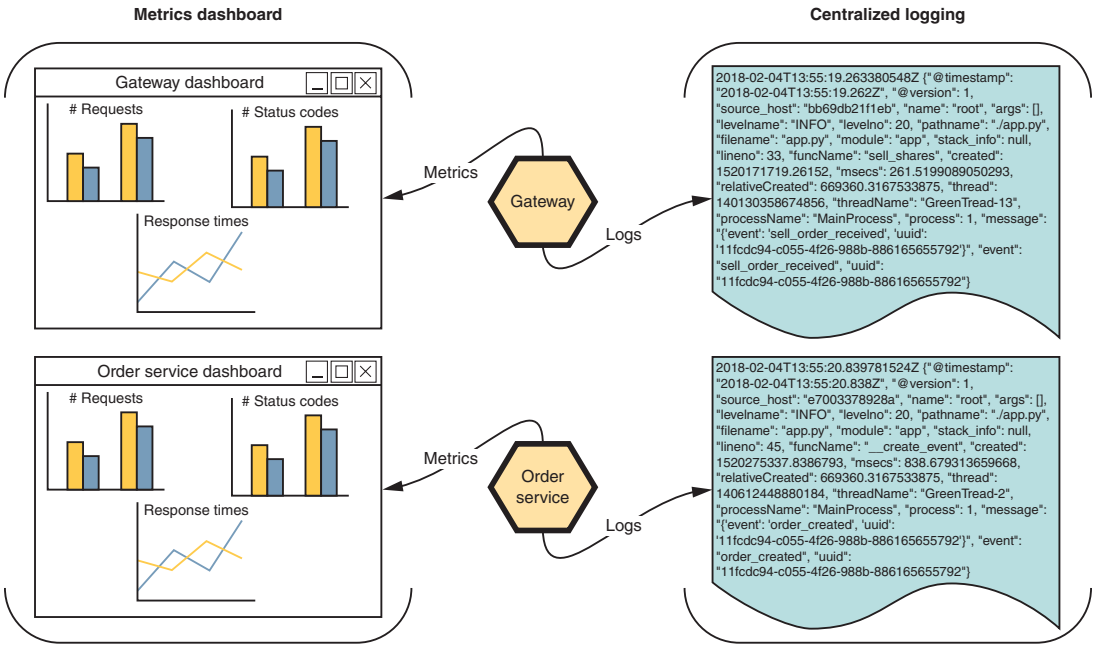


Figure 12.4 Services making use of centralized metrics and a centralized log for easy access to data

Once you set up the log aggregation capability, like you did for metrics in chapter 11, you'll have all services sending both metrics and logs to centralized systems that'll allow you to improve observability. You'll be able to observe data about a running system and dig deeper to collect more information in case you need to audit or debug a particular request. You'll set up a solution commonly called ELK (Elasticsearch, Logstash, and Kibana) and will use a data collector called Fluentd.

12.3.1 ELK- and Fluentd-based solution

You'll build the logging infrastructure we propose using Elasticsearch, Logstash, and Kibana. Also, you'll use Fluentd for pushing logs from the apps to your centralized logging solution. Before we get into more details about these technologies, have a look at figure 12.5 to get an overview of what we want to enable you to achieve.

In figure 12.5, you can see how you can collect the logs for multiple instances of the gateway service and forward them to your centralized logging system. We represent multiple instances of the same service, but this will work for any of the services you have running. Services will redirect all the log information to `STDOUT` (standard output), and an agent running the Fluentd daemon will be responsible for pushing those logs into Elasticsearch.

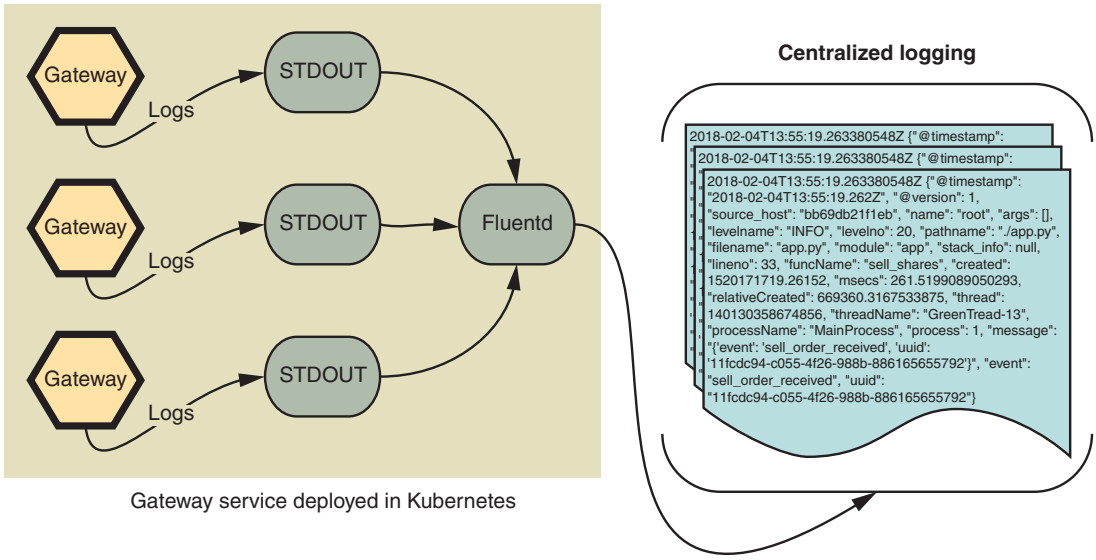


Figure 12.5 Collecting logs from multiple service instances and forwarding them to a centralized location

By following this pattern when deploying any new services, you'll make sure log data gets collected and indexed and becomes searchable. But before we move to implementation, we'll take a little time to introduce each of the technologies you'll be using.

ELASTICSEARCH

Elasticsearch (www.elastic.co/products/elasticsearch) is a search and analytics engine that stores data centrally. It indexes data, in your case log data, and allows you to perform efficient search and aggregation operations on the data it has stored.

LOGSTASH

Logstash (www.elastic.co/products/logstash) is a server-side processing pipeline that allows data ingestion from multiple sources and transforms that data prior to sending it to Elasticsearch. In your case, you'll be using the Logstash formatting and data collection capabilities by taking advantage of client libraries. In this chapter's earlier examples, you already observed its ability to provide consistent data that you can send to Elasticsearch. But here you won't be using Logstash to send data; you'll be using Fluentd instead.

KIBANA

Kibana (www.elastic.co/products/kibana) is a UI for visualizing Elasticsearch data. It's a tool you can use to query data and explore its associations. In your use case, it'll operate on log data. You can use Kibana to derive visualizations from gathered data, so it's more than a search tool. Figure 12.6 shows an example of a dashboard powered by Kibana.

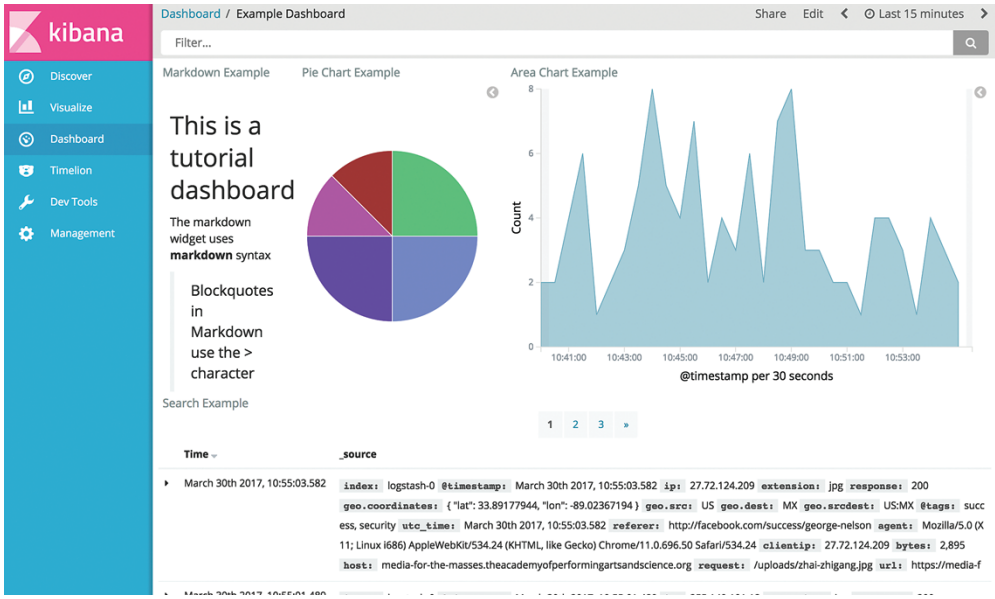


Figure 12.6 Kibana tutorial dashboard showing visualizations created from log data

FLUENTD

Fluentd (www.fluentd.org) is an open source data collector that you'll be using to push data from your services to Elasticsearch. You'll combine the data formatting and collecting capabilities of Logstash and use Fluentd to push that data. One of its advantages is the fact that you can use it as a logging provider for Dockerfiles if you declare it in Docker compose files.

12.3.2 Setting up your logging solution

You'll set up your solution via the Docker compose file, like you already did in chapter 11 to create the metrics collecting and alert infrastructure. You can find all the code used in this chapter at Github (http://mng.bz/k191). There, you'll find the docker-compose.yml file, where you'll be declaring the new dependencies. The following listing shows the new components added to the compose file.

Listing 12.2 Docker compose file with Elasticsearch, Kibana, and Fluentd containers

```
version: '2.1'
services:

  gateway:
    container_name: simplebank-gateway
    restart: always
    build: ./gateway
    ports:
```

```
      - 5001:5000
    volumes:
      - ./gateway:/usr/src/app
    links:
      - "rabbitmq:simplebank-rabbitmq"
      - "fluentd"
    logging:
      driver: "fluentd"
      options:
        fluentd-address: localhost:24224
        tag: simplebank.gateway

(…)
  kibana:
    image: kibana
    links:
      - "elasticsearch"
    ports:
      - "5601:5601"

  elasticsearch:
    image: elasticsearch
    expose:
      - 9200
    ports:
      - "9200:9200"

  fluentd:
    build: ./fluentd
    volumes:
      - ./fluentd/conf:/fluentd/etc
    links:
      - "elasticsearch"
    ports:
      - "24224:24224"
      - "24224:24224/udp"
(…)
```

Adds a logging directive to each service to force Docker to push the output of each container running a service to Fluentd, which in turn will make sure data gets pushed to Elasticsearch

For Kibana, uses the default image in Docker Hub with the defaults set

Links Kibana to the Elasticsearch container, because it'll consume data from it

Like you did for Kibana, uses the default image for Elasticsearch

Builds Fluentd from a custom Docker image

Injects the configuration for Fluentd into the built container, allowing you to tweak the default configuration

Links the Fluentd container to the Elasticsearch container, because it'll be pushing data into it

Once you've added this content to the docker compose file, you're almost ready to boot your logging infrastructure alongside your services. But first let's cover the missing tweaks we mentioned previously that'll allow you to configure Fluentd to your needs. The Dockerfile you use for building Fluentd follows.

Listing 12.3 Fluentd Dockerfile (Fluentd/Dockerfile)

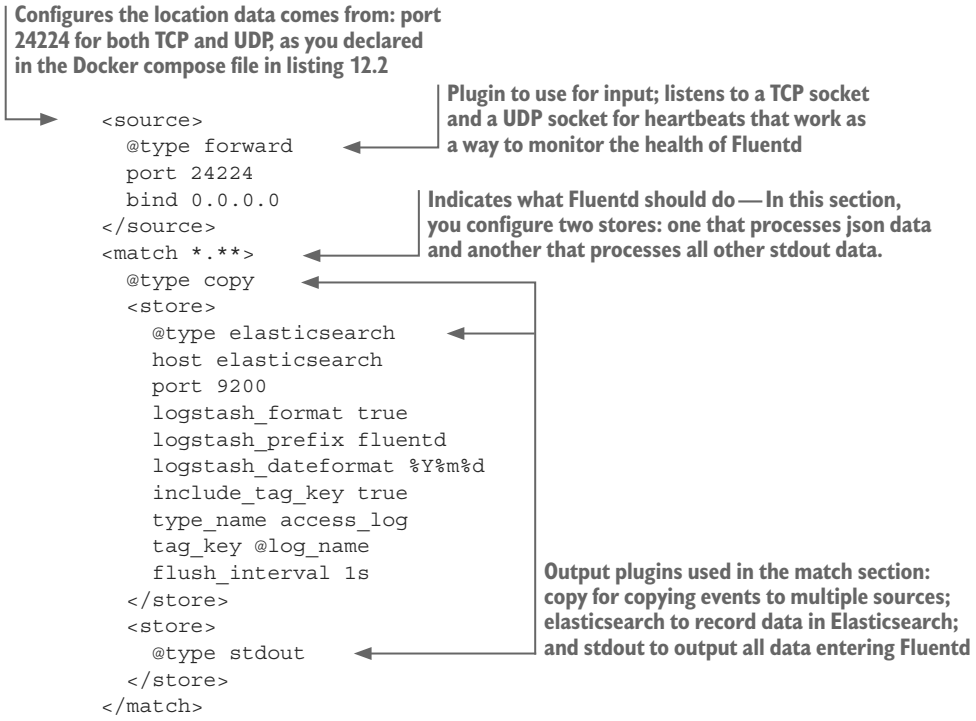
```
FROM fluent/fluentd:v0.12-debian
RUN ["gem", "install", "fluent-plugin-elasticsearch",
➥"--no-rdoc", "--no-ri", "--version", "1.9.2"]
```

Pulls the Fluentd base image

Installs the Elasticsearch plugin for Fluentd

Now all you need is to create a configuration file for Fluentd. The following listing shows the config file.

Listing 12.4 Fluentd configuration (fluentd/conf/fluent.conf)




```
<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>
<match *.**>
  @type copy
  <store>
    @type elasticsearch
    host elasticsearch
    port 9200
    logstash_format true
    logstash_prefix fluentd
    logstash_dateformat %Y%m%d
    include_tag_key true
    type_name access_log
    tag_key @log_name
    flush_interval 1s
  </store>
  <store>
    @type stdout
  </store>
</match>
```

The match section in the Fluentd configuration file contains all the needed configuration for connecting with `elasticsearch`, `port`, and `host`, as well as the format used. You're using logstash format, as you may recall.

With all the needed setup done, you're now ready to boot your services using the new Docker compose file. Before doing so, let's go through your services and change code to enable you to send data to your centralized logging infrastructure. In the next section, you'll configure your services to make use of the Logstash logger. You'll also set log levels.

12.3.3 Configure what logs to collect

In your services, you can control the log level via environment variables, so you can have different levels for development and production environments. Using different log levels allows you to enable more verbose logs in production, in case you need to investigate any issue.

Let's look at your gateway service for its logging configuration and also at the service code to understand how you can emit log messages. The logging configuration is shown in the following listing.

Listing 12.5 Configuration file for the gateway service (gateway/config.yml)

```
AMQP_URI: amqp://${RABBIT_USER:guest}:${RABBIT_PASSWORD:guest}@${RABBIT_HOST:
➥localhost}:${RABBIT_PORT:5672}/
WEB_SERVER_ADDRESS: '0.0.0.0:5000'
RPC_EXCHANGE: 'simplebank-rpc'
LOGGING:
    version: 1
    handlers:
        console:
            class: logging.StreamHandler
    root:
        level: ${LOG_LEVEL:INFO}
        handlers: [console]
```

Logging configuration section

Defines the handler class for console logging, which you'll use as the handler in listing 12.6

Reads the log level from an environment variable (LOG_LEVEL) and sets a default value of INFO in case the environment variable isn't defined

Registers only a `console` handler, because you won't be reading from log files

This configuration will allow setting the log level when the application boots. In the Docker compose file, you've set the environment variable `LOG_LEVEL` as `INFO` for all services except for the Gateway, which has a `DEBUG` value. Let's now look into the gateway code to set up logging, as shown in the following listing.

Listing 12.6 Enable logging in the gateway service (gateway/app.py)

```
import datetime
import json
import logging
import uuid

from logstash_formatter import LogstashFormatterV1
from nameko.rpc import RpcProxy, rpc
from nameko.web.handlers import http
from statsd import StatsClient
from werkzeug.wrappers import Request, Response

class Gateway:

    name = "gateway"
    orders = RpcProxy("orders_service")
    statsd = StatsClient('statsd', 8125,
                         prefix='simplebank-demo.gateway')

    logger = logging.getLogger()
    handler = logging.StreamHandler()
    formatter = LogstashFormatterV1()

    handler.setFormatter(formatter)
    logger.addHandler(handler)

    @http('POST', '/shares/sell')
    @statsd.timer('sell_shares')
```

Imports Python's logging facility (https://docs.python.org/3/library/logging.html)

Imports the Logstash Formatter so you can emit logs in logstash format

Initializes and configures logger

Examples of log messages emitted with the INFO log level

Example of how to log a message with the DEBUG level—This message will only be sent if the application log level is set to DEBUG. If you set it to INFO, this message won't be sent.

```
    def sell_shares(self, request):
        req_id = uuid.uuid4()
        res = u"{}".format(req_id)

        self.logger.debug(
            "this is a debug message from gateway",
➥extra={"uuid": res})

        self.logger.info("placing sell order", extra=
➥{"uuid": res})

        self.__sell_shares(res)

        return Response(json.dumps(
            {"ok": "sell order {} placed".format(req_id)}),
            ➥mimetype='application/json')

    @rpc
    def __sell_shares(self, uuid):
        self.logger.info("contacting orders service", extra={
            ➥"uuid": uuid})

        res = u"{}".format(uuid)
        return self.orders.sell_shares(res)

    @http('GET', '/health')
    @statsd.timer('health')
    def health(self, _request):
        return json.dumps({'ok': datetime.datetime.utcnow().__str__()})
```

In listing 12.2, you saw how to enable the Fluentd driver for logging with Docker. This means you're ready to send log data generated from your services to Elasticsearch using Fluentd, and afterwards you'll be able to explore that log data using Kibana. To start all the services, metrics, and logging infrastructure from a console in the root directory, execute the following command:

```
docker-compose up --build --remove-orphans
```

Once all is ready, you need to complete one last step, which is configuring Kibana to use the logs collected via Fluentd and stored in Elasticsearch. To do so, access the Kibana web dashboard (http://localhost:5601). On the first access, you'll be redirected to the management page, where you'll need to configure an index pattern. You need to tell Kibana where to find your data. If you recall, in the Fluentd configuration, you set an option for the logstash prefix with the value `fluentd`. This is what you need to enter in the index text box presented to you. Figure 12.7 shows the Kibana dashboard management section and the value you need to input.

After inserting `fluentd-*` as the index pattern and clicking the Create button, you'll be ready to explore all the log data that your multiple services create. Elasticsearch will forward all data to a central location, and you'll be able to access it in a convenient way.

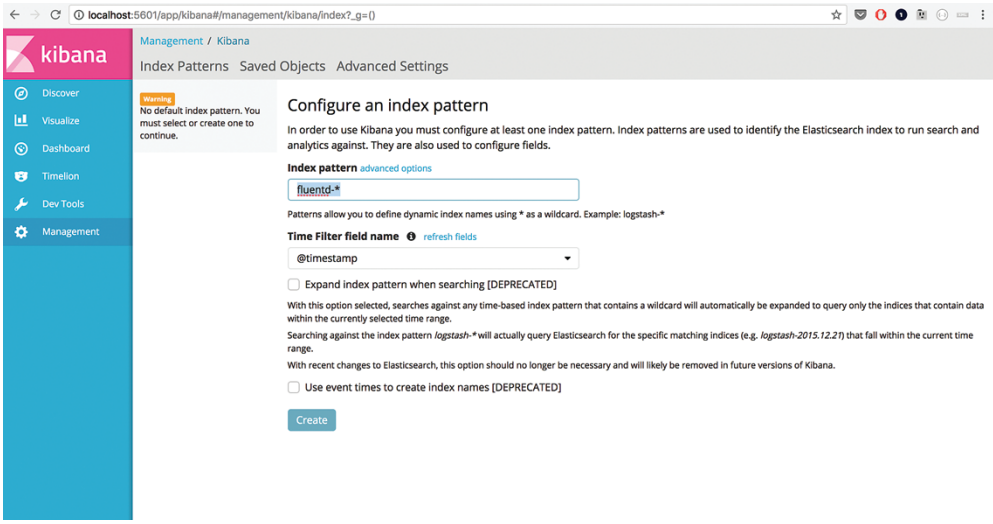


Figure 12.7 The Kibana management section where you need to indicate the index pattern for it to be able to fetch data from Elasticsearch

To generate some log data, all you need to do is create a sell request to your gateway service. To do so, you need to issue a `POST` request to the gateway. The following shows presenting a request via curl, but any tool capable of generating a `POST` request will do:

```
chapter-12$ curl -X POST http://localhost:5001/shares/sell \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/json'
```

curl command to the gateway service

```
chapter-12$ {"ok": "sell order e11f4713-8bd8-4882-b645
➥-55f96d220e44 placed"}
```

Response from the service — the UUID you receive allows you to identify your sell order, and you can use it as a search term on Kibana. (The UUID shown is randomly generated, so please use the one you receive as a response to the request you issued.)

Now that you have log data collected, you can explore it using Kibana. Clicking on the Discover section on the left side of Kibana's web dashboard will take you to a page where you can perform searches. In the search box, insert the request UUID you received as the sell order response. In the case of this example, you'd be using `e11f4713-8bd8-4882-b645-55f96d220e44` as your search parameter. In the next section, we'll show you how to use Kibana to follow the execution of a sell request through the different services involved.

12.3.4 Finding a needle in the haystack

In the code example for this chapter, you have five independent services collaborating to allow SimpleBank users to sell shares. All services are logging their operations and using the request UUID as a unique identifier to allow you to aggregate all log

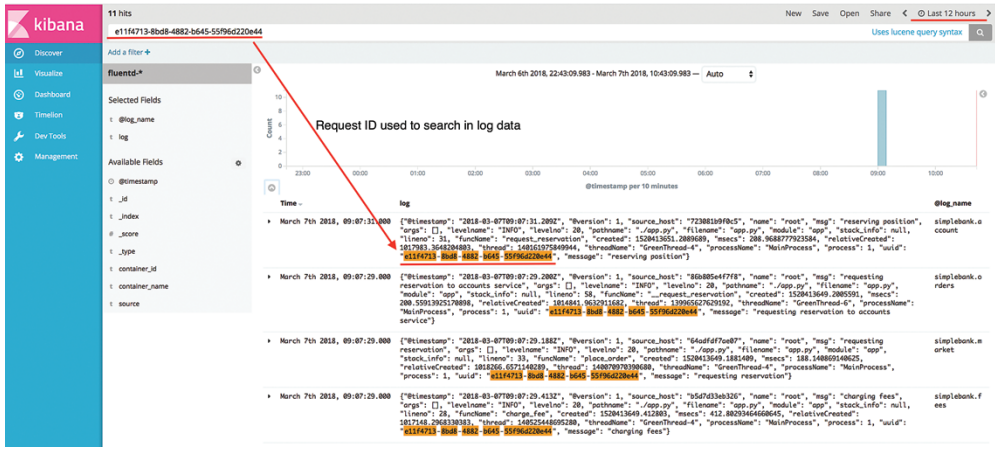


Figure 12.8 Searching the log data using a request ID

messages referring to a sell order that's processing. You can make use of Kibana to explore your logs and track down the execution of a request. In figure 12.8, you use the order ID that the gateway service returns to perform the search.

When you use the request ID as the search parameter, Kibana filters the log data, and you get 11 hits that allow you to follow the execution of a request through different services. Kibana allows you to use complex queries to be able to uncover insights. You get the ability to filter per service, sort by time, and even use log data—for example, execution times present in the log entries—to create dashboards to track performance. This use is beyond the scope of this chapter, but do feel free to explore the possibilities offered to get new perspectives on the collected data.

We'll now zoom in a bit and focus on some of the log entries that your query shows. Figure 12.9 shows some of those entries with a bit more detail.

Figure 12.9 Detailed view of log messages in the Kibana search page

In figure 12.9, you'll find messages from the market and gateway services. For the latter, given that the log level selected was DEBUG, you'll find both info and debug messages. As mentioned previously, by using the `logstash-formatter` library in your Python code, you get more information for free. You can find data regarding execution times and scoping of the execution by module, function, line, process, and thread. All of this info can be useful if you need to diagnose any issues in the future.

12.3.5 Logging the right information

Now that you're able to collect logs and store them, you need to be careful about what information you send via logs. Things like passwords, credit card numbers, ID card numbers, and potentially sensitive personal data that get sent will be stored and accessible by anyone who can use the logging infrastructure. In your case, you're hosting and controlling the logging infrastructure, but if you were using a third-party provider, you'd need to pay extra attention to these details. You have no easy way to delete only some of the data already sent. In most cases, if you want to delete something specific, it means deleting all the log data for a given period of time.

Data privacy is a hot topic at the moment, and with the EU General Data Protection Regulation (GDPR) (www.eugdpr.org) now in effect, you need to take extra care when considering which data to log and how to log it. We won't explore the needed steps in depth here, but both Fluentd and Elasticsearch allow you to apply filtering to data so that any sensitive fields get masked, encrypted, or removed from the data that they receive and that Elasticsearch indexes. The general rule would be to log as little information as you can, avoid any personal data in logs, and take extra care with reviewing what gets logged before any changes make it into the production environment. Once your services send data, it's hard to erase it and doing so will have associated costs.

That said, you can and should use logs to communicate useful information to allow you to understand system behavior. Sending IDs that allow you to correlate actions of different systems and terse log messages indicating actions performed in or by systems can help you keep track of what's happened.

12.4 Tracing interactions between services

When you were setting up your log infrastructure and making the code changes to emit log messages, you already took care of propagating an ID field that allows you to follow the execution path of a request through your system. With this set up, you can group log entries under the same context. You may even use log data to create visualizations that'll allow you to understand how much time each component took to process a request. In addition, you can use it to help you identify bottlenecks and places where you can improve your code to have extra performance. But logs aren't the only tool available—you have another method at your disposal for doing this that doesn't rely on log data.

You can do better by reconstructing the journey of requests through your microservices. In this section, you'll be setting up distributed tracing to allow you to visualize the flow of execution between services and at the same time, provide insights on how long each operation takes. This can be valuable, not only to understand the order in which a request flows

through multiple services, but also to identify possible bottlenecks. For this, you'll use Jaeger and libraries compatible with the OpenTracing API (http://opentracing.io).

OpenTracing API

The OpenTracing API is a vendor-neutral open standard for distributed tracing. A lot of distributed tracing systems (for example, Dapper, Zipkin, HTrace, X-Trace) provide tracing capabilities but do so using incompatible APIs. Choosing one of those systems would generally mean tightly coupling systems potentially using different programming languages to a single solution. The purpose of the OpenTracing initiative is to provide a set of conventions and a standard API for collection of traces. Libraries are available for multiple languages and frameworks. You can find some of the supported tracer systems at http://mng.bz/Gvr3.

12.4.1 Correlating requests: traces and spans

A *trace* is a direct acyclic graph (DAG) of one or more *spans*, where the edges of those spans are called *references*. Traces are used to aggregate and correlate execution flow through the system. To do so, some information needs to be propagated. A trace captures the whole flow.

Let's look at figures 12.10 and 12.11. In these figures, we represent a trace made up of multiple spans, from both a dependency perspective and a temporal perspective.

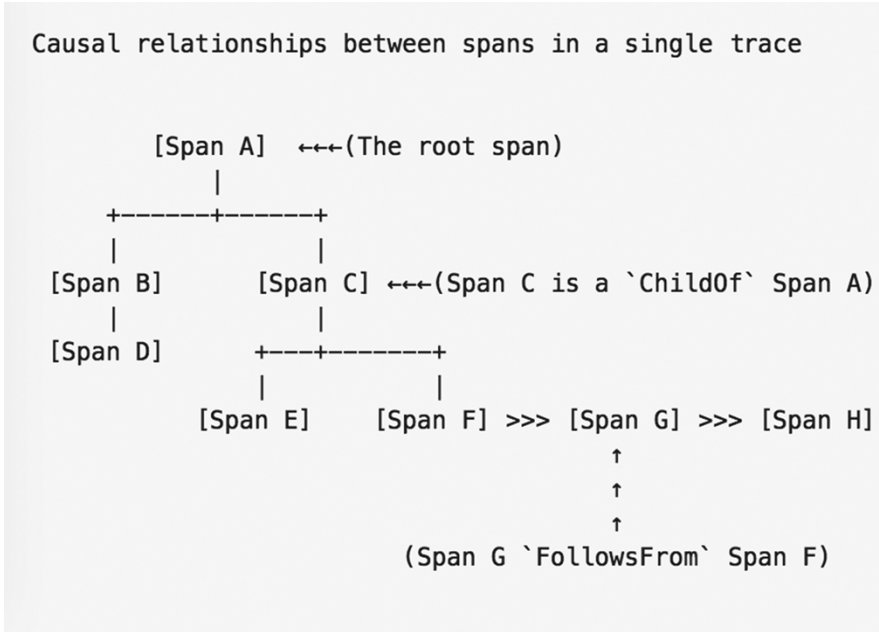


Figure 12.10 A trace made up of eight different spans from a dependency perspective

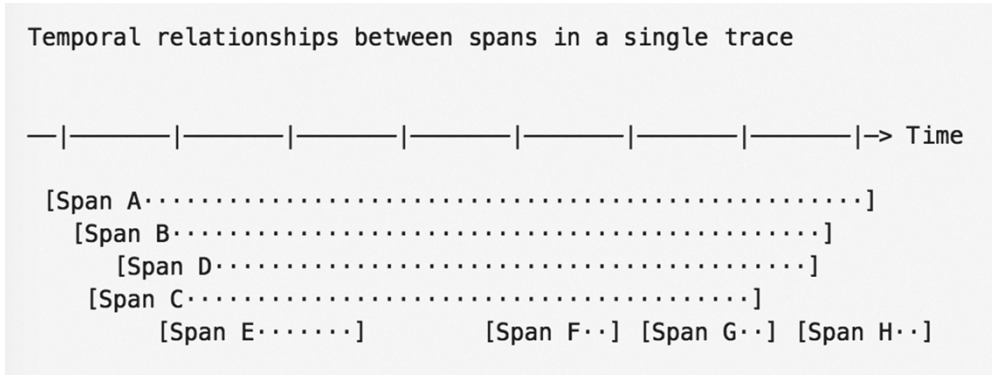


Figure 12.11 The temporal relationships in an eight-span trace

In figure 12.10, you can observe the dependency relationship between different spans. These spans can be triggered either in the same application or in different ones. The only requirement is for the parent span ID to be propagated, so when a new span is triggered, it'll hold a reference to its parent span.

In figure 12.11, you have a view of spans from a temporal perspective. By using temporal information contained in spans, you can organize them in a timeline. You can see not only when each span happened relative to other spans but also how long each operation that a span encapsulates took to complete.

Each span contains the following information:

- An operation name
- A start and a finish timestamp
- Zero or more span tags (key value pairs)
- Zero or more span logs (key value pairs with a timestamp)
- A span context
- References to zero or more spans (via the span context)

The span context contains the needed data to refer to different spans, either locally or across service boundaries.

Let's now move on to setting up tracing between services. You'll be using Jaeger (www.jaegertracing.io), a distributed tracing system, as well as a set of Python libraries that are OpenTracing compatible.

12.4.2 Setting up tracing in your services

To be able to display tracing information and correlate requests between different services, you'll need to set up a collector and a UI for traces, as well as including some libraries and setting them up in your services. The services we'll use as an example for distributed tracing will be the SimpleBank profile and settings services. In figure 12.12, we give an overview of the interactions you'll be tracing.

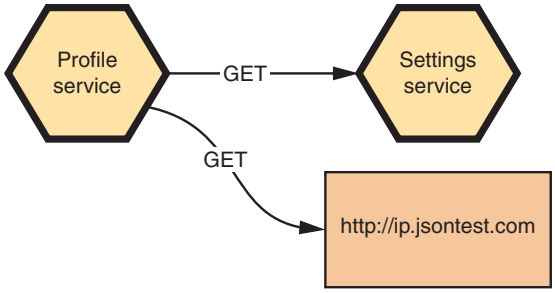


Figure 12.12 Interactions of the profile service

The profile service will contact an external service, in this case jsontest.com, to retrieve its IP and will also be fetching user settings from the settings service. You'll be setting up a tracing system (Jaeger) and making the code changes needed to display the trace and its spans and to be able to correlate those spans. Correlating them will allow you to understand in detail how long each operation took and how it contributed to the overall execution time of a call to the profile service. You'll begin by setting up Jaeger, the distributed tracing system collector and UI, by adding a Docker image into your docker-compose.yml file (listing 12.7).

Jaeger

Inspired by Dapper and OpenZipkin, Jaeger is a distributed tracing system released as open source by Uber Technologies. You use it for monitoring and troubleshooting microservice-based distributed systems.

Listing 12.7 Adding Jaeger to the docker-compose.yml file

```
(...)
jaeger:
    container_name: jaeger
    image: jaegertracing/all-in-one:latest
    ports:
       - 5775:5775/udp
       - 6831:6831/udp
       - 6832:6832/udp
       - 5778:5778
       - 16686:16686
       - 14268:14268
       - 9411:9411
    environment:
      COLLECTOR_ZIPKIN_HTTP_PORT: "9411"
```

You'll be using a jaeger image containing all the needed components because it'll be easier to set up. This all-in-one image has in-memory-only storage for spans.

Port for communicating spans

Port for accessing the Jaeger UI

Port used by Zipkin, another distributed tracing system — One of the advantages of the OpenTracing initiative is the fact that you can use different systems without the need to change all implementations or be locked to a particular one.

With the Docker image added to your docker-compose file, once you boot all the SimpleBank infrastructure, you'll have a distributed tracing system in place. Now you need to make sure that the SimpleBank profile and settings services are able to create traces and spans and communicate them to Jaeger.

Let's add the needed libraries to both the settings and profile services and initialize the tracer. The following listing adds the tracing libraries.

Listing 12.8 Adding the tracing libraries to the services via a requirements.txt file

```
Flask==0.12.0
requests==2.18.4
jaeger-client==3.7.1
opentracing>=1.2,<2
opentracing_instrumentation>=2.2,<3
```

Jaeger client library that connects the service to the tracer system

Python OpenTracing platform library

Collection of instrumentation tools to simplify integration with different frameworks and applications

By adding these libraries, you're now able to create traces and spans from both services. To make the process easier, you can also create a module to provide a convenient setup function to initialize the tracer, as shown in the following listing.

Listing 12.9 Tracer initializer lib/tracing.py

Receives the service name as an argument

Imports the Jaeger client that allows establishing communication between the app and the tracing collector system

```
import logging
from jaeger_client import Config

def init_tracer(service):
    logging.getLogger('').handlers = []
    logging.basicConfig(format='%(message)s', level=logging.DEBUG)

    config = Config(
        config={
            'sampler': {
                'type': 'const',
                'param': 1,
            },
            'local_agent': {
                'reporting_host': "jaeger",
                'reporting_port': 5775,
            },
            'logging': True,
            'reporter_batch_size': 1,
        },

        service_name=service,
    )
    return config.initialize_tracer()
```

Sets up both the host and the port where traces and spans will be sent—In the Docker compose file, you have Jaeger running as "jaeger" and receiving metrics via UDP on port 5775. This is necessary because you'll have one collector agent running for all services.

Sets the service name to the one received as the init function argument

In addition to collecting metrics in Jaeger, you're also emitting the trace events to the logs.

The SimpleBank profile and settings services will both be using the tracer initialization function shown in listing 12.9. This allows them to establish the connection to Jaeger. As we showed in figure 12.12, the profile service contacts both an external service and the settings internal service. You'll be tracing the interaction of the profile service with both of these collaborators. In the case of the interaction with the SimpleBank settings service, you'll need to pass along the context of the initial trace so you can visualize the full cycle of a request.

Listing 12.10 shows the profile service code where you set up the spans for both the external http service and the settings service. For the former, you create a span, and for the latter, you pass along the current span as a header so the settings service can make use of it and create child spans.

Listing 12.10 Profile service code

Imports the initializer function as defined in listing 12.9 to set up the connection to Jaeger

```
from urlparse import urljoin

import opentracing
import requests
from flask import Flask, jsonify, request
from opentracing.ext import tags
from opentracing.propagation import Format
from opentracing_instrumentation.request_context import
➥get_current_span
from opentracing_instrumentation.request_context import
➥span_in_context

from lib.tracing import init_tracer

app = Flask(__name__)
tracer = init_tracer('simplebank-profile')

@app.route('/profile/<uuid:uuid>')
def profile(uuid):
    with tracer.start_span('settings') as span:
        span.set_tag('uuid', uuid)
        with span_in_context(span):
            ip = get_ip(uuid)
            settings = get_user_settings(uuid)
            return jsonify({'ip': ip, 'settings': settings})

def get_ip(uuid):
    with tracer.start_span('get_ip', child_of=
➥get_current_span()) as span:
        span.set_tag('uuid', uuid)
        with span_in_context(span):
            jsontest_url = "http://ip.jsontest.com/"
            r = requests.get(jsontest_url)
            return r.json()
```

Imports the OpenTracing libraries to allow you to set up spans and tags

Calls the tracer initializer passing the service name, which will create a tracer object you can use

Sets the initial span associated with the tracer—The created span will be the parent for spans in both the call to the external service and the call to the settings service.

Creates a new span for the call to the external service, a child of the parent span initialized above.

Wraps code execution under the newly created span

```
def get_user_settings(uuid):
    settings_url = urljoin("http://settings:5000/
➥settings/", "{}".format(uuid))

    span = get_current_span()
    span.set_tag(tags.HTTP_METHOD, 'GET')
    span.set_tag(tags.HTTP_URL, settings_url)
    span.set_tag(tags.SPAN_KIND, tags.SPAN_KIND_RPC
➥_CLIENT)
    span.set_tag('uuid', uuid)
    headers = {}
    tracer.inject(span, Format.HTTP_HEADERS, headers)

    r = requests.get(settings_url, headers=headers)
    return r.json()

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

Sets tags for the span

Injects the span context before the call to the SimpleBank settings service—The span context will be passed in the headers, and the downstream service will use it to initialize its own spans under the proper context.

The SimpleBank profile service initializes a trace that'll be used to group different spans. It creates spans for the calls to `"http://ip.jsontest.com/"` and for the call to the SimpleBank settings service. For the former, given that you don't own the service, you execute the call wrapped in a span. But for the latter, because you control it, you can pass on span information that'll be used to create child spans. This will allow you to group all related calls in Jaeger.

Let's now look into how you can make use of the injected span in the SimpleBank settings service, as shown in the following listing.

Listing 12.11 Using a parent span in the settings service

```
import time
from random import randint

import requests
from flask import Flask, jsonify, request
from opentracing.ext import tags
from opentracing.propagation import Format
from opentracing_instrumentation.request_context import get_current_span
from opentracing_instrumentation.request_context import span_in_context

from lib.tracing import init_tracer

app = Flask(__name__)
tracer = init_tracer('simplebank-settings')

@app.route('/settings/<uuid:uuid>')
def settings(uuid):
    span_ctx = tracer.extract(Format.HTTP_HEADERS,
➥request.headers)
    span_tags = {tags.SPAN_KIND: tags.SPAN_KIND_RPC
➥_SERVER, 'uuid': uuid}
```

Initializes the tracer for the service

Extracts the span context from the request headers

Sets up the tags for a new span

```
        with tracer.start_span('settings', child_of=span
➥_ctx, tags=span_tags):
            time.sleep(randint(0, 2))
            return jsonify({'settings': {'name': 'demo user', 'uuid': uuid}})

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

Starts a new span as a child of the one propagated to the service via the request headers

By extracting the span context from the request the settings service receives, you can then make use of it as the parent to new spans. You can later visualize this new child span independently. But you'll also be able to take advantage of the fact that Jaeger will show the span as both an independent span in the context of the SimpleBank settings service and a child span in the context of the SimpleBank profile service.

12.5 Visualizing traces

With all the setup out of the way, all you need to do to start collecting traces is to issue a request to the SimpleBank profile endpoint. You can use the command line or a browser. To access traces via the command line, you can use curl to issue the following request:

```
$ curl http://localhost:5007/profile/26bc34c2-5959-4679-9d4d-491be0f3c0c0
{
  "ip": {
    "ip": "178.166.53.17"
  },
  "settings": {
    "settings": {
      "name": "demo user",
      "uuid": "26bc34c2-5959-4679-9d4d-491be0f3c0c0"
    }
  }
}
```

Here's a brief recap of what's going on when you hit the profile endpoint:

- The profile service creates a span A.
- The profile service contacts an external service to fetch the IP, wrapping it under a new span B.
- The profile service contacts the internal SimpleBank settings service to get user info under a new span C and passes the context of the parent span to the downstream service.
- Both services communicate spans to Jaeger.

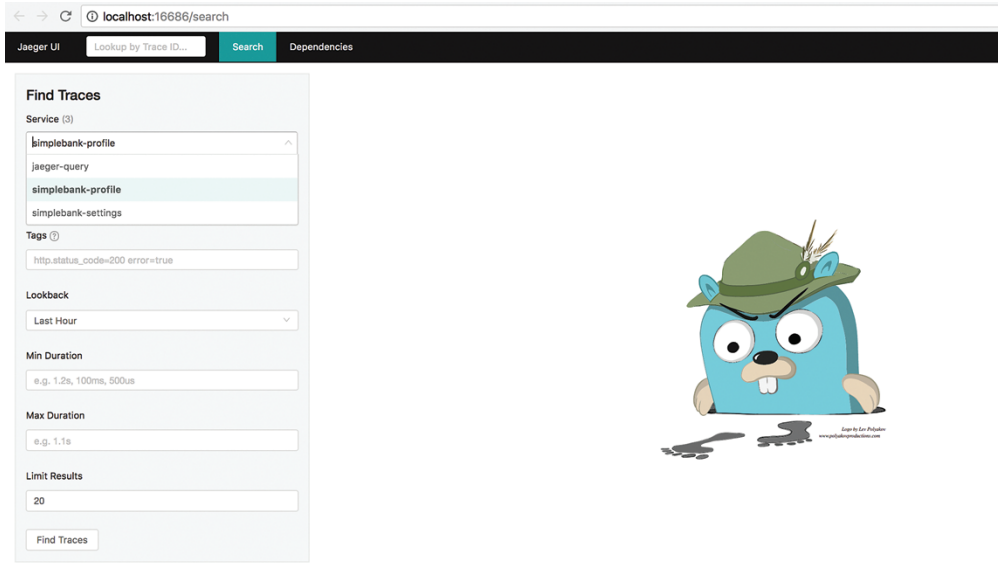


Figure 12.13 Jaeger UI search page showing the services that have traces available

To visualize the traces, you need to access the Jaeger UI that'll be running on port 16686. Figure 12.13 shows the Jaeger UI and the list of services that have traces available.

In the Service section, you see three services for which trace information is available: two SimpleBank services and one called jaeger-query. The latter gathers Jaeger internal traces and is of little use to you. You're interested in the other two services listed: simplebank-profile and simplebank-settings. If you recall, the profile service was creating spans for the execution of an external call, as well as for the call to the settings service. Go ahead and select simplebank-profile and click Find Traces at the bottom. Figure 12.14 shows the traces for the profile service.

The page lists six traces, and all of them have three spans across two services. This means you were able to collect information about the collaboration between two internal services and to get timing information about the execution. Figure 12.15 shows a detailed view of one of those traces.

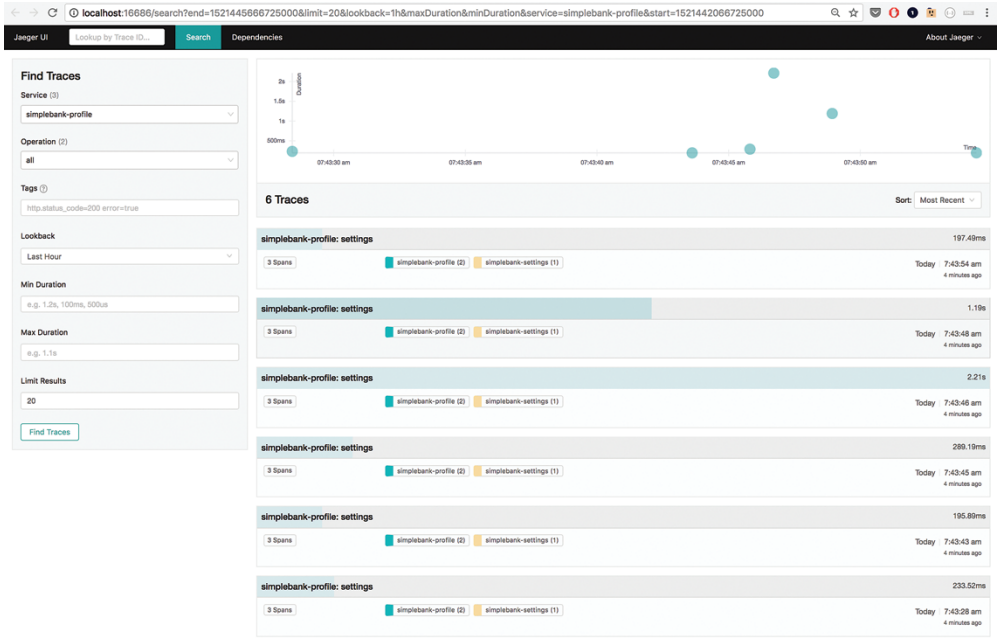


Figure 12.14 The simplebank-profile traces information

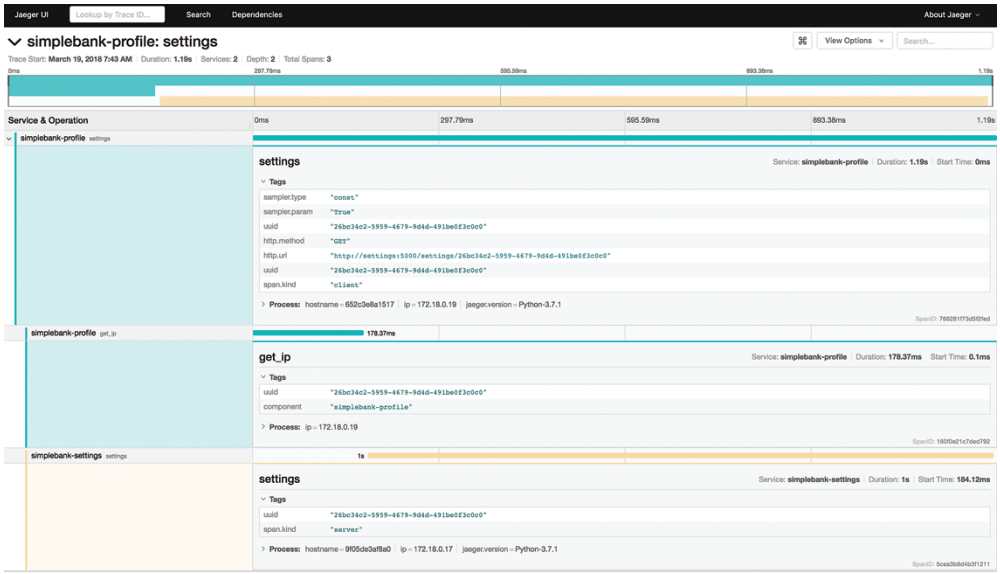


Figure 12.15 The timing information and execution sequence of a call to the profile service

In figure 12.15, you can see a timeline for the different steps of execution in a call to the profile service. You have information about the overall time of execution, as well as when each suboperation took place and for how long. The spans contain information about the operation, the component that generated them, and their execution times and relative positions, both in the timeline and in regards to dependencies with parent spans.

This information can be invaluable in order to know what's going on in a distributed system. You can now visualize the flow of requests through different services and know how long each operation takes to complete. This simple setup allows you to both understand the flow of execution in a microservice architecture and identify potential bottlenecks that you can improve.

You also can use Jaeger to understand how different components in your system relate to each other. The top navigation menu bar has a Dependencies link. By clicking it and then, in the page that comes up, selecting the DAG (direct acyclic graph) tab, you have access to the view illustrated in figure 12.16.

The example we used was a simple one, but it allows you to understand the power of tracing in a microservice architecture. Along with logging and metrics, it allows you to have an informed view of both the performance and the behavior of your system.

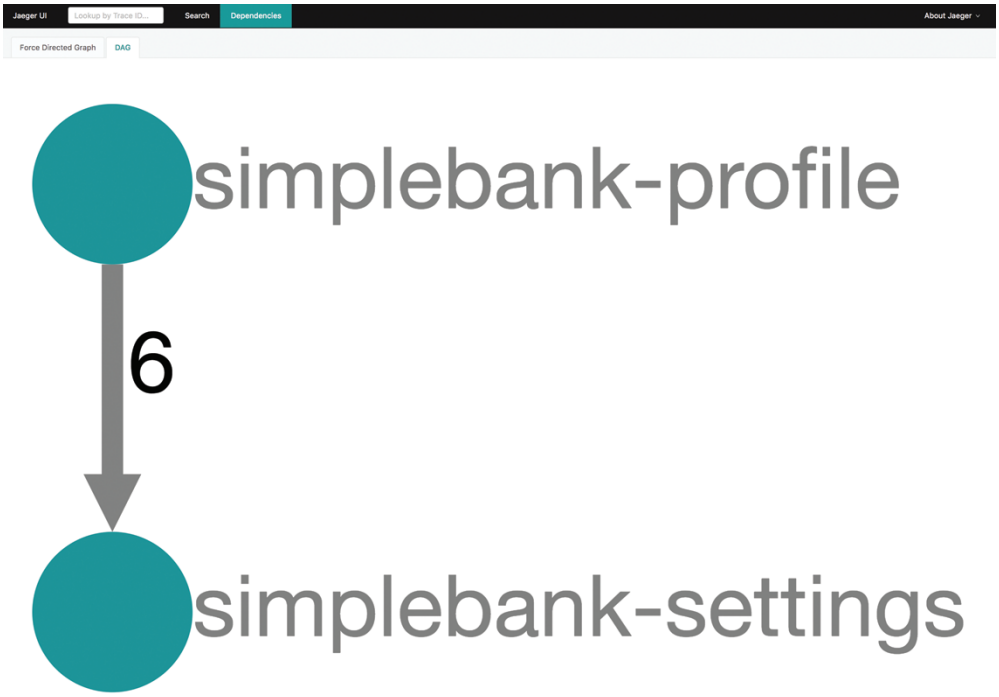


Figure 12.16 The service dependency view in the Jaeger UI

Summary

- You can set up a logging infrastructure using Elasticsearch, Kibana, and Fluentd, and distributed tracing using Jaeger.
- A logging infrastructure can generate, forward, and store indexed log data that allows searching and correlating requests.
- Distributed tracing allows you to follow the journey of execution of requests through different microservices.
- Alongside metrics collection, tracing allows you to better understand how the system is behaving, identify potential issues, and audit your system anytime.

13 Building microservice teams

This chapter covers

- How a microservice architecture affects your engineering culture and organization
- Strategies and techniques for building effective microservice teams
- Common pitfalls in microservice development
- Governance and best practice in large microservice applications

Throughout this book, we've focused on the technical side of microservices: how to design, deploy, and operate services. But it'd be a mistake to examine the technical nature of microservices alone. People implement software, and building great software is as much about effective communication, alignment, and collaboration as implementation choices.

A microservice architecture is great for getting things done. It allows you to build new services and capabilities rapidly and independently of existing functionality. Conversely, it increases the scope and complexity of day-to-day tasks, such as operations, security, and on-call support. It can significantly change an organization's technical strategy. It demands a strong culture of ownership and accountability from

engineers. Achieving this culture, while minimizing friction and increasing pace, is vital to a successful microservice implementation.

In this chapter, we'll begin by discussing team formation in software engineering and the principles that make teams effective. We'll then examine different models for engineering team structure and how they apply to microservice development. Lastly, we'll explore recommended practices for governance and engineering culture within microservice teams. Throughout the chapter, we'll touch on and explain how to mitigate some common pitfalls of microservice development.

Although you might not currently work as an engineering manager, a team lead, or a director, we think it's essential to understand how these dynamics—and the choices you and your organization make—impact the pace and quality of microservice development.

13.1 Building effective teams

Splitting engineers into independent teams is a natural outcome of organizational growth. Doing so is necessary to help an organization scale effectively, as limiting team size has several benefits:

- It ensures lines of communication remain manageable—figure 13.1 illustrates how these grow—which aids team dynamism and collaboration while easing conflict resolution. Many heuristics exist for "right size," such as Jeff Bezos' two-pizza rule or Michael Lopp's 7 +/– 3 formula.
- It clearly delineates responsibility and accountability while encouraging independence and agility.

Small, independent teams can typically move faster than large teams. They also gel faster and gain effectiveness more quickly. Contrastingly, distinct engineering teams can also cause new problems:

- Teams can become culturally isolated, following and accepting different practices of quality or engineering values.
- Teams may need to invest extra effort to align on competing priorities when they collaborate with other teams.
- Separate teams may isolate specialist knowledge to the detriment of global understanding or effectiveness.
- Teams can duplicate work, leading to inefficiency.

Microservices can exacerbate these divisions. Different teams will likely no longer work on the same shared body of code. Teams will have different, competing priorities—and be less likely to have a global understanding of the application.

Building an effective engineering organization beyond a small group of people—and developing great software products—is a balancing act between these two tension points: autonomy and collaboration. If boundaries between teams overlap and ownership is unclear, tension can increase; conversely, independent teams still need to collaborate to deliver the whole application.

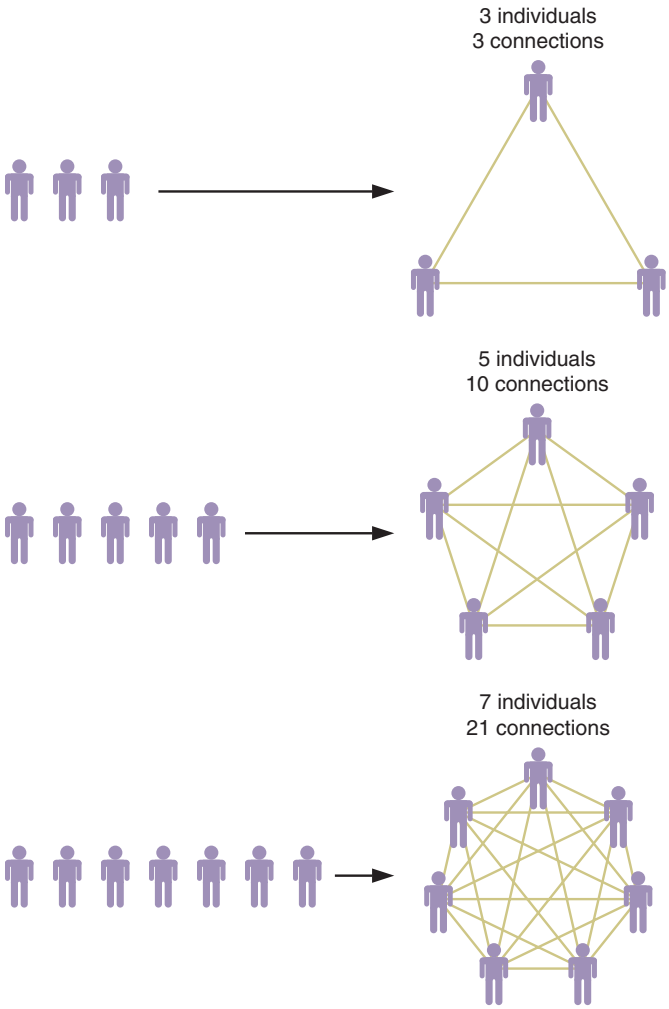


Figure 13.1 Lines of communication by group size

13.1.1 Conway's Law

It can be difficult to separate cause and effect in organizations that have successfully built microservice applications. Was the development of fine-grained services a logical outcome of their organizational structure and the behavior of their teams? Or did that structure and behavior arise from their experiences building fine-grained services?

The answer is: a bit of both! A long-running system isn't only an accumulation of features requested, designed, and built. It also reflects the preferences, opinions, and objectives of its builders and operators. This indicates that structure—what teams work on, what goals they set, and how they interact—will have a significant impact on how successfully you build and run a microservice application.

Conway's Law expresses this relationship between team and system:

> *…organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations…*

"Constrained" might suggest that these communication structures limit and constrict the effective development of a system. But the inverse of the rule is also true: you can take advantage of changes to team structure to produce a desired architecture. Team structure and microservice architecture are symbiotic: both can and should influence each other. This is a powerful technique, which we'll consider throughout this chapter.

13.1.2 Principles for effective teams

At a macro level, it's best to think of teams as units of achievement and communication. They're how stuff gets done and how people relate to each other within an organization. To realize benefits from microservices and adequately manage their complexity, your teams will need to adopt new working principles and practices, rather than using the same techniques they used to build monoliths.

There's no single right, perfect way to organize your teams. You'll always suffer from constraints: headcount, budget, personalities, skill sets, and priorities. Sometimes you can hire to fill a gap; sometimes you can't. The nature of your application and business domain will demand different approaches and skills. Your organization may be limited in its capacity to change. The best approach we've found is to guide the formation of teams using a small set of shared principles: ownership, autonomy, and end-to-end responsibility.

> **NOTE** Making a move to microservices—or indeed, any large-scale architectural change—in many enterprises will be challenging and disruptive. You won't be successful in isolation: you'll need to find sponsorship, build trust, and be prepared to argue your case—a lot! Richard Rodger's book, *The Tao of Microservices* (ISBN 9781617293146), goes into more (if slightly cynical) detail on navigating these institutional politics.

OWNERSHIP

Teams with a strong sense of ownership have high intrinsic motivation and exercise a considerable degree of responsibility for the area they own. Because microservice applications are typically long-lived, teams that have long-term ownership of an area support the evolution of that code while developing deep understanding and knowledge.

In a monolithic application, ownership is typically n:1. Many teams own one service: the monolith. This ownership is often split between different layers (such as frontend and backend) or between functional areas (such as orders and payments). In a microservice application, ownership is usually 1:n, meaning a team might own many services. Figure 13.2 depicts these ownership models.

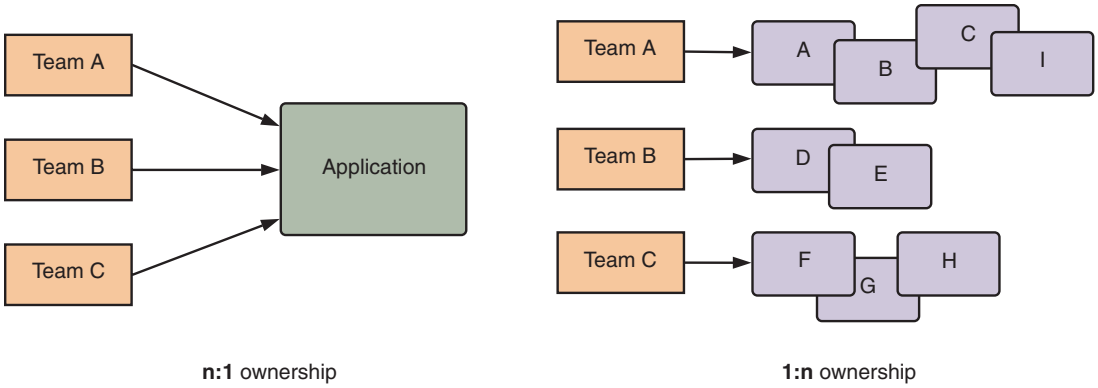


Figure 13.2 Team ownership in monolithic versus microservice codebases

> **WARNING** In the 1:n ownership model, it's usually bad practice for multiple teams to own one service. This can make accountability unclear and lead to conflict about technical choices and feature priority.

As an organization's codebase grows and the makeup of the engineering team fluctuates, the risk of code that no one knows—or code that no one can fix when it breaks—increases. Clear ownership helps you avoid this risk by placing natural, reasonable bounds on a team's knowledge while ensuring that ownership is the responsibility of a group, not individual developers.

AUTONOMY

It's not coincidental that these three principles reflect some of the principles of microservices themselves. Teams that can work autonomously—with limited dependencies on other teams—can work with less friction. These types of teams are highly aligned but loosely coupled.

Autonomy is important for scale. For an engineering manager, it's exhausting to control the work of multiple teams (not to mention, disempowering for the teams themselves); instead, you can empower teams to self-manage.

END-TO-END RESPONSIBILITY

A development team should own the full ideate-build-run loop of a product. With control over what's being built, a team can make rational, local priority decisions; experiment; and achieve a short cycle time between coming up with an idea and validating that idea with real code and users.

Most software spends significantly longer in operation than it ever spent being built. But many software engineers focus on the build stage, throwing code over the fence for a separate team to run it. This ultimately results in poorer quality and slower delivery. How software operates—how you observe its behavior in the real world—should feed back into improving that software (figure 13.3). Without responsibility for operation, this information is often lost. This tenet is also central to the DevOps movement.

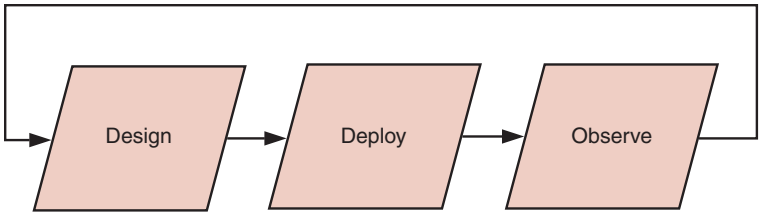


Figure 13.3 Software operation should continually inform future design and build.

End-to-end responsibility correlates closely with autonomy and ownership:

- The fewer cross-team dependencies in a team's path to production, the more likely it can control and optimize the pace of its delivery.
- A wider scope of ownership enables the team to reasonably and productively take on more responsibility for overall delivery.

13.2 Team models

In this section, we'll explore two approaches for structuring teams—by function or across function—and their benefits and disadvantages in developing microservices.

- In a *functional* approach, you group employees by specialization, with a functional reporting line, and assign them to time-bound projects. Most organizations fund projects for a specific scope and length of time. They measure success by the on-time delivery of that scope.
- Teams that you build *cross-functionally*—from a combination of different skillsets—typically are aligned to long-term product goals or aspirational missions, with freedom within that scope to prioritize projects and build features as needed to achieve those missions. You typically measure success through impact on business key performance indicators (KPIs) and outcomes.

The latter approach is a natural fit with microservices development.

13.2.1 Grouping by function

Traditionally, many engineering organizations have been grouped along horizontal, functional lines: backend engineers, frontend engineers, designers, testers, product (or project) management, and sysadmin/ops. Figure 13.4 illustrates this type of organization. In other cases, teams or individuals may move between any number of time-bounded projects.

This approach optimizes for expertise:

- It ensures that communication loops between specialists are short, so they share knowledge and solutions effectively and apply their skills consistently.
- Similar work and approaches are grouped together, providing clear career growth and skill development.

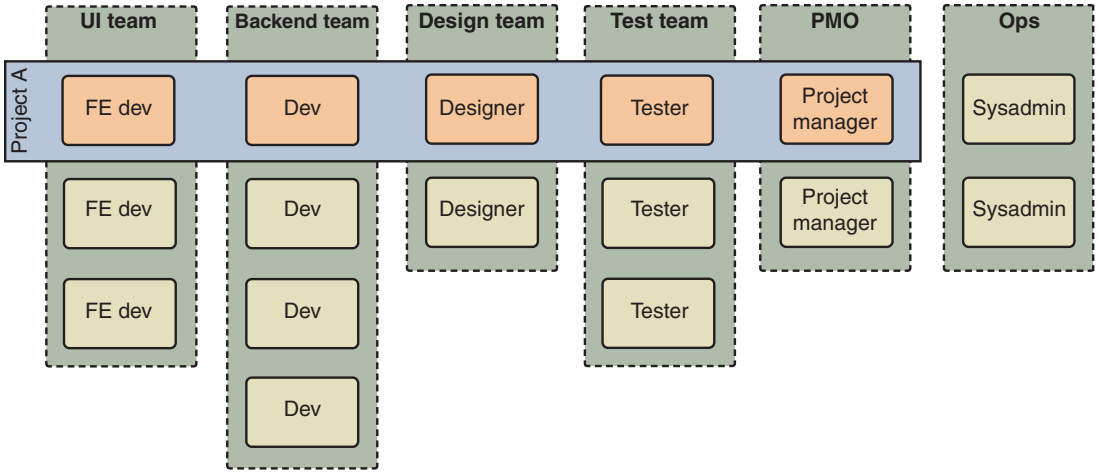


Figure 13.4 Grouping into teams by function and project

Now imagine you're building a new feature. This functional approach almost looks like a chain: the analyst team gathers requirements, engineers build backend services, testing windows are scheduled with the QA team, and sysadmins deploy the service. You can see that this approach involves a high coordination burden—delivering a feature relies on synchronization across several independent teams (figure 13.5).[1] This approach fails to meet our three principles for effective organization.

UNCLEAR OWNERSHIP

No team has clear ownership of business outcomes or value—they're only cogs in the value chain. As such, ownership of individual services is unclear: once a project is finished, who maintains the services that were built? How are these iterated on, improved, or discarded? Work allocation based on projects tends to shortchange long-term thinking and encourages ownership of code by individual engineers, which you want to avoid.

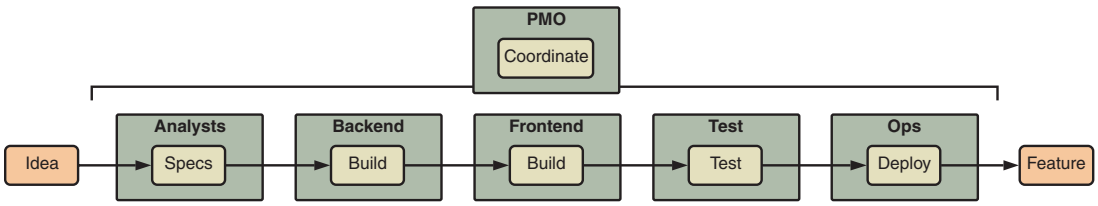


Figure 13.5 Functional teams contributing to the implementation of a feature

[1] And lo, the organization invented project managers!

LACK OF AUTONOMY

These teams are tightly coupled, not autonomous. Their priorities are set elsewhere, and every time work crosses a team boundary, the chance increases that a team will be blocked and development will be hampered. This leads to long lead times, rework, quality issues, and delays. Without alignment to the system architecture they're building, the team will be unable to evolve their application without being encumbered by other teams.

NO LONG-TERM RESPONSIBILITY

A project-oriented approach isn't conducive to long-term responsibility for the code produced or for the quality of a product. If the team is only together for a time-bound project, they might hand off their code to another department to run the application, so the original team won't be able to iterate on their original ideas and implementation. The organization will also fail to realize benefits from knowledge retention in the original team.

Lastly, a new team requires time to normalize productive working behaviors—the longer people work together, the better the team gels, and the more effective it becomes. A team that stays together longer will maintain a longer period of high performance.

> **TIP** There's also a risk that long-lived teams can become too comfortable or set in their ways. It's important to balance long-term bonding and bringing people with new perspectives and skills into the team.

RISK OF SILOS

Lastly, this approach also risks the formation of silos—teams diverge in goals and become incapable of effective, empathetic collaboration. Hopefully you've never worked someplace where the relationship between test and dev, or dev and ops, is almost adversarial, but it's been known to happen.

Ultimately, it's unlikely that a functional, project-oriented organization will deliver a microservice application without incurring significant friction and substantial cost.

13.2.2 Grouping across functions

By optimizing for expertise, the functional approach aims to eliminate duplicated work and skill-based inefficiencies, in turn reducing overall cost. But this can cause gridlock: increasing friction and reducing your speed in achieving organizational goals. This isn't great—your microservice architecture was meant to *increase* pace and *reduce* friction.

Let's look at an alternative. Instead of grouping by function, you can work cross-functionally. A cross-functional team is made up of people with different specialties and roles intended to achieve a specific business goal. You could call these teams market-driven: they might aim toward a specific, long-term mission; build a product; or connect directly with the needs of their end customer. Figure 13.6 depicts a typical cross-functional team.

> **NOTE** We won't cover team leadership or reporting lines in any detail in this book. A product owner, an engineering lead, a technical lead, a project manager, or a partnership between those roles might lead a cross-functional team. For example, at Onfido, a product manager—who focuses on *what* the team should do—and an engineering lead—who focuses on *how* to achieve it—lead our teams in partnership.

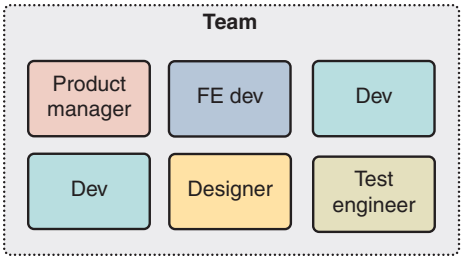


Figure 13.6 A typical cross-functional development team

Compared to the functional approach, a cross-functional team can be more closely aligned with the end goal of the team's activity. The multidisciplinary nature of the team is conducive to ownership. By taking on end-to-end responsibility for specification, deployment, and operation, the team can work autonomously to deliver features. The team gains clear accountability by taking on a mission that has a meaningful impact on the business's success. Day-to-day partnership between different specialists eliminates silos, as team members share ownership for the ultimate product of the team's work.

Designing these teams to be long-lived (for example, at least six months) is also beneficial. A long-lived team builds rapport, which increases their effectiveness, and shared knowledge, which increases their ability to optimize and improve the system under development. They also take long-term responsibility for the operation of the microservice application, rather than handing it off to another team.

The cross-functional, end-to-end approach to structuring teams is advantageous to microservice development:

- Aligning teams with business value will be reflected in the application developed; the teams will build services that explicitly implement business capabilities.
- Individual services will have clear ownership.
- Service architecture will reflect low coupling and high cohesiveness of teams.
- Functional specialists in different teams can collaborate informally to develop shared practices and ways of working.

This approach is common in modern web enterprises and is often cited as a reason for their success. For example, Amazon's CTO described the company's approach to architecture in 2006:

> *In the fine grained services approach that we use at Amazon, services do not only represent a software structure but also the organizational structure. The services have a strong ownership model, which combined with the small team size is intended to make it very easy to innovate. In some sense you can see these services as small startups within the walls of a bigger company. Each of these services require a strong focus on who their customers are, regardless whether they are externally or internally.*
>
> —*Werner Vogels*

Perhaps most importantly, a well-formed cross-functional team will be faster at delivering features than a group of functional teams, as lines of communication are shorter, coordination is local, and team members are aligned. The cross-functional approach prioritizes pace—but not at the expense of quality!

13.2.3 Setting team boundaries

A cross-functional team should have a mission. A mission is inspirational: it gives the team something to strive toward but also sets the boundaries of a team's responsibilities. Determining what a team is (and isn't) responsible for encourages autonomy and ownership while helping other teams align with each other. A mission is usually a business problem; for example, a growth team might aim to maximize recurring spend by customers, whereas a security team might aim to protect its codebase and data from known and novel threats. Based on this mission, each team prioritizes its own roadmap in collaboration with relevant partners within the business. Cross-cutting initiatives are driven by product or technical leadership.

> **NOTE** This type of team organization is also known as *product mode*—which doesn't mean each team is working on a self-contained product. Teams might own different vertical slices or different horizontal components of the same product. A particular component might be technically complex enough to demand a dedicated team.[2]

If your company offers a range of small products—that a team of 7 +/– 3 can productively work on—each team can be responsible for one product (figure 13.7). This isn't the case in many companies such as those that offer a large, complex product to market, requiring the effort of multiple teams.

For larger scale scenarios, bounded contexts—covered in chapter 4—are an effective starting point for setting loose boundaries for different teams in an organization. They also have the benefit of creating teams that map closely to business teams within the enterprise; for example, a warehouse product team will interact closely with warehouse operations.[3] Figure 13.8 illustrates a possible model for teams within SimpleBank.

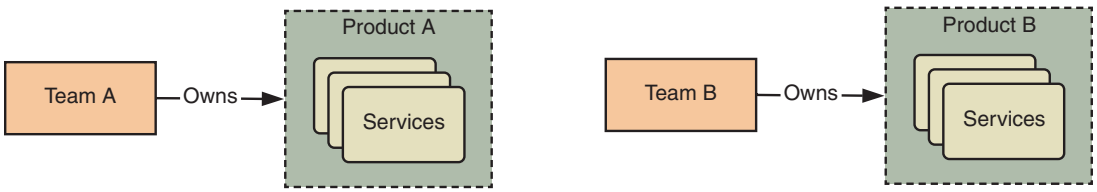


Figure 13.7 A team-per-product model

[2] A recent *ThoughtWorks* article describes these product-mode teams: Sriram Narayan, "Products Over Projects," February 20, 2018, http://mng.bz/r0v4.

[3] Be careful about how you approach this: the organizational structure itself might be suboptimal!

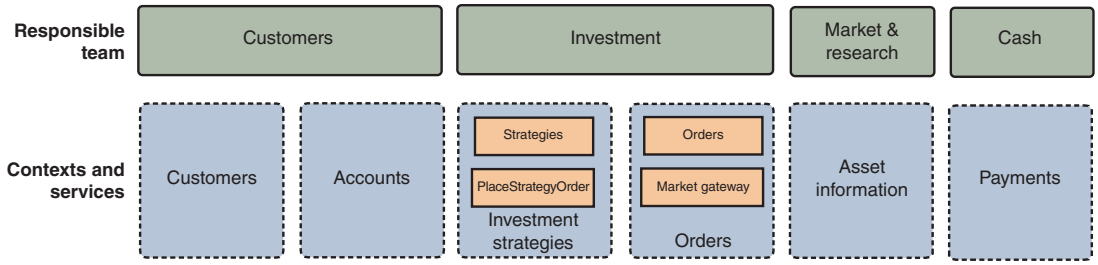


Figure 13.8 A possible model of service and capability ownership by different engineering teams for SimpleBank

Forming teams that own services in specific bounded contexts makes use of the inverse version of Conway's Law: if systems reflect the organizational structure that produces them, then you can attain a desirable system architecture by first shaping the structure and responsibilities of your organization.

As with services themselves, the right boundaries between teams may not always be obvious. We keep two general rules in mind:

- *Watch the team size.* If it approaches or surpasses nine people, it's likely that a team is doing too much or beginning to suffer from communication overhead.
- *Consider coherence.* Are the activities the team does cohesive and closely related? If not, a natural split may exist within the team between different groups of coherent work.

13.2.4 Infrastructure, platform, and product

Although we've advocated strongly for end-to-end ownership, it isn't always practical. For example, the underlying infrastructure—or microservice platform—of a large company is typically complex and requires a joined-up roadmap and dedicated effort, rather than loose collaboration between DevOps specialists spread across distinct teams.

As we outlined earlier in the book, building a microservice platform—deployment processes, chassis, tooling, and monitoring—is vital to sustainably and rapidly building a great microservice application. When you first start working with microservices, the team building the application will usually own the task of building the platform too (figure 13.9). Over time, this platform will need to serve the needs of multiple teams, at which stage you might establish a platform team (figure 13.10).

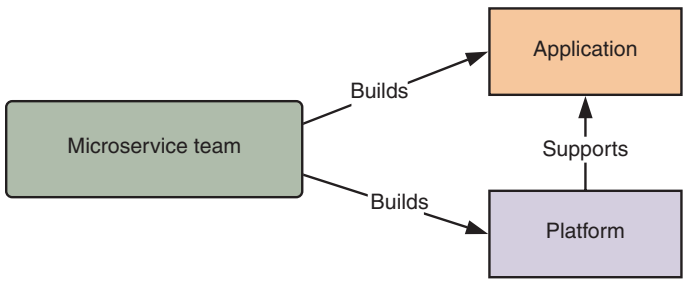


Figure 13.9 Early on, one team builds both the microservice application and the supporting platform.

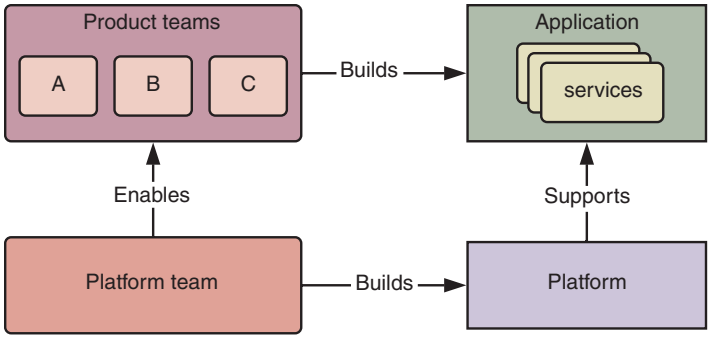


Figure 13.10 Establishing a platform team

Depending on the needs of your company and your technical choices, you might split this platform team further (figure 13.11) to distinguish core infrastructural concerns (such as cloud management and security) from specific microservice platform concerns (such as deployment and cluster operation). This is especially common in companies that operate their own infrastructure, rather than using a cloud provider.

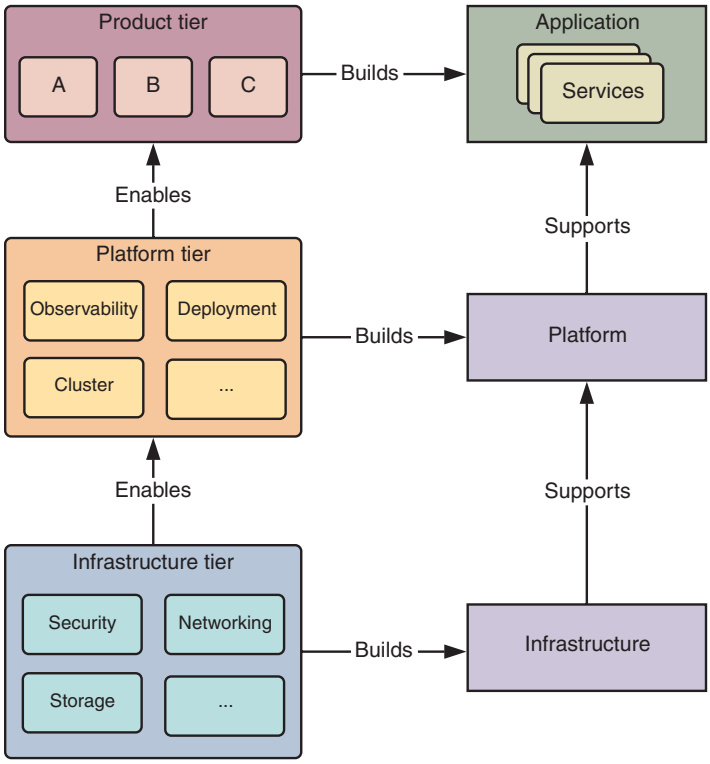


Figure 13.11 Establishing an infrastructure team as one tier in a three-tier model

In an even larger engineering organization, these tiers might be separated further; for example, different platform teams might focus on deployment tools, observability, or inter-service communication. This is also illustrated in figure 13.11.

The three-tier model shown in the figure provides economies of scale and specialization. This isn't a service relationship, where teams log tickets to each other. Instead, the output of each tier is a "product" that enables teams in the layer above to be more effective and productive.

13.2.5 Who's on-call?

The DevOps movement has been a strong influence on microservice approaches. A DevOps mentality—breaking down the barriers between build and runtime—is vital for doing microservices well, as deploying and operating multiple applications increases the cost and complexity of operational work. This movement encourages a "you build it, you run it" mindset; a team that takes responsibility for the operational lifetime of their services will build a better, more stable and more reliable application. This includes being on-call—ready to answer alerts—for your production services.

> **TIP** Chapter 11 covers best practices for triggering useful and actionable alerts from microservices.

For example, in the three-tier model:

- Engineering teams would be on-call for alerts from their own services.
- Platform and infrastructure teams would be on-call for issues in underlying infrastructure or shared services, such as deployment.
- An escalation path would exist between those two teams to support investigation.

This on-call model is illustrated in figure 13.12.

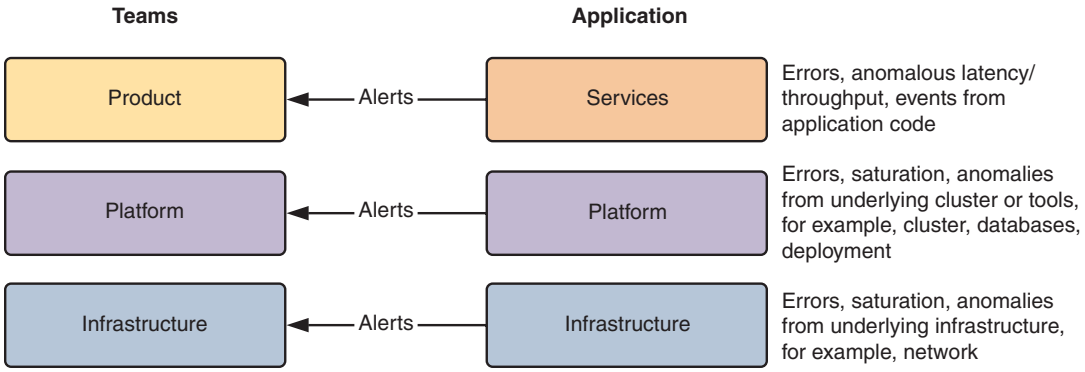


Figure 13.12 On-call model in a three-tier microservice team structure

Of the many changes that microservices bring, this may the most difficult to roll out: engineers are likely to resist being on-call, even for their own code. A successful on-call rotation should be

- *Inclusive*—Everyone who can do it, should do it, including VPs and directors.
- *Fair*—On-call work should be remunerated in addition to normal working hours.
- *Sustainable*—Enough engineers should be in a rotation to avoid burnout and avoid disruption to work-life balance or day-to-work in the office.
- *Reflective*—Your team should constantly review alerts and pages to ensure only alerts that matter wake someone up.

In this model, we split alerts across teams, because running software at scale is complex. Operational effort might be beyond the scope or knowledge of engineers within any one team. Many operational tasks—such as operating an Elasticsearch cluster, deploying a Kafka instance, or tuning a database—require specific expertise that would be unreasonable to expect product engineers to gain uniformly. Operational work also runs at a cadence different from the pace of product delivery.

> **WARNING** Historically, infrastructure operations teams have been responsible for running applications in production: keeping them stable and waking up when they break. This leads to tension: operations teams resent developers throwing unstable applications over the wall, whereas developers curse the lack of engineering skills in the operations team. This separate dev and ops model puts the onus for fixing production issues on the wrong team. Instead, if developers are responsible for how their code operates, they'll be better able to fix incidents and optimize that code in the long term.

The right choice for an on-call model that balances responsibility and expertise will depend on the types of applications you build, the throughput of those applications, and the underlying architecture you choose. If you're interested in learning more, *Increment* recently published an in-depth review (https://increment.com/on-call/who-owns-on-call/) of on-call approaches used at Google, PagerDuty, Airbnb, and other organizations.

13.2.6 Sharing knowledge

Although autonomous teams increase development pace, they have two downsides:

- Different teams may solve the same problem multiple times in different ways.
- Team members will have less engagement with their specialist peers on other teams.
- Team members may make local decisions without considering the global context or the needs of the wider organization.

You can mitigate these issues. We've had success applying Spotify's model of chapters and guilds.[4] These are communities of practice:

- Chapters group people by functional specialties, for example, mobile development.[5]
- A guild shares practice around a cross-cutting theme, for example, performance, security.

Figure 13.13 depicts this model.

Comparably, some organizations use matrix management to establish a formal identity for functional units. This adds a line of management responsibility (head of QA, head of design…) for functions, at the cost of building a more complicated management structure.

> **TIP** Most engineers have been taught to follow the DRY tenet—don't repeat yourself. Within a service, this is still important—there's no point in writing the same code twice! Across multiple services, this is much less of an imperative because writing shared code that's truly reusable is a costly endeavor, as is coordinating the rollout of that code across multiple consumers. A degree of duplication is acceptable if it means you can deliver features more rapidly.

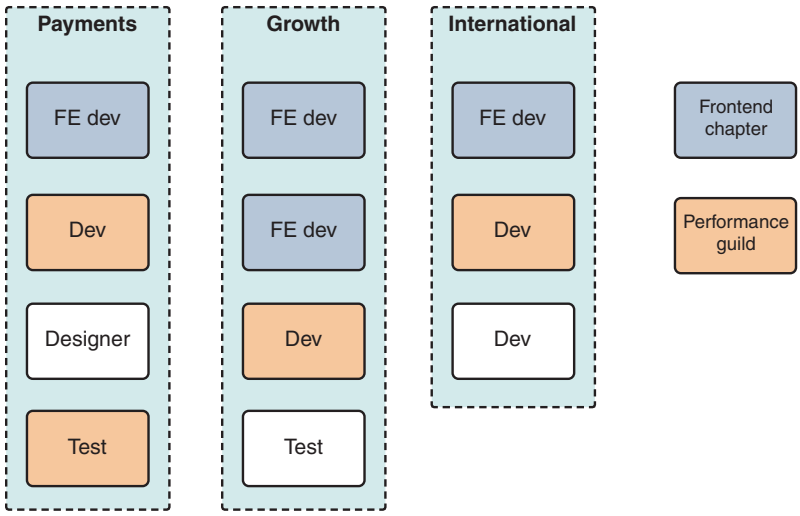


Figure 13.13 The chapters, guilds, and teams model

[4] See Henrik Kniberg, "Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds," *Crisp's Blog*, November 14, 2012, http://mng.bz/94Lv.

[5] In larger organizations, a chapter may group by functional specialty within an engineering division. (Spotify calls this a tribe.)

Either approach works well to disseminate knowledge and develop shared working practices. This helps to prevent the isolation that can arise in highly autonomous teams, ensuring teams remain aligned technically and culturally. Cross-pollination of ideas, solutions, and techniques also supports people moving between teams and reduces organization-level bus factor risks.

It's also important to strike a balance between team lifetime and team fluidity. In the long run, regularly rotating engineers between teams helps to share knowledge and skills and is a good complement for the chapter and guild model.

13.3 Recommended practices for microservice teams

The scale of change in a microservice application can be tremendous. It can be difficult to keep up! It's unreasonable to expect any engineer to have a deep understanding of all services and how they interact, especially because the topography of those services may change without warning. Likewise, grouping people into independent teams can be detrimental to forming a global perspective. These factors lead to some interesting cultural implications:

- Engineers will design solutions that are locally optimal—good for them or their team—but not always right for the wider engineering organization or company.
- It's possible to build around problems rather than fixing them, or to deploy new services instead of correcting issues with existing services.
- Practices on teams might become highly local, making it difficult for engineers to move between teams.
- It's challenging for architects or engineering leads to gain visibility and make effective decisions across the entire application.

Good engineering practices can help you avoid these problems. In this section, we'll walk through some of the practices that your teams should follow when building and maintaining services.

13.3.1 Drivers of change in microservices

Take a moment and consider the type of build items you might work on day to day. If you're on a product team, the items in your backlog are primarily functional additions or changes. You want to launch a new feature; support a new request from a customer; enter a new market; and so on. As such, you build and change microservices in response to these new functional requirements. And, thankfully, microservices are intended to ensure your application is flexible in the face of change.

But functional requirements—changes from your business domain—aren't the only driver of change in services. Each microservice will change for many reasons (figure 13.14):

- Underlying frameworks and dependencies (such as Rails, Spring, or Django) may require upgrades for performance, security, or new features.
- The service may no longer be fit for the purpose—for example, hitting natural scalability limits—and may require change or replacement.
- You discover defects in the service or the service's dependencies.

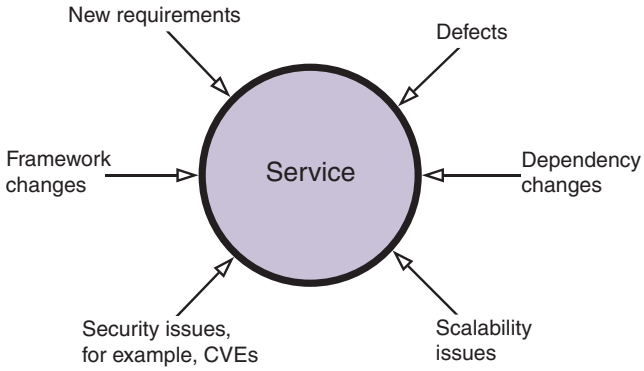


Figure 13.14 Drivers of change to a microservice

All this change increases complexity. For example, instead of tracking security vulnerabilities against a single monolithic application, you need to ensure your tooling supports static analysis and alerting across several applications (and likely several distinct programming languages and frameworks). Every new service generates more work.

Alternatively, some microservice practitioners have advocated *immutable* services—once a service is considered mature, put it under feature freeze, and add new services if change is required. There's a tricky cost-benefit decision here: is the risk of breaking a service through modification more than the cost of building a new service? It's a difficult question to answer definitively and will depend on both your business context and appetite for risk.

13.3.2 The role of architecture

Microservice applications evolve over time: teams build new services; decommission existing services; refactor existing functionality; and so on. The faster pace and more fluid environment that microservices enable change the role of architects and technical leads.

Architects have an important role to play in guiding the scope and overall shape of an application. But they need to perform that role without becoming a bottleneck. A prescriptive and centralized approach to major technical decisions doesn't always work well in a microservice application:

- The microservice approach and the team model we've outlined should empower local teams to make rapid, context-aware decisions without layers of approval.
- The fluidity of a microservice environment means that any overarching technical plan or desired model of the intended system will quickly pass its use-by date, as requirements change, services evolve, and the business itself matures.
- The volume of decisions increases with the number of services, which can overwhelm an architect and make them a bottleneck.

That doesn't mean that architecture isn't useful or necessary. An architect should have a global perspective and make sure the global needs of the application are met, guiding its evolution so that

- The application is aligned to the wider strategic goals of the organization.
- Technical choices within one team don't conflict with choices in another.
- Teams share a common set of technical values and expectations.
- Cross-cutting concerns—such as observability, deployment, and interservice communication—meet the needs of multiple teams.
- The whole application is flexible and malleable in the face of change.

The best starting point for architecture is to set *principles*. Principles are guidelines (or sometimes rules) that teams should follow to achieve higher level goals. They inform team practice. Figure 13.15 illustrates this model.

For example, if your product goal is to sell to privacy- and security-sensitive enterprises, you might set principles of compliance with recognized external standards, data portability, and clear tracking of personal information. If your goal is to enter a new market, you might mandate flexibility around regional requirements, design for multiple cloud regions, and out-of-the-box support for i18n (figure 13.16).

Principles are flexible. They can and should change to reflect the priorities of the business and the technical evolution of your application. For example, early development might prioritize validating product-market fit, whereas a more mature application might require a focus on performance and scalability.

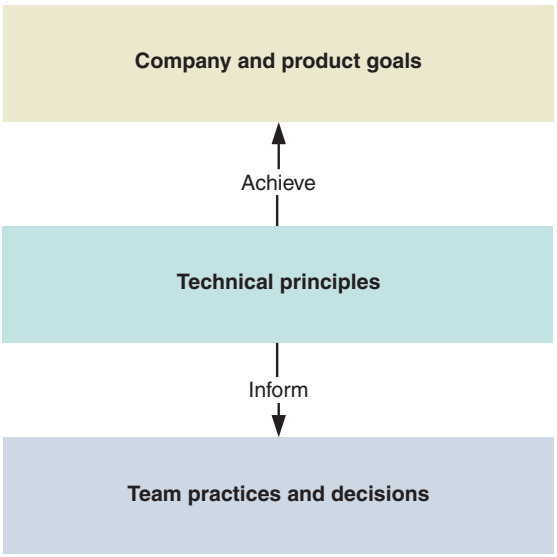


Figure 13.15 An architectural approach based on technical principles

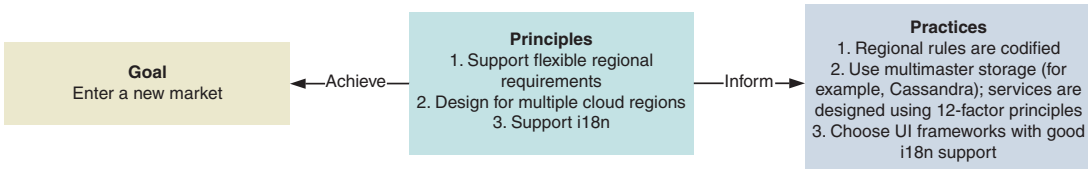


Figure 13.16 Principles and practices to support entering a new market

Several day-to-day practices support this evolutionary approach to architecture, such as design review, an inner-source model, and living documentation. We'll discuss them over the next few sections.

13.3.3 Homogeneity versus technical flexibility

A tricky decision you'll face is which languages to use to write microservices. Although microservices provide for technical freedom, using a wide range of languages and frameworks can increase risk:

- Bus factor and key person dependencies may increase because of limited shared knowledge, making it difficult to maintain and support services.
- Services in new languages may not meet production readiness standards.

In practice, you'll always encounter scenarios where you need to pick a different language, such as specialist features or performance needs. For example, Java would be ill-suited to writing systems infrastructure, just as Ruby doesn't have the depth of scientific and machine learning libraries available to Python. In these scenarios, it's important to share the development of services in new languages/frameworks across many team members to reduce bus factor risk: rotate team members, have a pair program, write documentation, and mentor new engineers.

Picking a single primary language, or a small set, allows you to better optimize practices and approach for that language. The creation of service templates, chassis, and/or exemplars will naturally ease development in your favored language, leading more developers to write services using it. Lowering friction this way creates a virtuous circle. Even if you don't explicitly choose a favored language, this can happen organically (although it'll take longer).

> **TIP** Microservices should be replaceable. If needed, you should be able to rewrite any service in a more favorable programming language.

13.3.4 Open source model

Applying open source principles to microservice code can help to alleviate contention and technical isolation while improving knowledge sharing. As we mentioned earlier, each team in a microservice organization typically owns multiple services. But each service you run in production must have a clear owner: a team that takes long-term responsibility for that service's functionality, maintenance, and stability.

That doesn't mean those people must be the only contributors to that service. Other teams might need to tweak functionality to meet their needs or fix defects. If these changes all needed the same group of people to make them, those people would be at the mercy of their own priorities, which in turn would slow other teams down.

Instead, an *inner-source* model—open source within your organization—balances ownership and visibility:

- Source code should be available internally for any service.[6]
- Any engineer can submit pull requests to any service, as long as the service owner reviews them.

This model (figure 13.17) closely resembles most open source projects, where a core group of committers make most commits and key decisions, and others can submit changes for approval. Imagine an engineer on Team A needs to make a change to a service that Team B owns. They could argue for the priority of their change against everything else on Team A's backlog, or they could pull the code, make the change themselves, and submit a pull request for Team B to review.

This approach has three benefits:

- Alleviates contention and priority negotiation between teams
- Reduces the sense of technical isolation and possessiveness that can develop when service work is limited to a small number of people within an organization
- Shares knowledge within an organization by helping engineers understand other teams' services and better understand the needs of their internal consumers

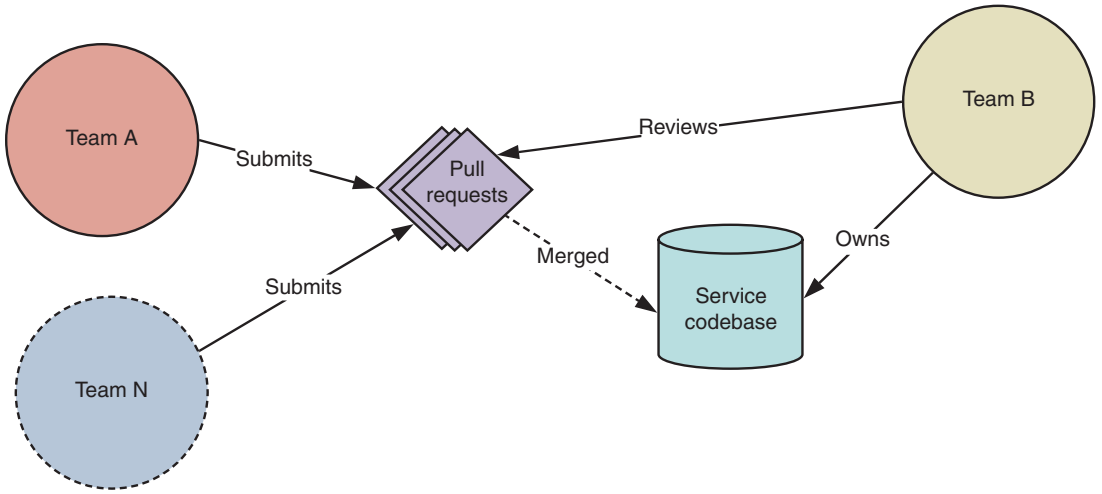


Figure 13.17 Applying an open source model to service development

[6] In some organizations, reasonable exceptions to this rule may apply, such as when code is highly sensitive.

NOTE Contributing across multiple services is significantly easier when those services follow common architectural and deployment conventions—like the ones we've discussed throughout this book!

13.3.5 Design review

Each new microservice is a blank slate. Each service will have different performance characteristics; might be written in a different language; might require new infrastructure; and so on. A new feature might be possible to write in several ways: as a new service, as many services, or within an existing service. This freedom is terrific, but a lack of oversight can result in

- *Inconsistency*—For example, a service might not log requests consistently, hampering common operational tasks, such as investigating defects.
- *Suboptimal design decisions*—You might build multiple services, when a single service would be more maintainable and perform better.

A few methods can help you get around this issue. In chapter 7, we discussed using service chassis and service exemplars as best practice starting points. But that's only a partial solution.

In our own company—comparable to practices at Uber and Criteo—we follow a design review process. For any new service or substantial new feature, the engineer responsible produces a design document (we call this an RFC, or request for comments) and asks for feedback from a group of reviewers, both in and outside of their own team. Table 13.1 outlines the sections in a typical design review document.

Table 13.1 Sections in a design review document for a new microservice

Section	Purpose
Problem & Context	What technical and/or business problem does this feature solve? Why are we doing this?
Solution	How are you intending to solve this problem?
Dependencies & Integration	How does it interact with existing or planned services/functionality/components?
Interfaces	What operations might this service expose?
Scale & Performance	How does the feature scale? What are the rough operational costs?
Reliability	What level of reliability are you aiming for?
Redundancy	Backups, restores, deployment, fallbacks
Monitoring & Instrumentation	How will you understand this service's behavior?
Failure Scenarios	How will you mitigate the impact of possible failures?
Security	Threat model, protection of data, and so on
Rollout	How will you launch this feature?
Risks & Open Questions	What risks have you identified? What don't you know?

This process catches suboptimal design decisions early in the development cycle. Although writing a document may seem like extra effort, having a semiformal prompt to consider service design tends to result in faster overall development, as the team brings to light the full range of considerations and tradeoffs before committing to an implementation direction.

13.3.6 Living documentation

As we've mentioned, it's difficult to keep a microservice architecture in your head. The scale of a microservice application demands that your team invest time in documentation. For each service, we recommend a four-layered approach: overviews, contracts, runbooks, and metadata. Table 13.2 details these four layers.

Table 13.2 Recommended minimum layers for documenting microservices

Type	Summary
Overview	An overview of the service's purpose, intended usage and overall architecture. Service overviews should be an entry point for team members and service users.
Contract	A service contract should describe the API that a service provides. Depending on transport mechanism, this can be machine-readable, for example, using Swagger (HTTP APIs) or protocol buffers (gRPC).
Runbooks	Documented runbooks for production support detailing common operational and failure scenarios
Metadata	Facts about a service's technical implementation, such as the programming language, major framework versions, links to supporting tools, and deployment URLs

This documentation should be discoverable in a *registry*—a single website where details for all services are available. Good microservice documentation serves many purposes:

- Developers can discover the capabilities of existing services, such as the contracts they expose. This speeds up development and may reduce wasted or duplicated work.
- On-call staff can use runbooks and service overviews to diagnose issues in production, as different services will vary operationally.
- Teams can use metadata to track service infrastructure and answer questions, for example, "How many services are running Ruby 2.2?"

Many tools exist for writing project documentation, such as MkDocs (www.mkdocs.org). You could combine them with service metadata approaches, as described in table 13.2, to build a microservice registry.

> **TIP** Documentation is notoriously hard to keep up to date, even for a single application. As much as feasible, you should aim to autogenerate documentation from application state. For example, you can generate contract documentation from Swagger YML files using the `swagger-ui` library.

13.3.7 Answering questions about your application

As a service owner or an architect, you'll often want to get an overarching view of the state of your application to answer questions like

- How many services are written in each language?
- Which services have security vulnerabilities or outdated dependencies?
- What upstream and downstream collaborators use Service A?
- Which services are production-critical? Which are spikes and experiments, or less important to critical application paths?

At the time of this writing, few tools exist in the wild that combine this information to make it readily available. When it's available, it's typically spread across multiple locations:

- Language and framework choices require code analysis or repository tagging.
- Dependency management tools (for example, Dependabot) scan for outdated libraries.
- Continuous integration jobs run arbitrary static analysis tasks.
- Network metrics and code instrumentation surface relationships between services.

Similar information might be kept in spreadsheets or architectural diagrams, which, sadly, are often out of date.

A recent presentation from John Arthorne at Shopify[7] proposed embedding a file, service.yml, in each code repository and using that as a source of service metadata. This is a promising idea, but at the time of this writing, you'll need to roll your own.

13.4 Further reading

Forming, growing, and improving engineering teams is a broad topic, and in this chapter we've only scratched the surface. If you're interested in learning more, we recommend the following books as good places to start:

- *Elastic Leadership*, by Roy Osherove (ISBN 9781617293085)
- *Managing Humans*, by Michael Lopp (ISBN 9781430243144)
- *Managing the Unmanageable*, by Mickey W. Mantle and Ron Lichty (ISBN 9780321822031)
- *PeopleWare*, by Tom DeMarco and Timothy Lister (ISBN 9780932633439)

We've covered a lot of ground in this chapter. Choosing a microservice engineering approach is great for getting things done and empowering engineers, but changing your technical foundation is only half the battle. Any system is deeply intertwined with

[7] See John Arthorne, "Tracking Service Infrastructure at Scale," SRECon San Francisco, March 13, 2017, http://mng.bz/Z6d0.

the people building it—successful, sustainable development requires close collaboration, communication, and rigorous and responsible engineering practices.

In the end, people deliver software. Getting the best product out requires getting the best out of your team.

Summary

- Building great software is as much about effective communication, alignment, and collaboration as implementation choices.
- Application architecture and team structure have a symbiotic relationship. You can use the latter to change the former.
- If you want teams to be effective, you should organize them to maximize autonomy, ownership, and end-to-end responsibility.
- Cross-functional teams are faster and more efficient at delivering microservices than a traditional, functional approach.
- A larger engineering organization should develop a tiered model of infrastructure, platform, and product teams. Teams in lower tiers enable higher tier teams to work more effectively.
- Communities of practice, such as guilds and chapters, can share functional knowledge.
- A microservice application is difficult to fit in your head, which leads to challenges for global decision making and on-call engineers.
- Architects should guide and shape the evolution of an application, not dictate direction and outcomes.
- Inner-source models improve cross-team collaboration, weaken feelings of possessiveness, and reduce bus factor risks.
- Design reviews improve the quality, accessibility, and consistency of microservices.
- Microservice documentation should include overviews, runbooks, metadata, and service contracts.

appendix
Installing Jenkins on Minikube

This appendix covers

- Running Jenkins on Minikube
- A short introduction to Helm

This appendix will walk you through the process of running Jenkins on your local Minikube cluster, which we use in the examples in chapter 10.

Running Jenkins on Kubernetes

You can run Jenkins as another service on the local Kubernetes cluster—Minikube—you set up in chapter 9. If you're working from scratch, follow the installation instructions on GitHub to get Minikube running (https://github.com/kubernetes/minikube). Once you have it installed, bring up the cluster by running `minikube start` at your terminal.

The Jenkins application consists of a master node and, optionally, any number of agent nodes. Running a Jenkins job executes scripts (such as `make`) across agent nodes to perform deployment activities. A *job* operates within a *workspace*—a local copy of your code repository. Figure A.1 illustrates this architecture.

You'll use Helm to install an "official" Kubernetes-ready configuration of Jenkins on your Minikube cluster.

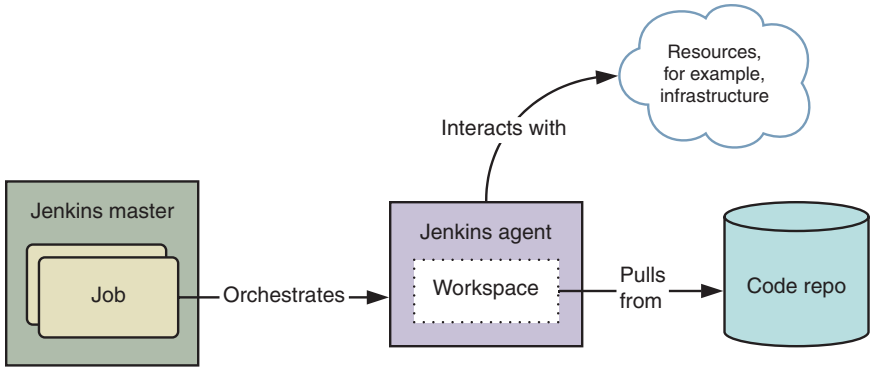


Figure A.1 A high-level Jenkins architecture

Setting up Helm

You can think of Helm (https://helm.sh/) as a package manager for Kubernetes. Helm's package format is a *chart*, which defines a set of Kubernetes object templates. Community-developed charts, like the one you'll use for Jenkins, are stored on Github (https://github.com/helm/charts).

Helm consists of two components:

- A client, which you'll use to interact with Helm charts
- A server-side application (also known as Tiller), which performs installation of charts

This is illustrated in figure A.2.

Installation instructions for Helm are on Github (https://github.com/helm/helm). Follow them to get the Helm client running on your machine. Once you've installed Helm, you'll need to set up Tiller on Minikube. Run `helm init` on the command line to set up this component.

Create a namespace and a volume

Before you install the Jenkins chart, you need to create two things:

- A new namespace to logically segregate your Jenkins objects within the cluster
- A persistent volume to store Jenkins configuration, even if you restart Minikube

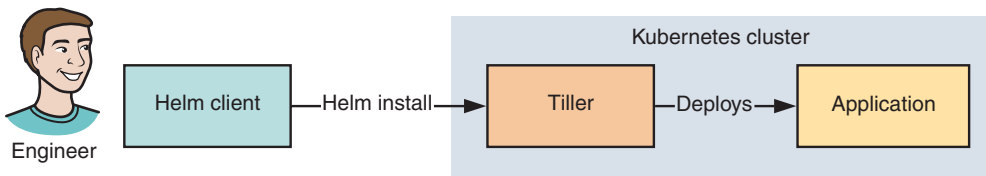


Figure A.2 Components of Helm, a package manager for Kubernetes

TIP You can find all the templates we use in this chapter in the book's code repo on Github: https://github.com/morganjbruce/microservices-in-action.

To create the namespace, apply the following template to your Minikube cluster, using `kubectl apply -f <file_name>`.

Listing A.1 jenkins-namespace.yml

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: jenkins
```

And do the same again for the persistent volume, as follows.

Listing A.2 jenkins-volume.yml

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: jenkins-volume
  namespace: jenkins
spec:
  storageClassName: jenkins-volume
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 10Gi
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: /data/jenkins/
```

Installing Jenkins

You'll install Jenkins with the community Helm chart. This chart is pretty complex; if you're interested, you can explore it on Github: https://github.com/helm/charts/tree/master/stable/jenkins.

First, create a values.yml file. Helm will interpolate the following code into the Jenkins chart to set appropriate defaults for running on Minikube.

Listing A.3 values.yml

```
Master:
  ServicePort: 8080
  ServiceType: NodePort
  NodePort: 32123
  ScriptApproval:
```

```
    - "method groovy.json.JsonSlurperClassic parseText java.lang.String"
    - "new groovy.json.JsonSlurperClassic"
    - "staticMethod org.codehaus.groovy.runtime.DefaultGroovyMethods
     leftShift java.util.Map java.util.Map"
    - "staticMethod org.codehaus.groovy.runtime.DefaultGroovyMethods split
     java.lang.String"
  InstallPlugins:
    - kubernetes:1.7.1
    - workflow-aggregator:2.5
    - workflow-job:2.21
    - credentials-binding:1.16
    - git:3.9.1
Agent:
  volumes:
    - type: HostPath
      hostPath: /var/run/docker.sock
      mountPath: /var/run/docker.sock

Persistence:
  Enabled: true
  StorageClass: jenkins-volume
  Size: 10Gi

NetworkPolicy:
  Enabled: false
  ApiVersion: extensions/v1beta1

rbac:
  install: true
  serviceAccountName: default
  apiVersion: v1beta1
  roleRef: cluster-admin
```

This default set of plugins will support running Jenkins Pipeline jobs.

The persistence settings refer to your persistent volume.

Now, to install Jenkins, run the following `helm` command:

```
helm install
  --name jenkins
  --namespace jenkins
  --values values.yml
  stable/jenkins
```

The chart to install

If successful, this will output a list of created resources that looks like figure A.3.

Give Jenkins a few minutes to start up. To access the server, you'll need a password. You can retrieve it using the following command:

```
printf $(kubectl get secret --namespace jenkins jenkins -o jsonpath="{.data.
     jenkins-admin-password}" | base64 --decode);echo
```

Then, navigate to the login page:

```
minikube --namespace=jenkins service jenkins
```

Log in with the username "admin" and the password you retrieved. Terrific—you've set up Jenkins!

Configuring RBAC

Minikube uses RBAC—role-based access control—by default, which requires an additional configuration step to ensure Jenkins can perform operations on the Kubernetes cluster.

```
NAME:   jenkins
LAST DEPLOYED: Thu Jun  7 17:50:53 2018
NAMESPACE: jenkins
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME     TYPE    DATA  AGE
jenkins  Opaque  2     0s

==> v1/ConfigMap
NAME           DATA  AGE
jenkins        5     0s
jenkins-tests  1     0s

==> v1/PersistentVolumeClaim
NAME     STATUS  VOLUME          CAPACITY  ACCESS MODES  STORAGECLASS    AGE
jenkins  Bound   jenkins-volume  10Gi      RWO           jenkins-volume  0s

==> v1/Service
NAME           TYPE       CLUSTER-IP     EXTERNAL-IP  PORT(S)         AGE
jenkins-agent  ClusterIP  10.99.195.109  <none>       50000/TCP       0s
jenkins        NodePort   10.110.150.27  <none>       8080:32123/TCP  0s

==> v1beta1/Deployment
NAME     DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
jenkins  1        1        1           0          0s

==> v1/Pod(related)
NAME                       READY  STATUS    RESTARTS  AGE
jenkins-69575dd96f-sfd59   0/1    Init:0/1  0         0s
```

Figure A.3 Kubernetes objects that the stable/Jenkins Helm chart installed.

To configure this appropriately on the Jenkins server:

1. Log in to the Jenkins dashboard.
2. Navigate to Credentials > System > Global Credentials > Add Credentials.
3. Add a Kubernetes Service Account credential, setting the value of the ID field to `jenkins`.
4. Save and navigate to Jenkins > Manage Jenkins > System.
5. Under the Kubernetes section, configure the credentials to those you created in step 3 (figure A.4) and click Save.

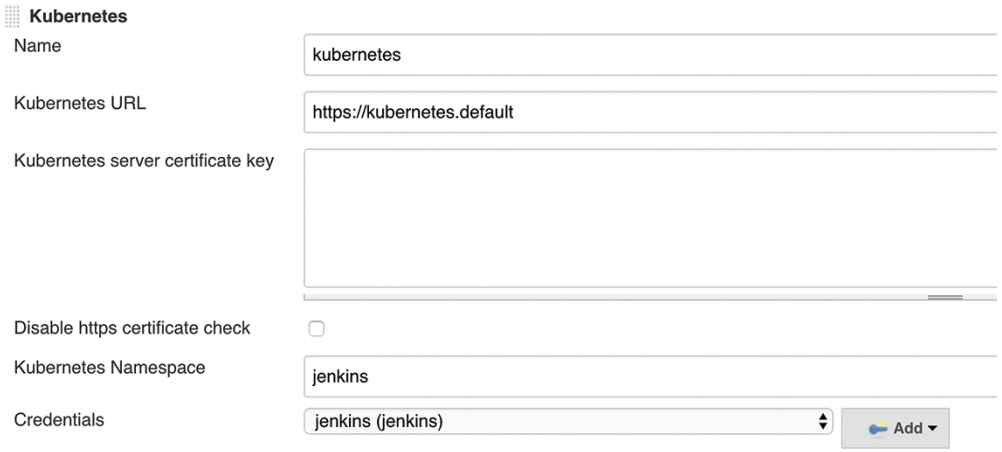


Figure A.4 Kubernetes cloud credentials

Testing it all works

You can run a simple build to make sure everything's working. First, log in to your new Jenkins dashboard and navigate to New Item in the left-hand column.

Create a new pipeline job named "test-job" per the configuration in figure A.5. Click OK to move to the next page and configure that job with the following script in the Pipeline Script field.

Listing A.4 Test pipeline script

```
podTemplate(label: 'build', containers: [
    containerTemplate(name: 'docker', image: 'docker', command: 'cat',
     ttyEnabled: true)
  ],
  volumes: [
    hostPathVolume(mountPath: '/var/run/docker.sock', hostPath: '/var/run/
     docker.sock'),
```

```
]
) {
  node('build') {
    container('docker') {
      sh 'docker version'
    }
  }
}
```

Click Save, then, on the following page, click Build Now. This will execute your job.

TIP The first build run may take some time!

The script you added will

1. Create a new pod, containing a Docker container
2. Execute the `docker version` command inside that container and output the results to the console

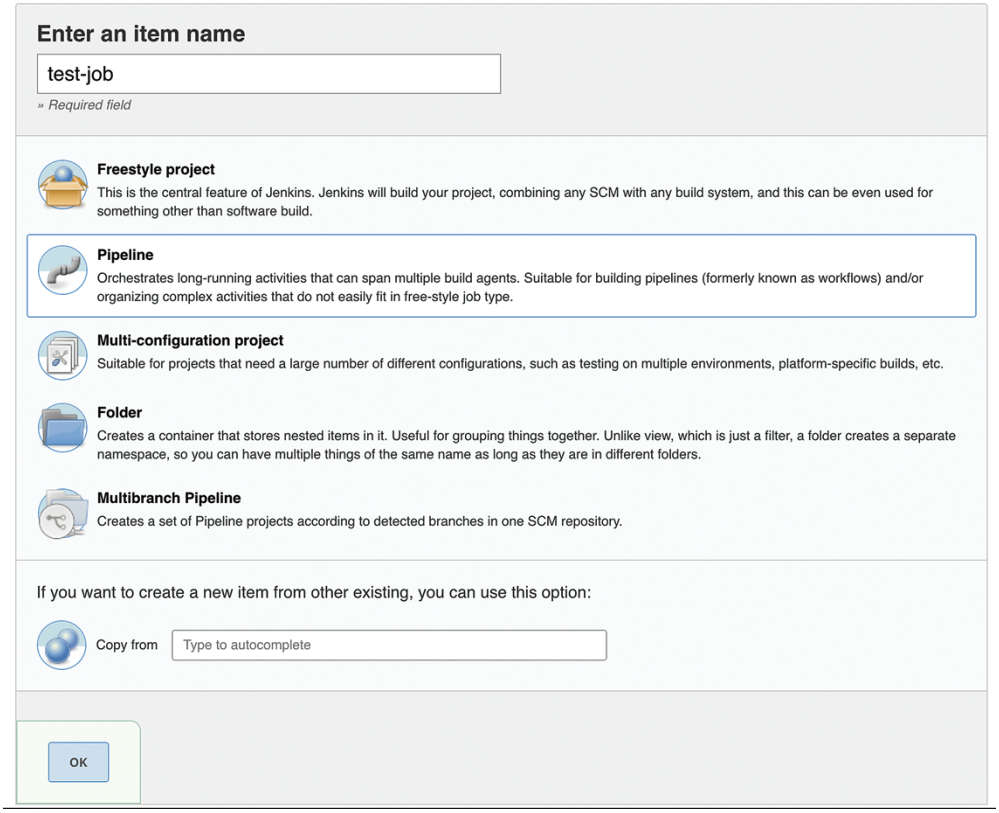


Figure A.5 New job page on Jenkins

Once the build job has completed, navigate to the build's console output (http://<insert Jenkins ip here>/job/test/1/console). You should see output similar to figure A.6, showing the output of the job script commands.

If your output looks like the figure, fantastic—everything's in working order! If not, your first point of call to diagnose any issues should be the Jenkins logs: http://<insert Jenkins ip here>/log/all.

```
Started by user admin
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] podTemplate
[Pipeline] {
[Pipeline] node
Still waiting to schedule task
jenkins-slave-83vg6-gvs6d is offline
Agent jenkins-slave-83vg6-gvs6d is provisioned from template Kubernetes Pod Template
Agent specification [Kubernetes Pod Template] (build):
* [docker] docker

Running on jenkins-slave-83vg6-gvs6d in /home/jenkins/workspace/test
[Pipeline] {
[Pipeline] container
[Pipeline] {
[Pipeline] sh
[test] Running shell script
+ docker version
Client:
 Version:      18.05.0-ce
 API version:  1.35 (downgraded from 1.37)
 Go version:   go1.9.2
 Git commit:   f150324
 Built:        Wed May  9 22:11:29 2018
```

Figure A.6 Console output from the test build job

index

Symbols

2PC (two-phase commit) protocol 107

A

abstraction 66
acbuild command-line tool 218
acceptance testing 257
accidental complexity 12
ACID transaction 106
actions 92–94
activity feeds 126
Advanced Message Queuing Protocol (AMQP) 63, 162
aggregating data 59
aggregation 59–60
alerts
  raising 291–293
    symptoms versus causes 292–293
    who needs to know 292
  setting up 287–291
alignment 9–10
ambiguity 99–103
  migration 100–103
  preparing for decomposition 100
  retirement 100–103
  starting with coarse-grained services 99–100
AMQP (Advanced Message Queuing Protocol) 63, 162
analytics 127
anemic services 30
API gateways 39, 68–69
Apollo 71
application boundaries 66–71
  API gateways 68–69
  BFF (backends for frontends) 69–70
  consumer-driven gateways 70–71
applications
  distributed 106–108
  observing 293–295
architecture
  of CQRS pattern 124–125
  of microservices 52–56
    architectural principles 54
    four tiers of microservice applications 55–56
    role of architect 54
  role of 341–343
artifacts 199–207
  configuring 206–207
  defining 200
  immutability of 201
  publishing 254–255
  standardizing 22–23
  types of 202–206
    containers 205–206
    operating system packages 203
    server images 203–204
asset information 81
asynchronous communication patterns 63–65
  job queues 63–64
  overview of 149
  publish-subscribe 64–65
asynchronous messages 62–63
autohealing 190
automated deployments 42–43
automated tests 257
automation
  of production environments 191
  overview of 9, 189
autonomy
  lack of 332
  overview of 8, 329
autoscaling 190, 194
availability 189
  automation 189
  calculating 130
  manual change management 189
  prioritizing 123
  small releases 189

B

backends for frontends. *See* BFF
balancing 177–180
BDD (behavior-driven development) 78
BEAM (Erlang virtual machine) 183
BFF (backends for frontends) 69–70

binary dependencies 200
blue-green deploys pattern 210
bootstrapping 164–165
bottlenecks 154
boundaries. *See* application boundaries
boundary layer 66
bounded contexts 67, 79
build pipelines
  configuring 247–251
  procedural versus declarative 263–264
bus factor 41
business capabilities, scoping by 78–87
  capabilities 78–79
  challenges and limitations 87
  creating investment strategies 79–86
  domain modeling 78–79
  nested contexts and services 86
busybox 241

C

cachetools library 144
caching 67, 143–144
canaries
  on GCE 211–213
  overview of 236–240
capacity 196
CAP theorem 108, 123
cascading failures 136–139
categories 301
chaos testing 152, 154–155
Chaos Toolkit 155
ChargeFailed event 110
chassis 160–163
  designing 165–180
    balancing 177–180
    limiting 177–180
    observability 171–177
    service discovery 167–171
  exploring features 180–182
  purpose of 163–165
    faster bootstrapping 164–165
    reduced risk 164
choreographed interactions 110
choreographed sagas 112–115
choreography 37–39, 94, 109–110
circuit breakers 146–148
Clair 220
clients 71–74
  frontend monoliths 71
  micro-frontends 72–74
clusters, deploying to 224–242
  components of master nodes 230
  components of worker nodes 231
  connecting multiple services 241–242
  deploying new versions 235–240
  designing pods 226–228
  health checks 233–235
  load balancing 228–230
  overview of 230–233
  rolling back 241
  running pods 228–232, 226–233
  state changes 231
coarse-grained services
  overview of 87
  starting with 99–100
codebases 329
code deployment 259
coherence 335
collaboration. *See* service collaboration
collaborators 37
command-query responsibility segregation pattern. *See* CQRS pattern
commands, separating queries from 123–125
commit phase, 2PC 108
communication 60–65
  as sources of failures 135
  asynchronous communication 149
  asynchronous communication patterns 63–65
    job queues 63–64
    publish-subscribe 64–65
  asynchronous messages 62–63
  by group size 327
  designing 139–149
    circuit breakers 146–148
    fallbacks 143–145
    retries 140–143
    timeouts 145–146
  event-based communication 108–110
    choreography 109–110
    events 109–110
  locating services 65
  overview of 21
  standardizing 156
  synchronous messages
    choosing transport 62
    drawbacks of 62
    when to use 61–62
  using service mesh 157
communication broker 63
compensating actions 113
components
  adding to Docker compose files 276
  of master nodes 230
  of worker nodes 231
compose files 276
configuration drift 202
configuring
  build pipelines 247–251
  Grafana 280–282
  Prometheus 280
  service artifacts 206–207
  StatsD exporter 277–279
ConnectionError exception 179
consistency patterns 118–119
constraints 103
Consul 65
consumer-driven gateways 70–71
containerizing services 215–224
  building images 218–220
  running containers 220–222
  storing images 223–224
  working with images 216–218
containers
  overview of 23, 205–206
  running 220–222

container schedulers 210
contexts, nested 86
continuous delivery 245
continuous deployment 245
contracts
  between services 16
  overview of 37
control 103
Conway's Law 327
coordinators 59
correlation IDs 121
CQRS (command-query responsibility segregation) pattern
  architecture of 124–125
  challenges 125–127
  optimistic updates 126
  overview of 63
  polling 127
  publish-subscribe 127
critical paths 60
cross-functionally 330
CRUD (create, read, update, delete) 30, 108
cyclic dependencies 114

D

DAG (direct acyclic graph) 314
dark launches 264
data
  aggregating 59
  storing copies of 122–123
  stubbed 145
data composition 121
data warehouses 127
DDD (domain-driven design) 79
deadlines 146
decomposition
  preparing for 100
  scaling through 6–7
degradation 143
delivery pipelines, building 242, 264–266
  dark launches 264
  deployment pipeline 244–246
  feature flags 23–24, 265–266
dependencies 135–136
deploying
  microservices 21–24
    implementing continuous delivery pipelines 23–24
    standardizing microservices deployment artifacts 22–23
  services 191–199, 187–213
    adding load balancers 196–198
    importance of 188–189
    provisioning virtual machines 192–193
    run multiple instances of service 194–196
    service startup 192
  to clusters 224–242
    components of master nodes 230
    components of worker nodes 231
    connecting multiple services 241–242
    deploying new versions 235–240
    designing pods 226–228
    health checks 233–235
    load balancing 228–230
    overview of 230–233
    rolling back 241
    running pods 228–232, 226–233
    state changes 231
  to production 259–261
  to staging 255–257
  to underlying hosts 207–210
    multiple scheduled services per host 209–210
    multiple static services per host 208–209
    single service to host 207
  without downtime 210–213
    canaries on GCE 211–213
    rolling deploys on GCE 211–213
deployment pipelines 57, 190, 244–246
deployment process 8, 43
deployments 235–236
  automated 42–43
  quality-controlled 42–43
deployment target 190
described feature 166
design reviews 345–346
development process 43
direct acyclic graph (DAG) 314
distributed applications 106–108
distributed computing 14, 130
distributed systems 16–17
divergence. *See* technical divergence
docker-compose, installing 140
docker-compose up command 180, 281
Docker platform 276
docker version command 355
documentation 346
domain-driven design (DDD) 79
domain modeling 78–79
domains. *See* modeling domains
domain-specific language (DSL) 217, 247
downtime 130
DRY (don't repeat yourself) 104, 183, 339
DSL (domain-specific language) 217, 247

E

edge capabilities 67
edge side includes (ESI) 73
elastic load balancer (ELB) 65
Elasticsearch engine 305
Elasticsearch, Logstash, and Kibana. *See* ELK-based solutions
ELB (elastic load balancer) 65
ELK (Elasticsearch, Logstash, and Kibana)-based solutions 304–306
  Elasticsearch 305
  Kibana 305
  Logstash 305
end-to-end responsibility 71, 329
engineering culture 26–27
enterprise service buses (ESBs) 6

environments, staging 258. *See also* production environments
Erlang virtual machine (BEAM) 183
ERROR level 301
error reporting 174–175
errors 273
ESBs (enterprise service buses) 6
ESI (edge side includes) 73
event backbone 63
event-based communication 108–110
  choreography 109–110
  events 109–110
events 109–110
event sourcing 119–120
eventual consistency 108
expectations, setting 104
explicit interfaces 104
exponential back-off strategy 142

F

failures
  cascading 136–139
  sources of 133–136
    communication 135
    dependencies 135–136
    hardware 134
    service practices 136
failure zones 195
fallbacks 143–145
  caching 143–144
  functional redundancy 144
  graceful degradation 143
  stubbed data 145
fault tolerance
  overview of 42
  validating 152–155
    chaos testing 154–155
    load testing 153–154
feature flags 265–266
features
  building 32–39
    identifying microservices by modeling domains 33–35
    service choreography 37–39
    service collaboration 36–37
  designing 75–104
  taking to production 40–45
    automated deployments 42–43
    quality-controlled deployments 42–43
    resilience 43
    transparency 43–45
feedback loops 139
flags 265–266
flat services 59
Flipper 265
Fluentd-based solutions 306
Fluentd daemon 304, 307
frameworks
  building reusable 159–183
  overview of 156
friction 13, 258
frontends
  BFF (backends for frontends) 69–70
  frontend monoliths 71
  micro-frontends 72–74
full staging environments 258
functional redundancy 144
functions
  grouping teams across 332–333
  grouping teams by 330–332
    lack of autonomy 332
    no long-term responsibility 332
    risk of silos 332
    unclear ownership 331

G

gateways
  API gateways 39, 68–69
  consumer-driven gateways 70–71
gauges 273
GCE (Google Kubernetes Engine)
  canaries on 211–213
  rolling deploys on 211–213
GCP (Google Cloud Platform) 191
GDPR (General Data Protection Regulation) 313
GKE (Google Kubernetes Engine). *See* GCE
Google Cloud Platform (GCP) 191
Google Compute Engine (GCE) 192
Google Kubernetes Engine (GKE) 225
graceful degradation 143
Grafana tool
  configuring 280–282
  monitoring systems with 275–291
    RabbitMQ 282–285
    setting up alerts 287–291
    setting up metric collection infrastructure 276–283
GraphQL 70
Gunicorn web server 226

H

half open circuit breaker 148
HAProxy 197
hardware as source of failure 134
health checks 135, 150, 233–235
Helm application package manager 350
heterogeneity
  overview of 182–183
  technical heterogeneity 12
high cohesion 4
higher order services 59–60
histograms 274
holdings 130, 136
homogeneity 343
horizontal decomposition 9
hosts
  multiple scheduled services per 209–210
    advantages of scheduling model 210
    container schedulers 210
  multiple static services per 208–209
  single service to 207
  underlying, deploying services to 207–210
HTTP liveness check 151

I

IaaS (infrastructure as a service) 191
idempotent 140
identifiers 301
images
  building 218–220, 251–252
  overview of 216–218
  storing 223–224
immutability 201, 341
infrastructure as a service (IaaS) 191
infrastructures 9, 191, 335–337
inner-source model 344
installing Jenkins, on Minikube 349–355
instance group 194
interactions, choreographed 110
interfaces 104
internal load balancing 198
interruption 118
interwoven sagas 117–118
  interruption 118
  locking 118
  short-circuiting 118
investment strategies
  creating 79–86
  placing orders 88–92
InvestmentStrategies service 96
isolated staging environments 258
isolation 46

J

Jaeger 315
Java virtual machine (JVM) 183
Jenkins Pipeline 263
Jenkins tool
  building pipelines with 246–261
    building images 251–252
    configuring build pipelines 247–251
    deploying to production 259–261
    deploying to staging 255–257
    publishing artifacts 254–255
    running tests 252–254
    staging environments 258
  installing on Minikube 349–355
  running on Kubernetes 349–355
    configuring RBAC 353–354
    creating namespaces 350–351
    creating volume 350–351
    setting up Helm 350
    testing 354–356
jitter 142, 148
job queues 63–64
JVM (Java virtual machine) 183

K

Kibana plugin 305
KPIs (key performance indicators) 330
ksonnet 256
kubectl command-line tool 227
kube-proxy 231
Kubernetes platform, running Jenkins on 349–355. *See also* GCE
  configuring RBAC 353–354
  creating namespaces 350–351
  creating volume 350–351
  installing Jenkins 351–352
  setting up Helm 350
  testing 354–356

L

latency 272–273
launches 264
levels 301
limiting 177–180
Linkerd 157
liveness health check 233
living documentation 346
load balancers 56, 150–151, 196–198, 228–230
load/capacity tests 257
load testing 152–154
locking 118
logging 176–177
  generating logs 300–303
    readability 301–303
    structure 301–303
    useful information to include in log entries 300–301
  information 313
  searching 311–313
  setting up infrastructure 303–313
    configure which logs to collect 308–311
    ELK based solutions 304–306
    Fluentd-based solutions 306
  setting up solutions 306–308
login screen, Grafana 282
logs 67, 305
logstash-formatter library 176, 313
Logstash tool 302, 305
loose coupling 4, 8

M

managed registries 223
manual tests 257
mapping runtime platforms 57
market-data 130, 136
MarketDataClient class 140
master nodes 230
messages. *See also* asynchronous messages
metric collection infrastructure 276–283
  adding components to Docker compose file 276
  configuring Prometheus 280
  configuring StatsD exporter 277–279
  setting up Grafana 280–282
metrics
  overview of 67, 172–174
  types of 273–274
    gauges 273
    histograms 274
micro-frontends 72–74
microservice platforms 56–57
microservices. *See also* services
  advantages of 11–13, 30–32
    delivering sustainable value 31–32
    reducing friction 31–32
    reducing friction and risk 13
    risk and inertia in financial software 31
    technical heterogeneity 12

architecture of 52–56
architectural principles 54
four tiers of microservice applications 55–56
role of architect 54
challenges of 14–18
design challenges 14–17
operational challenges 17–18
deploying 21–24
implementing continuous delivery pipelines 23–24
standardizing microservices deployment artifacts 22–23
designing 19–21
communication 21
monoliths 19–20
resiliency 21
scoping services 20
development lifecycle of 18–25
identifying by modeling domains 33–35
key principles of 7–10
alignment 9–10
automation 9
autonomy 8
resilience 9
transparency 9
observing 24–25
behaviors across hundreds of services 25
identifying and refactoring fragile implementations 24–25
scaling through decomposition 6–7
scaling up development of 45–46
isolation 46
technical divergence 45–46
users of 10–11
microservice scoping 52
microservice teams 325–348
building 326–329
Conway's Law 327
principles for 328–330
autonomy 329
end-to-end responsibility 329
ownership 328–329
recommended practices for 340–347
answering questions about applications 347
design reviews 345–346
drivers of change in microservices 340–341
homogeneity versus technical flexibility 343
living documentation 346
open source models 343–344
role of architecture 341–343
team models 330–340
grouping across functions 332–333
grouping by function 330–332
infrastructures 335–337
on-call support 337–338
platforms 335–337
product 335–337
setting team boundaries 334–336
sharing knowledge 338–340
migration 100–103
Minikube tool, installing Jenkins on 349–355
MkDocs 346
modeling domains 33–35
model-view controller (MVC) 19
modes of failure 14
monitoring systems 269–295
observing whole application 293–295
raising alerts 291–293
symptoms versus causes 292–293
who needs to know 292
robust monitoring stacks 270–275
golden signals 272–273
layered monitoring 270–272
recommended practices 274–275
with Grafana 275–291
RabbitMQ 282–285
setting up alerts 287–291
setting up metric collection infrastructure 276–283
with Prometheus 275–291
RabbitMQ 282–285
setting up alerts 287–291
setting up metric collection infrastructure 276–283
monoliths 19–20, 71, 106, 111
multispeed development 103
MVC (model-view controller) 19

N

nameko framework 166
namespaces 350–351
nested contexts 86
network communication 61
nginx.conf file 222
NGINX container 222
NodePort service 230
nonbackwards-compatible functionality 135
noncritical paths 60
nonfunctional testing 257
notifications, sending 96–98
nslookup 241

O

object-oriented applications 93
object-relational mapping (ORM) 161
observability 18, 171–177
error reporting 174–175
logging 176–177
metrics 172–174
observable services 43
on-call support 337–338
OpenAPI specification 82
open source models 343–344
OpenTracing API 314
operating system packages 203
optimistic updates 126
orchestrated sagas 115–117
orchestration 94
OrderCreated event 64, 109

OrderPlaced event 39
OrderRequested event 109
organizations, ownership in 103–104
ORM (object-relational mapping) 161
ownership 103–104, 329–331

P

PaaS (platform as a service) 57, 209
paths
  critical 60
  noncritical 60
patterns, consistency patterns 118–119. *See also* CQRS pattern
pipelines. *See also* delivery pipelines
  building reusable 262–264
  building with Jenkins 246–261
    building images 251–252
    configuring build pipelines 247–251
    deploying to production 259–261
    deploying to staging 255–257
    publishing artifacts 254–255
    running tests 252–254
    staging environments 258
  overview of 57, 246
  procedural versus declarative build pipelines 263–264
pip tool 219
PlaceStrategyOrders service 88, 91
platform as a service (PaaS) 57, 209
platforms, mapping runtime platforms 57. *See also* microservice platforms
pods
  designing 226–228
  running 228–232, 226–233
point-to-point communication 36
polling 127
positive feedback 136
prepare phase, 2PC 108
principles 342
production
  deploying to 259–261
  taking features to 40–45
    automated deployments 42–43
    quality-controlled deployments 42–43
    resilience 43
    transparency 43–45
production environments 189–190
  automation 191
  features of 190–191
  speed 191
product mode 334
Prometheus ecosystem
  configuring 280
  monitoring systems with 275–291
    RabbitMQ 282–285
    setting up alerts 287–291
    setting up metric collection infrastructure 276–283
protocol buffers 346
provisioning virtual machines 192–193
public images, Docker 220
publish-subscribe 64–65, 127
Python chassis 166

Q

quality-controlled deployments 42–43
queries 120–127, 105–128
  analytics 127
  CQRS pattern challenges 125–127
    optimistic updates 126
    polling 127
    publish-subscribe 127
  reporting 127
  separating from commands 123–125
    CQRS pattern architecture 124
  storing copies of data 122–123
queues. *See* job queues

R

RabbitMQ broker 167, 282–285
rabbitmq_queue_messages metric 290
rate limits 67, 152
RBAC (role-based access control) 353–354
readability of logs 301–303
readiness checks 151, 233
recovery_timeout parameter 180
redundancy 144
releases 189
reliability
  defining 130–132
  maximizing 150–155
    load balancing 150–151
    rate limits 152
    service health 150–151
  overview of 42
  validating 152–155
    chaos testing 154–155
    load testing 153–154
remote_hello method 168
ReplicaSet 227
replication lag 125
reporting 127
request for comments (RFC) 345
resiliency 9–21, 43
retirement 100–103
retries 140–143
RFC (request for comments) 345
risk, reducing 13, 164
robust monitoring stacks 270–275
  golden signals 272–273
    errors 273
    latency 272–273
    saturation 273
    traffic 273
  layered monitoring 270–272
  recommended practices 274–275
  types of metrics 273–274
    gauges 273
    histograms 274

role-based access control. *See* RBAC
rolling back 114, 241, 259
rolling deploys 211–213
round-robin algorithm 167
routing components 190
RPC-facing services 150
rpc-market_service queue, RabbitMQ 178
run the packer build command 204
runtime management 190
runtime platforms 57, 190

S

sagas 111–120
  choreographed 112–115
  consistency patterns 118–119
  event sourcing 119–120
  interwoven 117–118
    interruption 118
    locking 118
    short-circuiting 118
  orchestrated 115–117
saturation 273
scalability 42
scaling up microservice development 45–46
  isolation 46
  technical divergence 45–46
scanning tools 220
scheduled services, multiple per host 209–210
  advantages of scheduling model 210
  container schedulers 210
scheduling models 210
scoping 15–16
  by business capabilities 78–87
    capabilities 78–79
    challenges and limitations 87
    creating investment strategies 79–86
    domain modeling 78–79
    nested contexts and services 86
  by use case 87–94
    actions 92–94
    choreography 94
    orchestration 94
    placing investment strategy orders 88–92
    stores 92–94
  by volatility 94–96
  services 20
Scripted Pipeline 247
secure operation 190
security 220
security tests 257
self-healing 195–196
sending notifications 96–98
server images 203–204
server management 201
service artifact 199
service collaboration 36–37
  service contracts 37
  service responsibility 37
service discovery 65, 167–171
service health 150–151
service mesh 157–158
service migration 102
service-oriented architectures (SOAs) 6
service partitioning 99
service practices as source of failures 136
service responsibility 37
services 58–60
  aggregation 59–60
  API gateways 39
  behavior across 297–300
  building artifacts 199–207
    configuring 206–207
    defining 200
    immutability 201
    types of service artifacts 202–206
  capabilities of 58–59
  challenges of designing 132–139
    cascading failures 136–139
    sources of failure 133–136
coarse-grained
  overview of 87
  starting with 99–100
connecting multiple 241–242
containerizing 215–224
  building images 218–220
  running containers 220–222
  storing images 223–224
  working with images 216–218
contracts between 16
critical paths 60
deploying 191–199, 187–213
  adding load balancers 196–198
  importance of 188–189
  provisioning virtual machines 192–193
  service startup 192
designing 129–158
  defining reliability 130–132
designing reliable communication 139–149
  asynchronous communication 149
  circuit breakers 146–148
  fallbacks 143–145
  retries 140–143
  timeouts 145–146
higher order services 59–60
interactions between tracing and 313–320
locating 65
maximizing reliability of 150–155
  load balancing 150–151
  rate limits 152
  service health 150–151
  validating reliability and fault tolerance 152–155
nested contexts and 86
noncritical paths 60
ownership in organizations 103–104
run multiple instances of 194–196
  adding capacity 196

failure zones 195
self-healing 195–196
safety of 156–158
frameworks 156
service mesh 157–158
scoping 20
setting up tracing in 315–320
to underlying hosts 207–210
multiple scheduled services per host 209–210
multiple static services per host 208–209
single service to host 207
without downtime 210–213
canaries on GCE 211–213
rolling deploys on GCE 211–213
service teams 87
service to host models 207–210
multiple scheduled services per host 209–210
advantages of scheduling model 210
container schedulers 210
multiple static services per host 208–209
single service to host 207
sharding 6
short-circuiting 118
silos 332
smoke tests 259
SOAs (service-oriented architectures) 6
sources 301
spans 314–315
stability 189
automation 189
manual change management 189
small releases 189
stacks. *See* robust monitoring stacks
staging
deploying to 255–257
environments 258
standardizing communication 156
startup logs 205
startups 192
state changes 231
static services 208–209
StatsD exporter 172–173, 277–279
statsd-exporter container 277
STDOUT (standard output) 304
stores 92–94
storing
copies of data 122–123
images 223–224
stubbed data 145
subtasks, sagas 112
supporting notifications 97
synchronous communication 36, 163
synchronous messages
choosing transport 62
drawbacks of 62
when to use 61–62
synchronous requests 62

T

tarballs 223
team models 330–340
grouping across functions 332–333
grouping by function 330–332
lack of autonomy 332
no long-term responsibility 332
risk of silos 332
unclear ownership 331
infrastructures 335–337
on-call support 337–338
platforms 335–337
product 335–337
setting team boundaries 334–336
sharing knowledge 338–340
teams, challenges of 16. *See also* microservice teams
technical capabilities 96–98
overview of 59
sending notifications 96–98
when to use 98
technical divergence 45–46
technical heterogeneity 12
tenacity library 140
testing 252–254
chaos testing 154–155
Jenkins on Minikube 354–356
load testing 153–154
three-tier architecture 19, 52
timeouts 145–146
timestamps 300
Togglz 265
traces, visualizing 320–324
tracing
interactions between services and 313–320
setting up in services 315–320
spans and 314–315
TradeExecuted event 39
traffic 273
transactions 105–128
consistent 106–108
distributed 107–108
overview of 130, 136
transparency 9, 42–45
transport 62
transport-related boilerplate 163
truck factor 41
try-finally statement 254
two-phase commit (2PC) protocol 107

U

updates, optimistic 126
upstream collaborators 37
uptime 130
use case, scoping by 87–94
actions 92–94
choreography 94
orchestration 94
placing investment strategy orders 88–92
stores 92–94
user management 80

V

vault 207
VMs (virtual machines) 56, 192–193
volatility 78, 94–96
volume 350–351

W

worker nodes, components of 231
workflow tools 117
workspaces 247

Z

zero-downtime deployments 210

SOFTWARE DEVELOPMENT

Microservices IN ACTION

Bruce • Pereira

Invest your time in designing great applications, improving infrastructure, and making the most out of your dev teams. Microservices are easier to write, scale, and maintain than traditional enterprise applications because they're built as a system of independent components. Master a few important new patterns and processes, and you'll be ready to develop, deploy, and run production-quality microservices.

Microservices in Action teaches you how to write and maintain microservice-based applications. Created with day-to-day development in mind, this informative guide immerses you in real-world use cases from design to deployment. You'll discover how microservices enable an efficient continuous delivery pipeline, and explore examples using Kubernetes, Docker, and Google Container Engine.

What's Inside

- An overview of microservice architecture
- Building a delivery pipeline
- Best practices for designing multi-service transactions and queries
- Deploying with containers
- Monitoring your microservices

Written for intermediate developers familiar with enterprise architecture and cloud platforms like AWS and GCP.

Morgan Bruce and **Paulo A. Pereira** are experienced engineering leaders. They work daily with microservices in a production environment, using the techniques detailed in this book.

"The one [and only] book on implementing microservices with a real-world, cover-to-cover example you can relate to."
—Christian Bach, Swiss Re

"A perfect fit for those who want to move their majestic monolith to a scalable microservice architecture."
—Akshat Paul
McKinsey & Company

"Shows not only how to write microservices, but also how to prepare your business and infrastructure for this change."
—Maciej Jurkowski, Grupa Pracuj

"A deep dive into microservice development with many real and useful examples."
—Antonio Pessolano
Consoft Sistemi

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/microservices-in-action

ISBN-13: 978-1-61729-445-7
ISBN-10: 1-61729-445-4
54999
9 781617 294457

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]