



2ND EDITION

Microservices with Go

The expert's guide to building secure, scalable, and reliable microservices with Go



ALEXANDER SHUISKOV

Microservices with Go

Second Edition

The expert's guide to building secure, scalable,
and reliable microservices with Go

Alexander Shuiskov



Microservices with Go

Second Edition

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Kunal Chaudhari

Relationship Lead: Dhruv J. Kataria

Project Manager: Ashwin Dinesh Kharwa

Content Engineer: Deepayan Bhattacharjee, Nisha Cleetus

Technical Editor: Nithik Cheruvakodan

Copy Editor: Safis Editing

Indexer: Rekha Nair

Proofreader: Deepayan Bhattacharjee

Production Designer: Deepak Chavan

Growth Lead: Sayantani Saha

First published: November 2022

Second edition: June 2025

Production reference: 1220525

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83620-733-7

www.packtpub.com

I dedicate this book to my family and to my source of passion, inspiration, and love, Vera.

– Alexander Shuiskov

Contributors

About the author

Alexander Shuiskov is an independent researcher and software expert specializing in distributed systems, microservices, and observability. He has worked at major tech companies, including Uber, Booking.com and eBay, where he helped solve large-scale infrastructure challenges, as well as establish reliability and monitoring of thousands of microservices.

About the reviewer

Sadman Khan is a Staff Software Engineer at Uber, working in the Observability organization. With a strong background in building scalable and reliable systems, he co-created Uber's alerting engine alongside Alex, a platform that powers over 3 million alerts, ensuring Uber's infrastructure remains highly reliable.

Sadman holds a bachelor's degree in computer engineering from the University of Waterloo, Canada. Before joining Uber, he gained valuable experience at LinkedIn and Coinbase, working on high-impact systems in the tech industry.

Table of Contents

Preface	xvii
<hr/>	
Part 1: Introduction	1
<hr/>	
Chapter 1: Introduction to Microservices	3
<hr/>	
What is a microservice?	3
Motivation to use microservices	4
Pros and cons of microservices	6
Benefits of microservices • 6	
Common issues of microservices • 8	
When to use microservice architecture	9
The role of Go in microservice development	11
Summary	12
Further reading	12
<hr/>	
Part 2: Foundation	13
<hr/>	
Chapter 2: Scaffolding a Go Microservice	15
<hr/>	
Technical requirements	16
Go basics	16
Core principles • 16	

Writing idiomatic Go code • 17	
<i>Naming</i> • 17	
<i>Comments</i> • 18	
<i>Errors</i> • 19	
Interfaces • 20	
Tests • 20	
Context • 20	
Project structure	21
Private packages • 22	
Public packages • 22	
Executable packages • 23	
Other commonly used directories • 23	
Common files • 23	
Best practices • 24	
Scaffolding an example application	24
Movie application • 25	
<i>Movie metadata</i> • 25	
<i>Ratings</i> • 25	
<i>Should we split the application?</i> • 27	
Application code structure • 29	
Movie metadata service • 31	
<i>Model</i> • 32	
<i>Repository</i> • 32	
<i>Controller</i> • 34	
<i>Handler</i> • 36	
<i>Main file</i> • 37	
<i>Rating service</i> • 38	
<i>Model</i> • 39	
<i>Repository</i> • 40	
<i>Controller</i> • 41	
<i>Handler</i> • 43	

<i>Main</i> • 45	
Movie service • 46	
<i>Model</i> • 46	
<i>Gateways</i> • 47	
<i>Controller</i> • 51	
<i>Handler</i> • 53	
<i>Mainfile</i> • 54	
Summary	55
Further reading	56
Chapter 3: Service Discovery	57
Technical requirements	57
Service discovery overview	58
Registry • 59	
Service discovery models • 59	
<i>Client-side service discovery</i> • 60	
<i>Server-side service discovery</i> • 60	
Service health monitoring • 61	
Service discovery solutions	62
HashiCorp Consul • 62	
Kubernetes • 62	
etcd • 62	
Apache ZooKeeper • 63	
Adopting service discovery	63
Preparing the application • 63	
Implementing the discovery logic • 66	
<i>In-memory implementation</i> • 67	
<i>Consul-based implementation</i> • 70	
Using the discovery logic • 72	
Summary	79
Further reading	79

Chapter 4: Serialization	81
Technical requirements	82
The basics of serialization	82
Popular serialization formats	84
XML • 84	
YAML • 84	
Apache Thrift • 85	
Apache Avro • 86	
Protocol Buffers • 87	
Using Protocol Buffers	87
Serialization best practices	94
Summary	95
Further reading	95
Chapter 5: Synchronous Communication	97
Technical requirements	98
Introduction to synchronous communication	98
Go RPC frameworks and libraries • 100	
<i>Apache Thrift</i> • 100	
<i>gRPC</i> • 100	
Defining a service API using Protocol Buffers	101
Implementing gateways and clients	105
Metadata service • 105	
Rating service • 110	
Movie service • 112	
Synchronous communication best practices	118
Perform excessive request validation and return correct error codes • 118	
Ensure idempotency • 119	
Summary	120
Further reading	120

Chapter 6: Asynchronous Communication	121
Technical requirements	121
Asynchronous communication basics	122
Benefits and challenges of asynchronous communication • 122	
Techniques and patterns of asynchronous communication • 124	
<i>Message broker</i> • 124	
<i>The publisher-subscriber model</i> • 125	
Using Apache Kafka for messaging	126
Apache Kafka basics • 126	
Adopting Kafka for our microservices • 128	
Asynchronous communication best practices	138
Versioning • 138	
Leverage partitioning • 139	
Use explicit message acknowledgment whenever necessary • 140	
Use a separate topic for unprocessed messages • 142	
Summary	143
Further reading	143
Chapter 7: Storing Service Data	145
Technical requirements	145
Introduction to databases	146
Common database features • 148	
Common database types • 148	
Using MySQL to store our service data	150
Implementing data caching	158
Summary	161
Further reading	161

Chapter 8: Setting Up Service Deployments	163
Technical requirements	164
Preparing application code for deployments	164
Deployment basics • 164	
Application configuration • 165	
Service deployment solutions	169
<i>Docker Swarm</i> • 169	
HashiCorp Nomad • 170	
Kubernetes • 170	
Deploying via Kubernetes	171
Introduction to the Kubernetes data model • 171	
Setting up our microservices for Kubernetes deployments • 172	
Deployment best practices	179
Automated rollbacks • 179	
Canary deployments • 180	
Continuous deployment • 180	
Use feature flags to decouple deployments from feature releases • 181	
Summary	182
Further reading	182
Chapter 9: Unit and Integration Testing	183
Technical requirements	183
Go testing overview	184
Subtests • 186	
Skipping • 187	
Unit tests	189
Mocking • 190	
Implementing unit tests • 193	
Integration tests	196

Testing best practices	207
Using helpful messages • 207	
Avoiding the use of Fatal in your logs • 208	
Comparing structures using a cmp library • 209	
Detecting and fixing flaky tests • 210	
Detecting race conditions • 211	
Tracking and maintaining high code coverage • 212	
Summary	213
Further reading	213
 Chapter 10: Security and Compliance	 215
 Technical requirements	 216
Security basics	216
Key areas of software security • 216	
Authentication and access control • 217	
Secure service communication • 218	
Implementing secure service communication with TLS	219
Additional improvements • 224	
Implementing authentication and access control with JWT	225
The basics of JWT • 225	
Implementing JWT issuance and validation in microservices • 226	
Security analysis of Go services	235
Using gosec to automate security analysis • 236	
Security best practices	237
Secure secret management • 238	
Use automated vulnerability scanning • 239	
Perform periodic threat modeling exercises • 240	
Compliance basics	241
Summary	243
Further reading	243

Part 3: Maintenance 245**Chapter 11: Reliability Overview 247**

Technical requirements	248
Reliability basics	248
Achieving reliability through automation	249
Communication error handling • 250	
<i>Implementing request retries</i> • 250	
<i>Deadlines and timeouts</i> • 254	
<i>Fallbacks</i> • 256	
<i>Rate limiting</i> • 257	
Graceful shutdown • 260	
Achieving reliability through development processes and culture	263
On-call process • 264	
Incident management • 266	
Reliability drills • 268	
Disaster recovery plans • 269	
Set and track the reliability objectives of your services • 269	
Summary	272
Further reading	272

Chapter 12: Collecting Service Telemetry Data 273

Technical requirements	274
Telemetry overview	274
Collecting service logs	276
Choosing a logging library • 279	
<i>Elasticsearch</i> • 283	
<i>OpenSearch</i> • 283	
Storing microservice logs • 282	
<i>OpenTelemetry Collector</i> • 284	

Logging best practices • 284	
<i>Avoiding using interpolated strings</i> • 285	
<i>Standardizing the format of your log messages</i> • 285	
<i>Periodically reviewing your log data</i> • 287	
<i>Setting up appropriate log retention</i> • 287	
<i>Identifying the message source in logs</i> • 287	
Collecting service metrics	288
Storing metrics • 291	
<i>Prometheus</i> • 291	
<i>Graphite</i> • 292	
<i>OpenTelemetry Collector</i> • 292	
Popular Go metrics libraries • 292	
Emitting service metrics • 293	
Metrics best practices • 294	
<i>Keeping tag cardinality in mind</i> • 295	
<i>Standardizing metric and tag names</i> • 295	
<i>Setting the appropriate retention</i> • 296	
Collecting service traces	296
Tracing tools • 298	
Collecting tracing data with the OpenTelemetry SDK • 299	
Summary	308
Further reading	309
Chapter 13: Setting Up Service Alerting	311
Technical requirements	311
Alerting basics	312
Alerting use cases • 313	
Introduction to Prometheus	315
Setting up Prometheus alerting for our microservices	317
Alerting best practices	325
Summary	327
Further reading	327

Chapter 14: Performance Monitoring	329
Technical requirements	329
Creating dashboards to visualize service telemetry data	330
Introduction to dashboards • 330	
Creating performance dashboards using the Grafana tool • 331	
Adding service-level metrics to dashboards • 334	
Profiling Go services	340
Profiling CPU usage using the pprof tool • 340	
Profiling heap memory usage using the pprof tool • 344	
Summary	345
Further reading	346
 Part 4: Advanced Topics	347
 Chapter 15: Implementing Distributed System Scenarios	349
Technical requirements	350
Introduction to distributed system problems	350
Consensus in distributed systems • 351	
<i>Paxos</i> • 351	
<i>Raft</i> • 352	
Distributed locking and leader election • 352	
Distributed system tools	354
Apache Zookeeper • 354	
etcd • 355	
HashiCorp Consul • 355	
Implementing distributed locking with HashiCorp Consul	356
Distributed system best practices	362
Build systems with minimal necessary coordination between components • 363	
Avoid distributed transactions whenever possible and find the right level of consistency for your data • 364	

Summary	365
Further reading	365
Chapter 16: Advanced Topics	367
Technical requirements	367
Static analysis of Go service code	368
Implementing data validation	371
Implement streaming APIs	375
Frameworks	379
Storing microservice ownership data	382
Summary	384
Further reading	385
Other Books You May Enjoy	388
Index	391

Preface

Since its release, the Go programming language has gained popularity among all types of software developers. Simple language syntax, ease of use, and a rich set of libraries make Go one of the primary languages for writing different kinds of software, from small tools to large-scale systems consisting of hundreds of components.

Among the primary Go use cases is microservice development – the development of individual applications, called microservices, that can play various roles, from processing payments to storing user data. Organizing a large system as a set of microservices often brings multiple advantages, such as increasing the development and deployment speed, but also brings multiple types of challenges. Among such challenges are service discovery and communication, integration testing, and service monitoring.

In this book, we will illustrate how to implement Go microservices and establish communication between them, how to enable the deployment of individual microservices and secure their interactions, and how to store and retrieve service data and provide service APIs, enabling other applications to use our microservices. You will learn about some of the industry's best practices related to all of these topics and get a detailed overview of the possible challenges along the way, as well as the possible benefits. The knowledge that you will gain by reading this book will help you to both create new microservices and efficiently maintain the existing ones. I hope this journey will be exciting for you!

Who this book is for

This book is for all types of developers: from people interested in learning how to write microservices in Go to seasoned professionals who want to take the next step in mastering the art of writing scalable and reliable microservice-based systems. The first two parts of the book, covering the development of microservices, will be useful to developers who are just starting their experience with Go or those who are interested in the best practices of organizing the Go application code base according to the industry's best standards; the last two parts will be useful for all developers, even the most experienced ones, as it provides great insights on maintaining and operating microservices at scale.

What this book covers

Chapter 1, Introduction to Microservices, covers the key benefits of and common issues with a microservice architecture, helping you understand which problems microservices solve and which challenges they usually introduce. The chapter emphasizes the role of the Go programming language in microservice development and lays down the foundation for the rest of the book.

Chapter 2, Scaffolding Go Microservices, introduces you to the main principles of the Go programming language and provides the most important recommendations for writing Go code. It covers the process of setting up the right structure to organize the microservice code in Go and introduces you to an example application consisting of three microservices. Finally, the chapter illustrates how to scaffold the code for each of the example microservices. The example microservices implemented in this chapter are going to be used throughout the book, with each chapter adding new features to them.

Chapter 3, Service Discovery, talks about the problem of service discovery and illustrates how different services can find each other in a microservice environment. It covers the most popular service discovery tools and walks you through the steps of adding service discovery logic to the example microservices from the previous chapter.

Chapter 4, Serialization, brings us to the concept of data serialization, which is required for understanding upcoming chapters covering microservice communication. It introduces the Protocol Buffers data format, which is going to be used for encoding and decoding the data transferred between our example microservices. The chapter provides examples of how to define serializable data types and generate code for them, and how to use the generated code in Go microservices.

Chapter 5, Synchronous Communication, covers the topic of synchronous communication between microservices. It illustrates how to define service APIs using the Protocol Buffers format and introduces you to gRPC, a service communication framework. The chapter wraps up with examples of how to implement microservice gateways and clients and perform remote calls between our microservices.

Chapter 6, Asynchronous Communication, talks about asynchronous communication between microservices. It introduces you to a popular asynchronous communication tool, Apache Kafka, and provides examples of sending and receiving messages, using it for our example microservices. The chapter wraps up with an overview of best practices for using asynchronous communication in microservice environments.

Chapter 7, Persistence and Databases, covers the topic of persisting service data in databases. You will learn about the common types of databases and the benefits they bring to software developers. The chapter walks you through the process of implementing the logic for storing service data in a MySQL database.

Chapter 8, Setting Up Service Deployments, talks about service deployment and provides an overview of a popular deployment and orchestration platform, Kubernetes. The chapter illustrates how to prepare service code for deployment and how to deploy it using Kubernetes. It also includes best practices for deploying microservice applications.

Chapter 9, Unit and Integration Testing, describes the common techniques of testing Go microservice code. It covers the basics of Go unit and integration testing and demonstrates how to test the microservice code from the previous chapters. The chapter wraps up with the industry's best practices for writing and organizing tests.

Chapter 10, Security and Compliance, covers the main aspects of Go microservice security, such as authentication, authorization, secret management, and vulnerability analysis. The chapter illustrates how to implement service-level authentication and access control with JSON Web Tokens.

Chapter 11, Reliability Overview, introduces you to the topic of system reliability and describes the core principles, instruments, and industry best practices for building reliable and highly available microservices. It illustrates how to automate service responses to various types of failures, as well as how to establish the processes for keeping service reliability under control.

Chapter 12, Collecting Service Telemetry Data, provides a detailed overview of modern instruments and solutions for collecting service telemetry data, such as logs, metrics, and traces. The chapter provides lots of detailed examples of collecting all different types of telemetry data and lists some of the best practices for working with them.

Chapter 13, Setting Up Service Alerting, illustrates how to set up automated incident detection and notification for microservices, using the telemetry data collected in the previous chapter. It introduces you to a popular alerting and monitoring tool, Prometheus, and shows how to set up Prometheus alerts for our example microservices.

Chapter 14, Performance Monitoring, describes the common techniques of tracking Go microservice performance, such as memory and CPU profiling and dashboards. The chapter provides practical examples of setting up service performance dashboards using a popular Grafana tool.

Chapter 15, Implementing Distributed System Scenarios, covers some common distributed system problems and solutions that Go service developers might face while building advanced applications.

Chapter 16, Advanced Topics, wraps up the last part of the book and covers some of the advanced topics in microservice development, such as static code analysis, data validation, streaming APIs, service ownership, and frameworks. The chapter includes examples of data validation techniques, static code checks, and streaming API client and server logic.

To get the most out of this book

I suggest you get some familiarity with Go by implementing a few applications, such as simple web services. Familiarity with a Docker tool would be a plus because we will be using it for running some of the tools that our microservices will be using. Finally, I strongly suggest implementing, running, and playing with the example microservices that we will be implementing so that all your knowledge will be cemented by practice.

Software/hardware covered in the book	Operating system requirements
Go 1.18 or above	Windows, macOS, or Linux
Docker	Windows, macOS, or Linux
grpcurl	Windows, macOS, or Linux
Kubernetes	Windows, macOS, or Linux
Prometheus	Windows, macOS, or Linux
Jaeger	Windows, macOS, or Linux
Graphviz	Windows, macOS, or Linux

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781836207337>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: " XML represents data as a tree of nodes called elements. An element example would be <example>Some value</example>.

A block of code is set as follows:

```
Metadata{  
    ID:      "123",  
    Title:   "The Movie 2",  
    Description: "Sequel of the legendary The Movie",  
    Director: "Foo Bars",  
}
```

Any command-line input or output is written as follows:

```
protoc -I=api --go_out=. movie.proto
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "These solutions combine two roles: they act as both serialization formats and **communication protocols** – mechanisms for sending and receiving arbitrary data over the network."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com/>.

Share your thoughts

Once you've read *Microservices with Go - Second Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781836207337>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

Part 1

Introduction

This part provides an overview of the microservice architecture model. It covers the key benefits and common issues of microservices, helping you understand which problems microservice architecture helps to solve and which issues it usually introduces. The chapter emphasizes the role of the Go programming language in microservice development and lays down the foundation for the rest of the book.

This part of the book includes the following chapters:

- *Chapter 1, Introduction to Microservices*

1

Introduction to Microservices

In this chapter, you will be introduced to **microservices** and the motivation behind them. You will understand the key benefits and common issues of the microservice architecture model and learn when to use it, as well as learning about some microservice development best practices. This knowledge will help you establish a solid foundation for reading the next chapters and give you some ideas on what challenges you may face with microservices in the future.

In this chapter, we will cover the following topics:

- What is a microservice?
- Motivation to use microservices
- Pros and cons of microservices
- When to use microservice architecture
- The role of Go in microservice development

What is a microservice?

Companies worldwide have used the **microservice architecture model** so widely that it has almost become a default way of software development. There are many companies having tens, hundreds, and even thousands of microservices at their disposal.

So, what exactly is the microservice model?

The microservice architecture model is organizing an application as a collection of services, called microservices, each of which is further responsible for a certain part of application logic, usually defined by a particular business capability.

As an example, consider an online marketplace application. The application may have multiple features, including search, a shopping cart, payments, order history, and much more. Each feature can be so different and complex on its own that it could be developed and maintained by a separate team – like search and payments in our example. In the microservice architecture model, each component would be an independent service playing its own role in the system.

Organizing each part of the application as a separate service is not necessarily a requirement. As with any architecture model or any aspect of software development, engineers need to be careful with choosing a particular approach or solution – doing an initial analysis and understanding the solution under the given conditions.

Before we proceed to the key benefits and downsides of microservices, let's see what challenges you could face when the application is not separated into multiple services.

Motivation to use microservices

In order to understand the motivation behind using the microservice architecture, it is very important to see the opposite approach – when the application is built and executed as a single program. Such applications are called **monolithic applications** or **monoliths**.

Monolithic architecture is, in most ways, the simplest model to implement since it does not involve splitting the application into multiple parts that need to coordinate with each other. This can provide you with major advantages in many cases, such as the following:

- **Application logic is still loosely defined:** It is very common that parts of the application or the entire system go through major structural or logical changes, especially at the very early stages of development. This might be caused by a sudden change of requirements, priorities, changes in the business model, or a different approach to development. In such cases, the ability to make quick changes to the application structure can be critical not only to the development process but also to the entire company.
- **Narrow scope of the application:** Certain applications can have a limited scope (for example, a photo upload service) or have their features integrated tightly (payment processing and invoice generation). Such applications might benefit from being monoliths: it is much easier to perform end-to-end testing and development without extra boundaries between the parts of the application.
- **Performance constraints:** There are types of applications (for example, trading systems) that can be very sensitive to the latency of each operation. Time-sensitive parts of such applications could benefit from monolithic architecture because there wouldn't be any API calls between the components.

In all of the preceding cases, monolithic architecture would be a better fit for the application. However, at some point, services get too big to remain monolithic. Developers start experiencing the following issues:

- **Large application size and slow deployments:** At a certain point, an application can become so big that it can take minutes or even hours to build, start, or deploy.
- **Inability to deploy a particular part of the application independently:** Not being able to replace a part of a large application can easily become a bottleneck, slowing down the development and release process.
- **Higher blast radius:** If there is a bug in a certain function or library widely used across the application code, it is going to affect all parts of the system at once, potentially causing major issues.
- **Inability to scale components separately:** If some part of a monolithic application requires more resources (CPU, RAM, or disk space), the entire application would need to be scaled – this could be very expensive.
- **Vertical scalability bottleneck:** The more logic the application has, the more resources it needs in order to run. At a certain point, it can get hard or impossible to scale the application up even further, given the possible limits on CPU and RAM.
- **Interference:** Certain parts of the application can perform heavy operations (for example, CPU-intensive data processing), reducing the performance of the rest of the system.
- **Higher security risks:** A possible security issue in the application may result in unauthorized access to all components at once.
- In addition to the possible issues we just described, different components may have different requirements, such as the following:
 - **Resources and hardware requirements:** Certain components are more CPU-intensive or memory-intensive and may perform I/O operations at a higher rate. Separating such components may reduce the load on the entire system, increasing system availability and reducing latency.
 - **Deployment cadence:** Some parts of the system mostly remain unchanged while others require multiple deployments per day.
 - **Deployment monitoring and automated testing:** Certain components may require stricter checks and monitoring and can be subject to slower deployments due to multi-step rollouts.

- **Technologies or programming languages:** It is not uncommon that different parts of the system can be written in different programming languages or use fundamentally different technologies, libraries, and frameworks.
- **Change management process:** Some components may be subject to a stricter code review, deployment, or data management processes, as well as additional requirements.
- **Security:** Components may have different security requirements and may require additional isolation from the rest of the application for security reasons.
- **Compliance:** Some parts of the system may be subject to stricter compliance requirements. For example, handling **personally identifiable information (PII)** for users from a certain region can put stricter requirements on the entire system. Logical separation of such components helps to reduce the scope of work required to keep the system compliant.

With all the preceding issues described, we can see that, at a certain point, monolithic applications can become too big for a *one-size-fits-all* model. As the application grows, certain parts of it may start becoming independent and have different requirements, benefiting from a logical separation from the rest of the application.

In the next section, we are going to see how splitting the application into microservices can solve the aforementioned problems and which aspects of it you should be careful with.

Pros and cons of microservices

In order to understand how to get the best results from using microservices and which issues to be aware of, let's review the pros and cons of the microservice model.

Benefits of microservices

As previously described, different application components may have fundamentally different requirements and, at certain points, diverge so much that it would be beneficial to separate them. In this case, microservice architecture provides a clear solution by decoupling the parts of the system.

Microservices provide the following benefits to developers:

- **Faster compilation and build time:** Faster build and compilation time may play a key role in speeding up all development processes.
- **Faster deployments, smaller deployable size:** When each part of the system is deployed separately, the deployable size can get so significantly smaller that individual deployments can take just a fraction of the time compared to monolithic applications.

- **Custom deployment cadence:** The microservice model solves the problem of following a custom deployment schedule. Each service can be deployed independently and follow its own schedule.
- **Custom deployment monitoring:** Some services can perform more critical roles in the system than others and may require more fine-grained monitoring and extra checks.
- **Independent and configurable automated testing:** Services may be configured to perform different automated tests as a part of the build and deployment pipeline. Additionally, the scope of checks can be reduced for individual microservices, that is, we don't need to perform tests for the entire application, which may take longer.
- **Cross-language support:** It is no longer required to run an application as a single executable, so it is possible to implement different parts of the system using different technologies, finding the best fit for each problem.
- **Simpler APIs:** Fine-grained APIs are one of the key aspects of microservice development and having clear and efficient APIs helps to enforce the right composition of the system.
- **Horizontal scaling:** Microservices are easier and often cheaper to scale horizontally. Monolithic applications are usually resource-heavy and running them on numerous instances could be quite expensive due to high hardware requirements. Microservices, however, can be scaled independently. So, if a particular part of the system requires running on hundreds or thousands of servers, other parts don't need to follow the same requirements.
- **Hardware flexibility:** With microservices, you can run application components on fully different hardware configurations: in-memory storage could be running on hosts with high amounts of RAM, while data processing logic might be executed on hosts with expensive GPUs that the rest of the application would not use at all.
- **Fault isolation:** Service decoupling provides an efficient safety mechanism to prevent major issues on partial system failures.
- **Understandability:** Services are easier to understand and maintain due to smaller code base sizes.
- **Cost optimization:** Matching each service to its optimal resource configuration often results in significant cost savings.
- **Distributed development:** Removing the coupling between the components helps achieve more independence in code development, which can play an important role in distributed teams.

- **Ease of refactoring:** In general, it is much easier to perform refactoring for microservices due to the lower scope of changes and independent release and testing processes, which helps detect possible issues and reduce the scope of failures.
- **Independent decision-making:** Developers are free to choose libraries, frameworks, and tools that fit their needs the best. This does not, however, imply that there should be no standardization, but it is often highly beneficial to achieve a certain degree of freedom for distributed decision-making.
- **Removing unnecessary dependencies:** It is easy to miss detecting unwanted dependencies between the components of a monolithic application given the tighter coupling of the components. Microservice architecture helps you notice unwanted dependencies between components and restricts the use of certain services to particular parts of the application.

As we can see, microservices bring a high degree of flexibility and help to achieve a higher level of independence between the components. These aspects may be instrumental to the success of a large development team, allowing them to build and maintain independent components separately. However, any model comes at its own cost, and in the next section, we are going to see the challenges you could face with a collection of microservices.

Common issues of microservices

As with any solution, microservice architecture has its own issues and limitations. Some issues with microservice architecture include the following:

- **Higher resource overhead:** When an application consists of multiple components, instead of sharing the same process space, there is a need to communicate between the components that involve higher network use. This puts more load on the entire system and increases traffic, latency, and I/O usage. In addition, the total CPU and RAM are also higher due to the extra overhead of running each component separately.
- **Debugging difficulty:** Troubleshooting and debugging are often more difficult when you deal with multiple services. For example, if multiple services process a request that fails, a developer needs to access the logs of multiple services in order to understand what caused the failure.
- **Integration testing:** Separating a system requires building a large set of integration tests and other automated checks that would monitor the compatibility and availability of each component.

- **Consistency and transactions:** In microservice applications, the data is often scattered across the system. While this helps to separate the independent parts of the application, it makes it harder to do transactional and atomic changes in the system.
- **Divergence:** Different services may use different versions of libraries, which may include incompatible or outdated ones. Divergence makes it harder to perform system upgrades and resolve various issues, including software vulnerability fixes.
- **Possible duplication, overlapping functionality:** In a highly distributed development environment, it is not uncommon to have multiple components performing similar roles in the system. It is important to set clear boundaries within the system and decide in advance which particular roles the components are assigned.
- **Ownership and accountability:** Ownership becomes a major aspect of the development process when there are many different teams maintaining and developing independent components. It is crucial to define clear ownership contracts to address development requests, security and support issues, and all other types of maintenance work.

As we have just illustrated, the microservice model comes at a cost and you should expect that you will need to solve all these challenges at a certain point. Being aware of the possible challenges and being proactive in solving them is the key to success – the benefits that we have described earlier can easily outweigh the possible issues.

In the next section, we are going to summarize when to use microservices and learn about some best practices for working with them.

When to use microservice architecture

We have covered the benefits and common issues of microservices, providing a good overview of the applicability of using the microservice architecture model in an application. Let's summarize the key points of using the microservice model, which are the following:

- **Don't introduce microservices too early:** Don't use the microservice architecture too early if the product is loosely defined or could go through significant changes. Even when developers know the exact purpose of the system, there are high chances of various changes in the early stages of the development process. Starting from a monolithic application – and splitting it over time once there are clearly defined business capabilities and boundaries – helps reduce the amount of work and establish the right interfaces between the components.

- **No size fits all:** Each company is unique and the final decision should depend on many factors, including the size of the team, its distribution, and geography. A small local team may be comfortable working with a monolithic application, whereas a geographically distributed team may highly benefit from splitting the application into multiple microservices to achieve higher flexibility.

Additionally, let's summarize the best practices of using the microservice architecture model for applications, which are the following:

- **Design for failure:** In a microservice architecture, there are many interactions between the components, most of which happen via remote calls and events. This increases the chance of various failures, including network timeouts, client errors, and many more. Build the system thinking of every possible failure scenario and different ways to proceed with it.
- **Embrace automation:** Having more independent components requires much stricter checks in order to achieve stable integration between the services. Investing in solid automation is absolutely necessary in order to achieve a high degree of reliability and ensure all changes are safe to deploy.
- **Don't ship hierarchy:** It is a relatively common practice to split the application into services based on the organizational structure, where each team may be responsible for its own service. This model works well if the organizational structure perfectly aligns with the business capabilities of the microservices, but quite often this is not the case. Instead of using a service-per-team model, try to define the clear domains and business capabilities around which the code is structured and see how the components interact with each other. It is not easy to achieve perfect composition, but you will be highly rewarded for it.
- **Invest in integration testing:** Make sure you have comprehensive tests for the integrations between your microservices performing automatically.
- **Keep backward compatibility in mind:** Always remember to keep your changes backward compatible to ensure that new changes are safe to deploy. Additionally, use techniques such as versioning, which we are going to cover in the next chapter of the book.

At this point, we have covered the key aspects of microservice development, and you have learned about its benefits and the challenges you may face. Before we proceed to the next chapter, let's cover one more topic to ensure we are ready for the journey into microservice development. Let's get familiar with the Go programming language and its role in microservice development.

The role of Go in microservice development

Over the last decade, the Go programming language has become one of the most popular languages for application development. There have been many factors contributing to its success, including its simplicity, ease of writing network applications, and the ability to easily develop parallel and concurrent applications.

Additionally, the larger developer community has played a key role in raising its popularity across all types of developers. The Go community is welcoming to everybody, from people just starting their journeys into programming to seasoned experts with decades of experience building different types of applications.

The Go standard library provides a set of packages that can often be enough for building a complete web application or an entire service, sometimes without even requiring any external dependencies. Many developers have been fascinated by the ease of writing applications and tools performing network calls, data serialization and encoding, file processing, and many other types of common operations.

This simplicity, paired with fast and efficient compilation into native binaries, as well as rich tooling, made it one of the primary languages for writing web tools and services. The high adoption of the Go language for web service development made it one of the primary choices for writing microservices across the industry.

The biggest advantages of Go include the following:

- **Smooth learning curve:** As one of the critical aspects of application development in growing teams, the simplicity of the Go language helps reduce the onboarding time for new, inexperienced developers.
- **Explicit error handling:** While a hot topic in the Go community, error handling in Go encourages explicit handling of all application errors. It aligns with one of the key principles of microservice development of designing applications for failure.
- **Useful standard library:** The Go standard library includes lots of packages that can be used in production-grade systems without requiring external solutions.
- **Community support:** The Go community is among the biggest in the industry and the most popular libraries get enough support and maintenance.
- **Ease of writing concurrent code:** Concurrent calls are very common in microservice application logic – microservices often call multiple other services and combine their results. Writing concurrent code in Golang can be a fairly trivial task when utilizing the built-in sync package and core language features, such as channels and Goroutines.

Some Go advantages are especially useful in microservice development:

- **Native binary format:** Unlike virtual machine-based languages, such as Java, Kotlin, Scala, or Clojure, or interpreted languages, such as Python, Go applications are compiled into native binaries and don't require extra dependencies, such as JVMs or language interpreters, to run the applications.
- **Low startup time:** Another advantage of the Go binary format is low startup time (also called **cold start time**): Go applications often take just milliseconds to start.

The growth of the Go community has increased the rate of development of additional libraries for the language and resulted in the creation of the entire ecosystem of tools, powering application logging, debugging, and implementations of all widely used networking protocols and standards. For example, nearly 75 percent of Cloud Native Computing Foundation projects are written in Go. Additionally, Go is the primary language of some of the most popular development and orchestration software, including Docker and Kubernetes, which we will cover in *Chapter 8* of this book. The Go developer community keeps growing and the rate of new releases is only accelerating.

Summary

We have discussed the key aspects of the microservice development model, including the motivation to use it, the common benefits, and the possible challenges. You have learned that the microservice model brings many advantages, helping you achieve a higher degree of flexibility. It also comes with its own costs, which often include additional complexity and a lack of uniformity in the system. Microservice architecture requires you to think about these problems proactively in order to address them before they become big issues.

In the next chapter, we are going to start our journey into microservice development with the Go language. You will learn the important basics of the Go programming language and we will scaffold our microservices, which we are going to improve throughout the rest of the book.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- A collection of resources on microservice development: <https://microservices.io>
- Overview of the microservice architecture model: <https://martinfowler.com/articles/microservices.html>

Part 2

Foundation

This part covers the foundational aspects of Go microservice development, such as service discovery, data serialization, synchronous and asynchronous communication, deployment, testing, and security. You will learn how to scaffold Go microservices, establish communication between them, store service data, and implement service APIs, as well as many other important aspects of microservice development.

This part of the book includes the following chapters:

- *Chapter 2, Scaffolding a Go Microservice*
- *Chapter 3, Service Discovery*
- *Chapter 4, Serialization*
- *Chapter 5, Synchronous Communication*
- *Chapter 6, Asynchronous Communication*
- *Chapter 7, Storing Service Data*
- *Chapter 8, Setting Up Service Deployments*
- *Chapter 9, Unit and Integration Testing*
- *Chapter 10, Security and Compliance*

2

Scaffolding a Go Microservice

In this chapter, we will finally start scaffolding our microservice code. The goal of this chapter is to establish a solid foundation for writing Go microservices and setting the right structure for future changes. While Go makes it relatively easy to write small applications, there are multiple challenges that engineers may face along the way, including the following:

- Setting the right project structure to make it easier to evolve and maintain the code base
- Writing idiomatic Go code that is going to be consistent with common conventions and style guidelines that are widely accepted by the Go community
- Separating the components of a microservice and wiring them together

We are going to address each of these challenges. First, you will be introduced to the key aspects of writing idiomatic and conventional Go code. You will learn important recommendations for writing and organizing your code base, as well as how to set up the proper code structure for your services. Then, we are going to introduce you to an example application, which will consist of three microservices that we are going to use throughout the book. In the following chapters, we will add additional features to these services, illustrating all the important areas of microservice development.

In this chapter, we will cover the following topics:

- Go basics
- Project structure
- Scaffolding an example application

Technical requirements

To complete this chapter, you need to have Go 1.18 or above. If you don't have Go installed, you can download it from the official website at <https://go.dev/dl/>.

You can find the code examples for this chapter on GitHub: <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter02/src>.

Go basics

Go is a great language for writing microservices. It is relatively easy to learn and has a pretty smooth learning curve, making onboarding new engineers easier. While you may have already had some experience with Go, one of the purposes of this book is to provide enough information for all types of developers – from beginners to highly experienced professionals.

In this section, we are going to summarize important concepts of the language. If you already have experience with Go, you can still quickly scan through this part. It also includes some useful recommendations and best practices commonly missed even by experienced engineers.

Core principles

Before we proceed to look at the basics of Go, I'm going to share with you some fundamental principles that will help you make decisions when writing and organizing your code. These principles include the following:

- *Always follow the official guidelines.* It is not uncommon for us engineers to have strong opinions about various styling and coding practices. However, in any developer community, consistency is more important than individual opinions. Make sure you get familiar with the most fundamental Go programming guidelines, written by the Go team:
 - **Effective Go:** This is an official set of guidelines for Go developers and it can be found at https://go.dev/doc/effective_go
 - **Go Code Review Comments:** This is another useful source of information on Go development, covering various aspects including code style, naming, and error handling, and it can be found at <https://go.dev/wiki/CodeReviewComments>
- *Follow the style used in the standard library.* The standard Go library, which comes with any Go installation, is the best source of code examples and comments. Get familiar with some of the packages from the library, such as context and net. Following the coding style used in these packages will help you write consistent, readable, and maintainable code, regardless of who will be using it later.

- *Do not try to apply the ideas from other languages to Go.* Go does not support some features of other languages, such as C++/Java-style objects and object inheritance. Because of this, Go code is generally structured differently from its Java or C++ counterparts. If you are new to the language, try to understand the philosophy of Go by getting familiar with the implementation of some popular Go packages – you can check the `net` package for some good examples: <https://pkg.go.dev/net>.

Now, as we are aligned on the core principles, let's move on to the key recommendations for writing conventional and idiomatic Go code.

Writing idiomatic Go code

This section summarizes the key topics described in the *Effective Go* document. Following the suggestions provided in this section will help you to keep your code consistent with the official guidelines.

Naming

Naming is one of the most important aspects of Go development, as clear and concise names directly impact the readability and maintainability of your code, as well as help to maintain consistency with other packages. Writing Go code in an idiomatic way requires an understanding of its core naming principles:

- Exported names start with an uppercase character.
- When a variable, struct, or interface is imported from another package, its name includes a package name or alias, for example, `bytes.Buffer`.
- Since references include package names, you should not prefix your names with the package name. If the package name is `xml`, use the name `Reader`, not `XMLReader` – in the second case, the full name would be `xml.XMLReader`.
- Packages are generally given lowercase, single-word names.
- It is not idiomatic to start the names of getters with the `Get` prefix. If your function returns the user's age, call the function `Age()`, not `GetAge()`. Using the `Set` prefix, however, is fine; you can safely call your function `SetAge()`.
- Single-method interfaces are named using the method name plus an `er` suffix. For example, an interface with a `Write` function would be called `Writer`.
- Initialisms and acronyms should have a consistent case. The correct versions would be `URL`, `url`, `ID`, and `id` – `Url` and `Id` would be incorrect.
- Variable names should be short rather than long. In general, follow this simple rule – the closer to declaration a name is used, the shorter it should be. For iterating over an array, use `i` for the index variable.

Additional naming recommendations include the following:

- The package name should be short, concise, and evocative and should provide context for its contents, for example, `json`.
- Keep the contents of a package consistent with the name. If you start noticing that a package includes extra logic that has no relationship to the package name, consider exporting it to a separate one or using a more descriptive name.
- Use name abbreviations only if they are widely used (for example, `fmt` or `cmd`).
- Avoid name collisions when possible. For example, if you introduce a set of string functions, avoid calling it `strings` because a package with the same name exists in the Go standard library and is already widely used.
- Consider the client's point of view when giving names to your code. Think about how the code is going to be used when giving a name to it, for example, the `Writer` interface for proving the `Write` function. This helps to make your Go APIs more intuitive and easier to use.

In addition to these rules, remember to keep the naming consistent across your code base. It will help make it easier to read and write new code – good names will act as examples for other engineers as well.

Comments

Comments are the next important aspect of Go development. Go comments can be used in two different ways:

- Seeing the comments alongside the code
- Viewing the package documentation generated by the `godoc` tool

General principles for Go comments include the following:

- Every package should have a comment describing its contents
- Every exported name in Go should have a comment
- Comments should be complete sentences and end with a period
- The first sentence of the comment should start with the name being exported and provide a summary of it, as in the following example:

```
// ErrNotFound is returned when the record is not found.  
var ErrNotFound = errors.New("not found")
```

The Go standard library provides many good examples of code comments, so I always suggest getting familiar with some examples from it.

Errors

General recommendations for Go errors include the following:

- Only use panics in truly exceptional cases. Usage of panics makes the code harder to debug and recover from, so it's best to use them only in situations when your code can't continue to execute safely (for example, when an application is out of memory).
- Always handle each error; don't discard errors by using the `_` assignment.
- Error strings should start with a lowercase character, unless they begin with names requiring capitalization, such as acronyms.
- Error strings, unlike comments, should not end with punctuation marks, as in the following example:

```
return errors.New("user not found")
var errUserNotFound = errors.New("user not found")
```

- When calling a function returning an error, always handle the error first.
- Wrap errors if you want to add additional information to the clause. The conventional way of wrapping errors in Go is to use `%w` at the end of the formatted error:

```
if err != nil {
    return fmt.Errorf("upload failed: %w", err)
}
```

- While checking for errors, using the `==` operator may result in improper handling of the wrapped errors. There are two solutions for this. For a comparison to a sentinel error, such as `errors.New("some error")`, use `errors.Is`:

```
if errors.Is(err, ErrNotFound) {
    // err or some error it wraps is ErrNotFound.
}
```

For error types, use `errors.As`:

```
var e *QueryError
if errors.As(err, e) {
    // err has *QueryError type.
}
```

Additionally, keep errors descriptive yet compact. It should always be easy to understand what exactly went wrong by reading the error message.

Interfaces

Key principles of Go **interfaces** include the following:

- Do not define interfaces before they are used without a realistic example of usage to avoid unnecessary abstractions.
- Prefer defining interfaces on the receiving side, not where the types that implement them are defined. This practice offers lots of benefits, including more simple testing and mocking of dependencies, which we will cover in *Chapter 9*.
- Single-method interfaces should be called by the method name and er suffix, for example, the `Writer` interface with a `Write` function.

See some built-in interfaces, such as `Writer` and `Reader`, to get a good example of defining and using interfaces in Go.

Tests

We are going to cover testing in detail in *Chapter 9, Unit and Integration Testing*. Here are some key suggestions for writing Go tests in an idiomatic way:

- Tests should always provide information to the user on what exactly went wrong in case of a failure.
- Consider writing table-driven tests when you want to test the behavior of some function against a set of various inputs to enhance test readability and maintainability. See this example: https://github.com/golang/go/blob/master/src/fmt/errors_test.go.
- Generally, we should test only public functions. Your private function should be indirectly tested through public ones.

Make sure you always write tests for your code. Not only does this help with finding bugs earlier but it also helps to see how your code can be used. I find the latter especially useful.

Context

One of the key differences between the Go language and other popular languages is explicit context propagation. **Context propagation** is a mechanism for propagating an additional call argument, called **context**, into function calls, passing additional metadata.

Go context has a type called `context.Context`. There are multiple ways of using it:

- **Cancellation logic:** You can pass a special instance of a context that can get *canceled*. In that case, all functions you would call with it would be able to detect this. Such logic can be useful for handling application shutdown or stopping any processing.
- **Timeouts:** You can set the timeouts for your execution by using the corresponding context functions.
- **Propagating extra metadata:** You can propagate additional key-value metadata inside the context. This way, any downstream functions called would receive that metadata inside the context object. There are some useful applications of this approach, one of which is distributed tracing, which we are going to cover in the following chapters.

We will get back to context propagation in the following chapters. Now, we can define some important aspects of using context in Go:

- Context is immutable but can be cloned with extra metadata
- Functions using context should accept it as their first argument

Additionally, some context best practices are as follows:

- Always pass context to functions performing I/O calls.
- Limit the usage of context for passing any metadata. You should use metadata propagation for truly exceptional cases, such as distributed tracing, as mentioned earlier.
- Do not attach context to structures by storing it as a field. Such a practice is called **context leakage** and might result in unexpected timeouts or cancellations, as well as incorrect propagation of values. Instead, always pass it through function calls.

Now, as we have discussed the key recommendations for writing idiomatic Go code, we can move on to the next section, which is going to cover the project structure recommendations and standards for Go applications.

Project structure

The project structure is the foundation of your code that plays a key role in its readability and maintainability. As we discussed in the previous sections, in Go projects, the structure may play a more important role than in other languages, because each exported name generally includes the name of its package. This requires you to have good and descriptive naming for your packages and directories, as well as the right hierarchy of your code.

While the official guidelines define some strong recommendations for naming and coding style, there aren't that many rules constraining the Go project structure. Each project is unique by nature, and developers are generally free to choose the way they organize the code. However, there are some common practices and specifics of Go package organization that we are going to cover in this section.

Private packages

In Go, all code stored inside a directory called `internal` can be imported and used only by packages stored within the same directory or one of the directories it includes. Putting code into an internal directory can ensure your code is not exported and used by external packages. This can be useful for the following different cases:

- Hide the details of the implementation from the user if some of the types of functions need to be exported
- Ensure no external package relies on your types and functions, which you don't want to expose widely
- Remove possible unnecessary dependencies between the packages
- Avoid extra refactoring and maintenance difficulties if your code is unexpectedly used by other developers/teams

I have found it useful to use internal packages as a protection against unwanted dependencies. This plays a big role in large repositories and applications, where there is a high possibility of unexpected dependencies between the packages. Large code bases that don't have a separation between private and public packages often suffer from an effect called *spaghettification* – when packages depend on each other in an uncontrolled and chaotic way.

Public packages

There is another type of directory name with a semantic meaning in Go – a directory called `pkg`. It implies that it is OK to use the code from this package externally.

The `pkg` directory isn't recommended officially, but it is widely used. Ironically, the Go team used this in the library code and then got rid of this pattern, while the rest of the Go community adopted it so widely that it became a common practice.

It is up to you whether you use a `pkg` directory in your applications. In small projects, it might be unnecessary and redundant. In larger code bases, in tandem with the `internal` directory, it can help to organize your code so that it is clear what code can and cannot be reused, easing the code navigation for the developers.

Executable packages

The `cmd` package is commonly used in the Go community to store the code of one or multiple executable packages having a `main` function. This may include the code starting your application or any code for your executable tools. For a single-app directory, you can store your Go code directly in the `cmd` package:

```
cmd/  
cmd/main.go
```

For a multi-app directory, you can include sub-packages in `cmd` packages:

```
cmd/  
cmd/indexer/main.go  
cmd/crawler/main.go
```

Other commonly used directories

The following list includes some other commonly used directory or package names in the Go community:

- `api`: JSON schema files and definitions in various protocols, including gRPC. We are going to cover these topics in *Chapter 4*.
- `testdata`: Files containing the data used in tests.
- `web`: Web application components and assets.

Common files

Here is a list of common filenames, which will keep your packages consistent with the official library and lots of third-party libraries:

- `main.go`: A file containing the `main()` function
- `doc.go`: Package documentation (a separate file is not necessary for small packages)
- `*_test.go`: Test files
- `README.md`: A README file written in the Markdown language
- `LICENSE`: A license file, if there is one
- `CONTRIBUTING.md`: Instructions on how to contribute to the project (very common in open source projects)
- `CONTRIBUTORS.md/AUTHORS`: List of contributors and/or authors
- `CHANGELOG.md`: Chronological list of changes made to the project
- `SECURITY.md`: Guidelines on how to report security vulnerabilities

Now, let's cover the best practices for organizing the code base for Go applications.

Best practices

In this section, you can find a list of best practices for organizing the Go application project structure. It is going to help you keep your code aligned with thousands of other Go packages and keep it conventional and idiomatic. The best practices of Go project organization include the following:

- Separate private code using an `internal` directory to avoid its accidental usage by external packages.
- Get familiar with the way popular open source Go projects, such as <https://github.com/kubernetes/kubernetes>, are organized. This can provide you with great examples of how to structure your repository.
- Split the code in a sufficiently granular way to keep the balance between readability and maintainability. Don't split the packages too early but also avoid having a lot of logic in a single package. Generally, you will find that the easier it is to give a short and specific self-descriptive name to a package, the better your code composition is.
- Avoid long package names due to poor readability.
- Always be ready to change the structure if requirements are changed or if the structure no longer reflects the package name/original intent.

This sums up the part of the chapter describing the core principles and best practices of Go code organization. Now, we are ready to get to the practical side of this chapter.

Scaffolding an example application

We have covered the general recommendations for writing and organizing Go applications and we are finally ready to start writing the code! In this section, we are going to introduce an application, consisting of multiple microservices that are going to be used throughout the book. In each chapter, we are going to add to or improve them, converting them from small examples into production-grade services that are ready to be used.

You will learn how to scaffold microservice code and split the code into separate logical parts, each having its own role. We are going to apply the project structure and Go knowledge you gained in this chapter to illustrate how to set the right structure for each service and write its code in a conventional and idiomatic way.

Movie application

Let's imagine we are building an application for movie lovers. The application would provide the following features:

- A feature to get the movie metadata (such as title, year, description, and director) and the aggregated movie rating
- A feature to rate a movie

The listed features seem to be closely related. However, let's take a closer look at them.

Movie metadata

Let's assume we have the metadata for a collection of movies, which includes the following fields:

- ID
- Title
- Year
- Description
- Director
- List of actors

Such information about movies doesn't generally change unless somebody wants to update the description, but for simplicity, we will assume that we are dealing with a static dataset. We would retrieve the records based on their IDs, so we could use any key-value or document database to store and access the metadata.

Ratings

Let's now review the functionality required for storing and retrieving movie ratings.

Generally, we would need to perform the following rating operations:

- Store a movie rating
- Get the aggregated movie rating

Later, we would also need to support rating deletion, but for now, we can just keep this logic in mind while designing the application.

The ratings data is quite different from the movie metadata – we can both append and delete the records. In addition to this, we need to return the aggregated rating, so we should either be able to return all stored ratings for an item and perform the aggregation on the go or have separate logic for performing and storing the aggregations. You will notice that the ways we access ratings and movie metadata are different. This hints that the ratings data can, and probably should, be stored separately from the movie metadata.

While designing the application, it is beneficial to think one step further and imagine how the application may evolve in the future. This does not mean that you should necessarily build the application trying to predict future use cases because it can lead to unnecessary abstractions that may not be needed later if your plans change. However, thinking one step ahead may save you time later if you find efficient ways of modeling and storing your data, which would help you to adapt to changing requirements.

Let's see how the rating service might evolve. At some point, we may want to extend the rating functionality to other types of movie-related records. A user may be able to do the following:

- Rate an actor's performance in some movies
- Rate the movie soundtrack
- Rate the movie's costume design

When making decisions on supporting future use cases, you should ask yourself how likely it is that you will need to implement that logic in the observable future (6–12 months). You should generally avoid thinking much further ahead because the requirements and goals may change. However, if you are quite certain you have plans to support particular features, you should make sure your data model would support those features without major changes.

Let's assume we want to implement the additional ratings mentioned earlier. In this case, we want to make sure we can design our application in a way that would support the ratings for different types of objects.

Let's define the API for such a rating component:

- Store the rating record, including the following:
 - ID of the user who gave the rating
 - ID and type of the record being rated
 - Rating value
- Get the aggregated rating for a record by its ID and type

This API supports record types, so we can easily add more types of ratings without changing the system. The trade-off we made here is quite reasonable – the API is different by just one field (record type) from the API of a rating system designed just for movies. However, this gives us complete freedom in introducing new rating types in the future! Such a trade-off seems very reasonable given that we have decided we would certainly need those ratings in the future.

Should we split the application?

Let's provide a summary of the two parts of the application we have just described:

- Movie metadata:
 - Retrieve the metadata for a movie by its ID
- Ratings:
 - Store a rating for a record
 - Retrieve an aggregated rating for a record

After we abstracted the rating component by letting it support various record types, it stopped being a movie rating component and became a more generic record rating system. The movie metadata component is now loosely coupled to the rating system – the rating system can store the ratings for movies as well as for any other possible types of records.

As we discussed previously, the data models for both components are also quite different. The movie metadata component stores static data, which is going to be retrieved by ID, while the rating component stores dynamic data, which requires aggregation.

Both components seem to be relatively independent of each other. This is a perfect example of a situation where we may benefit from splitting the application into separate services:

- Logic is loosely coupled
- Data models are different
- Data is generally independent

This list is not complete, and you need to consider all the aspects described in *Chapter 1* to make a decision on splitting the application. However, since this book covers microservice development, let's make our decision here and decide to split the system into separate services.

Let's list the services we would split the application into:

- **Movie metadata service:** Store and retrieve the movie metadata records by movie ID
- **Rating service:** Store ratings for different types of records and retrieve aggregated ratings for records
- **Movie service:** Provide complete information to the callers about a movie or a set of movies, including the movie metadata and its rating

Why did we end up with three services here? We did this for the following reasons:

- The movie metadata service will be solely responsible for accessing the movie metadata records.
- The movie service will provide the client-facing API, aggregating two separate types of records – movie metadata and ratings. The records will be stored in two separate systems, so this component will join them together and return to the caller.
- If we introduce any other types of records in the system, such as likes, reviews, and recommendations, we will plug them into the movie service, not the movie metadata service. The movie metadata service will be used solely for accessing the static movie metadata, not any other types of records.
- Movie metadata services can potentially evolve in the future by getting more metadata-related functionality, such as editing or adding descriptions in different languages. This also hints that it is better to keep this component solely for the metadata-related features.

Let's illustrate these services in a diagram:

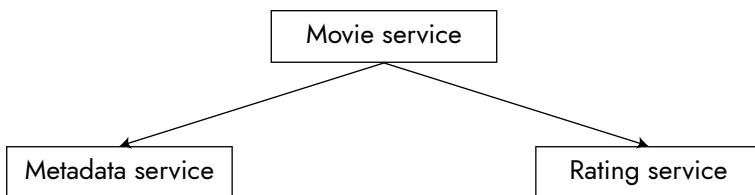


Figure 2.1 – Movie application services

Now that we have the definitions of the three microservices, let's finally proceed to the coding part.

Application code structure

Let's align on how we are going to structure the code of all microservices in relation to each other. I would suggest storing them inside a single directory, which would be our application root. Create a new directory (you may call it `movieapp`), and inside it, create the following directories for our microservices:

- `rating`
- `metadata`
- `movie`

Throughout the book, I will use the directory paths relative to the application directory you've created, so when you see a directory or filename, assume it is stored in the app directory you chose for this.

From the *Project structure* section, we know that the logic containing the `main` function generally resides in the `cmd` directory. We will use this approach in our microservices – for example, the main file for the rating service would be called `rating/cmd/main.go`.

Each service may contain one or multiple packages related to the following logical roles:

- API handlers
- Business/application logic
- Database logic
- Interaction with other services

Note that handler and business/application logic are separate, even though the primary purpose of the application may be to handle the API requests. This is not absolutely necessary, but it's a relatively good practice to separate the business logic from the API handling layer. This way, if you migrate from one type of API to another (for example, from HTTP to gRPC) or support both, you don't need to implement the same logic twice or rewrite it. Instead, you would just call the business logic from your handler, keeping the handler as simple as possible and making its primary purpose to pass the requests to the relevant interfaces.

We can illustrate this relationship with the help of a diagram:

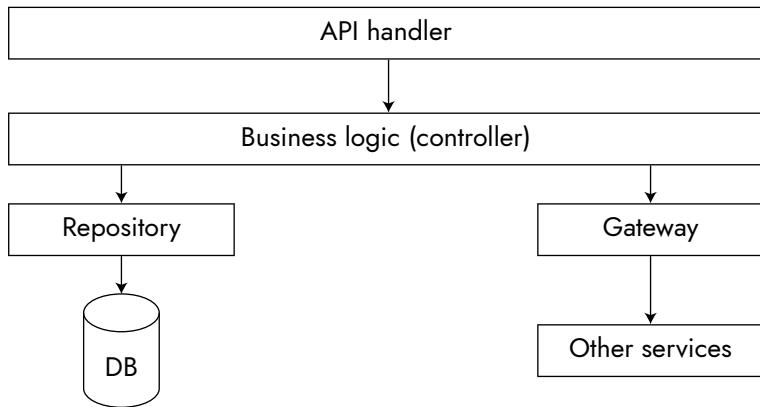


Figure 2.2 – Layers of a service

As you can see in the diagram, the API handler does not access the database directly. Instead, the database access is performed on a business logic layer.

There is no convention in the Go community on how to call packages serving these purposes, so we are free to choose the names for our packages providing such logic. It is, however, important that you keep these names consistent across all your microservices, so let's align on a common naming convention for these types of packages.

In this book, we are going to use the following names for our application components:

- **controller:** Business logic
- **gateway:** Logic for interacting with other services
- **handler:** API handlers
- **repository:** Database logic

Now, since we are aligned on the naming, let's proceed to the last step of setting up our project. Execute this command in the application root directory:

```
go mod init movieexample.com
```

This command creates a Go module called `movieexample.com`. A Go module is a collection of related packages stored in a file tree. They help manage dependencies for your project, and we are going to use this feature in all the chapters.

Now, we can proceed to code scaffolding for our first microservice.

Movie metadata service

Let's summarize the logic of the movie metadata service:

- **API:** Get metadata for a movie
- **Database:** Movie metadata database
- **Interacts with services:** None
- **Data model type:** Movie metadata

This logic would translate into the following packages:

- `cmd`: Contains the `main` function for starting the service
- `controller`: Our service logic (reads the movie metadata)
- `handler`: API handler for a service
- `repository`: Logic for accessing the movie metadata database

Let's store the logic of our service in a directory called `metadata`. Following the conventions we described earlier in the chapter, the executable code containing the `main` file is going to be stored in the `cmd` package. All code that we are not going to export will be stored in the `internal` directory and this will include most of our applications. The exported structures will reside in the `pkg` directory.

Applying the rules that we just described, we are going to structure our packages in the following way:

- `metadata/cmd`
- `metadata/internal/controller`
- `metadata/internal/handler`
- `metadata/internal/repository`
- `metadata/pkg`

Once you have created the directories listed here, let's proceed to implement the code for our microservice.

Model

First, we are going to implement the structure for the movie metadata. Inside the `metadata/pkg` directory, create a `metadata.go` file using the following code:

```
package model

// Metadata defines the movie metadata.
type Metadata struct {
    ID      string `json:"id"`
    Title   string `json:"title"`
    Description string `json:"description"`
    Director string `json:"director"`
}
```

This structure is going to be used by the callers of our service. It includes JSON annotations, which we are going to use later in this chapter.

Repository

Now, let's create the stub logic for handling the database logic. Inside the `metadata/internal/repository` directory, add an `error.go` file using the following code:

```
package repository

import "errors"

// ErrNotFound is returned when a requested record is not found.
var ErrNotFound = errors.New("not found")
```

This file defines an error for the case when the record is not found. We are going to use this error in our implementation.

In the next step, we are going to add the repository implementation. Even if you have some specific technology to work with for storing the data, it is often useful to provide more than one implementation of the database logic. I always find it useful to include an in-memory implementation of the database logic that can be used for testing and local development, reducing the need for any additional databases or extra libraries. I am going to illustrate how to do this.

Inside the `metadata/internal/repository` directory, create a directory called `memory` that will contain the in-memory implementation of our movie metadata database. Add a `memory.go` file to it, using the following code:

```
package memory

import (
    "context"
    "sync"

    "movieexample.com/metadata/internal/repository"
    "movieexample.com/metadata/pkg/model"
)

// Repository defines a memory movie metadata repository.
type Repository struct {
    sync.RWMutex
    data map[string]*model.Metadata
}

// New creates a new memory repository.
func New() *Repository {
    return &Repository{data: map[string]*model.Metadata{}}
}

// Get retrieves movie metadata by movie id.
func (r *Repository) Get(_ context.Context, id string) (*model.Metadata,
    error) {
    r.RLock()
    defer r.RUnlock()
    m, ok := r.data[id]
    if !ok {
        return nil, repository.ErrNotFound
    }
    return m, nil
}

// Put adds movie metadata for a given movie id.
```

```
func (r *Repository) Put(_ context.Context, id string, metadata *model.
Metadata) error {
    r.Lock()
    defer r.Unlock()
    r.data[id] = metadata
    return nil
}
```

Let's highlight some aspects of the code we've just added:

- First, we called the `Repository` structure because it provides a good name to the users when combined with the name of its package – `memory.Repository`.
- Second, we used the exported `ErrNotFound` that we previously defined, so callers can check their code. It is usually good practice to do so because it allows the developers to check for a specific error in their code. We will illustrate how to write tests for it in *Chapter 8, Testing*.
- Additionally, the function creating the repository is called `New`. This is often a good name for short packages when there is just one type being created.
- Our `Get` and `Put` functions accept `context` as the first argument. We mentioned this approach in the *Writing idiomatic Go code* section – all functions performing I/O operations must accept `context`.
- Our implementation is using a `sync.RWMutex` structure to protect against concurrent writes and reads.

Now, let's move on to the business logic layer.

Controller

The next step is to add a controller to encapsulate our business logic. Even if your logic is trivial, it is still a good practice to keep it separate from the handler from the beginning. This will help you avoid further changes and, more importantly, keep the structure of your applications consistent.

Inside the `metadata/internal/controller` package, a directory called `metadata`. Inside it, add a `controller.go` file with the following logic:

```
package metadata

import (
    "context"
    "errors"

    "movieexample.com/metadata/internal/repository"
    "movieexample.com/metadata/pkg/model"
)

// ErrNotFound is returned when a requested record is not found.
var ErrNotFound = errors.New("not found")

type metadataRepository interface {
    Get(ctx context.Context, id string) (*model.Metadata, error)
}

// Controller defines a metadata service controller.
type Controller struct {
    repo metadataRepository
}

// New creates a metadata service controller.
func New(repo metadataRepository) *Controller {
    return &Controller{repo}
}

// Get returns movie metadata by id.
func (c *Controller) Get(ctx context.Context, id string) (*model.Metadata, error) {
    res, err := c.repo.Get(ctx, id)
    if err != nil && errors.Is(err, repository.ErrNotFound) {
        return nil, ErrNotFound
    }
    return res, err
}
```

The controller we created is currently just a wrapper around the repository. However, the controller will generally have more logic, so it is preferable to keep it separate.

Handler

Now, we are going to create the API handler. Inside the `metadata/internal/handler` directory, create a directory called `http`. Inside it, create a file called `http.go` with the following logic:

```
package http

import (
    "encoding/json"
    "errors"
    "log"
    "net/http"

    "movieexample.com/metadata/internal/controller/metadata"
    "movieexample.com/metadata/internal/repository"
)

// Handler defines a movie metadata HTTP handler.
type Handler struct {
    ctrl *metadata.Controller
}

// New creates a new movie metadata HTTP handler.
func New(ctrl *metadata.Controller) *Handler {
    return &Handler{ctrl}
}
```

Now, let's implement the logic for retrieving movie metadata:

```
// GetMetadata handles GET /metadata requests.
func (h *Handler) GetMetadata(w http.ResponseWriter, req *http.Request) {
    id := req.FormValue("id")
    if id == "" {
        w.WriteHeader(http.StatusBadRequest)
        return
    }
    ctx := req.Context()
    m, err := h.ctrl.Get(ctx, id)
    if err != nil && errors.Is(err, repository.ErrNotFound) {
```

```
w.WriteHeader(http.StatusNotFound)
return
} else if err != nil {
    log.Printf("Repository get error for movie %s: %v\n", id, err)
    w.WriteHeader(http.StatusInternalServerError)
    return
}
if err := json.NewEncoder(w).Encode(m); err != nil {
    log.Printf("Response encode error: %v\n", err)
    w.WriteHeader(http.StatusInternalServerError)
}
}
```

The handler we just created uses our repository to retrieve the information and return it in JSON format. We chose JSON here just for simplicity. In *Chapter 4*, we are going to cover more data formats and illustrate how they can benefit your applications.

You may notice that we have called the package for our HTTP handler `http`. There is a trade-off here – while we are certainly colliding with its namesake standard library package, we get a pretty descriptive `http.Handler` exported name. Since our package is going to be used internally, this trade-off is reasonable.

Main file

Now, since we have created both a database and an API handler, let's create the executable for the metadata service. Inside the `metadata/cmd` directory, create the `main.go` file and add the following code:

```
package main

import (
    "log"
    "net/http"

    "movieexample.com/metadata/internal/controller/metadata"
    httphandler "movieexample.com/metadata/internal/handler/http"
    "movieexample.com/metadata/internal/repository/memory"
)
```

```
func main() {
    log.Println("Starting the movie metadata service")
    repo := memory.New()
    ctrl := metadata.New(repo)
    h := httphandler.New(ctrl)
    http.Handle("/metadata", http.HandlerFunc(h.GetMetadata))
    if err := http.ListenAndServe(":8081", nil); err != nil {
        panic(err)
    }
}
```

The function we just created initializes all structures of our service and starts the http API handler we implemented earlier. The service is ready to process user requests, so let's move on to the other services.

Rating service

Let's summarize the logic of the rating service:

- **API:** Get the aggregated rating for a record and write a rating
- **Database:** Rating database
- **Interacts with services:** None
- **Data model type:** Rating

This logic would translate into the following packages:

- **cmd:** Contains the `main` function for starting the service
- **controller:** Our service logic (read and write ratings)
- **handler:** API handler for a service
- **repository:** Logic for accessing the movie metadata database

We are going to use exactly the same directory structure as we used for the metadata service:

- `rating/cmd`
- `rating/internal/controller`
- `rating/internal/handler`
- `rating/internal/repository`
- `rating/pkg`

Once you have created these directories, let's move on to the implementation of the service.

Model

Create a `model` directory inside `rating/pkg` and create a `rating.go` file, using the following code:

```
package model

// RecordID defines a record id. Together with RecordType
// identifies unique records across all types.
type RecordID string

// RecordType defines a record type. Together with RecordID
// identifies unique records across all types.
type RecordType string

// Existing record types.
const (
    RecordTypeMovie = RecordType("movie")
)

// UserID defines a user id.
type UserID string

// RatingValue defines a value of a rating record.
type RatingValue int

// Rating defines an individual rating created by a user for
// some record.
type Rating struct {
    RecordID     string      'json:"recordId"'
    RecordType   string      'json:"recordType"'
    UserID       UserID      'json:"userId"'
    Value        RatingValue 'json:"value"'
}
```

The file contains the model for our rating service, which is also going to be used by other services interacting with it. Note that we created separate types, `RecordID`, `RecordType`, and `UserID`. This will help us with readability and add extra type protection, as you will see in the implementation.

Repository

Create the in-memory implementation for our rating repository inside the `rating/internal/repository/memory/memory.go` file:

```
package memory

import (
    "context"

    "movieexample.com/rating/internal/repository"
    "movieexample.com/rating/pkg/model"
)

// Repository defines a rating repository.
type Repository struct {
    data map[model.RecordType]map[model.RecordID][]model.Rating
}

// New creates a new memory repository.
func New() *Repository {
    return &Repository{map[model.RecordType]map[model.RecordID][]model.
Rating{}}
}
```

Then, add an implementation of the `Get` function to it, as shown in the following code block:

```
// Get retrieves all ratings for a given record.
func (r *Repository) Get(ctx context.Context, recordID model.RecordID,
recordType model.RecordType) ([]model.Rating, error) {
    if _, ok := r.data[recordType]; !ok {
        return nil, repository.ErrNotFound
    }
    if ratings, ok := r.data[recordType][recordID]; !ok || len(ratings) ==
0 {
        return nil, repository.ErrNotFound
    }
    return r.data[recordType][recordID], nil
}
```

Finally, let's implement a Put function inside it, as shown:

```
// Put adds a rating for a given record.
func (r *Repository) Put(ctx context.Context, recordID model.RecordID,
recordType model.RecordType, rating *model.Rating) error {
    if _, ok := r.data[recordType]; !ok {
        r.data[recordType] = map[model.RecordID][]model.Rating{}
    }
    r.data[recordType][recordID] =
append(r.data[recordType][recordID], *rating)
    return nil
}
```

The preceding implementation uses a nested map to store all records inside it. If we didn't define separate types (RatingID, RatingType, and UserID), it would be harder to understand the types of the keys in the map because we would be using primitives such as `string` and `int`, which are less self-descriptive.

Controller

Let's add a controller for our rating service. In the `rating/internal/controller/rating` package, create a `controller.go` file:

```
package rating

import (
    "context"
    "errors"

    "movieexample.com/rating/internal/repository"
    "movieexample.com/rating/pkg/model"
)

// ErrNotFound is returned when no ratings are found for a
// record.
var ErrNotFound = errors.New("ratings not found for a record")

type ratingRepository interface {
    Get(ctx context.Context, recordID model.RecordID, recordType model.
RecordType) ([]model.Rating, error)
```

```

        Put(ctx context.Context, recordID model.RecordID, recordType model.
RecordType, rating *model.Rating) error
    }

// Controller defines a rating service controller.
type Controller struct {
    repo ratingRepository
}

// New creates a rating service controller.
func New(repo ratingRepository) *Controller {
    return &Controller{repo}
}

```

Let's add functions for writing and getting an aggregated rating:

```

// GetAggregatedRating returns the aggregated rating for a
// record or ErrNotFound if there are no ratings for it.
func (c *Controller) GetAggregatedRating(ctx context.Context, recordID
model.RecordID, recordType model.RecordType) (float64, error) {
    ratings, err := c.repo.Get(ctx, recordID, recordType)
    if err != nil && err == repository.ErrNotFound {
        return 0, ErrNotFound
    } else if err != nil {
        return 0, err
    }
    sum := float64(0)
    for _, r := range ratings {
        sum += float64(r.Value)
    }
    return sum / float64(len(ratings)), nil
}

// PutRating writes a rating for a given record.
func (c *Controller) PutRating(ctx context.Context, recordID model.
RecordID, recordType model.RecordType, rating *model.Rating) error {
    return c.repo.Put(ctx, recordID, recordType, rating)
}

```

In this example, it is easy to see how the controller logic is different from the repository one. The repository provides an interface to get all ratings for a record and the controller implements the aggregation logic for them.

Handler

Let's implement the service handler in the `rating/internal/handler/http/http.go` file, using the following code:

```
package http

import (
    "encoding/json"
    "errors"
    "log"
    "net/http"
    "strconv"

    "movieexample.com/rating/internal/controller"
    "movieexample.com/rating/pkg/model"
)

// Handler defines a rating service controller.
type Handler struct {
    ctrl *rating.Controller
}

// New creates a new rating service HTTP handler.
func New(ctrl *rating.Controller) *Handler {
    return &Handler{ctrl}
}
```

Now, let's add a function for handling HTTP requests to our service:

```
// Handle handles PUT and GET /rating requests.
func (h *Handler) Handle(w http.ResponseWriter, req *http.Request) {
    recordID := model.RecordID(req.FormValue("id"))
    if recordID == "" {
        w.WriteHeader(http.StatusBadRequest)
        return
    }
```

```
recordType := model.RecordType(req.FormValue("type"))
if recordType == "" {
    w.WriteHeader(http.StatusBadRequest)
    return
}
switch req.Method {
case http.MethodGet:
    v, err := h.ctrl.GetAggregatedRating(req.Context(), recordID,
recordType)
    if err != nil && errors.Is(err, rating.ErrNotFound) {
        w.WriteHeader(http.StatusNotFound)
        return
    }
    if err := json.NewEncoder(w).Encode(v); err != nil {
        log.Printf("Response encode error: %v\n", err)
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
case http.MethodPut:
    userID := model.UserID(req.FormValue("userId"))
    v, err := strconv.ParseFloat(req.FormValue("value"), 64)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }
    if err := h.ctrl.PutRating(req.Context(), recordID, recordType,
&model.Rating{UserID: userID, Value: model.RatingValue(v)}); err != nil {
        log.Printf("Repository put error: %v\n", err)
        w.WriteHeader(http.StatusInternalServerError)
    }
default:
    w.WriteHeader(http.StatusBadRequest)
}
}
```

The handler we implemented handles both GET and PUT requests. Note the way we handle some special cases such as an empty id value from the request – in that case, we return a special error code, `http.StatusBadRequest`, indicating that the API request was invalid. If the record is not found, we return `http.StatusNotFound`, and if we encounter any unexpected errors when accessing our database, we return `http.StatusInternalServerError`.

Using such standard HTTP error codes helps clients differentiate between the types of errors and implement the logic for detecting and correctly handling such issues.

Main

Let's write the `main` file for our service. In `rating/cmd/main.go`, write the following logic:

```
package main

import (
    "log"
    "net/http"

    "movieexample.com/rating/internal/controller/rating"
    httphandler "movieexample.com/rating/internal/handler/http"
    "movieexample.com/rating/internal/repository/memory"
)

func main() {
    log.Println("Starting the rating service")
    repo := memory.New()
    ctrl := rating.New(repo)
    h := httphandler.New(ctrl)
    http.Handle("/rating", http.HandlerFunc(h.Handle))
    if err := http.ListenAndServe(":8082", nil); err != nil {
        panic(err)
    }
}
```

The `main` function we created is similar to the `main` function of the metadata service; it initializes all components of a service and starts an HTTP handler.

Now, we are ready to implement our last service.

Movie service

Let's summarize the logic of the movie service:

- **API:** Get the details for a movie, including the aggregated movie rating and movie metadata
- **Database:** None
- **Interacts with services:** Movie metadata and rating
- **Data model type:** Movie details

This logic would translate into the following packages:

- `cmd`: Contains the `main` function for starting the service
- `controller`: Our service logic (read rating and metadata)
- `gateway`: Logic for calling the other services
- `handler`: API handler for a service

The directory structure is as follows:

- `movie/cmd`
- `movie/internal/controller`
- `movie/internal/gateway`
- `movie/internal/handler`
- `movie/pkg`

Once you have created these directories, let's move on to the implementation of the service.

Model

Create a `model.go` file in the `movie/pkg/model` directory and write the following logic:

```
package model

import "movieexample.com/metadata/pkg/model"

// MovieDetails includes movie metadata its aggregated
// rating.
type MovieDetails struct {
    Rating    *float64    'json:"rating,omitempty"'
    Metadata model.Metadata 'json:"metadata"'
}
```

Note that the file imports the `model` package of a metadata service containing the `Metadata` structure that we can reuse in our service.

Gateways

In the previous examples, the services did not interact with each other and just provided an API for this. The movie service won't access any database by itself but instead is going to interact with both the movie metadata and the rating service.

Let's create the logic for interacting with both services.

First, let's create an error that we are going to use in our gateways. In the `movie/internal/gateway` package, create an `error.go` file, using the following code block:

```
package gateway

import "errors"

// ErrNotFound is returned when the data is not found.
var ErrNotFound = errors.New("not found")
```

Now, let's write an HTTP gateway for the movie metadata service. In the `movie/gateway/metadata/http` directory, create a `metadata.go` file:

```
package http

import (
    "context"
    "encoding/json"
    "fmt"
    "net/http"

    "movieexample.com/metadata/pkg/model"
    "movieexample.com/movie/internal/gateway"
)

// Gateway defines a movie metadata HTTP gateway.
type Gateway struct {
    addr string
}
```

```
// New creates a new HTTP gateway for a movie metadata
// service.
func New(addr string) *Gateway {
    return &Gateway{addr}
}
```

Let's implement a Get function in it:

```
// Get gets movie metadata by a movie id.
func (g *Gateway) Get(ctx context.Context, id string) (*model.Metadata,
error) {
    req, err := http.NewRequest(http.MethodGet, g.addr+"/metadata", nil)
    if err != nil {
        return nil, err
    }
    req = req.WithContext(ctx)
    values := req.URL.Query()
    values.Add("id", id)
    req.URL.RawQuery = values.Encode()
    resp, err := http.DefaultClient.Do(req)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()
    if resp.StatusCode == http.StatusNotFound {
        return nil, gateway.ErrNotFound
    } else if resp.StatusCode/100 != 2 {
        return nil, fmt.Errorf("non-2xx response: %v", resp)
    }
    var v *model.Metadata
    if err := json.NewDecoder(resp.Body).Decode(&v); err != nil {
        return nil, err
    }
    return v, nil
}
```

Now, let's write an HTTP gateway for the rating service. In the `movie/gateway/rating/http` directory, create a `rating.go` file:

```
package http

import (
    "context"
    "encoding/json"
    "fmt"
    "net/http"

    "movieexample.com/movie/internal/gateway"
    "movieexample.com/rating/pkg/model"
)

// Gateway defines an HTTP gateway for a rating service.
type Gateway struct {
    addr string
}

// New creates a new HTTP gateway for a rating service.
func New(addr string) *Gateway {
    return &Gateway{addr}
}
```

Let's add logic for getting the aggregated rating:

```
// GetAggregatedRating returns the aggregated rating for a
// record or ErrNotFound if there are no ratings for it.
func (g *Gateway) GetAggregatedRating(ctx context.Context, recordID model.
RecordID, recordType model.RecordType) (float64, error) {
    req, err := http.NewRequest(http.MethodGet, g.addr+"/rating", nil)
    if err != nil {
        return 0, err
    }
    req = req.WithContext(ctx)
    values := req.URL.Query()
    values.Add("id", string(recordID))
    values.Add("type", fmt.Sprintf("%v", recordType))
```

```

req.URL.RawQuery = values.Encode()
resp, err := http.DefaultClient.Do(req)
if err != nil {
    return 0, err
}
defer resp.Body.Close()
if resp.StatusCode == http.StatusNotFound {
    return 0, gateway.ErrNotFound
} else if resp.StatusCode/100 != 2 {
    return 0, fmt.Errorf("non-2xx response: %v", resp)
}
var v float64
if err := json.NewDecoder(resp.Body).Decode(&v); err != nil {
    return 0, err
}
return v, nil
}

```

Finally, let's add a function for handling a rating creation request:

```

// PutRating writes a rating.
func (g *Gateway) PutRating(ctx context.Context, recordID model.RecordID,
recordType model.RecordType, rating *model.Rating) error {
    req, err := http.NewRequest(http.MethodPut, g.addr+="/rating", nil)
    if err != nil {
        return err
    }
    req = req.WithContext(ctx)
    values := req.URL.Query()
    values.Add("id", string(recordID))
    values.Add("type", fmt.Sprintf("%v", recordType))
    values.Add("userId", string(rating.UserID))
    values.Add("value", fmt.Sprintf("%v", rating.Value))
    req.URL.RawQuery = values.Encode()
    resp, err := http.DefaultClient.Do(req)
    if err != nil {
        return err
    }
}

```

```
    defer resp.Body.Close()
    if resp.StatusCode/100 != 2 {
        return fmt.Errorf("non-2xx response: %v", resp)
    }
    return nil
}
```

At this point, we have both gateways and can implement the controller aggregating the data from them.

Controller

In the `movie/internal/controller/movie` directory, create a `controller.go` file:

```
package movie

import (
    "context"
    "errors"

    metadata "movieexample.com/metadata/pkg/model"
    "movieexample.com/movie/internal/gateway"
    "movieexample.com/movie/pkg/model"
    ratingmodel "movieexample.com/rating/pkg/model"
)

// ErrNotFound is returned when the movie metadata is not
// found.
var ErrNotFound = errors.New("movie metadata not found")
```

Let's define the interfaces for the services we will be calling:

```
type ratingGateway interface {
    GetAggregatedRating(ctx context.Context, recordID ratingmodel.RecordID, recordType ratingmodel.RecordType) (float64, error)
    PutRating(ctx context.Context, recordID ratingmodel.RecordID, recordType ratingmodel.RecordType, rating *ratingmodel.Rating) error
}

type metadataGateway interface {
    Get(ctx context.Context, id string) (*metadata.Metadata, error)
}
```

Now, we can define our service controller:

```
// Controller defines a movie service controller.
type Controller struct {
    ratingGateway    ratingGateway
    metadataGateway metadataGateway
}

// New creates a new movie service controller.
func New(ratingGateway ratingGateway, metadataGateway metadataGateway) *Controller {
    return &Controller{ratingGateway, metadataGateway}
}
```

Finally, let's implement the function for getting the movie details, including both its rating and metadata:

```
// Get returns the movie details including the aggregated
// rating and movie metadata.
// Get returns the movie details including the aggregated rating and movie
metadata.
func (c *Controller) Get(ctx context.Context, id string) (*model.MovieDetails, error) {
    metadata, err := c.metadataGateway.Get(ctx, id)
    if err != nil && errors.Is(err, gateway.ErrNotFound) {
        return nil, ErrNotFound
    } else if err != nil {
        return nil, err
    }
    details := &model.MovieDetails{Metadata: *metadata}
    rating, err := c.ratingGateway.GetAggregatedRating(ctx, ratingmodel.RecordID(id), ratingmodel.RecordTypeMovie)
    if err != nil && !errors.Is(err, gateway.ErrNotFound) {
        // Just proceed in this case, it's ok not to have ratings yet.
    } else if err != nil {
        return nil, err
    } else {
        details.Rating = &rating
    }
    return details, nil
}
```

Note that we redefine ErrNotFound in different components. While we could have just exported it to some shared package, sometimes it is better to keep it independent. Otherwise, we may confuse one error with another (for example, rating not found or metadata not found).

Handler

In the `movie/internal/handler/http` package, add the `http.go` file, using the following logic:

```
package http

import (
    "encoding/json"
    "errors"
    "log"
    "net/http"

    "movieexample.com/movie/internal/controller/movie"
)

// Handler defines a movie handler.
type Handler struct {
    ctrl *movie.Controller
}

// New creates a new movie HTTP handler.
func New(ctrl *movie.Controller) *Handler {
    return &Handler{ctrl}
}

// GetMovieDetails handles GET /movie requests.
func (h *Handler) GetMovieDetails(w http.ResponseWriter, req *http.Request) {
    id := req.FormValue("id")
    details, err := h.ctrl.Get(req.Context(), id)
    if err != nil && errors.Is(err, movie.ErrNotFound) {
        w.WriteHeader(http.StatusNotFound)
        return
    } else if err != nil {
        log.Printf("Repository get error: %v\n", err)
    }
}
```

```
w.WriteHeader(http.StatusInternalServerError)
    return
}
if err := json.NewEncoder(w).Encode(details); err != nil {
    log.Printf("Response encode error: %v\n", err)
    w.WriteHeader(http.StatusInternalServerError)
}
}
```

Now, we are finally ready to write a `main` file for the movie service.

Main file

In the `movie/cmd` package, create a `main.go` file, using the following code block:

```
package main

import (
    "log"
    "net/http"

    "movieexample.com/movie/internal/controller/movie"
    metadatagateway "movieexample.com/movie/internal/gateway/metadata/
http"
    ratinggateway "movieexample.com/movie/internal/gateway/rating/http"
    httphandler "movieexample.com/movie/internal/handler/http"
)

func main() {
    log.Println("Starting the movie service")
    metadataGateway := metadatagateway.New("localhost:8081")
    ratingGateway := ratinggateway.New("localhost:8082")
    ctrl := movie.New(ratingGateway, metadataGateway)
    h := httphandler.New(ctrl)
    http.Handle("/movie", http.HandlerFunc(h.GetMovieDetails))
    if err := http.ListenAndServe(":8083", nil); err != nil {
        panic(err)
    }
}
```

At this point, we have the logic for all three services. Note that we used static service addresses (`localhost:8081`, `localhost:8082`, and `localhost:8083`) in this example. This allows us to run the services locally; however, this would not work if we deployed our services to the cloud or any other deployment platform. In the next chapter, we are going to cover this aspect and continue improving our microservices. We can run the services we just created by executing this command inside the `cmd` directory of each service:

```
go run *.go
```

Then, we can call the metadata service API using the following command:

```
curl 'localhost:8081?id=1'
```

We can call the rating service API using a similar command:

```
curl 'localhost:8082?id=1&type=2'
```

Finally, we can call the movie service using the following command:

```
curl 'localhost:8083?id=1'
```

All of the preceding requests should return an HTTP 404 error, indicating that records are not found – we do not have any data yet, so this is expected.

At this point, we have illustrated how to bootstrap and manually test our example microservices and are ready to move on to the next chapter.

Summary

We covered lots of topics in this chapter, including the most important recommendations for writing Go applications and the standards for the project layout of Go applications. The knowledge we gained helped us during the code scaffolding of our microservices – we tried to implement our microservice code in an idiomatic way as much as possible.

You also learned how to split each of your microservices into multiple layers, each responsible for its own logic. We illustrated how to separate the business logic from the code accessing the database, and how to separate the API handler logic from both, as well as from the logic performing remote calls between the services.

While the amount of information in this chapter is quite overwhelming, we have made a solid start and are ready to move on to more advanced topics. In the next chapter, we are going to see how the microservices we created can explore each other, so we can finally test them.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Go Cookbook: <https://go-cookbook.com>
- Effective Go: https://go.dev/doc/effective_go
- Go Wiki: Go Code Review Comments: <https://go.dev/wiki/CodeReviewComments>
- Project layout: <https://github.com/golang-standards/project-layout>
- Project templates and the gonew tool: <https://go.dev/blog/gonew>
- Package names: <https://go.dev/blog/package-names>

3

Service Discovery

In the previous chapter, we created our example microservices and let them communicate with each other, using static local addresses hardcoded into each service. While a static-based approach was trivial to implement, it lacked some useful features, such as the ability to add or remove service instances dynamically without updating the service code. In this chapter, we will review a more advanced approach that would allow us to easily scale our services and let them locate each other in a dynamic environment.

In this chapter, we are going to cover the following topics:

- Service discovery overview
- Service discovery solutions
- Adopting service discovery

We will use the microservices we created in the previous chapter to illustrate how to implement service discovery. Now, let's move on to an overview of the service discovery concepts.

Technical requirements

In order to complete this chapter, you need Go 1.18 or above. Additionally, you will need a Docker tool, which you can download from <https://www.docker.com>.

You can find the source files for this chapter on GitHub: <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter03>.

Service discovery overview

In *Chapter 2, Scaffolding a Go Microservice*, we created an application consisting of three microservices. The relationship between the services is illustrated in the following diagram:

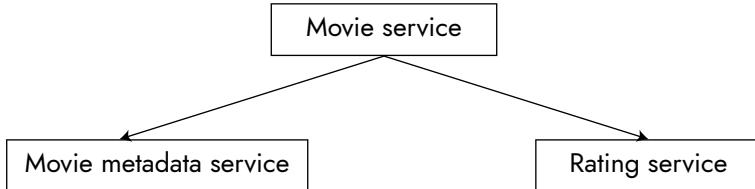


Figure 3.1 – Relationship between our microservices

To make requests to each service, we used static addresses that we set in our code. The settings we used were as follows:

- **Metadata service:** localhost:8081
- **Rating service:** localhost:8082
- **Movie service:** localhost:8083

This approach with pre-programmed service addresses has some significant limitations:

- **Impact on scalability:** Each time you need to add or remove instances, you need to update the code of each calling service
- **Reliability issues:** If an instance becomes unavailable for an extended period (for example, due to network failure), your services will keep calling it until you update their configuration

How do we address these limitations?

The problem we just described for our microservices is called **service discovery**. In general, service discovery addresses multiple problems, as follows:

- How to discover the active instance(s) of a particular service
- How to automate the addition and removal of service addresses in a dynamic environment
- How to handle issues when instances become unresponsive

Now, let's demonstrate how each of these problems is solved with service discovery.

Registry

The foundation of service discovery is a **registry** (also known as a **service registry**), which stores information about available service instances. It has the following features:

- Register an instance of a service
- Deregister an instance of a service
- Return the list of all instances of the service in the form of their network addresses

This is an example of service discovery registry data:

Service name	Address list
Movie service	172.18.10.2:2520
	172.18.12.55:8800
	172.18.89.10:2450
Rating service	172.18.25.11:1100
	172.18.9.55:2830
Movie metadata service	172.18.79.115:3512
	172.17.3.8:9900

Table 3.1 – Registry data

Each service can either register in the registry by itself or use some library or tool to register on service startup automatically. Once the service is registered, it starts being monitored via health checks to ensure the registry contains only available instances.

Let's now see the two common models of adopting service discovery.

Service discovery models

For applications, there are two ways of interacting with the registry:

- **Client-side service discovery:** Access the registry directly from the application using a registry client
- **Server-side service discovery:** Access the registry indirectly via a load balancer, a special server that forwards requests to available instances

Let's compare the pros and cons of each model.

Client-side service discovery

In the client-side service discovery model, each application or service accesses the service registry directly by requesting all available instances of a target service. When the application receives a response, it uses the addresses of the target service for making requests. The logic is illustrated in the following diagram:

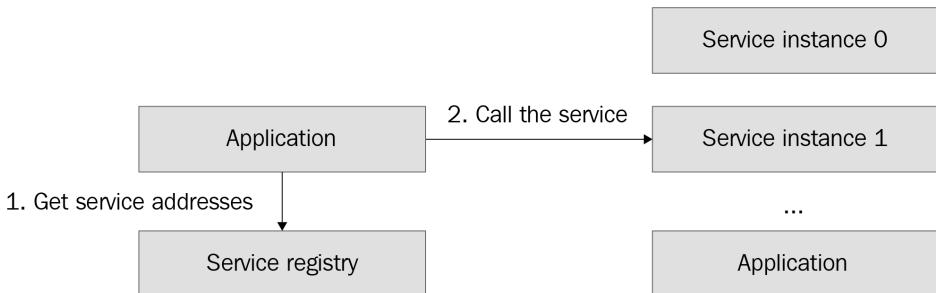


Figure 3.2 – Client-side service discovery

In this model, the application is responsible for balancing the load on the service it is calling – if an application picked just one instance from the list and kept calling it all the time, it would overload that instance and underutilize the other ones.

The downside of this model is that the calling application needs to be programmed with load-balancing and registry communication logic. While most service discovery libraries and tools provide support for this, the client-side service discovery model still adds some additional complexity to the application logic.

Server-side service discovery

The server-side service discovery model adds an extra layer to the interaction between the calling applications and the registry. Instead of calling the registry directly, applications send their requests to target microservices via a special server called a **load balancer**. The load balancer is responsible for interacting with the registry and distributing requests between all available instances.

The following diagram will help you understand the server-side service discovery model:

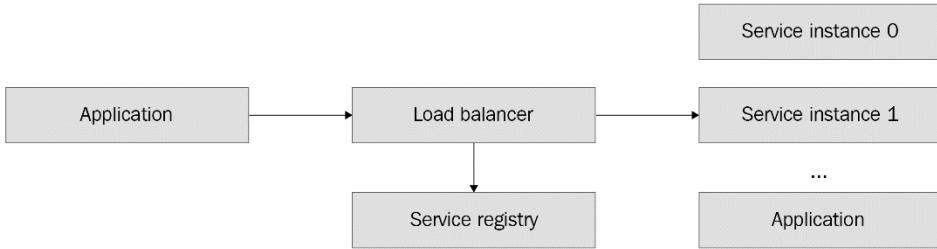


Figure 3.3 – Server-side service discovery

In the diagram, the application calls the target service via a load balancer, accessing the service registry to read the list of active service instances from it. In this model, the application does not need to know about the registry. This is the main benefit of the server-side service discovery model: it helps decouple the interaction with the service registry from each calling application, making the application logic simpler. The downside of the model is that it requires setting up and using a load balancer. The latter is a rather untrivial operation, and we are not going to cover it in this book.

Now, let's see how the registry can keep the list of only active instances of each service.

Service health monitoring

The registry keeps the information about the instances up to date via either a pull or a push model:

- **Pull model:** The registry periodically checks the availability of each known service instance by sending them requests called **health checks**
- **Push model:** Each service instance periodically notifies the registry of its availability

The pull model removes the need to implement status renewal on a service level. In a push model, the application is responsible for renewing its status or telling the service registry about its healthy status.

Now, as we have covered the theoretical basics of service discovery, let's see which existing solutions you can use to enable your microservices.

Service discovery solutions

In this section, we are going to describe the existing service discovery solutions available for your use.

HashiCorp Consul

HashiCorp Consul has been a pretty popular solution for service discovery for many years. Written in Go, this tool allows you to set up service discovery for your services and applications quite easily, using its clients or API.

Consul has a straightforward HTTP API, which includes the following endpoints:

- PUT `/catalog/register`: Register a service instance
- PUT `/catalog/deregister`: Deregister a service instance
- GET `/catalog/services`: Get available instances of a service

Client applications can access the Consul catalog either via the API or in a server-side service discovery mode, using a DNS service.

We are going to illustrate how to use Consul for service discovery later in this chapter. In the meantime, you can learn more about Consul by going to the official website: <https://consul.io>.

Kubernetes

Kubernetes is a popular open source platform for running, scaling, and managing collections of applications, such as microservices.

One of the features of Kubernetes is the ability to register and discover the services that are running in it. Kubernetes provides an API for retrieving the list of network addresses of each service that is being updated automatically, so users can use it in client-side discovery mode. Alternatively, it allows its users to plug in a load balancer to use it for server-side discovery instead.

We are going to cover more features of Kubernetes later, in *Chapter 8* of this book.

etcd

etcd is a distributed key-value store that is commonly used for storing critical system metadata, including dynamic service configuration and service registry. Unlike Consul or Kubernetes, it does not provide a service discovery API directly, but it is trivial to implement it by following the principles we will describe later in the chapter. Some other systems, including Kubernetes, use etcd internally for implementing service discovery. You can learn more about etcd on its website: <https://etcd.io>.

Apache ZooKeeper

Apache ZooKeeper is a system for coordinating and storing the metadata of distributed systems. Similar to etcd, it is also widely used for service discovery; use cases include some popular tools including Apache Kafka and Apache Hadoop. Among the service discovery solutions reviewed in this section, it is the only one that is not written in Go; however, it can be still used by a Go application via its API.

Now, let's see how we can add service discovery to the applications we created in the previous chapter.

Adopting service discovery

In this section, we are going to illustrate how you can start using service discovery for your applications. We will be using the microservices we created in the previous chapter as an example. Then, you will learn how to add the logic responsible for service discovery to your microservice code.

When you consider enabling service discovery for your services, you have multiple questions to answer, such as the following:

- Which model would you prefer to use – client-side or server-side discovery?
- Which platform will you use for the deployment and orchestration of your microservices?

Answering the second question may already give you a solution – various deployment platforms, including Kubernetes, as well as popular clouds such as AWS, offer service discovery for your services.

If you don't know which deployment platform you are going to use for your services and you are new to microservice development, you may consider client-side service discovery. The client-side discovery model is slightly simpler because your service directly coordinates with the service registry. Later, you will be able to switch to server-side service discovery, if you want.

Let's start preparing your applications for adding service discovery logic.

Preparing the application

Let's list what we want to achieve from our service discovery code:

- The ability to register a service that we are going to use on service startup
- The ability to deregister a service that we are going to use on service shutdown
- The ability to get a list of addresses of a particular service that we are going to use for making the calls to the other services
- Set up service health monitoring so the service registry would be able to remove inactive service instances

It would be great if our service discovery logic weren't directly tied to a particular tool we would be using. It is usually a good practice to abstract away the actual technology with a more generic interface, which would allow us to swap implementations. We can illustrate this using an example – imagine we are using the HashiCorp Consul library, which returns a list of service addresses in the following form:

```
func Service(string, string) ([]*consul.ServiceEntry, *consul.QueryMeta, error)
```

If we expose these Consul structures in our code and pass these structures around our code base, our service code will be heavily tied to Consul. If we ever decide to switch to another service discovery tool, we would need to replace not only the service discovery implementation logic but also all places using it in the code.

Instead, let's define a more generic and technology-agnostic interface. For providing the list of service instances, we can just return the list of URLs in `[]string` format.

The complete interface for our service discovery logic would be as follows:

```
// Registry defines a service registry.
type Registry interface {
    // Register creates a service instance record in the registry.
    Register(ctx context.Context, instanceID string, serviceName string,
    hostPort string) error
    // Deregister removes a service instance record from the registry.
    Deregister(ctx context.Context, instanceID string, serviceName string)
    error
    // ServiceAddresses returns the list of addresses of active instances
    // of the given service.
    ServiceAddresses(ctx context.Context, serviceID string) ([]string,
    error)
    // ReportHealthyState is a push mechanism for reporting healthy state
    // to the registry.
    ReportHealthyState(instanceID string, serviceName string) error
}
```

As you may notice, the interface is pretty generic, yet it allows you to create multiple implementations based on different technologies if needed.

You may also notice that the interface includes a `ReportHealthyState` function for reporting the healthy state of a service instance. This function allows us to implement the push-based service

health monitoring that we mentioned before, so each microservice would periodically report its health to the service registry. The registry would then be able to remove inactive instances of each service if they don't report a healthy state within some defined interval of time (we are going to assume the interval is five seconds in our implementation).

Let's now think about where we would store the service discovery logic for our microservices. I suggest using a package that all three services can access – let's create it in the root directory of our application, under the `pkg/discovery`. We can call it `pkg/discovery`. Inside it, add a `discovery.go` file and add the following code to it:

```
package discovery

import (
    "context"
    "errors"
    "fmt"
    "math/rand"
    "time"
)

// Registry defines a service registry.
type Registry interface {
    // Register creates a service instance record in the
    // registry.
    Register(ctx context.Context, instanceID string, serviceName string,
    hostPort string) error
    // Deregister removes a service instance record from
    // the registry.
    Deregister(ctx context.Context, instanceID string, serviceName string)
    error
    // ServiceAddresses returns the list of addresses of
    // active instances of the given service.
    ServiceAddresses(ctx context.Context, serviceID string) ([]string,
    error)
    // ReportHealthyState is a push mechanism for reporting
    // healthy state to the registry.
    ReportHealthyState(instanceID string, serviceName string) error
}
```

```
// ErrNotFound is returned when no service addresses are
// found.
var ErrNotFound = errors.New("no service addresses found")

// GenerateInstanceID generates a pseudo-random service
// instance identifier, using a service name
// suffixed by dash and a random number.
func GenerateInstanceID(serviceName string) string {
    return fmt.Sprintf("%s-%d", serviceName, rand.New(rand.NewSource(time.
Now()).UnixNano())).Int()
}
```

In the code that we just added, we have defined a `Registry` interface for the service registry. Additionally, we defined the `ErrNotFound` error that would be returned from the `ServiceAddresses` function if no active service addresses were found. Finally, we created a `GenerateInstanceID` function that would help us to generate randomized instance identifiers for use with the `Register` and `Deregister` functions that we will use for differentiating between multiple instances of each service.

You might wonder why we defined our `Registry` interface in a separate package instead of putting it into the packages where we would use it, such as our service directories. This is a common dilemma when designing Go APIs: while it is encouraged that interfaces should be defined on the caller side (for example, alongside the code using the interface), shared interfaces are fine as long as they are used for standardization and avoiding unnecessary duplication. In our case, we plan to use the `Registry` interface for each of our services. If we defined the `Registry` interface inside each service directory, we would create multiple duplicate implementations of it. Hence, in our case, the use of a shared interface is justified.

We are ready to begin our work on its implementation.

Implementing the discovery logic

One of the benefits of the interface that we defined earlier is that we can create multiple implementations and use them in our application. So, for example, we can create one implementation that we can later use in tests, whereas another implementation would be used in production. To illustrate this approach, we are going to create two implementations:

- **In-memory service discovery:** Use an in-memory registry to store the set of addresses
- **Consul-based service discovery:** Use the HashiCorp Consul service registry to store and retrieve service addresses

Now, let's proceed to implement the logic.

In-memory implementation

Let's start with the in-memory implementation first. In this implementation, we will be storing the service registry records in memory using a simple map data structure. Note that the data stored in our in-memory service registry would be unique to each process running the code, since each process would have its own memory. Such in-memory implementations are great for illustrative purposes and for use in tests – we are going to reuse our in-memory implementation later in *Chapter 9* of this book.

Let's implement our in-memory registry:

1. Create a `pkg/discovery/memory` package file and a `memory.go` file, then add the following:

```
package memory

import (
    "context"
    "errors"
    "log"
    "net"
    "sync"
    "time"

    "movieexample.com/pkg/discovery"
)

// Registry defines an in-memory service registry.
type Registry struct {
    sync.RWMutex
    serviceAddrs map[string]map[string]*serviceInstance
}

type serviceInstance struct {
    hostPort    string
    lastActive time.Time
}
```

```
// NewRegistry creates a new in-memory service
// registry instance.
func NewRegistry() *Registry {
    return &Registry{serviceAddrs: map[string]
map[string]*serviceInstance{}}
}
```

2. Let's implement our Register and Deregister functions:

```
// Register creates a service record in the registry.
func (r *Registry) Register(ctx context.Context, instanceID string,
serviceName string, hostPort string) error {
    r.Lock()
    defer r.Unlock()
    if _, ok := r.serviceAddrs[serviceName]; !ok {
        r.serviceAddrs[serviceName] = map[string]*serviceInstance{}
    }
    r.serviceAddrs[serviceName][instanceID] =
&serviceInstance{hostPort: hostPort, lastActive: time.Now()}
    return nil
}

// Deregister removes a service record from the
// registry.
func (r *Registry) Deregister(ctx context.Context, instanceID
string, serviceName string) error {
    r.Lock()
    defer r.Unlock()
    if _, ok := r.serviceAddrs[serviceName]; !ok {
        return nil
    }
    delete(r.serviceAddrs[serviceName], instanceID)
    return nil
}
```

3. Finally, let's implement the remaining two functions of the Registry interface:

```
// ReportHealthyState is a push mechanism for
// reporting healthy state to the registry.
func (r *Registry) ReportHealthyState(instanceID string, serviceName
string) error {
    r.Lock()
    defer r.Unlock()
    if _, ok := r.serviceAddrs[serviceName]; !ok {
        return errors.New("service is not registered yet")
    }
    if _, ok := r.serviceAddrs[serviceName][instanceID]; !ok {
        return errors.New("instance " + instanceID + " of service "
+ serviceName + " is not registered yet")
    }
    r.serviceAddrs[serviceName][instanceID].lastActive = time.Now()
    return nil
}

// ServiceAddresses returns the list of addresses of
// active instances of the given service.
func (r *Registry) ServiceAddresses(ctx context.Context, serviceName
string) ([]string, error) {
    r.RLock()
    defer r.RUnlock()
    if len(r.serviceAddrs[serviceName]) == 0 {
        return nil, discovery.ErrNotFound
    }
    var res []string
    for _, i := range r.serviceAddrs[serviceName] {
        if i.lastActive.Before(time.Now().Add(-5 * time.Second)) {
            log.Println("Instance " + instanceID + " of service " +
serviceName + " is not active, skipping")
            continue
        }
        res = append(res, i.hostPort)
    }
    return res, nil
}
```

This implementation can be used in tests or simple applications running on a single server. The implementation is based on a combination of a map data structure and `sync.RWMutex`, allowing us to read and write to the map concurrently. In the map, we store `serviceInstance` structures containing the instance address and the last time of a successful health check for it, which can be set by calling a `ReportHealthyState` function. In the `ServiceAddresses` function, we return only instances of the last time of a successful health check within the last five seconds.

Let's now move to the Consul-based service registry implementation.

Consul-based implementation

The implementation that we are going to work on now will be using HashiCorp Consul as a service registry:

1. First, create a `pkg/discovery/consul` package and add to it a file named `consul.go`:

```
package consul

import (
    "context"
    "errors"
    "fmt"
    "strconv"
    "strings"

    consul "github.com/hashicorp/consul/api"
    "movieexample.com/pkg/discovery"
)

// Registry defines a Consul-based service registry.
type Registry struct {
    client *consul.Client
}

// NewRegistry creates a new Consul-based service
// registry instance.
func NewRegistry(addr string) (*Registry, error) {
    config := consul.DefaultConfig()
    config.Address = addr
    client, err := consul.NewClient(config)
```

```
    if err != nil {
        return nil, err
    }
    return &Registry{client: client}, nil
}
```

2. Now, let's implement the functions of our interface to register and reregister the records:

```
// Register creates a service record in the registry.
func (r *Registry) Register(ctx context.Context, instanceID string,
    serviceName string, hostPort string) error {
    parts := strings.Split(hostPort, ":")
    if len(parts) != 2 {
        return errors.New("hostPort must be in a form of
<host>:<port>, example: localhost:8081")
    }
    port, err := strconv.Atoi(parts[1])
    if err != nil {
        return err
    }
    return r.client.Agent().ServiceRegister(&consul.
        AgentServiceRegistration{
            Address: parts[0],
            ID:      instanceID,
            Name:    serviceName,
            Port:    port,
            Check:   &consul.AgentServiceCheck{CheckID: instanceID, TTL:
                "5s"},
        })
}

// Deregister removes a service record from the
// registry.
func (r *Registry) Deregister(ctx context.Context, instanceID
    string, _ string) error {
    return r.client.Agent().ServiceDeregister(instanceID)
}
```

3. Finally, let's implement the remaining registry functions:

```
// ServiceAddresses returns the list of addresses of
// active instances of the given service.
func (r *Registry) ServiceAddresses(ctx context.Context, serviceName
string) ([]string, error) {
    entries, _, err := r.client.Health().Service(serviceName, "", true, nil)
    if err != nil {
        return nil, err
    } else if len(entries) == 0 {
        return nil, discovery.ErrNotFound
    }
    var res []string
    for _, e := range entries {
        res = append(res, fmt.Sprintf("%s:%d", e.Service.Address,
e.Service.Port))
    }
    return res, nil
}

// ReportHealthyState is a push mechanism for
// reporting healthy state to the registry.
func (r *Registry) ReportHealthyState(instanceID string, _ string)
error {
    return r.client.Agent().PassTTL(instanceID, "")
}
```

Our client depends on an external library, github.com/hashicorp/consul/api. We need to fetch it now by running `go mod tidy` inside our `src` directory. After this, Go should fetch the dependency, and our logic should be able to compile.

Now, we are ready to plug the logic we just created into our microservices.

Using the discovery logic

Now, we need to add logic for initializing and discovering the services. Currently, only the movie service communicates with the other two, so we are going to illustrate how to add service discovery using the movie service as an example.

Let's start with our gateways:

1. In the previous chapter, we created two gateways for calling the metadata and the rating services. Let's modify their structures to the one shown here:

```
type Gateway struct {
    registry discovery.Registry
}
```

2. Also, change the New function format to the following:

```
func New(registry discovery.Registry) *Gateway {
    return &Gateway{registry}
}
```

3. Now, the gateways require a registry on creation. We can change the beginning of the Get function of the metadata gateway to this now:

```
func (g *Gateway) Get(ctx context.Context, id string) (*model.Metadata, error) {
    addrs, err := g.registry.ServiceAddresses(ctx, "metadata")
    if err != nil {
        return nil, err
    }
    url := " HYPERLINK \h" + addrs[rand.Intn(len(addrs))] + "/metadata"
    log.Printf("Calling metadata service. Request: GET " + url)
    req, err := http.NewRequest(http.MethodGet, url, nil)
```

You can notice that instead of calling a static pre-configured address, we now first get the available addresses of the metadata from the registry. This is the essence of service discovery – we use the data from the registry to make remote calls between our services. After we get the list of service addresses, we pick a random one using the `rand.Intn` function. By doing this, we balance the load between the active instances, randomly selecting any available instance on each request.

4. Now, update the rating gateway in the same way we changed the metadata service.
5. The next step is to update the main functions of our services so that each service will register and deregister itself in the service registry. Let's update the metadata service first. Update its `main` function to the following:

```
const serviceName = "metadata"

func main() {
    var port int
    flag.IntVar(&port, "port", 8081, "API handler port")
    flag.Parse()
    log.Printf("Starting the metadata service on port %d", port)
    registry, err := consul.NewRegistry("localhost:8500")
    if err != nil {
        panic(err)
    }
    ctx := context.Background()
    instanceID := discovery.GenerateInstanceID(serviceName)
    if err := registry.Register(ctx, instanceID, serviceName, fmt.Sprintf("localhost:%d", port)); err != nil {
        panic(err)
    }
    go func() {
        for {
            if err := registry.ReportHealthyState(instanceID, serviceName); err != nil {
                log.Println("Failed to report healthy state: " + err.Error())
            }
            time.Sleep(1 * time.Second)
        }
    }()
    defer registry.Deregister(ctx, instanceID, serviceName)
    repo := memory.New()
    svc := metadata.New(repo)
    h := httphandler.New(svc)
    http.Handle("/metadata", http.HandlerFunc(h.GetMetadataByID))
    if err := http.ListenAndServe(fmt.Sprintf(":%d", port), nil);
    err != nil {
        panic(err)
    }
}
```

In the preceding code, we added the logic for registering and deregistering the service in the Consul-based service registry and reporting its healthy state to it every second. Note that we used the `flag.Parse` function to initialize the value of the `port` variable; this will allow us to override the default value of `8081` by using the `-port` command-line argument when we run our service. These mechanisms will allow us to run multiple instances of our metadata service locally, one for each separate port, to test the discovery of multiple instances.

6. Let's add similar logic to the rating service. Update its `main` function as follows:

```
func main() {
    var port int
    flag.IntVar(&port, "port", 8082, "API handler port")
    flag.Parse()
    log.Printf("Starting the rating service on port %d", port)
    registry, err := consul.NewRegistry("localhost:8500")
    if err != nil {
        panic(err)
    }
    ctx := context.Background()
    instanceID := discovery.GenerateInstanceID(serviceName)
    if err := registry.Register(ctx, instanceID, serviceName, fmt.Sprintf("localhost:%d", port)); err != nil {
        panic(err)
    }
    go func() {
        for {
            if err := registry.ReportHealthyState(instanceID, serviceName); err != nil {
                log.Println("Failed to report healthy state: " + err.Error())
            }
            time.Sleep(1 * time.Second)
        }
    }()
    defer registry.Deregister(ctx, instanceID, serviceName)
    repo := memory.New()
    svc := controller.New(repo)
```

```
    h := httphandler.New(svc)
    http.Handle("/rating", http.HandlerFunc(h.Handle))
    if err := http.ListenAndServe(fmt.Sprintf(":%d", port), nil);
err != nil {
    panic(err)
}
}
```

The changes that we just made are similar to the ones we did for the metadata service.

7. The last step is to modify the `main` function of the movie service, replacing it with the following:

```
func main() {
    var port int
    flag.IntVar(&port, "port", 8083, "API handler port")
    flag.Parse()
    log.Printf("Starting the movie service on port %d", port)
    registry, err := consul.NewRegistry("localhost:8500")
    if err != nil {
        panic(err)
    }
    ctx := context.Background()
    instanceID := discovery.GenerateInstanceID(serviceName)
    if err := registry.Register(ctx, instanceID, serviceName, fmt.
Sprintf("localhost:%d", port)); err != nil {
        panic(err)
    }
    go func() {
        for {
            if err := registry.ReportHealthyState(instanceID,
serviceName); err != nil {
                log.Println("Failed to report healthy state: " +
err.Error())
            }
            time.Sleep(1 * time.Second)
        }
    }()
    defer registry.Deregister(ctx, instanceID, serviceName)
```

```
metadataGateway := metadatagateway.New(registry)
ratingGateway := ratinggateway.New(registry)
svc := movie.New(ratingGateway, metadataGateway)
h := httphandler.New(svc)
http.Handle("/movie", http.HandlerFunc(h.GetMovieDetails))
if err := http.ListenAndServe(fmt.Sprintf(":%d", port), nil);
err != nil {
    panic(err)
}
}
```

At this point, we have successfully added Consul-based service discovery to our applications. Let's illustrate how it works in practice:

1. In order to run our applications now, you would need HashiCorp Consul to run locally. The easiest way would be to run it using a Docker tool. Assuming you have already installed Docker from its website, [docker.com](https://www.docker.com), you can run the following command:

```
docker run -d -p 8500:8500 -p 8600:8600/udp --name=dev-consul
hashicorp/consul agent -server -ui -node=server-1 -bootstrap-
expect=1 -client='0.0.0.0'
```

The preceding command runs HashiCorp Consul inside Docker in development mode, exposing its ports 8500 and 8600 for local use.

2. Run each microservice by executing this command inside each cmd directory:

```
go run *.go
```

3. Now, go to the Consul web UI using <http://localhost:8500/>. When you open the **Services** tab, you should see the list of our services and an active Consul instance:

The screenshot shows the 'Services' page in the Consul UI. At the top, there's a search bar with placeholder 'Search', a 'Search Across' dropdown, a 'Health Status' dropdown, and a 'Service Type' dropdown. Below the header, there's a table with four rows, each representing a service instance:

	consul	1 instance
	metadata	1 instance
	movie	1 instance

Figure 3.4 – Consul web view of active service instances

You can optionally add some additional instances of each service by running the following:

```
go run *.go --port <PORT>
```

If you run the preceding command, replace the <PORT> placeholder with unique port numbers that are not in use yet (in our examples, we used ports 8081, 8082, and 8083, so you can run with port numbers starting with 8084). The result of each command would be additional healthy instances in the Consul service view illustrated earlier.

You can also try shutting down any service manually by terminating the go run commands and seeing how instances change their states from **Passing** to **Critical**.

4. To test API requests, ensure you have at least one healthy instance of each service and make the following request to a movie service:

```
curl -v 'localhost:8083/movie?id=1'
```

5. Check the output logs of the movie service now (you should be able to see them in the terminal where you ran the go run command for the movie service). If you did everything correctly, you should see a line similar to this:

```
2022/06/08 13:37:42 Calling metadata service. Request: GET http://localhost:8081/metadata
```

The preceding line is the result of a call to the service registry backed by the Consul. In our metadata service gateway implementation, we select a random active instance from the registry and log its address before making a call. If you have more than one instance of a metadata service, you can make multiple `curl` requests, listed previously, and see that the movie service always picks a random instance among them.

At this point, we have illustrated how to use service discovery with our microservices. We can now dynamically scale our microservices by adding and removing their instances without any need to change the service code. We also have two working implementations of a service registry that you can use in your code. Now, we are ready to move to the next chapter, covering another important topic, data serialization.

Summary

In this chapter, we saw an overview of service discovery and compared its different models. You have learned what the service registry is and what its main service discovery models are. We have illustrated how to use a client-side service discovery model by providing two implementations, one using an in-memory set of data and another using HashiCorp Consul. We have also plugged the Consul-based implementation into our microservices to demonstrate how to use it in the microservice logic. Now, you know how to add and use service discovery in your applications: this knowledge will help you to scale up your services and establish service deployments, which we will cover in *Chapter 8, Setting Up Service Deployments*.

In the next chapter, we are going to discuss another important topic: serialization. You will learn how to encode and decode the data transferred between the services to establish communication between them.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Service discovery explained: <https://developer.hashicorp.com/consul/docs/concepts/service-discovery>
- Server-side service discovery: <https://microservices.io/patterns/server-side-discovery.html>

4

Serialization

In the previous chapters, we learned how to scaffold Go microservices, create HTTP API endpoints, and set up service discovery to let our microservices communicate with each other. This knowledge already provides a solid foundation for building microservices; however, we are going to continue our journey with more advanced topics.

In this chapter, we will explore **serialization**, a process that allows data to be encoded and decoded so that it can be stored or sent between services. To illustrate how to use it, we are going to define data structures that will be transferred between our microservices using the popular and efficient **Protocol Buffers** format. Then, we are going to show you how to generate code for the newly introduced data structures. Finally, we will demonstrate how fast our Protocol Buffers-based serialization is compared to some other formats, such as XML and JSON. By the end of this chapter, you will know how to leverage the serialization technique for service data encoding. This knowledge will help us establish efficient communication between our microservices in the following chapters.

In this chapter, we are going to cover the following topics:

- The basics of serialization
- Exploring popular serialization formats
- Using Protocol Buffers
- Serialization best practices

Technical requirements

To complete this chapter, you will need to have Go 1.18 or above and the Protocol Buffers compiler: <https://grpc.io/docs/protoc-installation/>.

You can find the code examples for this chapter on GitHub at <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter04>.

The basics of serialization

Serialization is the process of converting data into a format that allows you to transfer it, store it, and later deconstruct it.

Serialization has two primary use cases:

- Transferring data between services, acting as a common format between them
- Encoding and decoding arbitrary data for storage, allowing you to store complex data structures as byte arrays or regular strings

In *Chapter 2 Scaffolding a Go Microservice*, while scaffolding our applications, we created our HTTP API endpoints and set them to return JSON responses to the callers. In that case, JSON played the role of a **serialization format**, allowing us to transform our data structures into it and then decode them back.

Let's take our Metadata structure defined in the `metadata/pkg/model/metadata.go` file as an example:

```
// Metadata defines the movie metadata.  
type Metadata struct {  
    ID      string `json:"id"  
    Title   string `json:"title"  
    Description string `json:"description"  
    Director string `json:"director"  
}
```

Our structure includes records known as **annotations**, which help the JSON encoder transform our record into an output. For example, we can create an instance of our structure:

```
Metadata{  
    ID:      "123",  
    Title:   "The Movie 2",  
    Description: "Sequel of the legendary The Movie",
```

```
    Director: "Foo Bars",
}
```

When we then encode it with JSON, the result would be as follows:

```
{"id": "123", "title": "The Movie 2", "description": "Sequel of the legendary
The Movie", "director": "Foo Bars"}
```

Once the data has been serialized, it can be stored or transferred between services, even if they are written in different languages. In addition to data communication, serialization can be used for other purposes:

- **Store configuration:** You can define your service settings using serialization formats, such as JSON, and then read them in your service code.
- **Storing records in a database:** Serialization formats are frequently used for storing arbitrary data in databases. For example, key-value databases store records as strings or byte arrays, and developers often use serialization to encode and decode these record values.
- **Logging:** Application logs are often stored in JSON format, making them easy to read for both humans and various applications, such as data visualization software.

JSON is one of the most popular serialization formats at the time of writing and it has been essential to web development. It has the following benefits:

- **Language support:** Most programming languages include tools for encoding and decoding JSON data
- **Browser support:** JSON has been an integral part of web applications and all modern browsers include developer tools for working with it in the browser itself
- **Readability:** JSON records are easily readable and are often easy to use while both developing and debugging web applications

However, it has certain limitations as well:

- **Size:** JSON is not a size-efficient format. In this chapter, we are going to see which formats and protocols provide output records that are smaller in size.
- **Speed:** As with its output size, the encoding and decoding speed with JSON is not the fastest when set against other popular serialization protocols.

Many other popular serialization formats and protocols are used in the industry. In the next section, we are going to review some of the most popular ones.

Popular serialization formats

In addition to JSON, other popular serialization formats are available:

- **XML**
- **YAML**
- **Apache Thrift**
- **Apache Avro**
- **Protocol Buffers**

This section will provide a high-level overview of each one, as well as some key differences between these protocols.

XML

XML is one of the earliest serialization formats for web service development. It was created in 1998 and is still used in the industry, especially in enterprise applications.

XML represents data as a tree of nodes called elements. An element example would be `<example>Some value</example>`. If we were to serialize our `Metadata` structure, shown previously, the result would be as follows:

```
<Metadata><ID>123</ID><Title>The Movie 2</Title><Description>Sequel of the  
legendary The Movie</Description><Director>Foo Bars</Director></Metadata>
```

You may have noticed that the serialized XML representation of our data is slightly longer than the JSON one. This is one of the downsides of XML format – the output is often the largest among all popular serialization protocols, making it harder to read and transfer the data.

YAML

YAML is a serialization format that was first released in 2001. It gained popularity over the years, becoming one of the most popular serialization formats in the industry. The designers of the language took a strong focus on its readability and compactness, making it a perfect tool for defining arbitrary human-readable data. We can illustrate this in our `Metadata` structure. In YAML format, it would look like this:

```
metadata:  
  id: 123  
  title: The Movie 2  
  description: Sequel of the legendary The Movie  
  director: Foo Bars
```

YAML format is widely used for storing configuration data. One of the reasons for this is the ability to include comments, which is lacking in other formats, such as JSON. The use of YAML for service-to-service communication is less common, primarily due to the greater size of the serialized data.

Apache Thrift

So far, we have reviewed JSON, XML, and YAML, all of which are primarily used for defining and serializing arbitrary types of data. There are other solutions to a broader class of problems when we want to not only serialize and deserialize the data but also transfer it between multiple services. These solutions combine two roles: they act as both serialization formats and **communication protocols** – mechanisms for sending and receiving arbitrary data over the network. HTTP is an example of such a protocol but developers are not limited to using it in their applications.

Apache Thrift offers both serialization and a communication protocol that can be used for defining your data types and allowing your services to communicate with each other by passing them. It was initially created by Facebook but later became a community-supported open source project under the Apache Software Foundation.

Thrift, unlike JSON and XML, requires you to define your structures in their own format first. In our example, for the `Metadata` structure, we would need to create a file with the `.thrift` extension that includes the definition in Thrift language:

```
struct Metadata {  
    1: required string id,  
    2: required string title,  
    3: required string description,  
    4: required string director  
}
```

Once you have a Thrift file, you can use it with an automatic Thrift code generator to generate the code for most programming languages that would contain the defined structures and logic to encode and decode it. In addition to data structures, Thrift allows you to define **Thrift services** – sets of functions that can be called remotely. Here's an example of a Thrift service definition:

```
service MetadataService {  
    Metadata get(1: string id)  
}
```

This example defines a service called `MetadataService`, which provides a `get` function that returns a `Metadata` Thrift object. A Thrift-compatible server can act as such a Thrift service, processing incoming requests from the client applications. We are going to learn how to write such servers in *Chapter 5*.

Let's explore the benefits and limitations of Apache Thrift. First, we have the benefits:

- Thrift has a smaller output size and higher encoding and decoding speed compared to XML and JSON. As we are going to illustrate later in this chapter, Thrift-serialized data can be 30% to 50% smaller in size than XML and JSON.
- Thrift can define not only structures but entire services and generate code for them, allowing communication to occur between the servers and their clients.

The limitations include the following:

- Thrift has had relatively low popularity and adoption in recent years due to moving to more popular and efficient formats.
- It lacks official documentation. Thrift is a relatively complex technology, and most documentation is unofficial.
- Unlike JSON and XML, Thrift-serialized data is not readable, so it's trickier to use it for debugging.
- It has had nearly no support in recent years – Facebook keeps maintaining a separate branch of it called Facebook Thrift, but it is much less popular than its Apache counterpart.

Let's see the other popular serialization formats that are widely used across the industry.

Apache Avro

Apache Avro is a combination of a serialization format and a communication protocol that is somewhat similar to Apache Thrift. Apache Avro also requires a developer to define a schema (written either in JSON or in its own language, called Avro IDL) for their data. In our case, the `Metadata` structure would have the following schema:

```
{  
    "namespace": "example.avro",  
    "type": "record",  
    "name": "Metadata",  
    "fields": [  
        {"name": "id", "type": "string"},  
        {"name": "title", "type": "string"},  
    ]  
}
```

```
{ "name": "description", "type": "string"},  
  { "name": "director", "type": "string"},  
]  
}
```

Then, the schema would be used for translating the structures into a serialized state and back.

It is not uncommon for types and structures to change over time, and microservice API and structure definitions need to evolve. With Avro, developers can create a new version of a schema (represented as a separate file, often suffixed with an incremental version number), and keep both the old and new versions in the code base. This way, the application can encode and decode data in either format, even if there are some incompatible changes, such as changes in field names. This is one of the key benefits of using Apache Avro over many other serialization protocols.

Protocol Buffers

Protocol Buffers is a serialization format that was created at Google more than 20 years ago. In 2008, the format became public and immediately gained popularity among developers. The benefits of the format include the following:

- The simplicity of the definition language
- A small data output size
- High performance in terms of serialization and deserialization
- The ability to define services in addition to data structures and compile client and server code in multiple languages
- Protocol evolution and official support by Google

The popularity of Protocol Buffers and its simplicity, as well as the efficiency of its data encoding, makes it a great fit for use in microservice development. We are going to use Protocol Buffers to serialize and deserialize the data transferred between our services, as well as to define our service APIs. In the next section, you will learn how to start using Protocol Buffers and move our microservice logic to Protocol Buffers from JSON.

Using Protocol Buffers

In this section, we are going to illustrate how you can use Protocol Buffers for your applications. We will use the microservice examples from the previous chapters and define our data model in Protocol Buffers format. Then, we will use code generation tools alongside Protocol Buffers to generate our data structures. Finally, we will illustrate how to use our generated code to serialize and deserialize our data.

First, let's prepare our application. Create a directory called `api` under our application's `src` directory. Inside this directory, create a `movie.proto` file and add the following to it:

```
syntax = "proto3";
option go_package = "/gen";

message Metadata {
    string id = 1;
    string title = 2;
    string description = 3;
    string director = 4;
}

message MovieDetails {
    double rating = 1;
    Metadata metadata = 2;
}
```

In the first line of our schema definition, we set the syntax to `proto3`, the latest version of the Protocol Buffers protocol. The second line defines the output path for the code generated. The rest of the file includes two structures that we need for our microservices, similar to the Go structures we created in *Chapter 2*.



Note

Unlike Apache Thrift, we don't specify whether Protocol Buffers message fields are required or optional. This wasn't the case before the third version of the protocol; however, the authors of Protocol Buffers intentionally removed this feature. This is because required fields are difficult to deprecate: if you set the field as optional and stop returning it from a server, a client that is not aware of the change might consider the data invalid. To make serialization backward compatible, it is preferred to keep all fields optional, and provide explicit documentation on data validation and supported message values.

Now, let's generate the code for our structures. In the `src` directory of our application, run the following command:

```
protoc -I=api --go_out=. movie.proto
```

If the command executes successfully, you should find a new directory called `src/gen`. This directory should include a file called `movie.pb.go` containing the generated code, which includes our structures and the code to serialize and deserialize them. For example, the generated `MovieDetails` structure code would be as follows:

```
type Metadata struct {
    state      protoimpl.MessageState
    sizeCache  protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Id          string `protobuf:"bytes,1,opt,name=id,proto3"
json:"id,omitempty"`
    Title       string `protobuf:"bytes,2,opt,name=title,proto3"
json:"title,omitempty"`
    Description string `protobuf:"bytes,3,opt,name=description,proto3"
json:"description,omitempty"`
    Director    string `protobuf:"bytes,4,opt,name=director,proto3"
json:"director,omitempty"`
}
```

Now, let's describe what exactly we have just achieved. Here, we have created a `movie.proto` file that defines our **data schema** – the definition of our data structures. The schema is now defined independently from our Go code, providing the following benefits to us:

- **Explicit schema definition:** Our data schema is now decoupled from the code and explicitly defines the application data types. This makes it easier to see the data types provided by application APIs.
- **Code generation:** Our schema can be converted into code via code generation. We are going to use it later in *Chapter 5* to send the data between the services.
- **Cross-language support:** We can generate our code not only for Go but also for other programming languages. If our model changes, we won't need to rewrite our structures for all languages. Instead, we can just re-generate the code for all languages by running a single command.

Let's perform a quick benchmark and compare the size of the serialized data for three serialization protocols – XML, JSON, and Protocol Buffers. We'll write a small tool to do so.

Inside the `src` directory, create a directory called `cmd/sizecompare` and add a `main.go` file to it with the following contents:

```
package main

import (
    "encoding/json"
    "encoding/xml"
    "fmt"

    "github.com/golang/protobuf/proto"
    "movieexample.com/gen"
    "movieexample.com/metadata/pkg/model"
)

var metadata = &model.Metadata{
    ID:        "123",
    Title:     "The Movie 2",
    Description: "Sequel of the legendary The Movie",
    Director:   "Foo Bars",
}

var genMetadata = &gen.Metadata{
    Id:        "123",
    Title:     "The Movie 2",
    Description: "Sequel of the legendary The Movie",
    Director:   "Foo Bars",
}
```

Let's implement the `main` function:

```
func main() {
    jsonBytes, err := serializeToJson(metadata)
    if err != nil {
        panic(err)
    }
```

```
xmlBytes, err := serializeToXML(metadata)
if err != nil {
    panic(err)
}

protoBytes, err := serializeToProto(genMetadata)
if err != nil {
    panic(err)
}

fmt.Printf("JSON size:\t%d\n", len(jsonBytes))
fmt.Printf("XML size:\t%d\n", len(xmlBytes))
fmt.Printf("Proto size:\t%d\n", len(protoBytes))
}
```

Additionally, add the following functions:

```
func serializeToJson(m *model.Metadata) ([]byte, error) {
    return json.Marshal(m)
}

func serializeToXML(m *model.Metadata) ([]byte, error) {
    return xml.Marshal(m)
}

func serializeToProto(m *gen.Metadata) ([]byte, error) {
    return proto.Marshal(m)
}
```

In the preceding code, we encoded our `Metadata` structure using JSON, XML, and Protocol Buffers formats, and printed the output sizes in bytes for each encoded result.

You may need to fetch the `github.com/golang/protobuf/proto` package, which is required for our benchmark, by running the following command:

```
go mod tidy
```

Now, we can run our benchmark by executing `go run *.go` inside its directory. We will see following the output:

```
JSON size:      106B
XML size:      148B
Proto size:    63B
```

The result is quite interesting. The XML output is almost 40% bigger than the JSON one. At the same time, Protocol Buffers's output is more than 40% smaller than the JSON data and more than twice as small as the XML result. This illustrates how efficient the Protocol Buffers format is compared to the other two in terms of output size quite well. By switching from JSON to Protobuf, we reduce the amount of data that we need to send over the network and make our communication faster.

Now, let's do an additional experiment and test the serialization speed for all three formats. For this, we are going to do a **benchmark** – an automated performance check that is going to measure how fast a target operation is.

Create a file called `main_test.go` in the same directory and add the following to it:

```
package main

import (
    "testing"
)

func BenchmarkSerializeToJson(b *testing.B) {
    for i := 0; i < b.N; i++ {
        serializeToJson(metadata)
    }
}

func BenchmarkSerializeToXml(b *testing.B) {
    for i := 0; i < b.N; i++ {
        serializeToXml(metadata)
    }
}

func BenchmarkSerializeToProto(b *testing.B) {
```

```
    for i := 0; i < b.N; i++ {
        serializeToProto(genMetadata)
    }
}
```

We have just created a Go benchmark that is going to tell us how fast JSON, XML, and Protobuf are encoding. We are going to cover the details of benchmarking in *Chapter 8*. For now, run the code to see the output by executing the following command:

```
go test -bench=.
```

The result of the command should look as follows:

```
goos: darwin
goarch: amd64
pkg: movieexample.com/cmd/sizecompare
cpu: Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz
BenchmarkSerializeToJson-12           3308172          342.2 ns/op
BenchmarkSerializeToXml-12           480728          2519 ns/op
BenchmarkSerializeToProto-12         6596490          185.7 ns/op
PASS
ok      movieexample.com/cmd/sizecompare   5.239s
```

Here, you can see the names of three functions that we just implemented and two numbers next to them:

- The first one is the number of times the function was executed
- The second is the average processing speed, measured in nanoseconds per operation

From the output, we can see that, on average, Protocol Buffers serialization took 185.7 nanoseconds, while JSON serialization was almost two times slower at 342.2 nanoseconds. On average, XML serialization took 2,519 nanoseconds, being more than 13 times slower than Protocol Buffers, and more than seven times slower than JSON serialization.

The benchmark is indeed interesting – it illustrates how different the average encoding speeds for various serialization formats are. If performance is important for your services, you should consider faster serialization formats to achieve a higher encoding and decoding speed.

For now, we are going to leave the generated structures in our repository. We will be using them in the next chapter to replace our JSON API handlers.

Now, let's learn some best practices for using serialization.

Serialization best practices

This section briefly summarizes the best practices for serializing and deserializing data. These practices will help you to maintain and evolve your data schemas over time, as well as make them easy to use:

- **Keep your schema backward compatible:** Avoid making any changes in your data schema that would break any existing callers. Such changes include making modifications (renaming or removing) to field names and types.
- **Ensure that data schemas are kept in sync between clients and servers:** For serialization formats with explicit schema definitions, such as Apache Thrift, Protocol Buffers, and Apache Avro, you should keep clients and servers in sync with the latest schema versions.
- **Document implicit details:** Let the callers know any implicit details related to your data schema. For example, if your API does not allow an empty value of a certain field of a structure, include this in the comments in the schema file.
- **Use built-in structures to represent time whenever possible:** Protocol Buffers and some other serialization protocols provide built-in types for timestamps and durations. Taking Protocol Buffers as an example, having an `int timestamp` field would be considered a bad practice. The right approach would be to use `google.protobuf.Timestamp`.
- **Use consistent naming:** Opt for using consistent naming in your schema files, similar to your code.
- **Follow the official style guide:** Become familiar with the official style guide if you are using a schema definition language, such as Thrift or Protocol Buffers. You can find the link to the official style guide for Protocol Buffers in the *Further reading* section of this chapter.

This list provides some high-level recommendations applicable to all serialization protocols. For protocol-specific recommendations, follow the official documentation and check the popular open source projects to find some real-world code examples.

Summary

In this chapter, we covered the basics of serialization and illustrated how our data structures can be encoded using various serialization protocols, including XML, JSON, and Protocol Buffers. You learned about the differences between the most popular serialization protocols and their main advantages and disadvantages.

We also covered the basics of the Protocol Buffers serialization format and showed how to define custom data structures in its schema definition language. We used some example code to illustrate how to generate the schema files for the Go language. Finally, we compared the performance of XML, JSON, and Protocol Buffers formats, and discussed some common data serialization best practices.

In the next chapter, we are going to demonstrate how to use Protocol Buffers-based serialization for communication between services.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Go Cookbook: Serialization <https://go-cookbook.com/snippets/serialization>
- Comparison of serialization formats: https://en.wikipedia.org/wiki/Comparison_of_data-serialization_formats
- The Protocol Buffers documentation: <https://developers.google.com/protocol-buffers>
- The Protocol Buffers official style guide: <https://developers.google.com/protocol-buffers/docs/style>

5

Synchronous Communication

In this chapter, we are going to cover the most common way of communicating between microservices – synchronous communication. In *Chapter 2 Scaffolding a Go Microservice*, we implemented the logic for communicating between microservices via the HTTP protocol and returning results in JSON format. In *Chapter 4 Serialization*, we illustrated that the JSON format is not the most efficient in terms of data size, and many different formats provide additional benefits to developers, including code generation. In this chapter, we are going to show you how to define service APIs using the Protocol Buffers format and generate both client and server code for them.

By the end of this chapter, you will understand the key concepts of synchronous communication between microservices and will have learned how to implement microservice clients and servers.

The knowledge you will gain in this chapter will help you learn how to better organize client and server code, generate the code for serialization and communication, and then use it in your microservices. In this chapter, we will cover the following topics:

- Introduction to synchronous communication
- Defining a service API using Protocol Buffers
- Implementing gateways and clients
- Synchronous communication best practices

Technical requirements

To complete this chapter, you will need Go 1.18 or above and the Protocol Buffers compiler, available at <https://grpc.io/docs/protoc-installation/>.

You will also need a Go gRPC plugin, which you can install by running the following command:

```
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest  
export PATH="$PATH:$(go env GOPATH)/bin"
```

Additionally, you will need to install the grpcurl tool: <https://github.com/fullstorydev/grpcurl>.

You can find the GitHub code for this chapter at <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter05>.

Introduction to synchronous communication

In this section, we are going to cover the basics of synchronous communication and introduce you to some additional benefits of Protobuf that we are going to use for our microservices.

Synchronous communication explains how network applications, such as microservices, interact and is where services exchange data using a **request-response model**. This process is illustrated in the following diagram:

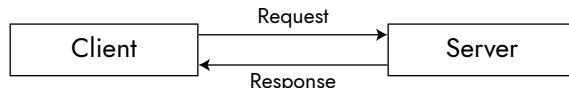


Figure 5.1 – Synchronous communication

Requests and responses can either take the form of individual messages or be represented as sequences of messages, called **streams**. Consider a video streaming service: a client would be sending a request to get a long video and get the response from a server as a sequence of messages, each containing a small chunk of video data.

There are many **protocols** for synchronous communication between services. HTTP is one of the most popular among them.

Each communication protocol defines its own data model for sending request and response data. In the case of HTTP, this includes the following:

- **URL parameters:** In the case of the `https://www.google.com/search?q=portugal` URL, `q` is a URL parameter.
- **Headers:** Each request and response includes optional key-value pairs called headers, allowing you to propagate additional contextual metadata, such as the client or browser name – for example, `User-Agent: Mozilla/5.0`.
- **Request and response body:** The request and response can include a body that contains arbitrary data. For example, when a client uploads a file to a server, the file's contents are usually sent as a request body.

When a server cannot handle a client request due to an error or the request is not received due to network issues, the client receives a specific response indicating an error. In the case of the HTTP protocol, there are two types of errors:

- **Client error:** This error is caused by the client. Examples of such errors include invalid request arguments (such as an incorrect username), unauthorized access, and access to a resource that cannot be found (for example, a non-existing web page).
- **Server error:** This error is caused by the server. This could be an application bug or an error with an upstream component, such as a database.

In *Chapter 2 Scaffolding a Go Microservice*, we implemented our API handlers by sending the result data as an HTTP response body in JSON format. We achieved this by using the Go JSON encoder:

```
if err := json.NewEncoder(w).Encode(details); err != nil {
    log.Printf("Response encode error: %v\n", err)
}
```

As discussed in the previous chapter, JSON is not the most optimal in terms of data size. Also, it does not offer useful tools, such as cross-language code generation for data structures, that are provided by the formats, such as Protobuf. Additionally, sending requests over HTTP and encoding the data manually is not the only form of communication that can occur between the services. There are some existing **remote procedure call (RPC)** libraries and frameworks that help to communicate between multiple services and offer some additional features to application developers:

- **Automated serialization and deserialization:** RPC libraries take responsibility for serializing and deserializing your communication data, as well as returning correct error codes in case message data is invalid.

- **Client and server code generation:** Developers can generate the client code for connecting and sending data to other microservices, as well as generate the server code for accepting incoming requests.
- **Authentication:** Most RPC libraries and frameworks offer authentication options for cross-service requests – we are going to review them in *Chapter 10 Security and Compliance*.
- **Context propagation:** This is the ability to send additional data with requests, such as traces, something we are going to cover in *Chapter 13 Setting Up Service Alerting*.
- **Documentation generation:** Thrift can generate HTML documentation for services and data structures.

In the next section, we are going to cover some of the RPC libraries that you can use in your Go services, along with the features they provide.

Go RPC frameworks and libraries

Let's review some popular RPC frameworks and libraries that are available for Go developers.

Apache Thrift

We covered Apache Thrift in *Chapter 4 Serialization* and mentioned its ability to define RPC services – sets of functions provided by an application, such as a microservice. Here is an example of a Thrift RPC service definition:

```
service MetadataService {  
    Metadata get(1: string id)  
}
```

The Thrift definition of a service can be used to generate client and server code. The client code would include the logic for connecting to an instance of a service, as well as making requests to it, serializing and deserializing the request and response structures. The advantage of using a library such as Apache Thrift over making HTTP requests manually is the ability to generate such code for multiple languages: a service written in Go could easily talk to a service written in Java, while both would use the generated code for communication, removing the need to implement serialization/deserialization logic.

gRPC

gRPC is an RPC framework that was created at Google. gRPC uses HTTP/2 as the transport protocol and Protobuf as a serialization format. Similar to Apache Thrift, it allows you to define RPC services and generate the client and server code for them. In addition to this, it offers some extra

features, such as the following:

- Authentication
- Context propagation
- Documentation generation

The gRPC protocol separates communication into four distinct types:

- **Unary**: A client sends a single message as a request and gets a single message as a response.
- **Client streaming**: A client sends a sequence of messages (for example, small chunks of a large file during the upload process) and gets a single message as a response (in the case of a file upload, this could include the identifier of the uploaded file).
- **Server streaming**: A client sends a single message as a request and gets a response as a sequence of messages (for example, a sequence of file chunks when downloading a large file).
- **Bi-directional streaming**: Both the client and the server send sequences of messages to each other.



Note

Streaming adds some extra overhead, such as increased memory and CPU usage for keeping client and server connections active, so it should only be used for specific limited use cases, such as large file uploads or media streaming. We are going to provide some streaming-specific examples in *Chapter 16 Advanced Topics*.

gRPC adoption is much higher than for Apache Thrift, and its support of the popular Protobuf format makes it a great fit for microservice developers. In this book, we are going to use gRPC as a framework for synchronous communication between our microservices. In the next section, we are going to illustrate how to leverage the features provided by Protobuf to define our service APIs.

Defining a service API using Protocol Buffers

Let's demonstrate how to define a service API using the Protocol Buffers format and generate the client and server gRPC code for communication with each of our services using a **proto** compiler. This knowledge will help you establish a foundation for both defining and implementing APIs for your microservices using one of the industry's most popular communication tools.

Let's start with our metadata service and write its API definition in the Protocol Buffers schema language.

Open the `api/movie.proto` file that we created in the previous chapter and add the following to it:

```
service MetadataService {
    rpc GetMetadata(GetMetadataRequest) returns (GetMetadataResponse);
    rpc PutMetadata(PutMetadataRequest) returns (PutMetadataResponse);
}

message GetMetadataRequest {
    string movie_id = 1;
}

message GetMetadataResponse {
    Metadata metadata = 1;
}
```

The code we just added defines our metadata service and its `GetMetadata` endpoint. We already have the `Metadata` structure from the previous chapter, so we can reuse it now.

Let's note some aspects of the code we just added:

- **Request and response structures:** It's good practice to create a new structure for both a request and a response. In our example, they are `GetMetadataRequest` and `GetMetadataResponse`.
- **Naming:** You should follow consistent naming rules for all your endpoints. We are going to prefix all request and response functions with the function name.

Now, let's add the definition of the rating service to the same file:

```
service RatingService {
    rpc GetAggregatedRating(GetAggregatedRatingRequest) returns
    (GetAggregatedRatingResponse);
    rpc PutRating(PutRatingRequest) returns (PutRatingResponse);
}

message GetAggregatedRatingRequest {
    string record_id = 1;
    int32 record_type = 2;
}

message GetAggregatedRatingResponse {
```

```
    double rating_value = 1;
}

message PutRatingRequest {
    string user_id = 1;
    string record_id = 2;
    int32 record_type = 3;
    int32 rating_value = 4;
}

message PutRatingResponse {
```

Here, our rating service has two endpoints, and we defined requests and responses for them in a similar way to the metadata service.



Note

As an alternative to using the `int32` type to store `record_type` fields, we could have used Protobuf enumerations: <https://protobuf.dev/programming-guides/proto3/#enum>.

Both options have some upsides: our rating format is more generic and could be used for storing ratings for any type of record, while enumerations would be more specific, requiring us to keep supported values in sync between clients and servers.

Finally, let's add the definition of the movie service to the same file:

```
service MovieService {
    rpc GetMovieDetails(GetMovieDetailsRequest) returns
    (GetMovieDetailsResponse);
}

message GetMovieDetailsRequest {
    string movie_id = 1;
}

message GetMovieDetailsResponse {
    MovieDetails movie_details = 1;
}
```

Now, our `movie.proto` file includes both our structure definitions and the API definitions for our services. At this point, we are ready to generate code for the newly added service definitions. In the `src` directory of the application, run the following:

```
protoc -I=api --go_out=. --go-grpc_out=. movie.proto
```

The preceding command is similar to the command that we used in the previous chapter to generate code for our data structures. However, it also passes a `--go-grpc_out` flag to the compiler. This flag tells the Protobuf compiler to generate the service code in gRPC format.

Let's see the compiled code that was generated as the output for our command. If the command is executed without any errors, you will find a `movie_grpc.pb.go` file inside the `src/gen` directory. This file will include the generated Go code for our services. Let's take a look at the generated client code:

```
type MetadataServiceClient interface {
    GetMetadata(ctx context.Context, in *GetMetadataRequest, opts ...grpc.CallOption) (*GetMetadataResponse, error)
}

type metadataServiceClient struct {
    cc grpc.ClientConnInterface
}

func NewMetadataServiceClient(cc grpc.ClientConnInterface) MetadataServiceClient {
    return &metadataServiceClient{cc}
}

func (c *metadataServiceClient) GetMetadata(ctx context.Context, in *GetMetadataRequest, opts ...grpc.CallOption) (*GetMetadataResponse, error) {
    out := new(GetMetadataResponse)
    err := c.cc.Invoke(ctx, "/MetadataService/GetMetadata", in, out, opts...)
    if err != nil {
        return nil, err
    }
    return out, nil
}
```

This generated code can be used in our applications to call our API from the Go applications. Additionally, we can generate such client code for other languages, such as Java, adding more arguments to the compiler command that we just executed. This is a great feature that can save us lots of time when writing microservice applications – instead of writing client logic for calling our services, we can use the generated clients and plug them into our applications.

In addition to the client code, the Protobuf compiler also generates some service code that can be used for handling the requests. In the same `movie_grpc.pb.go` file, you will find the following:

```
type MetadataServiceServer interface {
    GetMetadata(context.Context, *GetMetadataRequest)
    (*GetMetadataResponse, error)
    mustEmbedUnimplementedMetadataServiceServer()
}

func RegisterMetadataServiceServer(s grpc.ServiceRegistrar, srv
MetadataServiceServer) {
    s.RegisterService(&MetadataService_ServiceDesc, srv)
}
```

We are going to use both the client and server code that we just saw in our application. In the next section, we are going to modify our API handlers so that they can use the generated code and handle requests using the Protobuf format.

Implementing gateways and clients

In this section, we are going to illustrate how to plug the generated client and server gRPC code into our microservices. This will help us to switch communication between them from JSON-serialized HTTP to Protobuf gRPC calls.

Metadata service

In *Chapter 2 Scaffolding a Go Microservice*, we created our internal model structures, such as metadata, while in *Chapter 4 Serialization*, we created their Protobuf counterparts. Then, we generated the code for our Protobuf definitions. As a result, we have two versions of our model structures – internal ones, as defined in `metadata/pkg/model`, and the generated ones, which are located in the `gen` package.

You might think that having two similar structures is now redundant. While there is certainly some level of redundancy in having such duplicate definitions, practically, these structures serve different purposes:

- **Internal model:** The structures that you create manually for your application should be used across its code base, such as the repository, controller, and other logic.
- **Generated model:** Structures generated by tools such as the protoc compiler, which we used in the previous two chapters, should only be used for serialization. The use cases include transferring the data between the services or storing the serialized data.

You might be curious why it's not recommended to use the generated structures across the application code base. There are multiple reasons for this:

- **Unnecessary coupling between the application and serialization format:** If you ever want to switch from one serialization format to another (for example, from Thrift to Protocol Buffers), and all your application code base uses generated structures for the previous serialization format, you would need to rewrite not only the serialization code but the entire application.
- **The generated code's structure could vary between different versions:** While the field naming and high-level structure of the generated structures are generally stable between different versions of code generation tooling, the internal functions and structure of the generated code could vary from version to version. If any part of your application uses some generated functions that get changed, your application could break unexpectedly during a version update of a code generator.
- **Generated code is often harder to use:** In formats such as Protocol Buffers, all fields are always optional. In generated code, this results in lots of fields that can have nil values. For an application developer, this means doing more nil checks across all applications to prevent possible panics.

Because of these reasons, the best practice is to keep both internal structures and the generated ones and only use the generated structures for serialization. Let's illustrate how to achieve this.

We would need to add some **mapping** logic to translate the internal data structures and their generated counterparts. In the `metadata/pkg/model` directory, create a `mapper.go` file and add the following to it:

```
package model

import (
    "movieexample.com/gen"
)

// MetadataToProto converts a Metadata struct into a
// generated proto counterpart.
func MetadataToProto(m *Metadata) *gen.Metadata {
    return &gen.Metadata{
        Id:        m.ID,
        Title:     m.Title,
        Description: m.Description,
        Director:   m.Director,
    }
}

// MetadataFromProto converts a generated proto counterpart
// into a Metadata struct.
func MetadataFromProto(m *gen.Metadata) *Metadata {
    return &Metadata{
        ID:        m.Id,
        Title:     m.Title,
        Description: m.Description,
        Director:   m.Director,
    }
}
```

The code we just added transforms the internal model into the generated structures and back. In the following code block, we are going to use it in the server code.

Now, let's implement a gRPC handler for the metadata service that would handle the client requests to the service. In the `metadata/internal/handler` package, create a `grpc` directory and add a `grpc.go` file:

```
package grpc

import (
    "context"
```

```

"errors"

"google.golang.org/grpc/codes"
"google.golang.org/grpc/status"
"movieexample.com/gen"
"movieexample.com/metadata/internal/controller"
"movieexample.com/metadata/internal/repository"
"movieexample.com/metadata/pkg/model"
)

// Handler defines a movie metadata gRPC handler.
type Handler struct {
    gen.UnimplementedMetadataServiceServer
    svc *controller.MetadataService
}

// New creates a new movie metadata gRPC handler.
func New(ctrl *metadata.Controller) *Handler {
    return &Handler{svc: ctrl}
}

```

Let's implement the `GetMetadataByID` function:

```

// GetMetadataByID returns movie metadata by id.
func (h *Handler) GetMetadata(ctx context.Context, req *gen.GetMetadataRequest) (*gen.GetMetadataResponse, error) {
    if req == nil || req.MovieId == "" {
        return nil, status.Errorf(codes.InvalidArgument, "nil req or empty id")
    }
    m, err := h.svc.Get(ctx, req.MovieId)
    if err != nil && errors.Is(err, controller.ErrNotFound) {
        return nil, status.Errorf(codes.NotFound, err.Error())
    } else if err != nil {
        return nil, status.Errorf(codes.Internal, err.Error())
    }
    return &gen.GetMetadataResponse{Metadata: model.MetadataToProto(m)},
    nil
}

```

Let's highlight some parts of this implementation:

- The handler embeds the generated `gen.UnimplementedMetadataServiceServer` structure. This is required by a Protocol Buffers compiler to enforce future compatibility.
- Our handler implements the `GetMetadataByID` function in the same format as defined in the generated `MetadataServiceServer` interface.
- We are using the `MetadataToProto` mapping function to transform our internal structures into the generated ones.

Now, we are ready to update our main file and switch it to the gRPC handler. Update the `metadata/cmd/main.go` file, changing its contents to the following:

```
package main

import (
    "context"
    "flag"
    "fmt"
    "log"
    "net"
    "time"

    "google.golang.org/grpc"
    "movieexample.com/gen"
    "movieexample.com/metadata/internal/controller"
    grpchandler "movieexample.com/metadata/internal/handler/grpc"
    "movieexample.com/metadata/internal/repository/memory"
    "movieexample.com/metadata/pkg/model"
)

func main() {
    log.Println("Starting the movie metadata service")
    repo := memory.New()
    svc := controller.New(repo)
    h := grpchandler.New(svc)
    lis, err := net.Listen("tcp", "localhost:8081")
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
```

```

    }
    srv := grpc.NewServer()
    gen.RegisterMetadataServiceServer(srv, h)
    if err := srv.Serve(lis); err != nil {
        panic(err)
    }
}

```

The updated `main` function illustrates how we instantiate our gRPC server and start listening for requests in it. The rest of the function is similar to the one we had before.

We are done with the changes to the metadata service and can now proceed to the rating service.

Rating service

Let's create a gRPC handler for the rating service. In the `rating/internal/handler` package, create a `grpc` directory and add a `grpc.go` file with the following code:

```

package grpc

import (
    "context"
    "errors"

    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    "movieexample.com/gen"
    "movieexample.com/rating/internal/controller"
    "movieexample.com/rating/pkg/model"
)

// Handler defines a gRPC rating API handler.
type Handler struct {
    gen.UnimplementedRatingServiceServer
    svc *controller.RatingService
}

// New creates a new rating gRPC handler.
func New(svc *controller.RatingService) *Handler {
    return &Handler{svc: ctrl}
}

```

Now, let's implement the GetAggregatedRating endpoint:

```
// GetAggregatedRating returns the aggregated rating for a
// record.
func (h *Handler) GetAggregatedRating(ctx context.Context, req *gen.
GetAggregatedRatingRequest) (*gen.GetAggregatedRatingResponse, error) {
    if req == nil || req.RecordId == "" || req.RecordType == "" {
        return nil, status.Errorf(codes.InvalidArgument, "nil req or empty
id/type")
    }
    v, err := h.svc.GetAggregatedRating(ctx, model.RecordID(req.RecordId),
model.RecordType(req.RecordType))
    if err != nil && errors.Is(err, controller.ErrNotFound) {
        return nil, status.Errorf(codes.NotFound, err.Error())
    } else if err != nil {
        return nil, status.Errorf(codes.Internal, err.Error())
    }
    return &gen.GetAggregatedRatingResponse{RatingValue: v}, nil
}
```

Finally, let's implement the PutRating endpoint:

```
// PutRating writes a rating for a given record.
func (h *Handler) PutRating(ctx context.Context, req *gen.
PutRatingRequest) (*gen.PutRatingResponse, error) {
    if req == nil || req.RecordId == "" || req.UserId == "" {
        return nil, status.Errorf(codes.InvalidArgument, "nil req or empty
user id or record id")
    }
    if err := h.svc.PutRating(ctx, model.RecordID(req.RecordId), model.
RecordType(req.RecordType), &model.Rating{UserID: model.UserID(req.
UserId), Value: model.RatingValue(req.RatingValue)}); err != nil {
        return nil, status.Errorf(codes.Internal, err.Error())
    }
    return &gen.PutRatingResponse{}, nil
}
```

Now, we are ready to update our rating/cmd/main.go file. Replace it with the following:

```
package main

import (
    "log"
    "net"

    "google.golang.org/grpc"
    "movieexample.com/gen"
    "movieexample.com/rating/internal/controller"
    grpchandler "movieexample.com/rating/internal/handler/grpc"
    "movieexample.com/rating/internal/repository/memory"
)

func main() {
    log.Println("Starting the rating service")
    repo := memory.New()
    svc := controller.New(repo)
    h := grpchandler.New(svc)
    lis, err := net.Listen("tcp", "localhost:8082")
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    srv := grpc.NewServer()
    gen.RegisterRatingServiceServer(srv, h)
    if err := srv.Serve(lis); err != nil {
        panic(err)
    }
}
```

The way we start the service is similar to the metadata service. Now, we are ready to link the movie service to both the metadata and rating services.

Movie service

In the previous examples, we created gRPC servers to handle client requests. Now, let's illustrate how to add logic for calling our servers. This will help us to establish communication between our microservices via gRPC.

First, let's implement a function that we can reuse in our service gateways. Create the `src/internal/grpcutil` directory and add a file called `grpcutil.go` to it. Add the following code to it:

```
package grpcutil

import (
    "context"
    "math/rand"
    "pkg/discovery"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
    "movieexample.com/pkg/discovery"
)

// ServiceConnection attempts to select a random service
// instance and returns a gRPC connection to it.
func ServiceConnection(ctx context.Context, serviceName string, registry
discovery.Registry) (*grpc.ClientConn, error) {
    addrs, err := registry.ServiceAddresses(ctx, serviceName)
    if err != nil {
        return nil, err
    }
    return grpc.Dial(addrs[rand.Intn(len(addrs))], grpc.
WithTransportCredentials(insecure.NewCredentials()))
}
```

The function that we just implemented will try to pick a random instance of the target service using the provided service registry, and then it will create a gRPC connection for it.

Now, let's create a gateway for our metadata service. In the `movie/internal/gateway` package, create a directory called `metadata`. Inside it, create a `grpc` directory with a `metadata.go` file that contains the following code:

```
package grpc

import (
    "context"
```

```

"google.golang.org/grpc"
"movieexample.com/gen"
"movieexample.com/internal/grpcutil"
"movieexample.com/metadata/pkg/model"
"movieexample.com/pkg/discovery"
)

// Gateway defines a movie metadata gRPC gateway.
type Gateway struct {
    registry discovery.Registry
}

// New creates a new gRPC gateway for a movie metadata
// service.
func New(registry discovery.Registry) *Gateway {
    return &Gateway{registry}
}

```

Let's implement the function for getting the metadata from a remote gRPC service:

```

// Get returns movie metadata by a movie id.
func (g *Gateway) Get(ctx context.Context, id string) (*model.Metadata,
error) {
    conn, err := grpcutil.ServiceConnection(ctx, "metadata", g.registry)
    if err != nil {
        return nil, err
    }
    defer conn.Close()
    client := gen.NewMetadataServiceClient(conn)
    resp, err := client.GetMetadataByID(ctx, &gen.
GetMetadataByIDRequest{MovieId: id})
    if err != nil {
        return nil, err
    }
    return model.MetadataFromProto(resp.Metadata), nil
}

```

Let's highlight some details of our gateway implementation:

- We use the `grpcutil.ServiceConnection` function to create a connection to our metadata service
- We create a client using the generated client code from the `gen` package
- We use the `MetadataFromProto` mapping function to convert the generated structures into internal ones

Now, we are ready to create a gateway for our rating service. Inside the `movie/internal/gateway` package, create a `rating/grpc` directory and add a `grpc.go` file with the following contents:

```
package grpc

import (
    "context"
    "pkg/discovery"
    "rating/pkg/model"

    "google.golang.org/grpc"
    "movieexample.com/internal/grpcutil"
    "movieexample.com/gen"
)

// Gateway defines an gRPC gateway for a rating service.
type Gateway struct {
    registry discovery.Registry
}

// New creates a new gRPC gateway for a rating service.
func New(registry discovery.Registry) *Gateway {
    return &Gateway{registry}
}
```

Add the implementation of the `GetAggregatedRating` function:

```
// GetAggregatedRating returns the aggregated rating for a
// record or ErrNotFound if there are no ratings for it.
func (g *Gateway) GetAggregatedRating(ctx context.Context, recordID model.
RecordID, recordType model.RecordType) (float64, error) {
```

```
conn, err := grpcutil.ServiceConnection(ctx, "rating", g.registry)
if err != nil {
    return 0, err
}
defer conn.Close()
client := gen.NewRatingServiceClient(conn)
resp, err := client.GetAggregatedRating(ctx, &gen.
GetAggregatedRatingRequest{RecordId: string(recordID), RecordType:
string(recordType)})
if err != nil {
    return 0, err
}
return resp.RatingValue, nil
}
```

At this point, we are almost done with the changes. The last step is to update the `main` function of the movie service. Change it to the following:

```
package main

import (
    "context"
    "log"
    "net"

    "google.golang.org/grpc"
    "movieexample.com/gen"
    "movieexample.com/movie/internal/controller"
    metadatagateway "movieexample.com/movie/internal/gateway/metadata/
grpc"
    ratinggateway "movieexample.com/movie/internal/gateway/rating/grpc"
    grpchandler "movieexample.com/movie/internal/handler/grpc"
    "movieexample.com/pkg/discovery/static"
)
```

```
func main() {
    log.Println("Starting the movie service")
    registry := static.NewRegistry(map[string][]string{
        "metadata": {"localhost:8081"},
        "rating":   {"localhost:8082"},
        "movie":    {"localhost:8083"},
    })
    ctx := context.Background()
    if err := registry.Register(ctx, "movie", "localhost:8083"); err !=
nil {
        panic(err)
    }
    defer registry.Deregister(ctx, "movie")
    metadataGateway := metadatagateway.New(registry)
    ratingGateway := ratinggateway.New(registry)
    svc := controller.New(ratingGateway, metadataGateway)
    h := grpchandler.New(svc)
    lis, err := net.Listen("tcp", "localhost:8083")
    if err != nil {
        panic(err)
    }
    srv := grpc.NewServer()
    gen.RegisterMovieServiceServer(srv, h)
    if err := srv.Serve(lis); err != nil {
        panic(err)
    }
}
```

You might have noticed that the format hasn't changed, and we just updated the imports for our gateways, changing them from HTTP to gRPC.

We are done with the changes to our services. Now, the services can communicate with each other using the Protocol Buffers serialization, and you can run them using the `go run *.go` command inside each `cmd` directory. You can make a test gRPC request to one of our services by using the `grpcurl` tool:

```
grpcurl -plaintext -d '{"record_id":"1", "record_type": "movie"}' localhost:8082 RatingService/GetAggregatedRating
```

You will get an error indicating that there is no data for the provided record. However, this is currently expected since we aren't storing any data yet – we will do this soon in *Chapter 7, Storing Service Data*.

With that, we have covered the main aspects of synchronous communication between microservices. Now, let's review some of the best practices concerning synchronous communication.

Synchronous communication best practices

This section lists some synchronous communication best practices that will help you to improve the maintainability and reliability of your synchronous communication, as well as communicate any errors to service callers clearly.

Perform excessive request validation and return correct error codes

Earlier in this chapter, we mentioned that two basic types of errors can occur during synchronous communication: client" and server error. All communication protocols, including HTTP, gRPC, and Thrift, offer support for differentiating between different error types in client and server code. For example, in our metadata server code, we used the following code to return different types of errors to our callers:

```
func (h *Handler) GetMetadata(ctx context.Context, req *gen.GetMetadataRequest) (*gen.GetMetadataResponse, error) {
    if req == nil || req.MovieId == "" {
        return nil, status.Errorf(codes.InvalidArgument, "nil req or empty id")
    }
    m, err := h.svc.Get(ctx, req.MovieId)
    if err != nil && errors.Is(err, controller.ErrNotFound) {
        return nil, status.Errorf(codes.NotFound, err.Error())
    } else if err != nil {
        return nil, status.Errorf(codes.Internal, err.Error())
    }
    return &gen.GetMetadataResponse{Metadata: model.MetadataToProto(m)},
    nil
}
```

Our code returns three different types of error codes:

1. `codes.InvalidArgument`: A client error indicating incorrect request data.
2. `codes.NotFound`: A client error indicating that the requested data can't be found.
3. `codes.Internal`: A server error indicating some internal error while handling the request.

Each error code provides a hint to our clients regarding what exactly went wrong during the request processing. For example, our client might retry a request when a server returns the `codes.Internal` error code, while the `InvalidArgument` error would indicate that the problem is likely on the side of our client and requires a change to be made to the client code before the request can be processed successfully.

Leveraging specific error codes will help with investigating issues more easily, as well as adding various automations to your code (for example, automatic retries of requests in the case of internal server errors, something we will implement in *Chapter 10 Security and Compliance*. Additionally, excessive request validation in your server code will help you avoid lots of possible issues, including code panics (for example, when some request parameter has a `nil` value).

Ensure idempotency

Idempotency is one of the critical concepts in synchronous communication and API design. In simple terms, idempotency means that an operation, such as a network request, can be handled multiple times without any unintended side effects. Let's assume we have a system where a server processes payment requests from one of its clients. Now, assume that the client just submitted a payment request, but immediately after, the client's network connection becomes unavailable. If the server guarantees idempotency, the client can safely retry the same request without worrying about getting charged twice for the same payment.

How can we achieve idempotency? Some common techniques can be helpful:

- **Assign unique identifiers to requests:** In our payment example, a server might require the client to provide an extra argument (for example, `transaction_id`) that would act as the unique identifier of the request. Such identifiers are often called **idempotency keys**. Then, a server would guarantee that requests with the same idempotency key would be idempotent, hence safe to retry on the client side.

- **Design APIs with idempotency in mind:** When implementing service APIs, follow some basic rules for achieving idempotency. First, read operations (for example, `GetMetadata` requests) should not make extra modifications to the underlying data. Second, various update operations (for example, HTTP PUT and DELETE calls) should also produce the same results without causing any unexpected effects. The article at <https://stripe.com/blog/idempotency> provides a good summary on the importance of building idempotent APIs.

Also, be explicit about idempotency and all aspects of request handling in your API documentation. Your service clients should not be manually testing how the system would behave in various error cases; instead, you should provide a clear description of how each endpoint works and what the expected inputs and outputs are, as well as which operations are idempotent and how such idempotency is achieved. We are going to cover this more in *Chapter 10, Security and Compliance*.

This brief section summarized the best practices for synchronous communication. Some other useful tips can be found in the *Further reading* section of this chapter.

Summary

In this chapter, we covered the basics of synchronous communication and learned how to make microservices communicate with each other using the Protocol Buffers serialization format and gRPC communication framework. We illustrated how to define our service APIs using the Protocol Buffers schema language and generate code that can be reused in microservice applications written in Go and other languages. Then, we implemented gRPC client and server components and discussed some of the best practices for establishing synchronous communication.

The knowledge you have gained in this chapter should help you write and maintain the existing services using Protocol Buffers and gRPC, as well as serve as an example of how to use code generation for your services.

In the next chapter, we are going to continue our journey and look into different ways of communication by covering another model: asynchronous communication.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Go Cookbook: RPC: <https://go-cookbook.com/snippets/rpc>
- gRPC: <https://grpc.io>
- HTTP/2 detailed overview: <https://web.dev/performance-http2>
- Idempotent REST APIs: <https://restfulapi.net/idempotent-rest-apis/>

6

Asynchronous Communication

In the previous chapter, we illustrated how services can communicate with each other using a synchronous request-response model. Other communication models provide various benefits to application developers, such as asynchronous communication, something we are going to cover in this chapter.

Now you are going to learn the basics of asynchronous communication and some common techniques for using it, as well as some benefits and challenges it brings to microservice developers. We will cover a popular piece of asynchronous communication software, Apache Kafka, and illustrate how to use it to establish communication between our microservices.

In this chapter, we are going to cover the following topics:

- Asynchronous communication basics
- Using Apache Kafka for messaging
- Asynchronous communication best practices

Technical requirements

To complete this chapter, you will need Go 1.18+ or above, similar to in previous chapters. Additionally, you will need Docker, which you can download at <https://www.docker.com>. You will need to register on the Docker website to test service deployments in this chapter.

You can find the GitHub code for this chapter at <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter06>

Asynchronous communication basics

Asynchronous communication is communication between a sender and one or multiple receivers, where a sender does not necessarily expect an immediate response to their messages. In the synchronous communication model, which we covered in *Chapter 5 Synchronous Communication*, the caller sending the request would expect an immediate (or nearly immediate, considering network latency) response to it. In asynchronous communication, it may take an arbitrary amount of time for the receiver to respond to the request, or to not respond at all (for example, when receiving a no-reply notification).

We can illustrate the differences between the two models using two examples. An example of synchronous communication is a phone call – two people having a phone conversation are in direct and immediate communication with each other, and they expect to hear the responses in real time. An example of asynchronous communication is sending mail to people. It can take time to respond to such mail, and the sender does not expect an immediate response to their messages.

This does not mean, however, that asynchronous communication is necessarily slower than the synchronous model. In most cases, asynchronous processing is as quick as synchronous processing and often can be even faster: asynchronous processing is often much less interruptive and leads to higher processing efficiency. It's like replying to 10 emails, one by one, compared to switching between 10 parallel phone calls – the latter example of synchronous processing can sometimes be much slower due to context switching and frequent interruptions.

In the following section, we are going to review the common benefits and challenges of asynchronous communication to help you understand when to use it in microservice applications.

Benefits and challenges of asynchronous communication

The asynchronous communication model comes with several benefits and challenges. Developers need to consider both to decide on whether to use this model. So, what are the benefits of using asynchronous communication? Let's find out:

- **A more streamlined approach to processing messages:** Imagine you have a service whose purpose is to process data and report the status to another service. The reporting service does not necessarily need to wait for any response back from the service it is reporting to, as it would do in the synchronous model. In asynchronous mode, it just needs confirmation that the status message was sent successfully. This is similar to sending a large number of postcards to your relatives – if you send a dozen postcards, you don't want to wait until each card gets delivered before sending the next one!

- **The ability to decouple the operation of sending and processing requests:** Imagine a caller requesting that a server convert a large video file into a different format. In a synchronous model, the caller would be waiting in real time until the entire video is processed. This could easily take minutes, sometimes even hours, making such waiting very inefficient. Instead, such a task could be performed asynchronously, where the caller would send the task to a server, get an acknowledgment that the task was received, and perform any other activity until an eventual notification of completion (or processing failure) is received.
- **Better load balancing:** Certain applications can have uneven request loads and are prone to sudden spikes of requests. If communication is synchronous, the server needs to answer every request in real time, and this can easily overload it. Imagine responding to 10 messages from your friends simultaneously: it is much easier to do this sequentially, one by one.

The benefits that we just described are quite significant, and in many cases, asynchronous communication is the only way to perform certain types of tasks or to provide better system performance. Let's look at some examples of problems for which asynchronous communication is a good fit:

- **Long-running processing tasks:** Long-running tasks, such as video processing, are often better done asynchronously. The caller requesting such processing would not necessarily need to wait until it is completed and would eventually get notified of the final result.
- **Send once, processed by multiple components:** Certain types of messages, such as status reports, can be processed by multiple independent components. Imagine a system where multiple employees need to receive the same message – instead of sending it to each one independently, the message can be published to a component that can be consumed by everyone interested.
- **High-performance sequential processing:** Certain types of operations (for example, big data processing) are more efficient when performed sequentially and/or in batches. For such scenarios, asynchronous processing offers great performance improvements compared to more interactive and interruptive synchronous communication because the receiver of such requests can control the processing speed and process tasks one after another.

While these benefits of asynchronous communication may seem appealing, it is important to note that it often brings some difficult challenges to developers:

- **More complex error handling:** Imagine sending a message to your friend and not receiving a response. Was it because your friend did not receive the message? Did something happen during this time? Did the response get lost? In synchronous communication, such as a phone call, we would immediately know if the friend is not available and would be able to call back. In the case of an asynchronous scenario, we would need to think about more possible issues, such as the ones we described.

- **Reliance on additional components for message delivery:** Certain asynchronous communication use cases, such as the publisher-subscriber or message broker models described in the next section, require additional software for delivering messages. Such software often performs additional operations, such as message batching and storing, bringing additional complexity to the system in exchange for additional features it provides.
- **Asynchronous data flow may seem non-intuitive to many developers and be more complex:** Unlike the synchronous request-response model, where each request is logically followed by a response to it, asynchronous communication may be **unidirectional** (no responses are received at all) or may require the caller to perform additional steps to receive a response (for example, when the response is sent as a separate notification). Because of this, data flow in asynchronous systems may be more complex than in synchronous request-response interactions.

Now, let's cover some asynchronous communication techniques and patterns that can help you organize your services and establish asynchronous communication between them.

Techniques and patterns of asynchronous communication

Various techniques help to make the asynchronous interaction between multiple services more efficient in various scenarios, such as sending a message to multiple recipients. In this section, we are going to describe multiple patterns that help to facilitate such interactions.

Message broker

A **message broker** is an intermediary component in the communication chain that can play multiple roles:

- **Message delivery:** It delivers a message to one or multiple receivers.
- **Message transformation:** It transforms an incoming message into another format that can be later consumed by receivers.
- **Message batching:** It combines multiple messages into a single one for more efficient delivery or processing.
- **Message routing:** It routes incoming messages to the appropriate destination based on predefined rules.

When you send a postcard to your friend or a relative, the post office plays the role of a message broker, acting as an intermediary in delivering it to the destination. In this example, the main benefit of using the message broker would be the convenience of sending the message (a postcard, in our example) without any need to think about how to deliver it. Another benefit of using

message brokers is delivery guarantees. A message broker can provide various levels of guarantees for message delivery. Examples of these guarantees include the following:

- **At-least-once:** The message gets delivered at least once, but may be delivered multiple times in case of failures.
- **Exactly-once:** The message broker guarantees that the message gets delivered and it will be delivered exactly once.
- **At-most-once:** The message can be delivered 0 or 1 time.

The exactly-once guarantee is often harder to achieve in practice than the at-least-once and at-most-once guarantees. In the at-least-once model, a message broker can just re-send the message in case of any failure (such as a sudden power loss or a restart). In the exactly-once model, the message broker needs to perform additional checks or store extra metadata to ensure the message is never re-sent to the receiver in any possible case.

Another classification of message brokers is based on the possibility of them losing messages:

- **Lossy:** A message broker that can occasionally (for example, in case of failures) lose messages
- **Lossless:** A message broker that provides a guarantee of not losing any messages

The at-most-once guarantee is an example of a lossy message broker, whereas at-least-once and exactly-once brokers are examples of lossless ones. Lossy message brokers are faster than lossless ones because they don't need to handle extra logic for guaranteeing message delivery, such as persisting messages.

The publisher-subscriber model

The **publisher-subscriber model** is a model of communication between multiple components (such as microservices) where every component can publish messages and subscribe to the relevant ones.

Let's take X (formerly Twitter) as an example. Any user can publish messages to their feeds, and other users can subscribe to them. Similarly, microservices can communicate by publishing the data that other services can consume. Imagine that we have a set of services that process various types of user data, such as user photos, videos, and text messages. If a user deleted their profile, we would need to notify all services about this. Instead of notifying each service one by one, we could publish a single event that would indicate that a user profile has been deleted, and all services could consume it and perform any relevant actions, such as archiving user data.

The relationship between the publishers, the subscribers, and the data produced by the publisher is illustrated in the following diagram:

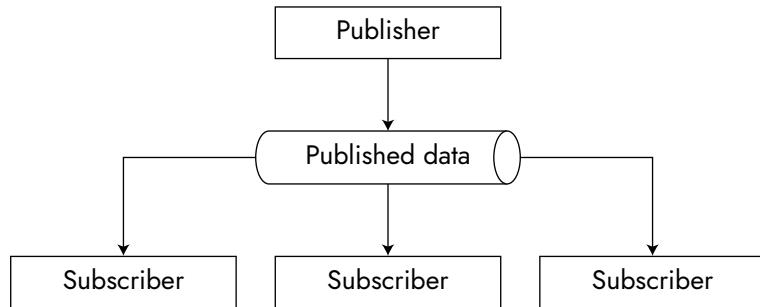


Figure 6.1 – The publisher-subscriber model

The publisher-subscriber model provides a flexible solution for sending and delivering data in a system where messages can be processed by multiple components. Each publisher can publish their messages without caring about the delivery process and any difficulties in delivering messages to an arbitrary number (even a very large one) of receivers. Each subscriber can subscribe to the relevant messages and get them delivered without needing to contact the publisher directly and check whether there is any new data to consume.

Now that we have covered some high-level asynchronous communication models, let's move on to the practical side of this chapter and illustrate how you can implement asynchronous communication in your microservices.

Using Apache Kafka for messaging

In this section, we are going to introduce you to Apache Kafka, a popular message broker system that we are going to use to establish asynchronous communication between our microservices. You will learn the basics of Kafka, how to publish messages to it, and how to consume such messages from the microservices we created in the previous chapters.

Apache Kafka basics

Apache Kafka is an open source message broker system that provides the ability to publish and subscribe to messages containing arbitrary data. Originally developed at LinkedIn, Kafka has become perhaps the most popular open source message broker software and is used by thousands of companies around the world.

In the Kafka model, a component that publishes messages is called a **producer**. Messages are published sequentially to objects called **topics**. Each message in a topic has a unique numerical **offset** in it. Kafka provides APIs for consuming messages (the component for consuming messages is called a **consumer**) for the existing topics. Topics can also be partitioned to allow multiple consumers to consume from them (for example, for parallel data processing).

The Kafka data model is illustrated in the following diagram:



Figure 6.2 – The Apache Kafka data model

With such a seemingly simple data model, Kafka is a powerful system that offers lots of benefits to its users:

- **High sequential write and read throughput:** Kafka is optimized for highly performant write and read operations. It achieves this by doing as many sequential writes and reads as possible, allowing it to make use of hardware such as hard disk drives efficiently, as well as send large amounts of data over the network sequentially.
- **Scalability:** Developers can leverage Kafka's topic partitioning to achieve more performant parallel processing of their data.
- **Flexible durability:** Kafka allows users to configure the policies for storing data, such as message retention. Messages can be stored for a fixed amount of time (for example, 7 days) or indefinitely until there is enough space on the data storage system.

Note



While Kafka provides many benefits for developers, it is important to note that it is a fairly complex infrastructure component that may be nontrivial to manage and maintain. We are going to use it in this chapter for illustrative purposes while considering its wide adoption and popularity in the developer community. In this chapter, we will avoid the difficulties of setting up a Kafka cluster by using its Docker version, but for production use cases, you may need to become familiar with the relevant Kafka maintenance documentation, available at <https://kafka.apache.org/documentation/>.

- Let's explore how we can leverage the benefits offered by Kafka for the microservices we developed in the previous chapters.

Adopting Kafka for our microservices

Let's get back to the rating service example from the previous chapters. The service provided a synchronous API for inserting rating records, allowing its callers to call an endpoint and get an immediate response from the service. Such an API would be useful in many practical use cases, including one where the user submits a rating from a user interface or a web form.

Now, consider a scenario where we work with a data provider that frequently publishes rating records (for example, movie ratings from a popular movie database, such as IMDb) that we can use in our rating service. Here, we would need to consume such records and ingest them into our system so that we could use them in addition to the data that was created through our API. The publisher-subscriber model that we described earlier in this chapter would be a great fit for this use case – the publisher would be the data provider that provides the rating data, and the subscriber would be a part of our application (such as a rating service), which would consume the data.

We can illustrate the described model using the following diagram:

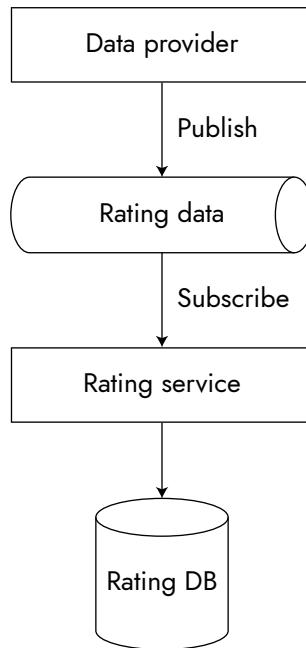


Figure 6.3 – The publisher-subscriber model of rating ingestion from a data provider

This model, which shows the interaction between the data provider and the rating service, is a perfect example of asynchronous communication – the rating service does not necessarily need to process the provider's data immediately. It is up to us when and how to consume this data – our

rating service could do this periodically (for example, once an hour or once a day), or handle the new rating data as soon as it gets published. We'll consider the second approach in this chapter.

The only missing piece in our model is the component that allows us to publish the rating data from the data provider and subscribe to it from our rating service. Apache Kafka, which we described earlier, is a great fit for this use case – it provides a performant, scalable, and durable solution for producing and consuming arbitrary data, allowing us to use it as a rating data message broker.

To illustrate the model that we have just described, let's implement the following logic:

- A new example application that will produce rating data for Apache Kafka
- Logic in the rating service to consume the rating data from Apache Kafka and save it to our rating database

Before we implement both components, we need to decide which data serialization format to use between them. For simplicity, let's assume the data provider provides us with the rating data in JSON format. An example of the provided rating data is as follows:

```
[{"userId": "105", "recordId": "1", "recordType": 1, "value": 5, "providerId": "test-provider", "eventType": "put"}, {"userId": "105", "recordId": "2", "recordType": 1, "value": 4, "providerId": "test-provider", "eventType": "put"}]
```

Let's define a Go structure for such rating records. In the `src/rating/pkg/model/rating.go` file, add the following code:

```
// RatingEvent defines an event containing rating information.
type RatingEvent struct {
    Rating
    ProviderID string      'json:"providerId"'
    EventType  RatingEventType 'json:"eventType"'
}

// RatingEventType defines the type of a rating event.
type RatingEventType string

// Rating event types.
const (
    RatingEventTypePut      = RatingEventType("put")
    RatingEventTypeDelete   = RatingEventType("delete")
)
```

In our code, we used a technique called **type embedding** – we embedded a Rating type into a RatingEvent structure so that we don't need to redefine the same fields that the Rating type has.

Now, let's implement the example application, which reads rating data from a provided file and produces it in Kafka. Create a cmd/ratingproducer directory and add a main.go file that contains the following code:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
    "time"

    "github.com/confluentinc/confluent-kafka-go/kafka"
    "movieexample.com/rating/pkg/model"
)

func main() {
    fmt.Println("Creating a Kafka producer")

    producer, err := kafka.NewProducer(&kafka.ConfigMap{"bootstrap.
servers": "localhost"})
    if err != nil {
        panic(err)
    }
    defer producer.Close()

    const fileName = "ratingsdata.json"
    fmt.Println("Reading rating events from file " + fileName)

    ratingEvents, err := readRatingEvents(fileName)
    if err != nil {
        panic(err)
    }
}
```

```
    const topic = "ratings"
    if err := produceRatingEvents(topic, producer, ratingEvents); err !=
nil {
    panic(err)
}

const timeout = 10 * time.Second
fmt.Println("Waiting " + timeout.String() + " until all events get
produced")

producer.Flush(int(timeout.Milliseconds()))
}
```

In the code that we just added, we initialize a Kafka producer by calling `kafka.NewProducer`, read the rating data from a file, and produce rating events containing the rating data in Kafka. Note that we import the `github.com/confluentinc/confluent-kafka-go/kafka` Kafka library – a Kafka client made by Confluent, a company founded by the creators of Kafka. There are multiple popular open source Kafka libraries for Go, including `github.com/Shopify/sarama`, which is well maintained and is widely used across many Go projects. You can use either library in your projects, depending on your preference.

Now, let's add a function for reading rating events to the file we just created:

```
func readRatingEvents(fileName string) ([]model.RatingEvent, error) {
    f, err := os.Open(fileName)
    if err != nil {
        return nil, err
    }
    defer f.Close()
    var ratings []model.RatingEvent
    if err := json.NewDecoder(f).Decode(&ratings); err != nil {
        return nil, err
    }
    return ratings, nil
}
```

Finally, add a function for producing rating events:

```
func produceRatingEvents(topic string, producer kafka.Producer, events []model.RatingEvent) error {
    for _, ratingEvent := range events {
        encodedEvent, err := json.Marshal(ratingEvent)
        if err != nil {
            return err
        }

        if err := producer.Produce(&kafka.Message{
            TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.PartitionAny},
            Value:          []byte(encodedEvent),
        }, nil); err != nil {
            return err
        }
    }
    return nil
}
```

Let's describe some parts of the code that we just wrote:

- We created a Kafka producer by calling a `kafka.NewProducer` function and providing `localhost` as the Kafka address for testing it locally.
- The program that we created is expected to read rating data from the `ratingsdata.json` file.
- When we produce events for Kafka using a `Produce` function, we specify a topic partition using a `kafka.TopicPartition` structure. In this structure, we provide the topic name (in our example, `ratings`) and the topic partition (in our example, we use `kafka.PartitionAny` to produce a partition – we will cover this later, in the *Asynchronous communication best practices* section).
- At the end of our `main` function, we call the `Flush` function to make sure all messages are sent to Kafka.
- The function that we just created uses the `github.com/confluentinc/confluent-kafka-go/kafka` library, which we need to include in our Go module. Let's do this by running the following code:

```
go mod tidy
```

Let's also add a file containing the rating events. In the directory that we just used, create a `ratingsdata.json` file that contains the following code:

```
[{"userId": "105", "recordId": "1", "recordType": 1, "value": 5, "providerId": "test-provider", "eventType": "put"}, {"userId": "105", "recordId": "2", "recordType": 1, "value": 4, "providerId": "test-provider", "eventType": "put"}]
```

With that, our application is ready. We have implemented the logic to read the rating data from a file and publish it to Apache Kafka for further consumption by the rating service. Now, let's implement the logic in the rating service for consuming the published data. Create a `rating/internal/ingester/kafka` directory and add an `ingester.go` file with the following contents:

```
package kafka

import (
    "context"
    "encoding/json"
    "fmt"
    "rating/pkg/model"

    "github.com/confluentinc/confluent-kafka-go/kafka"
    "movieexample.com/rating/pkg/model"
)

// Ingester defines a Kafka ingester.
type Ingester struct {
    consumer *kafka.Consumer
    topic    string
}

// NewIngester creates a new Kafka ingester.
func NewIngester(addr string, groupID string, topic string) (*Ingester, error) {
    consumer, err := kafka.NewConsumer(&kafka.ConfigMap{
        "bootstrap.servers": addr,
        "group.id":          groupID,
        "auto.offset.reset": "earliest",
    })
}
```

```
if err != nil {
    return nil, err
}
return &Ingestor{consumer, topic}, nil
}
```

Additionally, add this piece of code to it:

```
// Ingest starts ingestion from Kafka and returns a channel containing
// rating events representing the data consumed from the topic.
func (i *Ingestor) Ingest(ctx context.Context) (chan model.RatingEvent,
error) {
    fmt.Println("Starting Kafka ingestor")
    if err := i.consumer.SubscribeTopics([]string{i.topic}, nil); err !=
nil {
        return nil, err
}

ch := make(chan model.RatingEvent, 1)
go func() {
    for {
        select {
        case <-ctx.Done():
            close(ch)
            i.consumer.Close()
        default:
        }
        msg, err := i.consumer.ReadMessage(-1)
        if err != nil {
            fmt.Println("Consumer error: " + err.Error())
            continue
        }
        fmt.Println("Processing a message")
        var event model.RatingEvent
        if err := json.Unmarshal(msg.Value, &event); err != nil {
            fmt.Println("Unmarshal error: " + err.Error())
            continue
        }
    }
}
```

```
        ch <- event
    }
}()

return ch, nil
}
```

In the code we just created, we have implemented a `NewIngestor` function to create a new Kafka ingestor, the component that will ingest rating events from it. The `Ingest` function starts message ingestion in the background and returns a Go channel with `RatingEvent` structures.

You may have noticed that in our call to the `ReadMessage` function, we provided `-1` as an argument. Here, we specified a **consumer offset** – a checkpoint from which we should consume the messages from our topic. The value of `-1` is specific to Kafka and means that we will always consume from the beginning of the topic, reading all existing messages.

Let's use this structure in our rating service controller. In our `rating/internal/controller/controller.go` file, add the following code:

```
type ratingIngestor interface {
    Ingest(ctx context.Context) (chan model.RatingEvent, error)
}

// StartIngestion starts the ingestion of rating events.
func (s *RatingService) StartIngestion(ctx context.Context) error {
    ch, err := s.ingester.Ingest(ctx)
    if err != nil {
        return err
    }
    for e := range ch {
        fmt.Printf("Consumed a message: %v\n", e)
        if err := s.PutRating(ctx, e.RecordID, e.RecordType, &model.Rating{UserID: e.UserID, Value: e.Value}); err != nil {
            return err
        }
    }
    return nil
}
```

Here, we call the `Ingest` function and receive a Go channel containing rating events from the topic. We iterate over it using the `for` operator. It keeps returning available rating events until the channel is closed (for example, when the Kafka client is closed on service shutdown).

Now, update the existing `RatingService` structure and the `New` function in this file to the following:

```
// RatingService encapsulates the rating service business logic.
type RatingService struct {
    repo      ratingRepository
    ingestor  ratingIngestor
}

// New creates a rating service.
func New(repo ratingRepository, ingestor ratingIngestor) *RatingService {
    return &RatingService{repo, ingestor}
}
```

Now, our rating service can asynchronously consume rating events from Kafka and execute the `Put` function for each one, writing it to the rating database. The remaining step for the rating service is to update its `main.go` file:

1. First, add an extra import of `movieexample.com/rating/internal/ingester/kafka` package to it.
2. Find the line where we initialize the `ctrl` variable and replace it with the following code:

```
ingester, err := kafka.NewIngestor("localhost", "rating", "ratings")
if err != nil {
    log.Fatalf("failed to initialize ingestor: %v", err)
}
ctrl := rating.New(repo, ingestor)
if err := ctrl.StartIngestion(ctx); err != nil {
    log.Fatalf("failed to start ingestion: %v", err)
}
```

At this point, the rating service provides both a synchronous API for the callers that want to create ratings in real time and asynchronous logic for ingesting rating events from Apache Kafka.

Let's run our rating service so that we can test message ingestion. Navigate to the `rating/cmd` directory and run its `main.go` file. You will see regular messages indicating that the service has started, but we haven't produced any data for our topic yet – let's do this now without stopping the rating service.

Create a file called `docker-compose.yml` in our `cmd/ratingproducer` directory that contains the following code:

```
version: '3'

services:
  zookeeper:
    image: wurstmeister/zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka
    container_name: kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

Here, we define the Apache Kafka configuration required for running it locally in a Docker container. Note that the file also contains Zookeeper configuration – this is required by Kafka.



Credit

Big thanks to Jason Salas (<https://github.com/jasonsallas>), who shared this code example on GitHub after reading the first edition of this book.

Now, let's run Kafka in a Docker container. Go to our project's root directory and run the following commands:

```
docker-compose -d up
docker exec -it kafka /bin/sh
cd /opt/kafka_<YOUR KAFKA VERSION>/bin
kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1
--partitions 1 --topic ratings
```

Then, open a separate terminal and run the `ratingproducer` application:

```
cd cmd/ratingproducer
go run main.go
```

Our application produced rating events from a JSON file. Open the window that contains the running rating service; you should see the following messages in its output:

```
Consumed a message: {105 1 movie 5 test-provider put}  
Consumed a message: {105 2 movie 4 test-provider put}
```

These new messages from our rating service confirm that we successfully received the data we just asynchronously produced through our `ratingproducer` application via Kafka. Now, we can communicate between the services both synchronously and asynchronously.

At this point, we have covered the basics of asynchronous communication and can proceed to the final part of this chapter, where we'll cover some best practices you should keep in mind while using this model.

Asynchronous communication best practices

In this section, we are going to cover the best practices of using the asynchronous communication model. You will learn about some high-level recommendations for adopting the model in your applications and using it in a way that would maximize its benefits for you.

Versioning

Versioning is the technique of associating the format (or a schema) of the data with its version. Imagine you are working on a rating service, and you use a publisher-subscriber model for producing and consuming rating events. If at some point the format of your rating events gets changed, some of the events that are already being produced will have an old data format, and some will have the new one. This situation may be hard to handle because the logic consuming such data would need to know how to differentiate between such formats and how to handle each one. Differentiating between two formats without knowing the data schema or its version could be a nontrivial task. Imagine that we have two JSON events:

```
{"recordID": "1", "rating": 5}  
{"recordID": "2", "rating": 17, "userId": "alex"}
```

The second event has a `userId` field that is not present in the first. Is it because the producer did not provide it or because the data format did not have this field before?

Providing the schema version explicitly would help the data consumer handle this problem. Consider these updated examples:

```
{"recordID": "1", "rating": 5, "version": 1}  
{"recordID": "2", "rating": 17, "userId": "alex", "version": 2}
```

In these examples, we know the versions of the events and can now handle each one separately. For example, we may completely ignore events of a certain version (assume there was an application bug and we want to re-process events with an updated version instead) or use the version-specific validation (for instance, allow the records without a `userId` field for the first version, but disallow for the higher versions).

Versioning is very important to systems that can evolve over time because it makes dealing with different data formats easier. Even if you don't expect your data format to change, consider using versioning to increase your system's maintainability in the future.

Leverage partitioning

In the code examples provided in the *Adopting Apache Kafka for our microservices* section, we implemented the logic for producing our data to message topics in Apache Kafka. The function for producing a message was as follows:

```
if err := p.Produce(&kafka.Message{  
    TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.  
        PartitionAny},  
    Value:           []byte(encodedEvent),  
}, nil); err != nil {  
    return err  
}
```

In this function, we used the `kafka.PartitionAny` option. As we mentioned in the *Apache Kafka basics* section, Kafka topics can be partitioned to allow multiple consumers to consume different partitions of a topic. Imagine you have a topic with three partitions – you can consume each one independently, as illustrated in the following diagram:

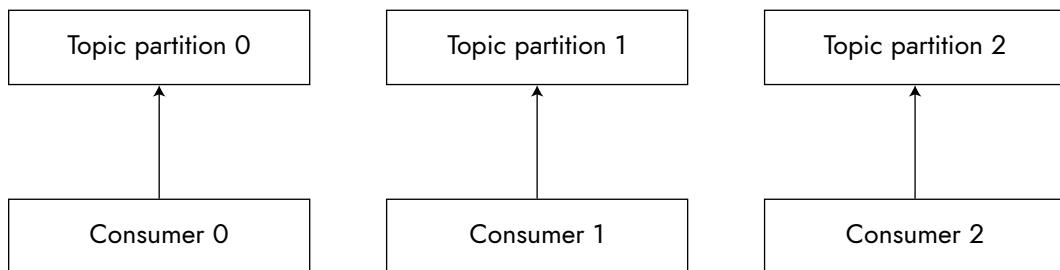


Figure 6.4 – A partitioned topic consumption example

You can control the number of topic partitions, as well as the partition for each message your services produce. Setting a partition manually may help you to achieve **data locality** – the ability to co-locate the data for various records, storing it together (in our use case, in the same topic partition). For example, you can partition the data using a user identifier, making sure the data for any user is stored on a single topic partition, helping you simplify the data search across the topic partitions.

It is important to note that topic partitioning might bring an extra challenge with message ordering. Messages within each partition will be ordered according to the time they were written to Kafka. However, if your data is produced into multiple partitions, order will only be kept within each partition. If you produce multiple messages – for example, A, B, C – and all of them get written into different partitions of the same topic, you can get them in various orders, including A, C, B, or even C, B, A. The exact order would depend on your producer and consumer configuration, as well as various factors, such as the timing of the events and even consumption speed. Hence, to achieve the correct ordering of events (for example, to make all messages for each movie come into the same partition), you can control which partition you write messages to, as shown in the following example:

```
const partitionCount := 3
partitionFunc := func(event model.RatingEvent) int32 {
    h := fnv.New32a()
    h.Write([]byte(event.RecordID))
    return int32(h.Sum32()) % partitionCount
}
producer.Produce(&kafka.Message{
    TopicPartition: kafka.TopicPartition{Topic: &topic, Partition:
partitionFunc(event)},
    Value:          []byte(encodedEvent),
})
```

In our example, we manually calculated a partition for input rating using the RatingID field to make sure that we always get the same partition number for each distinct RatingID value.

Use explicit message acknowledgment whenever necessary

Some message broker systems, such as Apache Kafka, allow you to control when each message is acknowledged as delivered. By default, many Kafka libraries, including confluent-kafka-go, mark all previously received messages as delivered every few seconds via a mechanism called **off-**

set commits. With this mechanism, for each topic partition that's consumed, the last consumed offset is periodically saved to avoid duplicate delivery. However, such periodic acknowledgment of received messages has some downsides, such as the possibility of message loss. Consider a scenario where you receive a message and run complex processing logic, which takes a long time to complete. In such a case, if message offset is committed before the message gets processed, there is a chance that any unexpected error might result in that message being unprocessed. To avoid this, Apache Kafka allows you to commit offsets manually. The following code illustrates this approach:

```
consumer, err := kafka.NewConsumer(&kafka.ConfigMap{
    "bootstrap.servers": addr,
    "group.id":          groupID,
    "auto.offset.reset": "earliest",
    "enable.auto.commit": false,
    "enable.auto.offset.store": false,
})
if err != nil {
    return err
}
msg, err := i.consumer.ReadMessage(time.Minute)
if err != nil {
    return err
}
// Process your message.
// ...
// Acknowledge message delivery.
_, err = consumer.CommitMessage(msg)
If err != nil {
    return err
}
```

In our example, we call the `CommitMessage` function only after we process the message so that we don't lose track of it in case we experience a processing error. Note that we also specified two configuration parameters to make this work: `enable.auto.commit` and `enable.auto.offset.store` should be set to `false` to make such explicit offset commits work.

Use a separate topic for unprocessed messages

One of the main challenges of asynchronous communication is error handling. Consider the following scenario: you read messages from a Kafka topic that contains updates to movie metadata. Most messages are processed successfully; however, one message contains metadata for a movie that is not stored in our system (let's assume its data is imported separately and the import has not happened yet). In such a scenario, you generally have two options:

1. Write an error and retry to process the failed message.
2. Skip the failed message and proceed to the next one.

The first option might not result in anything: even after the maximum number of retries, you might still experience the same error. In many cases, you also can't retry indefinitely: when you process messages sequentially, retrying to process a single message blocks the following ones from being processed. The second option would result in ignoring the failed message, and sometimes, this might cause other issues, such as data inconsistency.

A common solution to such a problem is to introduce a separate topic for unprocessed messages: if we experience an unexpected error and can't continue re-processing the same message after the maximum number of retries, we can write it to a separate topic and try to re-process it later (for example, after we've identified and fixed the underlying problem). This approach is often called **dead letter queue (DLQ)** and can be helpful in multiple other cases:

- **Message poisoning:** Some messages might contain corrupted or unexpected metadata and require separate fixing.
- **Message expiry:** Certain messages might take too long to be processed, so instead of blocking the execution, you might just store and re-process them separately.
- **Unexpected error handling:** Imagine that you get millions of input messages and only one of them causes your service to panic. Moving the message to a separate topic for further inspection would help to avoid blocking message processing.

The list of best practices provided here is not comprehensive. It does not cover all recommendations for using asynchronous communication in your microservices, but it provides some great ideas of what you should consider. Become familiar with the articles listed in the *Further reading* section for some additional ideas and recommendations.

Summary

In this chapter, we covered the basics of asynchronous communication and illustrated how to use it in your microservices. You learned the benefits of asynchronous communication and its common patterns, such as publisher-subscriber and message broker. In addition to this, we covered the basics of the Apache Kafka tool and illustrated how to use it in our microservices by implementing the logic for producing and consuming data from it. This knowledge should help you establish efficient communication between your microservices in a wide variety of scenarios, including complex distributed environments.

In the next chapter, we are going to cover another important topic of microservice development: deployment and orchestration. You will learn how to build and deploy your application so that you can use it in more advanced environments, such as cloud infrastructure.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Apache Kafka documentation: <https://kafka.apache.org/documentation/>
- Publish-subscribe pattern: https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern
- Asynchronous message-based communication: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication>

7

Storing Service Data

In this chapter, we are going to review the very important topic of establishing durable storage of our service data. In the previous chapters, we stored movie metadata and user ratings using in-memory repositories. While it was easy to implement in-memory storages of our data, using them would be impractical due to many reasons. One such reason is a lack of a **persistence** guarantee: if our service instances storing the data were restarted (for example, due to application failure or on host restart), we would lose all our data that was stored in the memory of a previously running instance. To guarantee that our data won't be lost over time, we need a solution that can persist our data and allow us to read and write it to our microservices. Among such solutions are databases, which we are going to review in this chapter.

We will cover the following topics:

- Introduction to databases
- Using MySQL to store our service data
- Implementing data caching

Technical requirements

To complete this chapter, you will need Go 1.18 or above. Additionally, you will need the following tools:

- **Docker:** <https://www.docker.com>
- **grpcurl:** <https://github.com/fullstorydev/grpcurl>

You can find the GitHub code for this chapter here: <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter07>.

Introduction to databases

- **Databases** are systems that allow us to store and retrieve different types of data. Databases offer a variety of guarantees related to data storage, among which is **durability** – a guarantee that all records and any related data changes will be persistent over time. A durability guarantee helps ensure that the data stored in a database won't be lost in case of various events such as software and hardware restarts, which are pretty common for microservices.
- Databases help solve lots of different other problems related to data storage. Let's illustrate one such problem using the metadata service that we created in *Chapter 2, Scaffolding a Go Microservice*. In our metadata service code, we implemented an in-memory repository for storing and retrieving the movie data that provides two functions, Get and Put. If we have just one instance of the metadata service, all its callers would be able to successfully write and read metadata records from the service memory, so long as a service instance does not perform a restart. However, let's imagine that we add another instance of a metadata service, as illustrated in the following diagram:

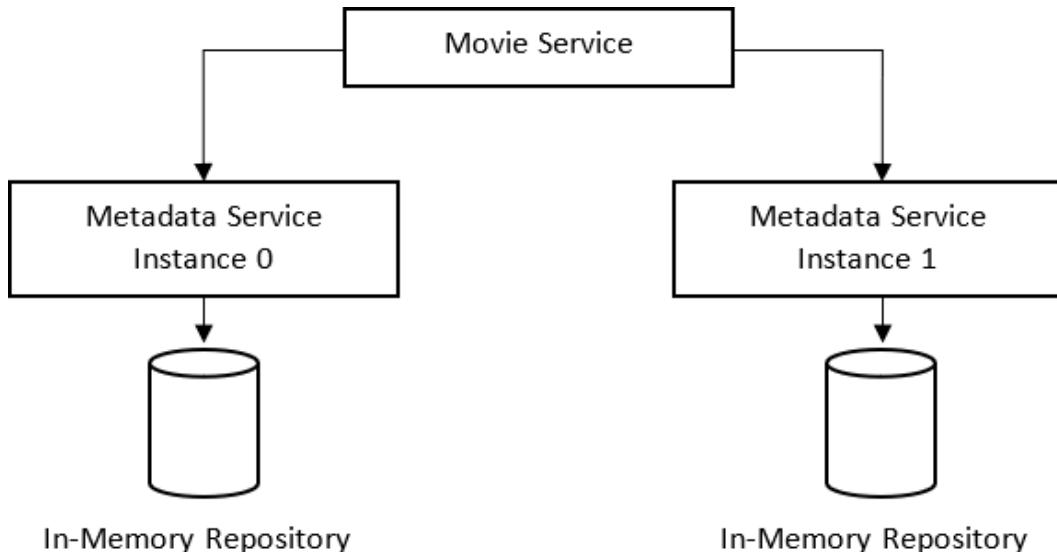


Figure 7.1 – Interaction between the movie service and two instances of the metadata service

Imagine that the movie service wants to write movie metadata and calls the metadata service to do this. The movie service instance would pick an instance of a metadata service (let's assume it picks instance 0) and send a write request to it, storing the record in memory of the instance that processes the request.

Now, let's assume the movie service wants to read the previously stored movie metadata and sends a read request to the metadata service. Depending on which instance handles the request, there would be two possible outcomes:

- **Instance 0:** Successfully return the previously saved movie metadata
- **Instance 1:** Return ErrNotFound

We just illustrated a case where the data is inconsistent between the two instances of the metadata repository. Because we have not implemented any coordination between our in-memory metadata repositories, each one acts as an independent data storage. Using our metadata service in such a way would be highly impractical: each newly added service instance would store a completely independent dataset. Additionally, on any service restart, in-memory data stored on each instance would get lost, now allowing us to durably store our service data.

To solve our data inconsistency and durability problems, we can use a database to store the movie metadata: a database will handle all writes and reads from metadata service instances, as well as provide additional durability and reliability guarantees.

A scenario where multiple service instances interact with a database is illustrated in the following diagram:

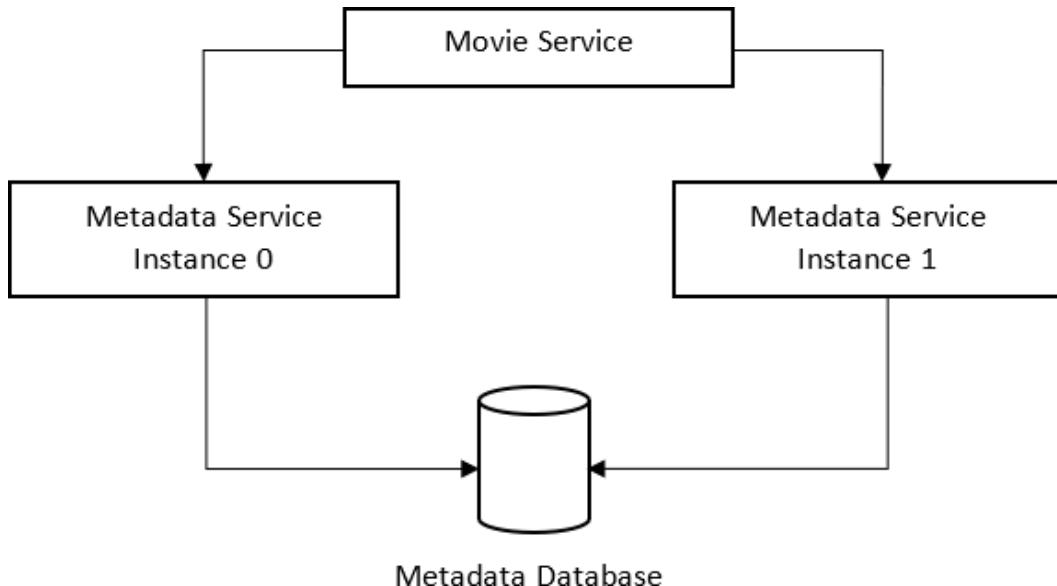


Figure 7.2 – Using a shared metadata database among multiple instances

In our diagram, multiple instances of a metadata service are using a shared database. This helps us persist the data coming from different metadata service instances, as well as establish a common mechanism of querying our data by them.

Common database features

In addition to durability, databases provide various additional features:

- **Transaction support:** Many databases support transactions – types of data changes – that have the following properties, abbreviated as ACID:
 - **Atomicity:** A change either happens entirely or does not happen at all
 - **Consistency:** A change brings a database from one valid state to another
 - **Isolation:** Concurrent changes get executed as if they were executed sequentially
 - **Durability:** All changes get stored permanently
- **Data replication:** A database can offer data replication – a mechanism of duplicating data to additional instances, called **replicas**. Replication can help make the database more resilient to data losses (for example, when a database host becomes unavailable, data can be accessed on its replica) and reduce read latency (for example, when a user reads data from a replica that is located closer to them).
- **Additional query capabilities:** There are many databases (such as MySQL and PostgreSQL), that offer support for different query languages, such as SQL (<https://en.wikipedia.org/wiki/SQL>).

The exact features provided by each database vary depending on its type. In the following section, we are going to provide a brief overview of common database types.

Common database types

The following list includes some widely used types of databases:

- **Key-value databases:** These are databases that store data in a key-value format, where each record contains a key (for example, user identifier) and a value (for example, user metadata). Keys and values are often represented as strings or byte arrays. Operations provided by key-value databases are often limited to key-based writes (store a value for the provided key) and key-based reads (read a value for the provided key). Because of their functional simplicity, key-value databases are among the most performant since they don't involve any complex data processing or indexing.

- **Relational databases:** These are databases that store the data as a collection of tables, each consisting of a set of rows and columns. Users can run SQL queries to retrieve the data from one or multiple tables, being able to join the data between them or perform complex searches based on a variety of conditions. Historically, relational databases have been the most popular across all database types due to their ability to execute queries of any complexity, as well as their ability to store various types of structured (having a well-defined schema that maps row data to table columns) data.
- **Document databases:** These are databases that store data in a document format, such as JSON or XML. Document databases don't require you to define data schemas, so they're a great fit for storing various kinds of differently structured documents (for example, a collection of YAML files containing movie metadata from multiple websites in different formats).
- **Graph databases:** These are databases that store information in the form of *vertices* – objects that have different properties (for example, user details) – and *edges* – the relationships between vertices (for example, if user A is following user B). Graph databases are different from relational databases in terms of the types of read queries they offer: most graph databases support *traversal* queries (checking each vertex in a graph), *connectivity* queries (getting all vertices connected to the target one), and many other ones.
- **Blob databases:** These are databases for storing **binary large object (blob)** data, such as audio or video files. Blob records are generally immutable (their content never gets changed after a successful write), so blob databases are well optimized for *append-only* writes (such as writing new files), as well as blob reads (such as retrieving the contents of a large file).

We won't go into the details of various types of databases because this is a topic for a separate book, but it is important to note that there is no *one-size-fits-all* solution among all of them, and each database provides a unique set of features that can be useful for solving a specific problem. For example, if the only purpose of your service is to store files, using a blob database would be sufficient, while a graph database could help with building a social network to store user relationship data. However, many use cases can be modeled using the relational model that powers relational databases: since it was introduced in 1970 by E. F. Codd, it has been used across the software development industry for nearly all types of problems. Popular relational databases, such as MySQL and PostgreSQL, remain the top-used software solutions, helping to build various types of applications, from tiny services running on a single host to large-scale clusters spanning hundreds of thousands of hosts. Because of the wide adoption and maturity of popular relational databases, many companies use them as a standard way to store various types of data. We are going to illustrate how to store our microservice data using a popular relational database, MySQL.

Using MySQL to store our service data

In this section, we are going to provide a brief overview of MySQL and demonstrate how to write and read data from the microservices that we created in the previous chapters.

MySQL is an open source relational database that was created in 1995 and since then has become one of the top-used databases across the development industry, according to DB-Engines Ranking (<https://db-engines.com/en/ranking>). It stores data as a set of tables, each consisting of rows and columns of predefined types (such as string, numeric, binary, or more), and allows you to access data via SQL queries. For example, assume you have the following data, stored as a table called **movies**:

id	title	director
922	New York Stories	John Jones
1055	Christmas Day	Ben Miles
1057	Sunny Weather 3	Ben Miles

Table 7.1 – Movie table example

A SQL query to get all movies filmed by a particular director would look as follows:

```
SELECT * FROM movies WHERE director = "Ben Miles"
```

Now, let's imagine we have a table called **ratings** that contains the following data:

record_id	record_type	user_id	rating
1055	movie	alex001	5
1055	movie	chris.rocks	3
1057	movie	alex001	4

Table 7.2 – Rating table example

Using SQL language, we can write a more complex query to get all ratings that are associated with movies of a certain director:

```
SELECT * FROM ratings r INNER JOIN movies m ON r.record_id = m.id WHERE
r.record_type = "movie" AND m.director = "Ben Miles"
```

In our SQL query, we perform a **join** of two tables – an operation that allows us to group the data belonging to two tables and perform additional filtering (in our case, we will only select rating events where the `record_type` column’s value is equal to *movie* and the `director` column’s value is equal to *Ben Miles*).

To demonstrate how to use MySQL to store our service data, let’s define which data we want to store and how we want to access it. Let’s start with the metadata service, which performs two data storage operations:

- Stores movie metadata for a given movie ID
- Gets movie metadata for a given movie ID

Now, let’s review the data schema of the movie metadata object. Our movie metadata contains the following fields:

- **ID**: String
- **Title**: String
- **Description**: String
- **Director**: String

Now, let’s see which data is stored by our rating service, which performs the following storage-related operations:

- Stores a rating for a given record (identified by a combination of a record ID and its type)
- Gets all ratings for a given record

Let’s review the data schema of a rating:

- **User ID**: String
- **Record ID**: String
- **Record type**: String
- **Rating value**: Integer

At this point, we know which data we want to store in the database and can set up our database. We will use a Docker version of MySQL that you can run by executing the following command:

```
docker run --name movieexample_db -e MYSQL_ROOT_PASSWORD=password -e  
MYSQL_DATABASE=movieexample -p 3306:3306 -d mysql:latest
```

In our command, we set the password for the MySQL root user to `password` so that we can use it for testing. We also set the database name to `movieexample` and exposed it on port 3306 so that we can use it to access our MySQL database.

Let's verify that our container started successfully. Run the following command to see a list of running Docker containers:

```
docker ps
```

The output should include a container with a `movieexample_db` name, a `mysql:latest` image, and an `Up` status.

The next step is to create our data schema. We will define it in a separate folder in our `src` directory, called `schema`. Create this directory and a `schema.sql` file in it, and add the following code to the newly created file:

```
CREATE TABLE IF NOT EXISTS movies (id VARCHAR(255) PRIMARY KEY, title  
VARCHAR(255), description TEXT, director VARCHAR(255));  
CREATE TABLE IF NOT EXISTS ratings (record_id VARCHAR(255), record_type  
VARCHAR(255), user_id VARCHAR(255), value INT, PRIMARY KEY (record_id,  
record_type, user_id));
```

In our schema file, we define two tables, called `movies` and `ratings`. The tables that we just defined consist of `VARCHAR(255)` and `TEXT` columns. The `VARCHAR` type is a MySQL type for storing string data, and 255 is the maximal size of a column value. The `TEXT` type is another MySQL type that is often used for storing long text records, so we used it for storing movie descriptions that may contain long texts.

Now, let's connect to our newly provisioned database and initialize our data schema. Run the following command inside the `src` directory of our project:

```
docker exec -i movieexample_db mysql movieexample -h localhost -P 3306  
--protocol=tcp -uroot -ppassword < schema/schema.sql
```

If everything worked correctly, our database should be ready to use. You can check if the tables were created successfully by running the following command:

```
docker exec -i movieexample_db mysql movieexample -h localhost -P 3306  
--protocol=tcp -uroot -ppassword -e "SHOW tables"
```

The output of the preceding command should include our two tables:

```
Tables_in_movieexample  
movies  
ratings
```

We are ready to implement the logic to write and read from it. Create a `metadata/internal/repository/mysql` directory and add a file called `mysql.go` to it with the following contents:

```
package mysql

import (
    "context"
    "database/sql"
    "metadata/pkg/model"

    _ "github.com/go-sql-driver/mysql"
    "movieexample.com/metadata/internal/repository"
    "movieexample.com/metadata/pkg/model"
)

// Repository defines a MySQL-based movie metadata repository.
type Repository struct {
    db *sql.DB
}

// New creates a new MySQL-based repository.
func New() (*Repository, error) {
    db, err := sql.Open("mysql", "root:password@/movieexample")
    if err != nil {
        return nil, err
    }
    return &Repository{db}, nil
}
```

In our code, we defined a MySQL-based repository that we will use to store and retrieve the movie metadata. Note that we added the following line to our imports:

```
_ "github.com/go-sql-driver/mysql"
```

The line that we added initializes a Go MySQL **driver**, which is required to access our MySQL database. We also used `root:password@/movieexample` inside our `New` function – the value is called a **connection string**, and it includes the name of the user, its password, and the name of the database to be connected. The connection string may also include the name of the host, MySQL port, and other values, but we don't need to set them as we are using the default values to access the local version of MySQL.



Note

Please note that storing database credentials in code is a bad practice, and it is recommended to store such data (often called *secrets*) separately: for example, as separate configuration files. In *Chapter 8 Setting Up Service Deployments*, we will review how to create and use configuration files with Go microservices.

Now, add the following code to the file that we just created:

```
// Get retrieves movie metadata for by movie id.  
func (r *Repository) Get(ctx context.Context, id string) (*model.Metadata, error) {  
    var title, description, director string  
    row := r.db.QueryRowContext(ctx, "SELECT title, description, director  
FROM movies WHERE id = ?", id)  
    if err := row.Scan(&title, &description, &director); err != nil {  
        if err == sql.ErrNoRows {  
            return nil, repository.ErrNotFound  
        }  
        return nil, err  
    }  
    return &model.Metadata{  
        ID:         id,  
        Title:      title,  
        Description: description,  
        Director:   director,  
    }, nil  
}  
  
// Put adds movie metadata for a given movie id.  
func (r *Repository) Put(ctx context.Context, id string, metadata *model.  
Metadata) error {  
    _, err := r.db.ExecContext(ctx, "INSERT INTO movies (id, title,  
description, director) VALUES (?, ?, ?, ?)",  
    id, metadata.Title, metadata.Description, metadata.Director)  
    return err  
}
```

In our code, we implemented the `Get` and `Put` functions so that we can store and retrieve the movie metadata from MySQL. Inside our `Get` function, we use the `QueryRowContext` function of our database instance to read a single row from our table. In the case of a query error, we check if it is equal to `sql.ErrNoRows`; if so, we return `ErrNotFound`.

Now, let's implement our MySQL rating repository. Create a `rating/internal/repository/mysql` directory and add a `mysql.go` file to it with the following contents:

```
package mysql

import (
    "context"
    "database/sql"
    "errors"

    _ "github.com/go-sql-driver/mysql"
    "movieexample.com/rating/pkg/model"
)

// Repository defines a MySQL-based rating repository.
type Repository struct {
    db *sql.DB
}

// New creates a new MySQL-based rating repository.
func New() (*Repository, error) {
    db, err := sql.Open("mysql", "root:password@/movieexample")
    if err != nil {
        return nil, err
    }
    return &Repository{db}, nil
}
```

So far, our rating repository code is similar to the metadata repository. Inside the same file, let's implement two functions to read and write rating data:

```
// Get retrieves all ratings for a given record.
func (r *Repository) Get(ctx context.Context, recordID model.RecordID,
recordType model.RecordType) ([]model.Rating, error) {
    rows, err := r.db.QueryContext(ctx, "SELECT user_id, value FROM
ratings WHERE record_id = ? AND record_type = ?", recordID, recordType)
    if err != nil {
        return nil, err
    }
    defer rows.Close()
    var res []model.Rating
    for rows.Next() {
        var userID string
        var value int32
        if err := rows.Scan(&userID, &value); err != nil {
            return nil, err
        }
        res = append(res, model.Rating{
            UserID: model.UserID(userID),
            Value:  model.RatingValue(value),
        })
    }
    return res, nil
}

// Put adds a rating for a given record.
func (r *Repository) Put(ctx context.Context, recordID model.RecordID,
recordType model.RecordType, rating *model.Rating) error {
    if rating == nil {
        return errors.New("rating is nil")
    }
    _, err := r.db.ExecContext(ctx, "INSERT INTO ratings (record_id,
record_type, user_id, value) VALUES (?, ?, ?, ?, ?)",
        recordID, recordType, rating.UserID, rating.Value)
    return err
}
```

In our Get handler, we use the Query function to read rating rows from our table. We scan each row by calling the rows.Scan function, converting MySQL data into the necessary structures.

Our repository code is ready, so we can import the newly used package, `github.com/go-sql-driver/mysql`, by running the following command:

```
go mod tidy
```

Let's verify that our logic is correct by manually testing the rating repository:

1. Inside the `rating/cmd/main.go` file, change the `movieexample.com/rating/internal/repository/memory` import to `movieexample.com/rating/internal/repository/mysql`.
2. Inside the same file, find the following block:

```
repo := memory.New()
```

3. Change it to the following:

```
repo := mysql.New()  
if err != nil {  
    panic(err)  
}
```

4. Navigate to the `cmd` directory of the rating service and run the following command:

```
go run *.go
```

5. Make a manual request to write a rating:

```
grpcurl -plaintext -d '{"record_id": "1", "record_type": "movie"}'  
localhost:8082 RatingService/GetAggregatedRating
```

You should see the following message:

```
ERROR:  
Code: NotFound  
Message: ratings not found for a record
```

6. Now, let's write a rating to test that our database works correctly. Execute the following command:

```
grpcurl -plaintext -d '{"record_id": "1", "record_type": "movie",  
"user_id": "alex", "rating_value": 5}' localhost:8082 RatingService/  
PutRating
```

7. Now, let's fetch an updated rating for the same movie. Execute the same command as in *step 4*:

```
grpcurl -plaintext -d '{"record_id": "1", "record_type": "movie"}'  
localhost:8082 RatingService/GetAggregatedRating
```

8. You should get the following response:

```
{  
    "ratingValue": 5  
}
```

Hooray – we just confirmed that our MySQL repository logic works! You can now shut down the rating service, rerun it, and repeat *step 6*. When you do this, you will get the same result, and this will confirm that our data is persistent now and does not get impacted by service restarts.

We have reviewed the very basics of working with databases in Go microservices. In the next section, we are going to review a data caching technique that can help you reduce the load on your database for applications performing lots of database operations.

Implementing data caching

In the previous section, we implemented the logic for storing our service data, including movie metadata and movie ratings, in a persistent database. Now, consider the scenario where we never update movie metadata once we save it to the database. In fact, this might be the case in a real movie metadata application since the movie metadata, such as its title or description, would hardly change at all unless we make small corrections to it. Let's also imagine that our movie metadata service becomes popular, and we start receiving lots of requests to it for movie details. If we start sending millions of requests to our database, it can easily get overloaded and stop responding to our requests at all. How would we solve such a load problem?

There is a powerful technique called **data caching** that can be helpful in reducing the load on our database. In essence, data caching is a technique of storing a copy or a part of our persistent data in the memory of a component called **cache**, which can either be an in-memory structure of our application or a separate application, similar to a database. In our movie metadata example, a data cache could store a set of movie metadata records to reduce the read load on our primary database. There are some popular solutions for caching data:

- **Redis:** Redis is a popular in-memory database that is often used for caching data for other databases, including MySQL. The reason for Redis' popularity is its simple minimalistic API and the simplicity of its installation. While Redis does not provide some complex features such as SQL support, it offers low latency for write and read operations.

- **Memcached:** Memcached is another popular caching solution that is used for caching data for various databases. Created more than 20 years ago, it remains one of the popular caching solutions together with Redis.



Note

Most caching solutions don't provide any durability guarantees or don't offer complex data coordination logic (such as coordinating writes between multiple instances) and are generally used for read optimization purposes.

In many cases, you don't need to use a separate caching solution such as Redis and Memcached and can implement caching inside your application. Such an approach has some great advantages: using a self-implemented solution can be quick and easy and won't require installing any external solutions.

Let's demonstrate how to implement data caching logic for your Go microservices. We will take the metadata service that we just mentioned in the section as an example. First, open the `metadata/internal/controller/metadata/controller.go` file and update the `metadataRepository` interface definition to the following:

```
type metadataRepository interface {
    Get(ctx context.Context, id string) (*model.Metadata, error)
    Put(ctx context.Context, id string, metadata *model.Metadata) error
}
```

Now, let's update our Controller structure code in the same file to the following:

```
// Controller defines a metadata service controller.
type Controller struct {
    repo metadataRepository
    cache metadataRepository
}

// New creates a metadata service controller.
func New(repo metadataRepository, cache metadataRepository) *Controller {
    return &Controller{repo, cache}
}
```

```
// Get returns movie metadata by id.
func (c *Controller) Get(ctx context.Context, id string) (*model.Metadata, error) {
    cacheRes, err := c.cache.Get(ctx, id)
    if err == nil {
        fmt.Println("Returning metadata from a cache for " + id)
        return cacheRes, nil
    }
    res, err := c.repo.Get(ctx, id)
    if err != nil && errors.Is(err, repository.ErrNotFound) {
        return nil, ErrNotFound
    }
    if err := c.cache.Put(ctx, id, res); err != nil {
        fmt.Println("Error updating cache: " + err.Error())
    }
    return res, err
}
```

In our code, we use a combination of a persistent repository and a cache, often called a **read-through cache**. First, we check if our cache contains the requested data, and immediately return the result if the data is found. Otherwise, we read the data from the persistent repository and save it to our cache.

Let's make a final edit to our metadata service code so that we can make our caching logic work. In the `metadata/cmd/main.go` file, find the line having a call to `metadata.New` and replace it with the following code:

```
cache := memory.New()
ctrl := metadata.New(repo, cache)
```

Our newly added code initializes the metadata service controller with both a repository and an in-memory cache. Now, you can start the metadata service and issue some `grpcurl` commands that we used in *Chapter 2 Scaffolding a Go Microservice*, to test the newly added logic.

Summary

In this chapter, we provided a brief overview of database storage solutions for storing microservice data. Then, we illustrated how to write the logic for writing and reading our service data into MySQL, a popular open source relational database that is widely used across the software development industry. Finally, we showed how to use a data caching technique for reducing the load on a database for services that perform lots of database operations. This knowledge should help you to establish a foundation for storing and retrieving various types of data in your microservices.

In the next chapter, we are going to illustrate how to build and run our service instances using a popular platform, Kubernetes, which allows us to coordinate various service-related operations, such as code updates, automated service instance count increases, and many more.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Types of databases: <https://www.prisma.io/dataguide/intro/comparing-database-types>
- DB-Engines Ranking: <https://db-engines.com/en/ranking>
- Go Cookbook: Databases <https://go-cookbook.com/snippets/databases>

8

Setting Up Service Deployments

By now, you know how to bootstrap microservices, set up the logic for accessing databases, implement service APIs, use serialization, and enable synchronous and asynchronous communication between your microservices. Now, we are ready to cover a topic that is very important in practice – microservice deployments.

Deployment is a technique of uploading and running your code to one or multiple servers that are often located remotely. Prior to this chapter, we assumed that all services are run locally. We implemented services using static hardcoded local addresses, such as `localhost` for Kafka. At some point, you will need to run your services remotely – for example, on a remote server or in a cloud, such as **Amazon Web Services (AWS)** or **Microsoft Azure**.

This chapter will help you to learn how to build and set up your applications for deployments to such remote infrastructure. Additionally, we are going to illustrate how to use one of the most popular deployment and orchestration systems, **Kubernetes**. You will learn about the benefits it provides, as well as how to set it up for the microservices that we created in the previous chapters.

In this chapter, we will cover the following topics:

- Preparing application code for deployments
- Service deployment solutions
- Deploying via Kubernetes
- Deployment best practices

Technical requirements

To complete this chapter, you need the following tools:

- **Go:** 1.18 or above, similar to the previous chapters
- **Docker:** <https://www.docker.com>
- **Kubernetes:** <https://kubernetes.io> (you will need the `kubectl` and `minikube` tools)
- **Helm:** <https://helm.sh>
- **grpcurl:** <https://github.com/fullstorydev/grpcurl>

You can find the GitHub code for this chapter here:

<https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter08/src>

Preparing application code for deployments

In this section, we are going to provide a high-level overview of a service deployment process and describe the actions required to prepare your microservices for deployments. You will learn how to configure Go microservices for running in different environments, how to build them for different operating systems, and some other tips for preparing your microservices for remote execution.

Let's proceed to the basics of the deployment process.

Deployment basics

As we mentioned in the introduction to this chapter, deployments allow you to run and update your applications on one or multiple servers. Such servers are usually located remotely (on clouds or dedicated web hosting) and run all the time to allow your applications to serve the request or process data 24/7.

The deployment process for each environment usually consists of multiple steps. The steps include the following:

1. **Build:** Build the service by compiling it (for compiled languages, such as Go) and including additional required files.
2. **Rollout:** Copy the newly created build to servers of the target environment and replace the existing running binary, if any, with the newly built one.

The rollout process is usually sequential; instead of replacing the build on all hosts parallelly, it performs one replacement at a time. For example, if you have 10 service instances, the rollout process would first update one instance, then verify that the instance is healthy and move to the second one, and continue until it updates the last service instance. This is done to increase service reliability because if a new version consists of a bug or entirely fails to start on some server, the rollout will not affect all servers at once.

In order to enable the testing of microservices, servers can be classified into multiple categories, called environments:

- **Local/development:** Servers that are used for running and testing code while working on the code. This environment should never handle any requests from users, and it often consists just of a developer's computer. It can also be configured to use simplified versions of a database and other components, such as single-server and in-memory implementations.
- **Production:** Servers that are intended to handle user requests.
- **Staging:** A mirror of a production environment, but it is used for pre-production testing. Staging often differs from the local/production environment in its configuration and separate data storages, which help to avoid any interference with production data during testing.

Production deployments can be done in **canary** mode – a deployment mode that performs the changes only on a small fraction (such as 1%) of production hosts. Canary deployments are useful for the final testing of new code before updating all production instances of a service.

Let's now see how developers can configure their microservices for deployments to multiple environments.

Application configuration

In the previous section, we described the differences between various environments, such as local, staging, and production. Each environment is usually configured differently – if your services have access to databases, each environment would generally have a separate database with different credentials. To enable your services to run in such environments, you would need to have multiple configurations of your services, one per environment.

There are two ways of configuring your services:

- **In-place/hardcode:** All required settings are stored in the service code (Go code, in our case)
- **Separate code and configuration:** Configuration is stored in separate files so that it can be modified independently

Separating service code and configuration often results in better readability, which makes configuration changes easier. Each environment can have a separate configuration file or a set of files, allowing you to read, review, and update environment-specific configurations easily. Additionally, various data formats, such as YAML, can help to keep configuration files compact. Here's a YAML configuration example:

```
mysql:  
  database: ratings  
kafka:  
  topic: ratings
```

In this book, we are going to use an approach that separates application code and configuration files and stores configuration in YAML format. This approach is common to many Go applications and can be seen in many popular open source Go projects.

Important note



Note that invalid configuration changes are among the top causes of service outages in most production systems. I suggest you explore various ways of automatically validating configuration files as a part of the code commit flow. An example of Git-based YAML configuration validation is provided in the following article: <https://ruleoftech.com/2017/git-pre-commit-and-pre-receive-hooks-validating-yaml>.

Let's review our microservice code and see which settings can be extracted from the application configuration:

1. Our metadata service does not have any settings other than its gRPC handler address, `localhost:8081`, which you can find in its `main.go` file:

```
lis, err := net.Listen("tcp", fmt.Sprintf("localhost:%v", port))
```

2. We can extract this setting to the service configuration. A YAML configuration file with this setting would look like this:

```
api:  
  port: 8081  
  serviceDiscovery:  
    consul:  
      address: http://consul-server.consul.svc.cluster.local:8500
```

3. Let's make the changes for reading the configuration from a file. Inside the `metadata/cmd` directory, create a `config.go` file and add the following code to it:

```
package main  
  
type config struct {  
    API apiConfig 'yaml:"api"'  
    ServiceDiscovery serviceDiscoveryConfig  
    'yaml:"serviceDiscovery"'  
}  
  
type apiConfig struct {  
    Port int 'yaml:"port"'  
}  
  
type serviceDiscoveryConfig struct {  
    Consul consulConfig 'yaml:"consul"'  
}  
  
type consulConfig struct {  
    Address string 'yaml:"address"'  
}
```

4. In addition to this, create a `configs` directory inside the `metadata` service directory and add a `default.yaml` file to it with the following contents:

```
api:  
  port: 8081  
  serviceDiscovery:  
    consul:  
      address: http://consul-server.consul.svc.cluster.local:8500
```

5. The file we just created contains the YAML configuration for our service. Now, let's add code to our `main.go` file to read the configuration. Replace the first line of the `main` function that prints a log message with this:

```
log.Println("Starting the movie metadata service")
f, err := os.Open("default.yaml")
if err != nil {
    panic(err)
}
defer f.Close()
var cfg serviceConfig
if err := yaml.NewDecoder(f).Decode(&cfg); err != nil {
    panic(err)
}
```

Additionally, replace the line with the `net.Listen` call with this:

```
lis, err := net.Listen("tcp", fmt.Sprintf("localhost:%d", cfg.
APIConfig.Port))
```

6. The code we have just added is using a `gopkg.in/yaml.v3` package to read a YAML file. Import it into our module by running the following command:

```
go mod tidy
```

7. Do the same changes that we just made for the other two services we created earlier. Use port number 8082 for the rating service and 8083 for the movie service in your YAML configuration.
8. Finally, find the code block in the `metadata/cmd/main.go` file where we initialize the `registry` variable and call `registry.Register`, and replace it with the following code:

```
    registry, err := consul.NewRegistry(cfg.ServiceDiscovery.Consul.
Address
)
if err != nil {
    panic(err)
}
ctx := context.Background()
instanceID := discovery.GenerateInstanceID(serviceName)
if err := registry.Register(ctx, instanceID, serviceName, fmt.
.Sprintf("metadata:%d", port)); err != nil {
```

```
    panic(err)
}
```

9. In our code, we set up the connection to the HashiCorp Consul service that we will run inside Kubernetes in the next section of this chapter. Also note that we updated the address of our service from `localhost` to `metadata` – this will be required for Kubernetes-based service discovery.
10. Apply the same changes that we did in *Step 7* to the rating and movie service. Don't forget to update service addresses inside the call to the `Register` function.

The changes we just made helped us to achieve two things. First, we prepared our services for Consul-based service discovery inside a Kubernetes cluster. Second, we introduced the application configuration that is separate from the service logic. In the future, to make any configuration changes, we would just need to update the YAML files without touching our service Go code.

Now that we have finished configuring our microservices for deployments, we are ready to move to the next section, which is going to provide a brief overview of popular service deployment solutions.

Service deployment solutions

In this section, we are going to review some popular solutions for establishing service deployments. This section will help us to choose a deployment solution for our microservices.

Docker Swarm

Docker Swarm is a lightweight tool provided by Docker that allows us to run multiple containers and perform service discovery, load balancing, and scaling. Its benefits include Docker CLI support, relatively simple configuration, and quick setup.

In order to set up a Docker Swarm deployment, you need to prepare a configuration of one or multiple services, such as the following:

```
version: "3.7"

services:
  example-service:
    image: example-image
    ports:
      - "8000:8000"
```

Having the configuration in place, a Docker Swarm deployment can be executed by running a single command:

```
docker stack deploy -c example-config.yaml demo
```

Docker Swarm is generally a great fit for small-scale applications that don't need complex configurations. Among its limitations is a lack of support for managed Docker Swarm clusters by popular cloud vendors such as Amazon, Google, and Microsoft.

You can get familiar with the Docker Swarm tool by reviewing <https://docs.docker.com/guides/swarm-deploy/>.

HashiCorp Nomad

HashiCorp Nomad (<https://www.nomadproject.io/>) is a relatively flexible tool that goes beyond container orchestration and also supports running virtual machines. It has a relatively simple configuration and integrates natively with HashiCorp Consul, which we used in the previous chapters. Similar to Docker Swarm, there is no offering of managed HashiCorp Nomad installations by the popular cloud providers, requiring its users to set up and run their own clusters manually.

Kubernetes

Kubernetes is an open source deployment and orchestration platform that was initially created at Google and later maintained by a large developer community backed by the Linux Foundation. Compared to the other solutions, such as Docker Swarm and HashiCorp Nomad, Kubernetes is much more flexible in its configuration and supports running and deploying applications of any size, from small, single-instance applications to ones with tens of thousands of instances. Because of this, Kubernetes can be considered the most scalable solution across popular open source deployment and orchestration solutions. In addition to this, there are managed Kubernetes cloud solutions, such as **AWS Elastic Kubernetes Service** and **Google Kubernetes Engine**, allowing us to use established cloud infrastructure for running our services in production without the need to manage our own deployment clusters.

Because of these benefits, we are going to choose Kubernetes as the deployment tool for our microservices. In the following section, we are going to provide a brief overview and demonstrate how to use its deployment and scaling mechanisms.

Deploying via Kubernetes

In this section, we are going to illustrate how to set up deployments for our microservices using Kubernetes. You will learn the basics of Kubernetes, how to set up our microservices for using it, and how to test our microservice deployments in Kubernetes.

Introduction to the Kubernetes data model

In **Kubernetes**, each application consists of one or multiple **Pods** – the smallest deployable units. Each Pod contains one or multiple **containers** – lightweight software blocks containing the application code. The deployment of a single container to multiple Pods is illustrated in the following diagram:

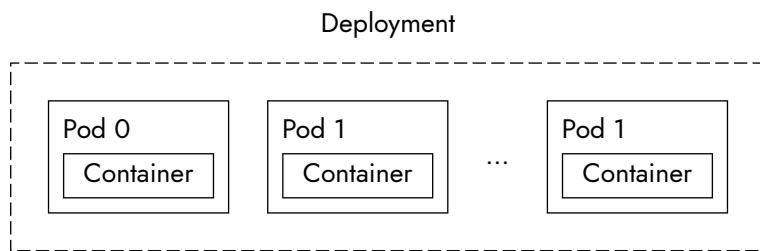


Figure 8.1 – Kubernetes deployment model

Kubernetes Pods can be run on one or multiple hosts, called **nodes**. A group of nodes is called a **cluster**, and the relationship between a cluster, nodes, and its Pods is illustrated in the following diagram:

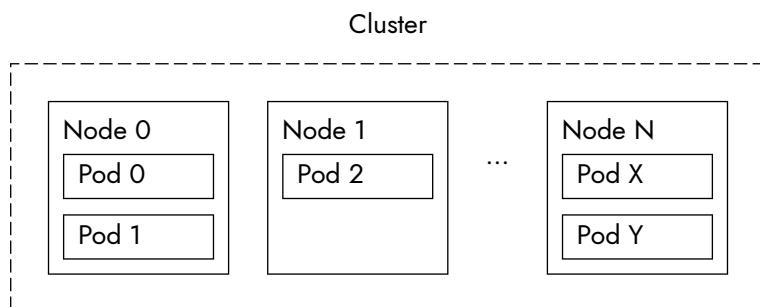


Figure 8.2 – Kubernetes cluster model

To deploy a service in Kubernetes, developers generally need to perform the following steps:

1. **Prepare a container image:** A **container image** contains either the application code or its compiled binary (both options can be used, as long as the container image contains the instructions and any tools to run the code), as well as any additional files required for running it. A container image is essentially a program ready for deployment.
2. **Create a deployment configuration:** A Kubernetes deployment configuration tells it how to run the application. It includes settings such as the number of replicas (number of Pods to run) and names of containers.
3. **Run a deployment command:** Kubernetes will apply the provided configuration by running the desired number of Pods with the target application(s).

One of the benefits of Kubernetes is abstracting away all the low-level details of deployments, such as selecting target servers to deploy (if you have many, you need to balance their load otherwise), copying and extracting your files, and running health checks. In addition to this, there are some other useful benefits:

- **Service discovery:** Kubernetes offers a built-in service discovery API for use in applications
- **Rollbacks:** If there are any issues with the deployment, Kubernetes allows you to roll back the changes to the previously deployed version
- **Automated restarts:** If any Pod experiences any issue, such as an application crash, Kubernetes will perform a restart of that Pod

Now, let's describe how we can set up deployments of our microservices using Kubernetes.

Setting up our microservices for Kubernetes deployments

Note



To make your work with Kubernetes more convenient, consider installing Kubernetes Dashboard locally so you can view your deployments, access Pod logs, and gain a lot of other useful information. For instructions, see <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>.

All the necessary steps for setting up deployments in Kubernetes for our three microservices are set out here:

1. Start Kubernetes locally:

```
minikube start
```

2. Install HashiCorp Consul to our Kubernetes cluster, so we can use it for service discovery of our services:

```
helm repo add hashicorp https://helm.releases.hashicorp.com
helm install consul hashicorp/consul --set global.name=consul
--create-namespace --namespace consul
kubectl port-forward service/consul-server 8500:8500 -n consul
```

3. If the previous commands are executed successfully, you should be able to access Consul's UI by opening `http://localhost:8500` in your browser.
4. Create a container image for each service. Kubernetes supports multiple types of containers, and Docker is currently the most popular container type. We already used Docker in *Chapter 3* and will now illustrate how to use it to create containers for our services.

Inside the `metadata` service directory, create a file called `Dockerfile` and add the following code to it:

```
FROM alpine:latest

# Create and set working directory
WORKDIR /app
# Copy our binary and configs to the container
COPY main ./
COPY configs/default.yaml ./
# Make sure the binary is executable
RUN chmod +x ./main
# Expose the port for accepting incoming requests
EXPOSE 8081
# Execute our service
CMD ["/app/main"]
```

In the file that we just added, we specified that to prepare the image for our container for the `metadata` service, Docker should use the `alpine:latest` base image. **Alpine** is a lightweight Linux distribution that has a size of just a few megabytes and is optimal for our services. The remaining commands in the file copy and execute our service binary.

5. As the next step, add a file with the same contents inside the `rating` and `movie` service directories. Make sure you use the right ports in the files (8082 and 8083, correspondingly).

Once you have created the Docker configuration files, run the `build` command inside each service directory:

```
GOOS=linux go build -o main cmd/*.go
```

The results of the previous command should be the executable file called `main`, stored in each service directory. Note that we used a `GOOS=linux` variable – this tells the `go` tool to build our code for the Linux operating system that will be used to run our Docker images inside containers.

6. Before we build our Docker images, let's point our Docker CLI to the `minikube` registry, so it would be able to access it during deployments:

```
eval $(minikube -p minikube docker-env)
```

7. Now, let's build service images. Run this command from the `metadata` service directory:

```
docker build -t metadata:latest .
```

Similarly, run this command from the `rating` service directory:

```
docker build -t rating:latest .
```

Finally, run this command from the `movie` service directory:

```
docker build -t movie:latest .
```

At this point, we are ready to create a Kubernetes deployment configuration that is going to tell Kubernetes how to deploy our services.

8. Inside the `metadata` service directory, create a file called `kubernetes-deployment.yaml` with the following contents:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: metadata
spec:
  replicas: 2
  # selector specifies what pods to deploy
  selector:
```

```
matchLabels:  
    app: metadata  
# template defines our pod configuration  
template:  
    metadata:  
        labels: # specifies Labels to add to our pods  
            app: metadata  
    spec:  
        containers:  
            - name: metadata  
                image: metadata:latest  
                imagePullPolicy: IfNotPresent  
                ports:  
                    - containerPort: 8081
```

The file that we just created provides instructions to Kubernetes on how to deploy our service. Here are some important settings:

- **Replicas:** The number of Pods to run
- **Image:** The name of the container image to deploy
- **Ports:** Container port to expose

Note that the container port is different from the application port (the one that we configured in our APIConfig structure). The mapping between these settings is done by Docker as a part of the `docker run` settings.

9. Now, create a file with the same name in the rating service directory with the following contents:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
    name: rating  
spec:  
    replicas: 2  
    selector:  
        matchLabels:  
            app: rating  
    template:
```

```
metadata:  
  labels:  
    app: rating  
spec:  
  containers:  
    - name: rating  
      image: rating:latest  
      imagePullPolicy: IfNotPresent  
      ports:  
        - containerPort: 8082
```

- Finally, create a `kubernetes-deployment.yaml` file in the `movie` service directory with the following contents:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: movie  
spec:  
  replicas: 2  
  selector:  
    matchLabels:  
      app: movie  
  template:  
    metadata:  
      labels:  
        app: movie  
    spec:  
      containers:  
        - name: movie  
          image: movie:latest  
          imagePullPolicy: IfNotPresent  
          ports:  
            - containerPort: 8083
```

- Let's apply our `metadata` deployment configuration by running the following command from the `metadata` service directory:

```
kubectl apply -f kubernetes-deployment.yaml
```



If you got an `ImagePullBackOff` error from Kubernetes, check that you built the Docker images correctly in *Steps 2 and 3*.

12. If the previous command is executed successfully, you should see the new deployment by running this command:

```
kubectl get deployments
```

The output of the command should be this:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
metadata	2/2	2	2	6s

Also, check the state of service Pods by running the following command:

```
kubectl get pods
```

The output should show the Running status for our `metadata` service Pods, as shown here:

NAME	READY	STATUS	RESTARTS	AGE
metadata-5f87cbbf65-st69m	1/1	Running	0	116s
metadata-5f87cbbf65-t4xsk	1/1	Running	0	116s

As you may notice, Kubernetes created two Pods for our service, the same number as we specified in the deployment configuration. Each Pod has a **unique identifier (UID)**, which is shown in the left column. You can see that Kubernetes created two Pods for our `metadata` service.

You can check the logs of each Pod by running the following command:

```
kubectl logs -f <POD_ID>
```

Now, perform the same changes that we did for the `metadata` service for the other two services, and verify that the Pods are running.

If you want to make some manual API requests to the services, you need to set up port forwarding by running the following command:

```
kubectl port-forward <POD_ID> 8081:8081
```

This command would work for the `metadata`, `rating`, and `movie` services; however, you would need to replace the `8081` port value with `8082` and `8083`, correspondingly.

Now you can test connectivity to our microservices. Run some `grpcurl` commands to check that their APIs are working:

```
grpcurl -plaintext -d '{"record_id": "1", "record_type": "movie", "user_id": "alex", "rating_value": 5}' localhost:8082 RatingService/PutRating  
grpcurl -plaintext -d '{"record_id": "1", "record_type": "movie"}' localhost:8082 RatingService/GetAggregatedRating
```

If you did everything well, congratulations! We have finished setting up basic Kubernetes deployments of our microservices. Our Kubernetes setup can be illustrated with the following diagram:

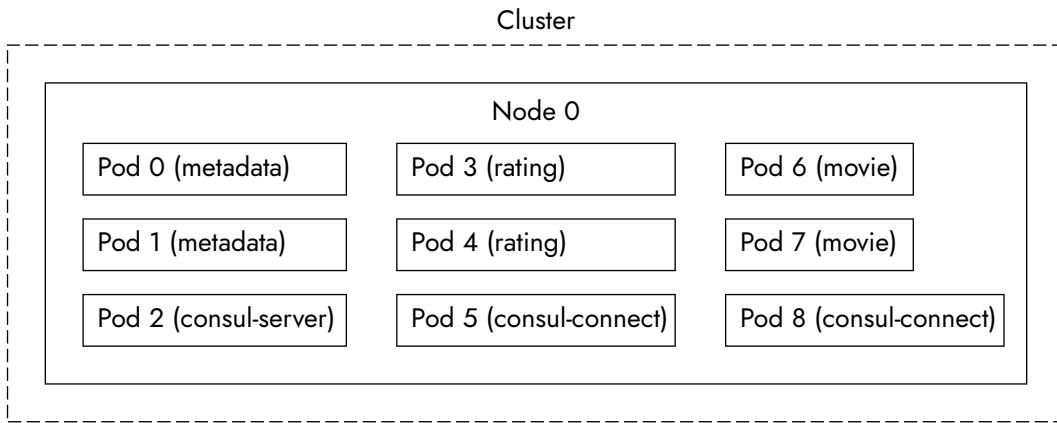


Figure 8.3 – Our Kubernetes deployment

Note that the actual number and names of HashiCorp Consul Pods can vary based on the Consul deployment configuration, but are generally similar to the ones presented in our diagram.

Let's summarize what we did in this section:

1. First, we started a local Kubernetes cluster using the `minikube` tool and installed HashiCorp Consul on it, so we could use it for service discovery.
2. Then, we created container images for each of our services so that we could deploy them.
3. We created a Kubernetes deployment configuration to tell it how to deploy our microservices.
4. Finally, we tested our Kubernetes deployments using a combination of the `kubectl` and `grpcurl` commands.

At this point, you have some understanding of Kubernetes deployments and know how to deploy your microservices using them. This knowledge will help you to run your services on many platforms, including all popular cloud platforms, such as AWS, Azure, and **Google Cloud Platform (GCP)**.

Deployment best practices

In this section, we are going to describe some best practices related to the deployment process. These practices, listed here, will help you to set up a reliable deployment process for your microservices:

- Automated rollbacks
- Canary deployments
- Continuous deployment
- Use feature flags for extra control over service changes

Automated rollbacks

Automated rollbacks are processes that automatically revert a deployment if a failure occurs during its execution. Imagine you are making a deployment of a new version of your service and that version has some application bug that is preventing it from starting successfully. In that case, the deployment process will replace your active instances of a service (if the service is already running) with the failing ones, making your services unavailable. Automated rollbacks are a way to detect and revert such bad deployments, helping you to avoid an outage in situations when your services become unavailable due to such issues.

Automated rollbacks are not offered by default in Kubernetes, at the time of writing this book. However, this should not stop you from using this technique, especially if you aim to achieve high reliability of your services. The high-level idea of implementing automated rollbacks with Kubernetes is as follows:

- Perform continuous health checks of your service (we are going to cover such logic in *Chapter 13* of this book).
- When you detect a health issue with your service, check whether there was a recent deployment of your service. For example, you can do so by running the `kubectl describe deployment` command.
- If there was a recent deployment and the time of it closely matches the time when the health check issues were detected, you can roll it back by executing this rollback command:
`kubectl rollout undo deployment <DEPLOYMENT_NAME>`.

Canary deployments

As we mentioned at the beginning of the chapter, canary is a special type of deployment, where you update only a small fraction (1–3%) of instances. The idea of canary deployments is to test a new version of your code on a subset of production instances and validate its correctness before doing a regular production deployment.

We won't cover the details of setting up canary deployments in Kubernetes, but can cover the basic ideas that would help you to do this once you want to enable canary deployments for your microservices, as set out here:

1. Create two separate Kubernetes deployment configurations, one for canary and one for production.
2. Specify the desired number of replicas in each configuration – if you want to run a service on 50 Pods and let canary handle 2% of traffic, set 1 replica for canary and 49 replicas for production.
3. You may also add environment-specific suffixes to deployment names. For example, you can call a canary deployment of a rating service `rating-canary` and the production environment `rating-production`.
4. When you perform a deployment of your service, deploy it using a canary configuration first.
5. Once you verify that the deployment was successful, make a deployment using a production configuration.

Canary deployments are strongly recommended for increasing the reliability of your deployments. Testing new changes on a small fraction of traffic helps to reduce the impact of various application bugs and other types of issues that your services can encounter.

Continuous deployment

Continuous Deployment (CD) is a technique of making frequent recurring deployments. With CD, services get deployed automatically – for example, on each code commit. The main benefit of CD is early deployment failure detection – if any change (such as a Git commit of a new service code) is causing a deployment failure, the failure would often get detected much sooner than in the case of less frequent manual deployments.

You can automate deployments by programmatically monitoring a change log (such as Git commit history), or by using **Git hooks** – configurable actions that are executed at specific stages of Git changes. With Kubernetes, once you detect a new version of your software, you can trigger a new deployment by using a `kubectl apply` command.

Due to the high cadence of version updates, CD requires some tooling for automated checks of service health. We are going to cover such tooling later in *Chapters 13 and 14* of this book.

Use feature flags to decouple deployments from feature releases

Consider a scenario where we release some new feature or a change in our microservice logic and want to test how it performs in a production environment. For example, let's assume that we want to add a feature that includes movie recommendations in our `GetMovieDetails` API handler. This feature might be computationally expensive (for example, involving some AI-based logic, or even making calls to some external API services) and we would want to control it by being able to turn it on and off dynamically. For this, without any additional mechanisms, we would generally have two options:

1. Enable or disable a new feature in code and deploy it.
2. Roll back the deployment in case we want to change the code to the previous state.

Both options might not be easy to use: making and deploying service code changes might be not trivial (for example, when there is a mandatory code review process or when there are deployment constraints, such as a specific deployment schedule). Because of this, it might be beneficial to have some dynamic control over service features so we could turn them on and off depending on our needs.

Such a concept is called a **dynamic configuration**, or **feature flags**. With a dynamic configuration, you could check in your code whether some feature is enabled or disabled, before executing some code:

```
if dynamicConfig.RecommendationsEnabled() {  
    movie.Recommendations = getRecommendations(movie.ID)  
}
```

How can you implement dynamic configuration for your microservices? There are multiple options:

- **Use a cloud service:** You might create or reuse some existing dynamic configuration service for your needs. Various cloud providers, such as AWS, Microsoft Azure, and Google Cloud, provide feature flag services, such as **AWS AppConfig**.
- **Implement dynamic configuration manually:** For dynamic configuration, you would need some persistent database, such as Redis or MySQL, to store your data. Then, you could either use it via some function/library or via a separate microservice that would provide the configuration check functionality to your microservices. We are going to cover dynamic configuration in more detail in *Chapter 11* of the book.

This concludes our deployment best practices. You may find references to some other practices, such as blue-green deployments, in the *Further reading* section of this chapter.

Summary

In this chapter, we have covered a very important topic – service deployments. You learned the basics of the service deployment process, as well as the necessary steps for preparing our microservices for deployments. Then, we introduced Kubernetes, a popular deployment and orchestration platform that is now provided by many companies and cloud providers. We illustrated how to set up a local Kubernetes cluster and deploy our microservices to it, running multiple instances of each service to illustrate how easy it is to run any arbitrary number of instances within the Kubernetes platform.

The knowledge you gained should help you to set up more complex deployment processes, as well as to work with the services that are already deployed via Kubernetes.

This chapter summarizes our material on service deployments. In the next chapter, we are going to describe another important topic: testing.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Kubernetes documentation: <https://kubernetes.io/docs/home/>
- Service deployment best practices: <https://codefresh.io/learn/software-deployment/>
- Setting up Kubernetes services: <https://kubernetes.io/docs/concepts/services-networking/service/>
- Install Consul on Kubernetes with Helm: <https://developer.hashicorp.com/consul/docs/k8s/installation/install>
- Deploy and Access the Kubernetes Dashboard: <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>
- Blue-green deployments: <https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment>

9

Unit and Integration Testing

Testing is an integral part of any development process. Writing good tests helps ensure that any changes made throughout the development process will keep the code working and reliable. Testing is especially important in microservice development: it's not enough to test each service; you also need to test integrations between the services, ensuring every service can work with the others.

In this chapter, we will cover both unit testing and integration testing and illustrate how to add tests to the microservices we created in the previous chapters. We will cover the following topics:

- Go testing overview
- Unit tests
- Integration tests
- Testing best practices

You will learn how to write unit and integration tests in Go, how to use the mocking technique, and how to organize the testing code for your microservices. This knowledge will help you to build more reliable services.

Let's proceed to an overview of Go testing tools and techniques.

Technical requirements

To complete this chapter, you need Go version 1.18+ or above.

You can find the GitHub code for this chapter here: <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter09>

Go testing overview

In this section, we are going to provide a high-level overview of Go's testing capabilities. We will cover the basics of writing tests for Go code, list the useful functions and libraries provided with the Go SDK, and describe various techniques for writing tests that will help you in microservice development.

First, let's cover the basics of writing tests for Go applications.

Go language has built-in support for writing automated tests and provides a package called `testing` for this purpose.

There is a conventional relationship between the Go code and its tests. If you have a file called `example.go`, its tests would reside in the same package in a file called `example_test.go`. Using a `_test` filename suffix allows you to differentiate between the code being tested and the tests for it, making it easier to navigate the source code.

Go test functions follow this conventional name format, with each test function name starting with the `Test` prefix:

```
func TestXxx(t *testing.T)
```

Inside these functions, you can use the `testing.T` structure to report test failures or use any additional helper functions provided by the `testing` package.

Let's take this test as an example:

```
func TestAdd(t *testing.T) {
    a, b := 1, 2
    if got, want := Add(1, 2), 3; got != want {
        t.Errorf("Add(%v, %v) = %v, want %v", a, b, got, want)
    }
}
```

In the preceding function, we used `testing.T` to report a test failure in case the `Add` function provides an unexpected output.

When it comes to execution, we can run the following command:

```
go test
```

The command executes each test in the target directory and prints the output, containing error messages for any failing tests or any other necessary data.

Developers are free to choose the format of their tests; however, there are some common techniques, such as **table-driven tests**, that often help organize test code elegantly.

Table-driven tests are tests in which inputs are stored in the form of a table or a set of rows. Let's take this example:

```
func TestAdd(t *testing.T) {
    tests := []struct {
        a      int
        b      int
        want  int
    }{
        {a: 1, b: 2, want: 3},
        {a: -1, b: -2, want: -3},
        {a: -3, b: 3, want: 0},
        {a: 0, b: 0, want: 0},
    }
    for _, tt := range tests {
        assert.Equal(t, tt.want, Add(tt.a, tt.b), fmt.Sprintf("Add(%v, %v)", tt.a, tt.b))
    }
}
```

In this code, we initialize the `tests` variable with the test cases for our function and then iterate over it. Note that we use the `assert.Equal` function provided by the `github.com/stretchr/testify` library to compare the expected and the actual result of the function being tested. This library provides a set of convenient functions that can simplify your test logic. Without using the `assert` library, the code comparing the test result would look like the following:

```
if got, want := Add(tt.a, tt.b), tt.want; got != want {
    t.Errorf ("Add(%v, %v) = %v, want %v", tt.a, tt.b, got, want)
}
```

Table-driven tests help reduce the repetitiveness of tests by separating test cases and the logic that performs actual checks. In general, these tests are good practice when you need to perform lots of similar checks against the defined goal states, as shown in our example.

The table-driven format also helps us improve the readability of test code, making it easier to see and compare different test cases for the same functions. The format is quite common in Go tests; however, you can always organize your test code in the way that is the best for your use case.

Now, let's review the basic features provided by Go's built-in testing library.

Subtests

One of the interesting features of the Go testing library is the ability to create **subtests** – tests that get executed inside other ones. Among the benefits of subtests is the ability to execute them separately, as well as to execute them in parallel for long-running tests and structure the test output in a more granular way.

Subtests are created by calling the `Run` function of the testing library:

```
func (t *T) Run(name string, f func(t *T)) bool
```

When using the `Run` function, you need to pass the name of the test case and the function to execute, and Go will take care of executing each test case separately. Here's an example of a test using the `Run` function:

```
func TestProcess(t *testing.T) {
    t.Run("test case 1", func(t *testing.T) {
        // Test case 1 logic.
    })
    t.Run("test case 2", func(t *testing.T) {
        // Test case 2 logic.
    })
}
```

In the preceding example, we created two subtests by calling the `Run` function twice, one time for each subtest.

To achieve more fine-grained control over subtests, you can use the following options:

- Each subtest, either passing or failing, can be shown separately in the output when running the `go test` command with the `-v` argument
- You can run an individual test case by using a `-run` argument of the `go test` command

There is one other interesting benefit of using the `Run` function. Let's imagine that you have a function called `Process` that takes seconds to complete. If you have a table test with lots of test cases and you execute them sequentially, the execution of the entire test may take a lot of time. In this case, you could let the Go test runner execute tests in parallel mode by calling the `t.Parallel()` function. Let's illustrate this in the following example:

```
func TestProcess(t *testing.T) {
    tests := []struct {
        name  string
        input string
        want  string
    }{
        {name: "empty", input: "", want: ""},
        {name: "dog", input: "animal that barks", want: "dog"},
        {name: "cat", input: "animal that meows", want: "cat"},
    }
    for _, tt := range tests {
        input := tt.input
        t.Run(tt.name, func(t *testing.T) {
            t.Parallel()
            assert.Equal(t, tt.want, Process(input), fmt.Sprintf("Process(%v)", input))
        })
    }
}
```

In our example, we call the `t.Run` function for each test case, passing the test case name and the function to be executed. Then, we call `t.Parallel()` to make each test case execute in parallel. This optimization can help to reduce the execution time in case our `Process` function is very slow.



Note

We use the additional input variable assignment to prevent our test from experiencing a **closure capture issue** – a case when our test function that is executed by `t.Run` gets an incorrect value due to a loop iterating to the next values.

Skipping

Imagine that you want to execute your Go tests after each change on your computer, but you have some slow tests that take a long time to run. In that case, you would want to find a way to skip running tests under certain conditions. The Go testing library has built-in support for this – the `Skip` function. Let's take this test function as an example:

```
func TestProcess(t *testing.T) {
    if os.Getenv("RUNTIME_ENV") == "development" {
        t.Skip("Skipping a test in development environment")
    }
    ...
}
```

In the preceding code, we skip the test execution if there is a `RUNTIME_ENV` runtime environment variable with the `development` value. Note that we also provide the reason for skipping it inside the `t.Skip` call so that it is logged on test execution.

The skipping feature can be particularly useful for bypassing the execution of long-running tests, such as tests performing slow I/O operations or doing lots of data processing. To support this, the Go testing library provides an ability to pass a specific flag, `-test.short`, to the `go test` command:

```
go test -test.short
```

With the `-test.short` flag, you can let the Go test runner know that you want to run tests in **short mode** – a testing mode when only selected tests with a `Short` setting are getting executed. You can add the following logic to all long-running tests to exclude them in short mode:

```
func TestLongRunningProcess(t *testing.T) {
    if testing.Short() {
        t.Skip("Skipping a test in short mode")
    }
    ...
}
```

In the preceding example, the test is skipped when the `-test.short` flag is passed to the `test` command.

Using the short testing mode is useful when some of your tests are much slower than others and you need to run tests very frequently. Skipping slow tests and executing them less frequently could significantly increase your development speed and make your development experience much better.

You can get familiar with other Go testing features by checking out the official documentation for the `testing` package: <https://pkg.go.dev/testing>. We are now going to proceed to the next section and focus on the details of implementing unit tests for our microservices.

Unit tests

We have covered many useful features for automated testing of Go applications and are now ready to illustrate how to use them in our microservice code. First, we are going to start with **unit tests** – tests of individual units of code, such as structures and individual functions.

Let's walk through the process of implementing unit tests for our code using the metadata service controller as an example. Currently, our controller file looks like this:

```
package metadata

import (
    "context"

    "movieexample.com/metadata/pkg/model"
)

type metadataRepository interface {
    Get(ctx context.Context, id string) (*model.Metadata, error)
}

// Controller defines a metadata service controller.
type Controller struct {
    repo metadataRepository
}

// New creates a metadata service controller.
func New(repo metadataRepository) *Controller {
    return &Controller{repo}
}

// Get returns movie metadata by id.
func (c *Controller) Get(ctx context.Context, id string) (*model.Metadata,
error) {
    return c.repo.Get(ctx, id)
}
```

Let's list what we would like to test in our code:

- A Get call when the repository returns ErrNotFound
- A Get call when the repository returns an error other than ErrNotFound
- A Get call when the repository returns metadata and no error

So far, we have three test cases to implement. All test cases need to perform operations on the metadata repository and we need to simulate three different responses from it. How exactly should we simulate the responses from our metadata repository in the test? Let's explore the powerful technique that allows us to achieve this with our testing code.

Mocking

The technique of simulating responses from a component is called **mocking**. Mocking is often used in tests to simulate various scenarios, such as returning specific results or errors. There are multiple ways of using mocking in Go code. The first one is to implement a *fake* version of components, called **Mocks**, manually. Let's illustrate how to implement these mocks using our metadata repository as an example. Our metadata repository interface is defined in the following way:

```
type metadataRepository interface {
    Get(ctx context.Context, id string) (*model.Metadata, error)
}
```

The mock implementation of this interface could look like this:

```
type mockMetadataRepository struct {
    returnRes *model.Metadata
    returnErr error
}

func (m *mockMetadataRepository) setReturnValues(res *model.Metadata, err error) {
    m.returnRes = res
    m.returnErr = err
}

func (m *mockMetadataRepository) Get(ctx context.Context, id string) (*model.Metadata, error) {
    return m.returnRes, m.returnErr
}
```

In our example mock of the metadata repository, we allow set values to be returned on the upcoming calls to the `Get` function by providing the `setReturnValues` function. The mock could be used to test our controller in the following way:

```
m := mockMetadataRepository{}
m.setReturnValues(nil, repository.ErrNotFound)
c := New(m)
res, err := c.Get(context.Background(), "some-id")
// Check res, err.
```

Manual implementation of mocks is a relatively simple way to test calls to various components that are outside of the scope of the package being tested. The downside of this approach is that you need to write mock code by yourself and update its code on any interface changes.

The other way of using mocks is to use libraries that generate mocking code. An example of this kind of library is <https://github.com/golang/mock>, which contains a mock generation tool called `mockgen`. You can install it by running the following command:

```
go install github.com/golang/mock/mockgen
```

The `mockgen` tool can then be used in the following way:

```
mockgen -source=foo.go [options]
```

Let's illustrate how to generate mock code for our metadata repository. Run the following command from the `src` directory of our project:

```
mockgen -package=repository -source=metadata/internal/controller/metadata/
controller.go
```

You should get the contents of a mock source file as the output. The contents would be similar to this:

```
// MockmetadataRepository is a mock of metadataRepository interface
type MockmetadataRepository struct {
    ctrl      *gomock.Controller
    recorder  *MockmetadataRepositoryMockRecorder
}

// NewMockmetadataRepository creates a new mock instance
func NewMockmetadataRepository(ctrl *gomock.Controller) *MockmetadataRepository {
```

```

        mock := &MockmetadataRepository{ctrl: ctrl}
        mock.recorder = &MockmetadataRepositoryMockRecorder{mock}
        return mock
    }

// EXPECT returns an object that allows the caller to indicate expected
use
func (m *MockmetadataRepository) EXPECT()
*MockmetadataRepositoryMockRecorder {
    return m.recorder
}

// Get mocks base method
func (m *MockmetadataRepository) Get(ctx context.Context, id string)
(*model.Metadata, error) {
    ret := m.ctrl.Call(m, "Get", ctx, id)
    ret0, _ := ret[0].(*model.Metadata)
    ret1, _ := ret[1].(error)
    return ret0, ret1
}

```

The generated mock code implements our interface and allows us to set the expected responses to our `Get` function in the following way:

```

ctrl := gomock.NewController(t)
defer ctrl.Finish()
m := NewMockmetadataRepository(gomock.NewController())
ctx := context.Background()
id := "some-id"
m.EXPECT().Get(ctx, id).Return(nil, repository.ErrNotFound)

```

The mock code generated by the `gomock` library provides some useful features that we have not implemented in our manually created mock version. One of them is the ability to set the expected number of times that the target function should be called using the `Times` function:

```
m.EXPECT().Get(ctx, id).Return(nil, repository.ErrNotFound).Times(1)
```

In the preceding example, we limit the number of times the `Get` function is called to one. The `gomock` library verifies these constraints at the end of the test execution and reports whether the function was called a different number of times. This mechanism is pretty useful when you want to make sure the target function has definitely been called in your test.

So far, we have shown how to use mocks in two different ways, and you may ask what the preferred way of using them is. Let's compare the two approaches to find out the answer.

The benefit of implementing mocks manually is the ability to do so without using any external libraries, such as `gomock`. However, the downsides of this approach would be the following:

- Manual implementation of mocks takes time
- Any changes to the mocked interfaces would require manual updates to the mock code
- Harder to implement extra features that are provided by libraries such as `gomock`, such as call count verification

Using a library such as `gomock` for providing mock code would be beneficial for the following reasons:

- Higher code consistency when all mocks are generated in the same way
- No need to write boilerplate code
- An extended mock feature set

In our comparison, automatic mock code generation seems to provide more advantages, so we will follow the `gomock`-based approach for automatic mock generation. In the next section, we are going to demonstrate how to do this for our services.

Implementing unit tests

We are going to illustrate how to implement controller unit tests using the generated `gomock` code. First, we will need to find a good place in our repository to put the generated code. We already have a directory called `gen` that is shared among the services. We can create a sub-directory called `mock` that we can use for various generated mocks. Run the mock generation command for the metadata repository again:

```
mockgen -package=repository -source=metadata/internal/controller/metadata/
controller.go
```

Copy its output to the file called `gen/mock/metadata/repository/repository.go`. Now, let's add a test for our metadata service controller. Create a file called `controller_test.go` in its directory and add to it the following code:

```
package metadata

import (
    "context"
    "errors"
    "testing"

    "github.com/golang/mock/gomock"
    "github.com/stretchr/testify/assert"
    gen "movieexample.com/gen/mock/metadata/repository"
    "movieexample.com/metadata/internal/repository"
    "movieexample.com/metadata/pkg/model"
)

)
```

Then, add the following code, containing the test cases in a table format:

```
func TestController(t *testing.T) {
    tests := []struct {
        name      string
        expRepoRes *model.Metadata
        expRepoErr error
        wantRes   *model.Metadata
        wantErr   error
    }{
        {
            name:      "not found",
            expRepoErr: repository.ErrNotFound,
            wantErr:   ErrNotFound,
        },
        {
            name:      "unexpected error",
            expRepoErr: errors.New("unexpected error"),
            wantErr:   errors.New("unexpected error"),
        },
        {
            name:      "success",
            expRepoRes: &model.Metadata{},
            wantRes:   &model.Metadata{},
        },
    }
}
```

```
    },  
}
```

Finally, add the code to execute our tests:

```
for _, tt := range tests {  
    t.Run(tt.name, func(t *testing.T) {  
        ctrl := gomock.NewController(t)  
        defer ctrl.Finish()  
        repoMock := gen.NewMockmetadataRepository(ctrl)  
        c := New(repoMock)  
        ctx := context.Background()  
        id := "id"  
        repoMock.EXPECT().Get(ctx, id).Return(tt.expRepoRes,  
        tt.expRepoErr)  
        res, err := c.Get(ctx, id)  
        assert.Equal(t, tt.wantRes, res, tt.name)  
        assert.Error(t, tt.wantErr, err, tt.name)  
    })  
}  
}
```

The code that we just added implements three different test cases for our `Get` function using the generated repository mock. We let the mock return the specific values by calling the `EXPECT` function and passing the desired values. We organized our test in a table-driven way, which we described earlier in the chapter.

To run the tests, use the regular command:

```
go test
```

If you did everything correctly, the output of the test should include `ok`.

Congratulations – we have just implemented the unit tests and demonstrated how to use mocks! We will let you implement the remaining tests for the microservices yourself – it's going to be a fair amount of work, but this is always a great investment for ensuring the code remains tested and reliable.

In the next section, we are going to work on another type of test – integration tests. Knowing why and how to write integration tests in addition to regular unit tests for your microservices will help you to write more stable code and make sure all services work well in integration with each other.

Integration tests

Integration tests are automated tests that verify the correctness of integrations between the individual units of your services and the services themselves. In this section, you are going to learn how to write integration tests and how to structure the logic inside them, as well as get some useful tips that will help you write your own integration tests in the future.

Unlike unit tests that test the individual pieces of code, such as functions and structures, integration tests help ensure that the combinations of individual pieces still work well together.

Let's provide an example of an integration test, taking our rating service as an example. The integration test for our service would instantiate both the service instance and the client for it and ensure that client requests would produce the expected results. As you remember, our rating service provides two API endpoints:

- `PutRating`: Writes a rating to the database
- `GetAggregatedRating`: Retrieves the ratings for a provided record (such as a movie) and returns the aggregated value

Our integration test for the rating service could have the following sequence of calls:

- Writes some data using the `PutRating` endpoint
- Verifies the data using the `GetAggregatedRating` endpoint
- Writes new data using the `PutRating` endpoint
- Calls the `GetAggregatedRating` endpoint and checks that the aggregated value reflects the latest rating update

In microservice development, integration tests usually test individual services or combinations of them – developers can write tests that target an arbitrary number of services.

Unlike unit tests – which generally reside together with the code being tested and can access some internal functions, structures, constants, and variables – integration tests often treat the components being tested as **black boxes**. Black boxes are logical blocks for which the implementation details remain unknown and can only be accessed through publicly exposed APIs or user interfaces. This way of testing is called **black box testing** – testing of a system using a public interface, such as an API, instead of calling individual internal functions or accessing internal components of the system.

Microservice integration tests are often performed by instantiating service instances and performing requests either by calling service APIs or via asynchronous events in case the system

handles requests in an asynchronous fashion. The structure of an integration test usually follows a similar pattern:

- **Set up the test:** Instantiate the components being tested and any clients that can access their interfaces
- **Perform test operations and verify the correctness of results:** Run an arbitrary number of operations and compare the outputs from the system being tested, such as a microservice, to the expected values
- **Tear down the test:** Gracefully terminate the test by tearing down the components instantiated in the setup, closing any clients if needed

To illustrate how to write an integration test, let's take three microservices from the previous chapters – metadata, movie, and rating services. To set up our test, we would need to instantiate six components – a server and a client for each microservice. To make it easier to run the test, we can instantiate servers using in-memory implementations of service registries and repositories.

Before you write the test, it's often helpful to write down the set of operations to be tested and determine the expected outputs for each step. Let's write down the plan for our integration test:

1. Write metadata for an example movie using the metadata service API (the `PutMetadata` endpoint) and check that the operation does not return any errors.
2. Retrieve the metadata for the same movie using the metadata service API (the `GetMetadata` endpoint) and check it matches the record that we submitted earlier.
3. Get the movie details (which should only consist of metadata) for our example movie using the movie service API (the `GetMovieDetails` endpoint) and make sure the result matches the data that we submitted earlier.
4. Write the first rating for our example movie using the rating service API (the `PutRating` endpoint) and check the operation does not return any errors.
5. Retrieve the initial aggregated rating for our movie using the rating service API (the `GetAggregatedRating` endpoint) and check that the value matches the one that we just submitted in the previous step.
6. Write the second rating for our example movie using the rating service API and check that the operation does not return any errors.
7. Retrieve the new aggregated rating for our movie using the rating service API and check that the value reflects the last rating.
8. Get the movie details for our example movie and check that the result includes the updated rating.

Note

Integration tests should be written in a way that their consecutive executions don't interfere with each other. There are different options for handling this, including performing data cleanups and using random values such as record identifiers. The exact solution depends on the logic of the test; however, you should keep this aspect in mind.

Having this kind of plan makes it easier to write the code for the integration test and brings us to the last step – actually implementing it:

1. Create a `test/integration` directory and add a file called `main.go` with the following code:

```
package main

import (
    "context"
    "log"
    "net"

    "github.com/google/go-cmp/cmp"
    "github.com/google/go-cmp/cmp/cmpopts"
    "google.golang.org/grpc"
    "movieexample.com/gen"
    metadata "movieexample.com/metadata/pkg/testutil"
    movie "movieexample.com/movie/pkg/testutil"
    "movieexample.com/pkg/discovery"
    "movieexample.com/pkg/discovery/memory"
    rating "movieexample.com/rating/pkg/testutil"
    "google.golang.org/grpc/credentials/insecure"
)
```

2. Let's add some constants with service names and addresses that we can use later in the test to the file:

```
const (
    metadataServiceName = "metadata"
    ratingServiceName   = "rating"
    movieServiceName     = "movie"
```

```
    metadataServiceAddr = "localhost:8081"
    ratingServiceAddr   = "localhost:8082"
    movieServiceAddr    = "localhost:8083"
)
```

3. The next step is to implement the setup code to instantiate our service servers:

```
func main() {
    log.Println("Starting the integration test")

    ctx := context.Background()
    registry := memory.NewRegistry()

    log.Println("Setting up service handlers and clients")

    metadataSrv := startMetadataService(ctx, registry)
    defer metadataSrv GracefulStop()
    ratingSrv := startRatingService(ctx, registry)
    defer ratingSrv GracefulStop()
    movieSrv := startMovieService(ctx, registry)
    defer movieSrv GracefulStop()
```

Note that `defer` calls to the `GracefulStop` function of each server – this code is a part of the tear-down logic of our test for terminating all servers gracefully.

4. Now, let's set up the test clients for our services:

```
    opts := grpc.WithTransportCredentials(insecure.NewCredentials())
    metadataConn, err := grpc.Dial(metadataServiceAddr, opts)
    if err != nil {
        panic(err)
    }
    defer metadataConn.Close()
    metadataClient := gen.NewMetadataServiceClient(metadataConn)

    ratingConn, err := grpc.Dial(ratingServiceAddr, opts)
    if err != nil {
        panic(err)
```

```

    }
    defer ratingConn.Close()
    ratingClient := gen.NewRatingServiceClient(ratingConn)

    movieConn, err := grpc.Dial(movieServiceAddr, opts)
    if err != nil {
        panic(err)
    }
    defer movieConn.Close()
    movieClient := gen.NewMovieServiceClient(movieConn)

```

Now, we are ready to implement the sequence of our test commands. The first step is to test, write, and read the operations of the metadata service:

```

log.Println("Saving test metadata via metadata service")

m := &gen.Metadata{
    Id:          "the-movie",
    Title:       "The Movie",
    Description: "The Movie, the one and only",
    Director:    "Mr. D",
}

if _, err := metadataClient.PutMetadata(ctx, &gen.PutMetadataRequest{Metadata: m}); err != nil {
    log.Fatalf("put metadata: %v", err)
}

log.Println("Retrieving test metadata via metadata service")

getMetadataResp, err := metadataClient.GetMetadata(ctx, &gen.GetMetadataRequest{MovieId: m.Id})
if err != nil {
    log.Fatalf("get metadata: %v", err)
}
if diff := cmp.Diff(getMetadataResp.Metadata, m, cmpopts.IgnoreUnexported(gen.Metadata{})); diff != "" {
    log.Fatalf("get metadata after put mismatch: %v", diff)
}

```

You may notice that we used the `cmpopts.IgnoreUnexported(gen.Metadata{})` option inside the call to the `cmp.Diff` function – this tells the `cmp` library to ignore unexported fields in the `gen.Metadata` structure. We have added this option because the `gen.Metadata` structure, generated by the Protocol Buffers code generator, includes some private fields that we want to ignore in the comparison.

The next test in our sequence would be to retrieve the movie details and check that the metadata matches the record that we submitted earlier:

```
log.Println("Getting movie details via movie service")

wantMovieDetails := &gen.MovieDetails{
    Metadata: m,
}

getMovieDetailsResp, err := movieClient.GetMovieDetails(ctx, &gen.GetMovieDetailsRequest{MovieId: m.Id})
if err != nil {
    log.Fatalf("get movie details: %v", err)
}
if diff := cmp.Diff(getMovieDetailsResp.MovieDetails,
wantMovieDetails, cmpopts.IgnoreUnexported(gen.MovieDetails{}, gen.Metadata{})); diff != "" {
    log.Fatalf("get movie details after put mismatch: %v", err)
}
```

Now, we are ready to test the rating service.

Let's implement two tests – one for writing a rating and one for retrieving the initial aggregated value, which should match the first rating:

```
log.Println("Saving first rating via rating service")

const userID = "user0"
const recordTypeMovie = "movie"
firstRating := int32(5)
if _, err = ratingClient.PutRating(ctx, &gen.PutRatingRequest{
    UserId:      userID,
    RecordId:   m.Id,
    RecordType: recordTypeMovie,
```

```

        RatingValue: firstRating,
}); err != nil {
    log.Fatalf("put rating: %v", err)
}

log.Println("Retrieving initial aggregated rating via rating service")

getAggregatedRatingResp, err := ratingClient.GetAggregatedRating(ctx,
&gen.GetAggregatedRatingRequest{
    RecordId:    m.Id,
    RecordType: recordTypeMovie,
})
if err != nil {
    log.Fatalf("get aggregated rating: %v", err)
}

if got, want := getAggregatedRatingResp.RatingValue, float64(5); got
!= want {
    log.Fatalf("rating mismatch: got %v want %v", got, want)
}

```

The next part of the test would be to submit the second rating and check that the aggregated value was changed:

```

log.Println("Saving second rating via rating service")

secondRating := int32(1)
if _, err = ratingClient.PutRating(ctx, &gen.PutRatingRequest{
    UserID:      userID,
    RecordId:   m.Id,
    RecordType: recordTypeMovie,
    RatingValue: secondRating,
}); err != nil {
    log.Fatalf("put rating: %v", err)
}

log.Println("Saving new aggregated rating via rating service")

```

```
getAggregatedRatingResp, err = ratingClient.GetAggregatedRating(ctx,
&gen.GetAggregatedRatingRequest{
    RecordId:   m.Id,
    RecordType: recordTypeMovie,
})
if err != nil {
    log.Fatalf("get aggregated rating: %v", err)
}

wantRating := float64((firstRating + secondRating) / 2)
if got, want := getAggregatedRatingResp.RatingValue, wantRating; got
!= want {
    log.Fatalf("rating mismatch: got %v want %v", got, want)
}
```

We are almost done with our `main` function – let's implement the last check:

```
log.Println("Getting updated movie details via movie service")

getMovieDetailsResp, err = movieClient.GetMovieDetails(ctx, &gen.
GetMovieDetailsRequest{MovieId: m.Id})
if err != nil {
    log.Fatalf("get movie details: %v", err)
}
wantMovieDetails.Rating = wantRating
if diff := cmp.Diff(getMovieDetailsResp.MovieDetails,
wantMovieDetails, cmpopts.IgnoreUnexported(gen.MovieDetails{}, gen.
Metadata{})); diff != "" {
    log.Fatalf("get movie details after update mismatch: %v", err)
}

log.Println("Integration test execution successful")
}
```

Our integration test is almost ready. Let's add functions for initializing the servers for our services below the `main` function. First, add a function for creating the server for a metadata service:

```
func startMetadataService(ctx context.Context, registry discovery.Registry) *grpc.Server {
    log.Println("Starting metadata service on " + metadataServiceAddr)
    h := metadatatest.NewTestMetadataGRPCServer()
    l, err := net.Listen("tcp", metadataServiceAddr)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    srv := grpc.NewServer()
    gen.RegisterMetadataServiceServer(srv, h)
    go func() {
        if err := srv.Serve(l); err != nil {
            panic(err)
        }
    }()
    id := discovery.GenerateInstanceID(metadataServiceName)
    if err := registry.Register(ctx, id, metadataServiceName,
        metadataServiceAddr); err != nil {
        panic(err)
    }
    return srv
}
```

You may notice that we call the `srv.Serve` function inside a goroutine – this way, it doesn't block the execution and allows us to immediately return from the function.

Let's add a similar implementation for the rating service server to the same file:

```
func startRatingService(ctx context.Context, registry discovery.Registry) *grpc.Server {
    log.Println("Starting rating service on " + ratingServiceAddr)
    h := ratingtest.NewTestRatingGRPCServer()
    l, err := net.Listen("tcp", ratingServiceAddr)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
}
```

```
    srv := grpc.NewServer()
    gen.RegisterRatingServiceServer(srv, h)
    go func() {
        if err := srv.Serve(l); err != nil {
            panic(err)
        }
    }()
    id := discovery.GenerateInstanceID(ratingServiceName)
    if err := registry.Register(ctx, id, ratingServiceName,
ratingServiceAddr); err != nil {
        panic(err)
    }
    return srv
}
```

Finally, let's add a function for initializing the movie server:

```
func startMovieService(ctx context.Context, registry discovery.Registry)
*grpc.Server {
    log.Println("Starting movie service on " + movieServiceAddr)
    h := movietest.NewTestMovieGRPCServer(registry)
    l, err := net.Listen("tcp", movieServiceAddr)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    srv := grpc.NewServer()
    gen.RegisterMovieServiceServer(srv, h)
    go func() {
        if err := srv.Serve(l); err != nil {
            panic(err)
        }
    }()
    id := discovery.GenerateInstanceID(movieServiceName)
    if err := registry.Register(ctx, id, movieServiceName,
movieServiceAddr); err != nil {
        panic(err)
    }
    return srv
}
```

Our integration test is ready! You can run it by executing the following command:

```
go run test/integration/*.go
```

If everything is correct, you should see the following output:

```
2022/07/16 16:20:46 Starting the integration test
2022/07/16 16:20:46 Setting up service handlers and clients
2022/07/16 16:20:46 Starting metadata service on localhost:8081
2022/07/16 16:20:46 Starting rating service on localhost:8082
2022/07/16 16:20:46 Starting movie service on localhost:8083
2022/07/16 16:20:46 Saving test metadata via metadata service
2022/07/16 16:20:46 Retrieving test metadata via metadata service
2022/07/16 16:20:46 Getting movie details via movie service
2022/07/16 16:20:46 Saving first rating via rating service
2022/07/16 16:20:46 Retrieving initial aggregated rating via rating
service
2022/07/16 16:20:46 Saving second rating via rating service
2022/07/16 16:20:46 Saving new aggregated rating via rating service
2022/07/16 16:20:46 Getting updated movie details via movie service
2022/07/16 16:20:46 Integration test execution successful
```

As you may notice, the structure of our integration test precisely matches the sequence of test operations that we defined earlier. We implemented our integration test as an executable command and added enough log messages to help you with debugging – if any step fails, it is therefore easier to understand at which step the failure occurred and which operations preceded that step.

It is important to note that we used the in-memory versions of the metadata and rating repositories in our integration test. An alternative approach would be to set up an integration test that stores the data in some persistent databases, such as MySQL. However, there are some challenges with using existing persistent databases in integration tests:

- Integration test data should not interfere with user data. Otherwise, it may cause unexpected effects on existing service users.
- Ideally, test data should be cleaned up after test execution so that the database does not get filled with unnecessary, temporary data.

In order to avoid interference with existing user data, I would suggest running integration tests on non-production environments, such as staging. Additionally, I would suggest always generating random identifiers for your test records to make sure that individual test executions don't

affect each other. For example, you can use the `github.com/google/uuid` library to generate new identifiers using the `uuid.New()` function. Lastly, I would recommend always including cleanup code at the end of each integration test that uses persistent data storage to clean up the created records, whenever this is possible.

Now, the question is when we should write integration tests. It is always up to you; however, I do have some general suggestions:

- **Test critical flows:** Make sure you test entire flows, such as user signups and logins
- **Test critical endpoints:** Perform tests of the most critical endpoints that your services provide to your users

Additionally, you may have integration tests that are executed after each code change. Systems such as Jenkins provide these kinds of features and allow you to plug any custom logic that would be executed into each update of your code. We won't cover Jenkins setup in this book, but you can familiarize yourself with its documentation on the official website (<https://www.jenkins.io>).

As we have illustrated how to write both unit and integration tests, let's proceed to the next section of the book, describing some of the best practices of Go testing.

Testing best practices

In this section, we are going to list some additional useful testing tips that are going to help you improve the quality of your tests.

Using helpful messages

One of the most important aspects of writing tests is providing enough information in error logs that it is easy to understand exactly what went wrong and which test case triggered the failure. Consider the following test case code:

```
if got, want := Process(tt.in), tt.want; got != want {  
    t.Errorf("Result mismatch")  
}
```

The error log does not include both the expected and the actual value received from the function being tested, making it harder to understand what the function returned and how it was different from the expected value.

A better log line would be as follows:

```
t.Errorf("got %v, want %v", got, want)
```

This log line includes the expected and the actual returned value of the function and provides much more context to you when you debug the test.

Important note



Note that in our test logs, first, we log the actual value and then the expected one. This order is recommended by the Go team as the conventional way of logging values in tests and is followed in all libraries and packages. Follow the same order in your logs for consistency.

An even better error message would be as follows:

```
t.Errorf("YourFunc(%v) = %v, want %v", tt.in, got, want)
```

This error log message includes some additional information – the function being called and the input argument that was passed to it.

To standardize the code for your test cases, you can use the `github.com/stretchr/testify` library. The following example illustrates how to compare the expected and the actual value and log the name of the function being tested, as well as the argument passed to it:

```
assert.Equal(t, want, got, fmt.Sprintf("YourFunc(%v)", tt.in))
```

The `assert` package of the `github.com/stretchr/testify` library prints both the expected and the actual value of the test result, as well as providing details about the test case (the `fmt.Sprintf` result, in our case).

Avoiding the use of Fatal in your logs

The built-in Go testing library includes different functions for logging errors, including `Error`, `Errorf`, `Fatal`, and `Fatalf`. The last two functions print logs and interrupt the execution of tests. Consider this test code:

```
if err := Process(tt.in); err != nil {
    t.Fatalf("Process(%v): %v, want nil", err)
}
```

The call to the `Fatalf` function interrupts the test execution. Interrupting test execution is often not the best idea because it leads to fewer tests being executed. Executing fewer tests leaves the developer with less information for the remaining failing test cases. Fixing one error and running all the tests again may be a suboptimal experience for many developers, and it is often better to continue the test execution whenever possible.

The previous example can be re-written as follows:

```
if err := Process(tt.in); err != nil {
    t.Errorf("Process(%v): %v, want nil", err)
}
```

If you use this code in a loop, you can add `continue` after the `Errorf` call to proceed to the next test cases.

Comparing structures using a cmp library

Imagine that you have a test that compares the `Metadata` structure we defined in *Chapter 2 Scaffolding a Go Microservice*:

```
want := &model.Metadata{ID: "123", Title: "\"Some title\""}
id := "123"
if got := GetMetadata(ctx, "123"); got != want {
    t.Errorf("GetMetadata(%v): %v, want %v", id, got, want)
}
```

The code here would not work for structure references – in our code, the `want` variable holds a pointer to the `model.Metadata` structure, so the `!=` operator will return true even for structures with the same field values if these structures are created separately.

A comparison of structure pointers can be made in Go using the `reflect.DeepEqual` function:

```
if !reflect.DeepEqual(GetMetadata(ctx, "123"), want) {
    t.Errorf("GetMetadata(%v): %v, want %v", id, *got, *want)
}
```

However, the output of the test may not be easy to read. Consider that you have lots of fields inside the `Metadata` structure – if only one field is different, you will need to scan through both structures to find the difference. There is a convenient library that simplifies comparison in tests called `cmp` (<https://pkg.go.dev/github.com/google/go-cmp/cmp>).

The `cmp` library allows you to compare arbitrary Go structures in the same way as with `reflect.DeepEqual`, but it also provides a human-readable output. Here's an example of using the function:

```
if diff := cmp.Diff(want, got); diff != "" {
    t.Errorf("GetMetadata(%v): mismatch (-want +got):\n%s", tt.in, diff)
}
```

If the structures don't match, the `diff` variable will be a non-empty string, including a printable representation of the differences between them. Here's an example of this kind of output:

```
GetMetadata(123) mismatch (-want +got):  
  model.Metadata{  
    ID:      "123",  
    -      Title: s"Title",  
    +      IPAddress: s"The Title",  
  }
```

Note how the `cmp` library highlighted the differences between both structures using the `-` and `+` prefixes. Now, it is easy to read the test output and notice the differences between the structures – this kind of optimization will save you lots of time during debugging.

Detecting and fixing flaky tests

Flaky tests are tests that don't produce consistent results when being executed over the same code. Consider the following test code:

```
func TestWriteAfterRead(t *testing.T) {  
    c := newMessageQueue()  
    err := c.ProduceMessage("some-message")  
    assert.NoError(t, err)  
    time.Sleep(time.Millisecond)  
    _, err = c.ConsumeMessage()  
    assert.NoError(t, err)  
}
```

Our test example produces a message to a message queue and tries to consume the data after waiting for 1 millisecond.

This test might not always work: assume that we get some unexpected delay (for example, an operating system running our Go code becomes temporarily unresponsive due to a high process count) and our message queue has not processed the message yet. In such a case, our test would fail, but in most other cases, it might run without any issues.

Writing non-flaky tests takes some practice. In general, the rule for avoiding flakiness in your tests is to avoid making assumptions about things you can't fully control, such as the exact time it takes to execute specific operations.

How to detect flaky tests? The simplest solution is to run the same tests many times to see which ones produce inconsistent results. The Go test runner provides a `-count` option for this:

```
go test -count=100 ./...
```

It is not necessary to use the `-count` option on each test execution, but it is generally a good practice to perform this action periodically so that you can detect flaky tests as soon as possible.

Detecting race conditions

Race conditions are special cases of code execution where multiple functions, goroutines, or processes attempt to modify and read some shared values, causing unexpected issues such as panics or incorrect read results.

Let's take the following test as an example:

```
func dataRace() {
    var counter int64
    var wg sync.WaitGroup
    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            counter++
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Printf("%v\n", counter)
}
```

Our code starts 100 goroutines that try incrementing the same `int` variable. If you run this code locally multiple times and print the value of the `counter` variable on each run, you will notice that the result of each execution might be different: sometimes you will be getting 100 as the result, while other runs will produce different values, such as 98.

The Go tool allows you to detect race conditions by using a `-race` option:

```
go test -race .
```

If race conditions are detected, you will get a similar output:

```
WARNING: DATA RACE
Read at 0x00b000243000 by goroutine 3:
    main.(*Counter).Value()
    ...
Previous write at 0x00b000243000 by goroutine 5:
    main.(*Counter).Increment()
    ...
```

Detecting and fixing race conditions is an important part of the development process to make sure there is no unexpected behavior in your application logic.

Tracking and maintaining high code coverage

The Go test tool allows you to report **code coverage** – the percentage of code lines that are covered by unit tests. Code coverage is generally a good indicator of code quality and reliability: code with high coverage provides guarantees of work against the defined test cases.

To report a code coverage, run the `go test` command with the `-cover` option. For example, you might navigate to our `metadata/internal/controller/metadata` directory and run the following:

```
go test -cover .
```

You will get the following result:

```
ok      movieexample.com/metadata/internal/controller/metadata 0.302s
coverage: 83.3% of statements
```

Code coverage for our package is 83.3% – this is generally considered a good result.

You might also report coverage for all packages inside your `src` directory by navigating to it and running the following:

```
go test -cover ./...
```

The output will be similar, with one line per each package of our code base.

The general suggestion is to perform code coverage checks periodically (you might even do this on each commit) and maintain solid coverage (for example, 85% and above) for all your code.

This summarizes our short collection of Go testing best practices – you can find more tips by reading the documents mentioned in the *Further reading* section. Make sure to familiarize yourself with the official recommendations and the comments for the `testing` package to learn how to write tests in a conventional way and leverage all the features provided by the built-in Go testing library.

Summary

In this chapter, we covered multiple topics related to Go testing, including common features of the Go testing library and the basics of writing unit and integration tests for your code. You learned how to add tests to your microservices, optimize test execution in various cases, create test mocks, and maximize the quality of your tests by following the best testing practices. The knowledge you gained from reading this chapter should help you to increase the efficiency of your testing logic and increase the reliability of your microservices.

In the next chapter, we will summarize our learning on writing and structuring microservice code and focus on the structural patterns of Go microservice development.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Golang test comments: <https://go.dev/wiki/TestComments>
- Golang testing package documentation: <https://pkg.go.dev/testing>
- Using subtests and sub-benchmarks: <https://go.dev/blog/subtests>
- Chaos testing: <https://www.pagerduty.com/resources/learn/what-is-chaos-testing/>
- Go Cookbook: Testing: <https://go-cookbook.com/snippets/testing>

10

Security and Compliance

In this chapter, we are going to review two topics of high practical significance: security and compliance. In the first part of this chapter, you will learn the main techniques you can implement to secure your Go microservices, such as authentication, authorization, and traffic encryption. In the second part of this chapter, we will cover the key principles of software regulatory compliance that might apply to microservice development. Upon completing this chapter, you will have a good understanding of how to implement security measures to protect communication between your microservices. Additionally, you will get an overview of common regulatory compliance challenges that you might face while developing or launching your microservice applications that might affect the design and implementation of your services.

This chapter contains the following topics:

- Security basics
- Implementing secure service communication with **Transport Layer Security (TLS)**
- Implementing authentication and access control with **JSON Web Tokens (JWT)**
- Security analysis of Go services
- Security best practices
- Compliance basics

Technical requirements

To complete this chapter, you will need Go 1.18 or above, as well as the following tools:

- **The Protobuf compiler:** <https://grpc.io/docs/protoc-installation/>
- **The Go gRPC plugin:** You should already have it installed since you were required to install it in *Chapter 5*, but you can always install or update it by running the following command:

```
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest  
export PATH="$PATH:$(go env GOPATH)/bin"
```

- **grpcurl:** <https://github.com/fullstorydev/grpcurl>
- **openssl:** <https://wiki.openssl.org/index.php/Binaries>

You can find the GitHub code for this chapter at <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter10>

Security basics

In microservice development, security is one of the critical challenges that affects the ways you store, process, and transmit data over the system. A breach in one service might compromise the entire system, making it vulnerable to various risks and attacks. Therefore, security should be seen as one of the critical aspects of service development from the very early phases of the service life cycle.

In this section, we are going to review the key areas of software security that apply to microservice development and demonstrate how to apply them to the services we created in the previous chapters.

Key areas of software security

First, we need to define a list of some of the key areas of software security that apply to microservices:

- **Authentication:** Verifying the identities of system users or services
- **Access control:** Defining and enforcing access permissions
- **Data security:** Encrypting data that is stored in the system (**data at rest**), as well as data that is transmitted over the system (**data in transit**)
- **Application security and vulnerability management:** Identifying and fixing common application security issues and **vulnerabilities**, such as code injection
- **Secure configuration management:** Mechanisms for storing and accessing secure configuration, such as database credentials or encryption keys

Next, let's review the basics of some of the key areas of software security, including authentication, access control, and communication data security.

Authentication and access control

Within the context of microservice development, there are two basic types of authentication:

- **Server authentication:** Verifying the identity of server-side applications, such as microservices
- **User authentication:** Verifying the identity of users

The mechanisms behind each of these types of communication are quite different. Service authentication is generally performed using **digital certificates** – special files obtained from trusted organizations called **Certificate Authorities (CAs)** (some of the most widely known CA organizations include DigiCert, GlobalSign, and GoDaddy) that can help to verify its owner's identity. Each digital certificate is associated with a **private key**, which must be known only by its owner. By using this private key, the certificate owner (for example, a microservice or a web server) can sign the certificate to verify its authenticity to its callers.

User authentication is generally based on other verification mechanisms, such as user credential checks. Such authentication can take one or multiple steps: some types of authentication, such as **two-factor authentication**, require some additional actions, including email or SMS verifications.

Successful authentication often results in the caller being granted access to some resources, such as service or user data (for example, user emails in Gmail). Additionally, the server performing this authentication may provide a security token to the caller that can be used on subsequent calls so that they can skip the verification process.

Once the caller's identity has been verified, we can perform additional **access control** checks for each operation in the system, such as data reads or calls to specific endpoints. This form of access control is also known as **authorization**, and it involves the following steps:

1. Specifying rights for accessing specific resources or performing defined operations.
2. Performing access checks on relevant actions in the system.

Authorization is often performed by using a security token that was obtained during authentication, as illustrated in the following diagram:

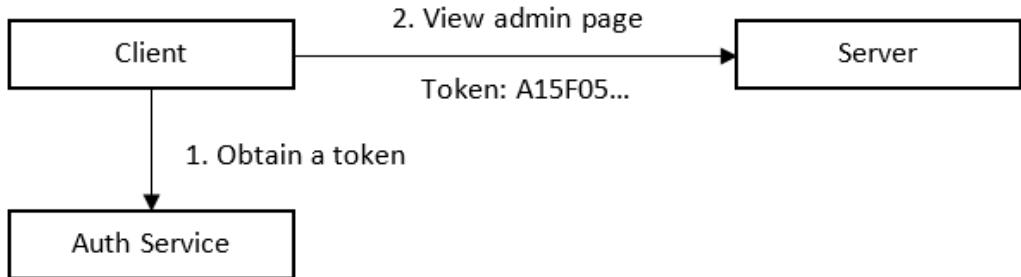


Figure 10.1 – Authorization request providing a token

Once a client obtains a security token, it can use it to request data from a server. A token might include some additional information that could be used for performing extra access checks, such as user identifiers, user roles, or permissions.

Secure service communication

Another key aspect of service security is secure service communication. Without establishing this, it might be impossible to secure other flows in the system, including authentication. Let's take user credential checks as an example – if you have an API that accepts user credentials without extra encryption, the data that's sent to it can easily be intercepted or manipulated, resulting in multiple negative outcomes, including compromised passwords and illegal system access (such kinds of attacks based on data interception are often called **man-in-the-middle** attacks).

There are some common ways of securing service communication, and among the most common ones is the TLS protocol. TLS solves the following problems regarding communication security:

- **Confidentiality:** TLS encrypts the communication data, making it impossible for unauthorized parties to access it, even in the case of interception
- **Integrity:** TLS provides mechanisms for checking that the data has not been tampered with during transit to ensure that the received data matches what was sent
- **Authenticity:** TLS verifies that the parties involved in the communication are who they claim to be

Internally, the TLS protocol uses the following mechanisms to enforce communication security:

- **Symmetric data encryption and key exchange:** TLS encrypts the data via sets of keys (a combination of a **private** and a **public key**) and involves performing a public key exchange between the components

- **Checksums:** TLS uses cryptographic checksum functions to perform checks for message tampering
- **Digital certificates:** TLS authentication is primarily based on the use of digital certificates issued by trusted CA organizations

The general process of establishing secure client-service TLS communication involves the following steps:

1. The server obtains a private key and a digital certificate from a CA. This digital certificate contains some additional information that can be used to verify its identity to server clients, such as the server's domain name.
2. When establishing a connection with the server, the client verifies the server's certificate by checking it against a trusted CA. In addition to this, the client sends a special sequence of data to the server. This is used to create a **session key** that will encrypt all communication data between them.
3. If the server possesses the original private key obtained in *step 1*, it can prove its identity by decoding the data that was sent by the client. After this, the client and the server establish a secure connection by using the session key created in *step 2*, which is known only to them.
4. There is an extension of this protocol that allows both client and server identities to be authenticated. This extension is called **Mutual TLS (mTLS)** and it can be used to add extra protection mechanisms for communication between multiple services. Since it involves a more complex configuration, we won't cover it in detail in this chapter (if you are interested, you can find a link to the relevant article in the *Further reading* section). Instead, we are going to illustrate how to implement basic secure client-server communication using a TLS protocol with server verification.

Implementing secure service communication with TLS

In this section, we are going to demonstrate how to establish secure client-server communication for our services using a TLS protocol. To do so, we are going to generate a digital test certificate that we will use via our microservices. Then, we will update the server and client code of our microservices so that they can use the certificate data to verify each other's identities and secure communication between them.

Let's begin the TLS onboarding process by following these steps:

1. First, let's generate a test certificate and a private key file that we will use to authenticate our services:

```
openssl req -x509 -nodes -newkey rsa:4096 \
  -keyout server.key -out server.crt -days 365 -nodes \
  -subj "/C=US/ST=State/L=City/O=Organization/OU=Department/
  CN=localhost" \
  -addext "subjectAltName=DNS:localhost,DNS:example.com,IP:127.0.0.1,
  IP:192.168.1.1" -config /dev/null
```

2. In the preceding code, we generated an **X.509** certificate (X.509 is a widely used certificate standard) that's valid for 365 days and provides the necessary options, including the metadata of the CA inside the `-subj` argument (we filled it with stub data that is sufficient for local testing). This command also generates a 4,096-bit private key associated with the certificate; our services are going to use this to verify their identities. We also provided some extra flags: `-nodes` to skip extra key encryption and `-config` to perform fully automated key generation without user prompts).
3. Move the files that were generated in the previous step into the `configs` directory of each of our microservices.

Note



Note that the private keys we just created should, in general, not be stored inside code repositories so that they're not accidentally exposed. To store keys, you typically need specialized encrypted storage solutions, such as HashiCorp Vault or AWS Secrets Manager. We are going to cover these briefly later in the *Security best practices* section.

4. Open the `metadata/cmd/main.go` file and add imports for the `crypto/tls` and `google.golang.org/grpc/credentials` packages to it.
5. Find the line containing the `net.Listen` call and add the following code block before it:

```
cert, err := tls.LoadX509KeyPair("server.crt", "server.key")
if err != nil {
    log.Fatalf("Failed to load key pair: %v", err)
}
creds := credentials.NewTLS(&tls.Config{Certificates: []tls.Certificate{cert}})
```

6. Replace the line containing the call to the `grpc.NewServer` function with the following:

```
srv := grpc.NewServer(grpc.Creds(creds))
```

Here, we made our metadata gRPC server read server TLS credentials from the files that we generated earlier. Our server is now ready to serve TLS connections.

7. Run our metadata microservice so that it has access to the newly created configuration. Similar to previous chapters, you can do this by running the following command from its `configs` directory:

```
go run ../cmd/*.go
```

8. Let's test that it is ready to accept calls by using the `grpcurl` tool. Run the following command:

```
grpcurl -cacert server.crt -d '{"movie_id":"1"}' localhost:8081  
MetadataService/GetMetadata
```

Our `grpcurl` command now contains the `-cacert` argument, which provides path to the certificate of our service. All the callers of the metadata service will need to have access to that certificate to make TLS calls to it. If you did everything correctly, you should see a valid response to our request.

Let's check what happens if we don't pass a certificate when we call our service. Remove the `-cacert` option and its argument so that the command looks like this:

```
grpcurl -d '{"movie_id":"1"}' localhost:8081 MetadataService/GetMetadata
```

If you run the command again, you will notice a similar message:

```
Failed to dial target host "localhost:8081": tls: failed to verify  
certificate: x509: certificate signed by unknown authority
```

The `grpcurl` tool and our server can't establish a secure connection with each other anymore because they can't verify the identity of the server. This is because they don't have the digital certificate data. You can perform a similar check by copying the digital certificate and modifying it: you still wouldn't be able to establish a secure communication without using the right certificate data.

Now, let's demonstrate how to update client-side gRPC code to establish secure communication with TLS-based servers. As you may recall from the previous chapters, the metadata service is called by our movie service, so we will also need to update its code to establish a certificate-based connection to our server. For this, let's update the movie service code:

1. Open the `main.go` file of the movie service and add the following imports:

```
"crypto/tls"
"crypto/x509"
"google.golang.org/grpc/credentials"
```

2. Find the line where we initialize the `metadataGateway` variable and replace it with the following code:

```
certBytes, err := os.ReadFile("server.crt")
if err != nil {
    log.Fatalf("failed to read server certificate: %v", err)
}
certPool := x509.NewCertPool()
if !certPool.AppendCertsFromPEM(certBytes) {
    log.Fatalf("Failed to append server certificate to pool")
}
cert, err := tls.LoadX509KeyPair("server.crt", "server.key")
if err != nil {
    log.Fatalf("Failed to load key pair: %v", err)
}
creds := credentials.NewTLS(&tls.Config{Certificates: []tls.Certificate{cert}})
metadataGateway := metadatagateway.New(registry, creds)
```

3. Also, find the line where we call the `NewServer` function and replace it with the following:

```
srv := grpc.NewServer(grpc.Creds(creds))
```

4. At this point, we can open the `internal/grpcutil/grpcutil.go` file and add the following import to it:

```
"google.golang.org/grpc/credentials"
```

5. Replace the `ServiceConnection` function inside it with this code:

```
// ServiceConnection attempts to select a random service instance
// and returns a gRPC connection to it.
func ServiceConnection(ctx context.Context, serviceName string,
    registry discovery.Registry, creds credentials.TransportCredentials)
(*grpc.ClientConn, error) {
```

```
    addrs, err := registry.ServiceAddresses(ctx, serviceName)
    if err != nil {
        return nil, err
    }
    return grpc.Dial(addrs[rand.Intn(len(addrs))], grpc.
        WithTransportCredentials(creds))
}
```

6. Now, let's update our metadata gateway code so that we can use certificate-based credentials. Open the `movie/internal/gateway/metadata/grpc/metadata.go` file and add the following line to its imports:

```
"google.golang.org/grpc/credentials"
```

7. Additionally, let's update the `Gateway` structure definition and the `New` function to the following:

```
// Gateway defines a movie metadata gRPC gateway.
type Gateway struct {
    registry discovery.Registry
    creds    credentials.TransportCredentials
}

// New creates a new gRPC gateway for a movie metadata service.
func New(registry discovery.Registry, creds credentials.
TransportCredentials) *Gateway {
    return &Gateway{registry, creds}
}
```

8. Find the line where we make a call to the `ServiceConnection` function and replace it with the following code:

```
    conn, err := grpcutil.ServiceConnection(ctx, "metadata",
g.registry, g.creds)
```

9. Inside the `movie/internal/gateway/rating/grpc/rating.go` file, replace the call to the `ServiceConnection` function with the following:

```
    conn, err := grpcutil.ServiceConnection(ctx, "rating",
g.registry, insecure.NewCredentials())
```

Let's briefly describe our changes:

1. First, we onboarded our movie gRPC service to TLS, similar to what we did for the metadata service.
2. Then, we updated our gRPC client library so that TLS certificate-based credentials can be passed to establish TLS connections.
3. Finally, we updated the code of the movie service, which calls the metadata service, so that it uses TLS for secure communication between the two services.

You can now run the movie service and make calls to it via `grpcurl`:

```
grpcurl -cacert server.crt -d '{"movie_id": "1"}' localhost:8083  
MovieService/GetMovieDetails
```

With that, calls between the `grpcurl` tool and our movie service, as well as a call between the movie service and the metadata service, have been secured with TLS. Optionally, you may wish to onboard the rating service to the TLS, similar to what you did for the metadata service, to make sure all three services have been secured.

Additional improvements

As mentioned earlier in this section, our TLS solution is rather basic and we can make some possible improvements. We won't implement them in this chapter, but it is worth noting them here so that you're aware of them:

- Each service might get a private key and a certificate, and they'll use these instead of the shared versions. This way, the risk of private key exposure would be reduced (even if the key of some service gets exposed, it wouldn't work for other services), and we would be able to perform more strict verification of service identities.
- Private keys would need to be moved out of the `configs` directory of each service to some form of secure secret storage. We will cover this in the *Security best practices* section.
- In a production environment, you would need to obtain a certificate from a trusted CA, such as DigiCert, instead of generating one yourself.

Now that we've covered the basics of secure communication between our microservices, let's move on to the next section, where we will explore how to establish user authentication and access checks.

Implementing authentication and access control with JWT

In this section, we will demonstrate some practical ways of establishing user authentication and access control. There are many different ways to implement such flows in a microservice environment, among which is JWT, a proposed internet standard for creating security tokens. First, let's review the basics of the protocol to help you understand how to use it in our microservices.

The basics of JWT

In JWT, security tokens are generated by components that perform authentication or authorization. Each token consists of three parts: a header, a payload, and a signature. The payload is the main part of the token and contains a set of **claims** – statements about the caller's identity, such as a user identifier or a role in the system. The following code shows an example of a token payload:

```
{  
  "name": "Alexander",  
  "role": "admin",  
  "iat": 1663880774  
}
```

Our example payload contains three claims: the user's name, role (in our case, `admin`), and token issuance time (`iat` is a standard field name that's part of the JWT protocol). Such claims can be used in various flows – for example, when checking whether a user has the `admin` role so that they can access a system dashboard.

To protect against modifications, each token contains a **signature** – a cryptographic function of its payload, header, and a special value, called a **secret**, that is known only to the authentication server. The following pseudocode provides an example of token signature calculation:

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret,  
)
```

The algorithm that is used for creating a token signature is defined in a **token header**. The following JSON record provides an example of a header:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

In our example, the token is using *HMAC-SHA256*, a cryptographic algorithm that is commonly used for signing JWTs. We primarily chose HMAC-SHA256 due to its popularity; if you wish to learn about the other available signing algorithms, you can find a link to an overview of them in the *Further reading* section of this chapter.



Tip

There is a JWT tool available at <https://jwt.io> that can help you understand the structure of JWT tokens. I strongly suggest that you try it out.

The resulting JWT is a concatenation of the token's header, payload, and signature, encoded with the *Base64uri* algorithm. For example, the following value is a JWT that's been created by combining the header and the payload from our code snippets, signed with a random secret string called `our-secret`:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJpY1IjoiQWxleGFuZGVyIiwicm9sZSI6ImFkbWluIiwiaWF0IjoxNjYzODgwNzc0fQ.  
FqogLyrV28wR5po6SMouJ7qs2Y3m6gmpaPg6MuthWpQ
```

You can use the JWT tool available at <https://jwt.io> to decode our JWT payload and see the resulting contents.

Implementing JWT issuance and validation in microservices

Let's demonstrate how to use JWT for authentication and basic access control for our microservices. First, let's start with the authentication process. A simple credential-based authentication flow can be summarized as follows:

1. The client initiating authentication would call a specified endpoint (for example, `GetToken`) while providing the necessary user credentials, such as username and password.

Note

Note that all endpoints that accept sensitive data, such as user credentials, must be secured (for example, by using TLS communication) to avoid accidental data exposure.

2. The server handling authentication would verify the credentials and perform one of two actions:
 - Return an error if the credentials are invalid (for example, a gRPC error with an `Unauthenticated` code)
 - Return a successful response containing a token that has been signed with the server's secret

If authentication is successful, the client can store the received token so that it can be used in the following requests.

- Let's illustrate how to implement the server logic for the authentication flow that we just described. For this, we are going to implement a new service called `AuthService` that will provide the following API endpoints:
 - `GetToken`: Perform credential verification and issue a new JWT token
 - `ValidateToken`: Validate the provided token and return the identifier of the user in case the token is valid

The steps for updating our service code are as follows:

1. First, let's create a `.proto` file that contains the definition of our API. Inside our `api` directory, create a file called `auth.proto` that contains the following code:

```
syntax = "proto3";
option go_package = "/gen";

message GetTokenRequest {
    string username = 1;
    string password = 2;
}
```

```
message GetTokenResponse {
    string token = 1;
}

message ValidateTokenRequest {
    string token = 1;
}

message ValidateTokenResponse {
    string username = 1;
}

service AuthService {
    rpc GetToken(GetTokenRequest) returns (GetTokenResponse);
    rpc ValidateToken(ValidateTokenRequest) returns
(ValidateTokenResponse);
}
```

2. Inside the `src` directory, run the `protoc` command to generate code for our API:

```
protoc -I=api --go_out=. --go-grpc_out=. auth.proto
```

3. Now, we can implement our auth service logic. Inside the `src` directory, create the following new directories:

```
auth/cmd  
auth/configs  
auth/internal/handler/grpc
```

4. Copy the `server.crt` and `server.key` files that we created in the previous section inside the `configs` directory of the auth service.
5. Create the `main.go` file inside the `auth/cmd` directory and add the following code to it:

```
package main

import (
    "crypto/tls"
    "fmt"
    "log"
    "net"
```

```
        "google.golang.org/grpc"
        "google.golang.org/grpc/credentials"
        "google.golang.org/grpc/reflection"
    grpchandler "movieexample.com/auth/internal/handler/grpc"
    "movieexample.com/gen"
)
```

6. Let's also implement the `main` function inside this file:

```
func main() {
    port := 8084
    log.Printf("Starting the auth service on port %d", port)
    cert, err := tls.LoadX509KeyPair("server.crt", "server.key")
    if err != nil {
        log.Fatalf("Failed to load key pair: %v", err)
    }
    creds := credentials.NewTLS(&tls.Config{Certificates: []tls.Certificate{cert}})
    lis, err := net.Listen("tcp", fmt.Sprintf(":%v", port))
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    h := grpchandler.New(func() []byte {
        // Initialize the handler with a secret.
        // In production, this would be replaced
        // by a secure secret retrieval function.
        return []byte("test-secret")
    })
    srv := grpc.NewServer(grpc.Creds(creds))
    reflection.Register(srv)
    gen.RegisterAuthServiceServer(srv, h)
    if err := srv.Serve(lis); err != nil {
        panic(err)
    }
}
```

7. We are one step away from implementing our new service. So, let's create a gRPC handler that will perform token validation. Inside the auth/internal/handler/grpc directory, create a file called grpc.go and add the following code to it:

```
package grpc

import (
    "context"
    "fmt"
    "time"

    "github.com/golang-jwt/jwt/v5"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    "movieexample.com/gen"
)

// SecretProvider defines a provider of secrets for our handler.
type SecretProvider func() []byte

// Handler defines an auth gRPC handler.
type Handler struct {
    secretProvider SecretProvider
    gen.UnimplementedAuthServiceServer
}

// New creates a new auth gRPC handler.
func New(secretProvider SecretProvider) *Handler {
    return &Handler{secretProvider: secretProvider}
}
```

8. Now, we can implement the GetToken function inside the same file:

```
// GetToken performs verification of user credentials and returns a
JWT token in case of success.
func (h *Handler) GetToken(ctx context.Context, req *gen.
GetTokenRequest) (*gen.GetTokenResponse, error) {
    username, password := req.Username, req.Password
    if !validCredentials(username, password) {
```

```
        return nil, status.Errorf(codes.Unauthenticated, "invalid
credentials")
    }
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.
MapClaims{
    "username": username,
    "iat":       time.Now().Unix(),
})
tokenString, err := token.SignedString(h.secretProvider())
if err != nil {
    return nil, status.Errorf(codes.Internal, err.Error())
}
return &gen.GetTokenResponse{Token: tokenString}, nil
}

func validCredentials(username string, password string) bool {
    if username == "" || password == "" {
        return false
    }
    // We intentionally skip verification of username and password
    // to simplify chapter code.
    return true
}
```

- Finally, let's implement the `ValidateToken` function inside the same file:

```
// ValidateToken performs JWT token validation.
func (h *Handler) ValidateToken(ctx context.Context, req *gen.
ValidateTokenRequest) (*gen.ValidateTokenResponse, error) {
    token, err := jwt.Parse(
        req.Token,
        func(token *jwt.Token) (interface{}, error) {
            if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
                return nil,
                fmt.Errorf(
                    "unexpected signing method: %v", token.
Header["alg"],
                )
            }
        }
    )
```

```
        return h.secretProvider(), nil
    })
    if err != nil {
        return nil, status.Errorf(codes.Unauthenticated, "invalid
token")
    }
    claims, ok := token.Claims.(jwt.MapClaims)
    if !ok || !token.Valid {
        return nil, status.Errorf(codes.Unauthenticated, "invalid
token")
    }
    var username string
    if v, ok := claims["username"]; ok {
        if u, ok := v.(string); ok {
            username = u
        }
    }
    return &gen.ValidateTokenResponse{Username: username}, nil
}
```

Let's take a look at the code that we just added.

First, in the `GetToken` API handler, we created a token using the `jwt.NewWithClaims` function. The token includes two fields:

- `username`: The name of the authenticated user
- `iat`: The time of token creation

The server code that we just created uses the `secret` value to sign the token. Any attempts to modify the token would be impossible without knowing the secret: the token signature allows us to check the correctness of the token. In a production environment, the value of the secret should be kept separately from the code so that it can be accessed securely (we are going to cover how to do this in the *Security best practices* section).

Note that we intentionally left `validCredentials` blank to demonstrate where you would need to implement credential verification logic. The complete version of this code would need to perform username and password checks against securely stored user data, which you might implement separately, or use some existing services, such as AWS Cognito (coverage of such services is outside of the scope of this chapter, but you can find a link to the relevant documentation in the *Further reading* section).

Now that our service code implementation is ready, we can test it:

1. Run the auth service by executing a regular `go run *.go` command.
2. Now, let's test that we can communicate with our service using TLS. For this, execute the following command inside the `configs` directory of our auth service:

```
grpcurl -cacert server.crt -d '{}' localhost:8084 AuthService/
GetToken
```

You should get a message indicating that the request hasn't been unauthenticated due to invalid credentials:

```
ERROR:
Code: Unauthenticated
Message: invalid credentials
```

This message is expected because we have not provided a valid username and password. Update our request so that they're included:

```
grpcurl -cacert server.crt -d '{"username": "user1", "password": "password"}' localhost:8084 AuthService/GetToken
```

3. The response to our request will now include a token:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJpYXQiOjE3MzIzNTUzMjIsInVzZXJuYW1lIjoidXNlcjEifQ.VXYnWw0oXQLG6qX2_
dIV7qXEiuZILsENeFzPoLlzX0o"
}
```

4. Now, let's verify the token that we just obtained. Run the following command:

```
grpcurl -cacert server.crt -d '{"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJpYXQiOjE3MzIzNTUzMjIsInVzZXJuYW1lIjoidXNlcjEifQ.VXYnWw0oXQLG6qX2_
dIV7qXEiuZILsENeFzPoLlzX0o"}' localhost:8084 AuthService/
ValidateToken
```

You should get the following message:

```
{
  "username": "user1"
}
```

This message includes the valid username claim that we provided in the `GetToken` request. If you try to update the value of the token and run the previous command again, you should get the following error:

```
ERROR:  
Code: Unauthenticated  
Message: invalid token
```

With that, we have confirmed that our authentication and token validation logic works correctly. How could we use this in our services?

The most useful way we could use our `ValidateToken` endpoint would be to protect our endpoints, such as `MetadataService/PutMetadata`, against unauthorized access. For this, we would need to make the following changes:

- Our `PutMetadataRequest` structure would need to include an additional token field
- On each `PutMetadata` call, we would need to call the `AuthService/ValidateToken` endpoint to validate the access token before we update the data in our repositories

To save space, we won't cover how to implement this logic here; you should be able to achieve this by following the same steps that we completed previously to establish communication between our existing services. However, we will outline some best practices for establishing JWT-based authentication and access control:

- **Set a token expiration time:** When issuing JWTs, it is useful to set the token expiration time (the `exp` JWT claim field) to avoid situations where users use old authorization records. By having an expiration time set in each token payload, you can verify it against authorization requests. For example, when a user authenticates as a system administrator, you can set a short token expiration time (for example, a few hours) to avoid situations where a former administrator can still perform critical actions in the system. Note that there is no built-in mechanism for invalidating JWT tokens, so expiration time is the simplest mechanism for controlling the lifetime of issued tokens.
- **Include the token issuance time:** Additional metadata, such as the token issuance time (the `iat` JWT claim field), can be useful in many practical situations. For example, if you identify a security breach that happened at a certain point in time, you can invalidate all access tokens that were issued before that moment by using the token issuance time metadata.

- **Prefer standard JWT claim fields to custom ones:** When including metadata in the JWT payload, make sure that there is no standard field for the same purpose. You can find a list of standard JWT claim fields at https://en.wikipedia.org/wiki/JSON_Web_Token#Standard_fields.

The JWT website (<https://jwt.io/>) contains some additional tips on using JWTs, as well as an online tool for encoding and decoding JWTs, that you can use to debug your service communication.

Now that we have covered the practical aspects of authentication and authorization using JWT tokens, let's proceed to the next section of this book, where we'll cover the basics of application security analysis regarding Go microservices.

Security analysis of Go services

In this section, we are going to briefly review another aspect of security that is often called **application security**. This area of security primarily focuses on various issues regarding application code and logic, including the following:

- **Code injection:** Executing arbitrary commands (for example, by running the `os.Exec` function with arguments provided by the user).
- **Hardcoded secrets:** Using hardcoded secrets that might be compromised when application logic is executed (for example, in service logs or API responses).
- **Weak cryptographic practices:** Some cryptographic libraries, such as `crypto/md5` or `crypto/sha1`, are considered to offer weak protection mechanisms against various types of inputs. It's suggested that they're replaced with more secure counterparts (for example, `crypto/sha256` or `crypto/sha512`).
- **SQL injection:** Running SQL commands without proper protection against user-provided arguments might expose your system to additional read or write operations that are executed unexpectedly.
- **Logging sensitive data:** Your services might be inadvertently logging some sensitive data, such as user emails.

There are some existing tools for automating the process of analyzing common application security issues. Here, we are going to review the popular `gosec` tool, which is often used for application security analysis of Go microservices.

Using gosec to automate security analysis

The gosec tool can detect Go application security issues by performing static code analysis. The tool can detect dozens of issues, as defined in the documentation: <https://github.com/securego/gosec?tab=readme-ov-file#available-rules>.

To install the gosec tool, run the following command:

```
go install github.com/securego/gosec/v2/cmd/gosec@latest
```

Once the tool has been installed, you can run it inside your source code directory:

```
gosec ./...
```

Let's run the tool inside our `src` directory. The output will include a list of possible security issues, including the following:

```
[src/movie/cmd/main.go:58] - G402 (CWE-295): TLS MinVersion too low.  
(Confidence: HIGH, Severity: HIGH)  
    57:      }  
    > 58:      creds := credentials.NewTLS(&tls.Config{Certificates: []tls.  
Certificate{cert}})  
    59:      metadataGateway := metadatagateway.New(registry, creds)  
  
[src/rating/internal/ingester/kafka/ingester.go:45] - G104 (CWE-703):  
Errors unhandled. (Confidence: HIGH, Severity: LOW)  
    44:          close(ch)  
    > 45:          i.consumer.Close()  
    46:          return
```

This tool helped us to identify some potential security issues, such as the minimal TLS version not being specified and the `consumer.Close` function call not being handled. At the end of the report, the tool also reports a summary table of these issues:

Summary:

```
Gosec  : dev  
Files   : 47  
Lines   : 3643  
Nosec   : 0  
Issues  : 12
```

The gosec tool offers some useful options that can be passed to the command with the `-` prefix.

Here are some of the most useful ones:

- **tests**: Runs the gosec tool against the tests defined in the code
- **exclude-dir**: Excludes specific directories from the report
- **exclude-generated**: Excludes the analysis of generated code

Additionally, the gosec tool supports AI-based suggestions via the following arguments:

```
gosec -ai-api-provider="gemini" -ai-api-key="your_key" ./...
```

Each gosec output also includes two fields that can be used to identify security risks or issues:

- **Confidence**: How confident is the tool that the detected issue is valid?
- **Severity**: How big is the security risk associated with the issue?

You can run the gosec tool by specifying what level of confidence and severity to use, as well as sort the final list so that you can see the high-severity issues first.

We are not going to cover additional features of the gosec tool in this chapter. Instead, we'll outline the high-level suggestions for using the tool for security analysis of Go microservices:

- Run the gosec tool periodically (for example, once a week) to ensure you are aware of each of your possible security issues
- Always update the gosec tool to the latest version to keep the list of its available rules as wide as possible
- Try including all types of code (including the generated code) in the security report to make sure you have full coverage of possible security issues

You can find more information on the gosec tool and its features on its GitHub page:
<https://github.com/securego/gosec>

Now that we have covered the key basic areas of microservice security, let's proceed to the final section, where we'll cover the best practices regarding Go microservice development.

Security best practices

In this section, we are going to review a list of best practices that are going to help you establish service management, automate how common service vulnerabilities are scanned, and perform threat modeling to identify the weaknesses in your microservice applications.

Secure secret management

One of the most common problems in software security is storing **secrets** – parts of application configuration containing private information, such as passwords or encryption keys. In micro-service environments, the number of secrets can often be much bigger than in monolithic applications: each service might have its own encryption keys and security certificates. Therefore, the problem of storing secret data securely is even more critical for microservice applications.

Unlike most parts of service configuration, such as Docker and Kubernetes configuration files, secrets can't be stored in regular code repositories, such as Git. The reasons for this are as follows:

- **Access control requirements:** Secrets should generally have restrictive permissions for viewing and editing them, especially in large organizations
- **Data encryption:** Secret data must be encrypted for additional protection against unauthorized access
- **Data rotation:** Secret data should be rotated periodically (for example, passwords should be changed every few months) to reduce the negative impact in case potential data exposure occurs

So, what are the options for storing secrets in microservice environments?

The first and easiest option is to store service secrets as environment variables directly on servers or containers where your services are being executed. Here is an example of how you could do this on a Unix-compatible operating system:

```
MYSQL_USERNAME=<username> MYSQL_PASSWORD=<password> ./main
```

In our example, when we run our service binary, we pass MySQL credentials as environment variables to it. By doing this, our application can access these variables internally:

```
username := os.Getenv("MYSQL_USERNAME")
password := os.Getenv("MYSQL_PASSWORD")
db, err := sql.Open(username + ":" + password + "@/movieexample")
```

An environment-variable-based solution is simple to use, but it has some disadvantages:

- **Possible accidental exposure and lack of encryption:** Without extra encryption, our secret data would be easy to obtain if an attacker gains access to our system (for example, they access the Docker container running the service)
- **Secret rotation difficulty:** On secret rotation, we would need to distribute our secrets to all our service instances, potentially increasing security risks and often requiring the service instances to be restarted for changes to take effect

A better alternative to environment variables is **centralized secret management** tools. These address the aforementioned problems – that is, lack of encryption and secret rotation. Such systems offer additional protection mechanisms, including the following:

- **Access control:** You can specify who has access to specific secret data
- **Audit of changes:** All operations containing secrets can be logged for further investigation

Some common examples of existing centralized secret management tools include HashiCorp Vault, AWS Secrets Manager, and Google Secret Manager.

To access your secrets via code, you generally need to initialize an API client that you can use to access your secrets data. Here is an example of accessing secrets stored in HashiCorp Vault:

```
client, err := api.NewClient(&api.Config{
    Address: vaultAddr,
})
if err != nil {
    log.Fatalf("Failed to initialize Vault client: %v", err)
}
client.SetToken(vaultToken)
secretPath :=
// Read the secret
secret, err := client.Logical().Read(secretPath)
```

As with most cloud providers, there might be usage limits and various pricing tiers for using such services, so you might need to check each provider's usage and pricing information to find the best service for your use case.

Use automated vulnerability scanning

In addition to static analysis of your Go service code, there is dedicated tooling for finding security vulnerabilities inside your service dependencies, such as the Go libraries your services import. Among such tools is **govulncheck**, a tool created by the Go team that can analyze the code imports inside your Go packages and modules and compare them against the database of common Go vulnerabilities (<https://vuln.go.dev/>).

To install govulncheck, run the following go install command:

```
go install golang.org/x/vuln/cmd/govulncheck@latest
```

Once the tool has been installed, you can execute it inside your source code directory:

```
govulncheck ./...
```

The example output will include the following information:

```
Found vulnerability in github.com/some/lib: v1.2.3
Vulnerability: CVE-2023-XXXXXX
Fix: Upgrade to v1.2.4
```

Conveniently, the `govulncheck` tool suggests how to fix each vulnerability so that you can address each of the issues shown in its output.

To minimize the security risks for your Go services, it is suggested that you run this tool periodically to detect issues as early as possible. Additionally, you need to perform careful checks of your service dependencies (for example, avoid using unpopular or rarely maintained open source libraries that have lots of open issues) and perform frequent dependency updates to ensure your dependencies include the latest security patches.

Perform periodic threat modeling exercises

Threat modeling is the process of identifying the potential security risks of your services by analyzing how your services would behave in various hypothetical scenarios. Assume you have a service that stores sensitive user information, such as user passport data. What happens if some unauthorized person accesses your network? Do you have any security mechanisms that would still protect user data against unauthorized access? Can a random employee from your company copy user data to some external device?

Various possible questions might be asked to assess your possible security risks and find the weak points in your system architecture. To simplify and standardize the threat modeling process, there is a standard database of possible security issues that you can use called **Common Weakness Enumeration (CWE)**, available at <https://cwe.mitre.org/>. This database contains various security risks not only for software but also for hardware applications. You can find the database of software security weaknesses at <https://cwe.mitre.org/data/definitions/699.html>. Each item includes a unique code. Some of the common security weaknesses are as follows:

- **CWE-20:** Improper Input Validation
- **CWE-89:** Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

- **CWE-78:** Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- **CWE-798:** Use of Hard-coded Credentials
- **CWE-311:** Missing Encryption of Sensitive Data
- **CWE-434:** Unrestricted Upload of File with Dangerous Type
- **CWE-615:** Inclusion of Sensitive Information in Source Code Comments

There are hundreds of possible security weaknesses in software, and the list is updated and extended continuously. The general suggestion is to perform periodic threat modeling exercises by going against the list of all software CWE items and analyzing whether each item is a potential security threat. Additionally, it is always helpful to have high-level diagrams of your microservices, illustrating how all types of requests are handled (for example, what services, databases, Kafka topics, and other system components are involved in processing each request).

Now that we have covered the key best practices of software security, let's move on to the final section of this chapter, which covers the key regulatory compliance aspects of microservice development.

Compliance basics

In the last section of this chapter, we are going to briefly review the topic of regulatory compliance of software. Often overlooked, this aspect of software development is becoming more and more important as various compliance regulations become mandatory in many countries. The reason we have included this section in this book on microservice development is that regulatory compliance may significantly affect what data you can store and process with your services, as well as your overall system architecture. Knowing the relevant compliance regulations for your software may significantly help you choose the right tools and technologies, as well as establish the right processes early in development, potentially saving you lots of time and money in the future.

Let's review the most common compliance regulations for software development:

- **General Data Protection Regulation (GDPR):** This regulation was introduced by the European Union and mandates specific requirements for storing and handling the data of European Union residents. Among the basic requirements are explicit permissions for storing user data, the ability for users to access or delete their personal data, mandatory notifications in the case of data breaches, and cross-border data transfer restrictions. Similar to many other types of compliance regulations, organizations that don't comply with GDPR are subject to high fines that can go as high as millions or even billions of dollars.

- **Sarbanes-Oxley Act (SOX):** SOX requires publicly traded companies based in the United States to establish security and audit mechanisms that aim to ensure the accuracy and proper retention of financial data, the efficiency of internal security processes and data auditing activities, and many more. Some key SOX compliance requirements may significantly affect the ways you store and handle your data – for example, financial and audit documents must be stored for at least 7 years.
- **California Consumer Privacy Act (CCPA):** This regulation is similar to GDPR and applies to California state users.
- **Payment Card Industry Data Security Standard (PCI DSS):** This standard establishes rules for storing and handling payment information, such as credit card numbers. Among the key requirements are data encryption of cardholder data, data access restrictions, data access monitoring, and periodic software vulnerability scans.
- **Health Insurance Portability and Accountability Act (HIPAA):** This regulation targets health data (for example, medical patient records) and mandates strict rules for storing and transferring the data.

Various compliance regulations match different geographies and mandate different sets of requirements. The general framework for ensuring your software is compliant with the necessary regulations is as follows:

1. **Identify the relevant compliance regulations for your regions:** Regulations might differ based on the regions you are operating in, as well as where your customers are from.
2. **Identify the relevant compliance regulations for your data:** Most compliance regulations target personal, financial, and health data.
3. **Minimize the necessary data that's stored:** To reduce security and compliance risks, try to limit the amount of personal data to the bare minimum.
4. **Set correct data retention policies:** Establish explicit data retention policies for your data. Limit the storage of personal data to the maximum necessary period and ensure that financial and audit data is stored for the minimal necessary period if required.
5. **Ensure data privacy and protection:** Establish storage for secret and personal data by using dedicated tooling such as HashiCorp Vault.

Doing this early in the design and development process will help you comply with all relevant regulations, as well as establish the right processes for storing, transferring, and accessing your data according to industry standards and best practices.

Summary

In this chapter, we covered two important topics regarding microservice development: security and regulatory compliance. Both areas are especially important in microservice environments where services may transfer and access various types of data independently, increasing the scope of potential security risks (generally, the more complex the system is, the more security risks there are). In the first part of this chapter, you learned how to establish secure communication between your microservices with TLS, perform JWT-based authentication and authorization, analyze your Go microservice code to identify security issues, and scan service dependencies for known software vulnerabilities. In the second part of this chapter, we briefly covered common compliance regulations that might be relevant to microservice development and outlined a simple framework for assessing and addressing compliance requirements.

In the next chapter, we are going to cover another practically important topic: software reliability.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Security principles: https://infosec.mozilla.org/fundamentals/security_principles.html
- What is mutual TLS (mTLS)?: <https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/>
- JSON Web Token (JWT) Signing Algorithms Overview: <https://auth0.com/blog/json-web-token-signing-algorithms-overview/>
- gosec: <https://github.com/securego/gosec>
- GDPR: <https://gdpr-info.eu/>
- Amazon Cognito Documentation: <https://docs.aws.amazon.com/cognito/>
- Go Cookbook: Security: <https://go-cookbook.com/snippets/security>

Part 3

Maintenance

This part covers the topics related to running and operating your existing Go microservices in a production environment. The topics include service reliability, observability, alerting, and performance monitoring. You will learn how to handle different types of service reliability issues, how to collect and analyze service performance data, and how to set up automated service alerting. The part includes lots of best practices and examples that will help apply the newly gained knowledge to your microservices.

This part of the book includes the following chapters:

- *Chapter 11, Reliability Overview*
- *Chapter 12, Collecting Service Telemetry Data*
- *Chapter 13, Setting Up Service Alerting*
- *Chapter 14, Performance Monitoring*

11

Reliability Overview

We have taken a long journey through all previous chapters of this book and completed the part of the book dedicated to microservice development basics. So far, you have learned how to bootstrap microservices, write tests, set up service discovery, use synchronous and asynchronous communication between your microservices, and serialize the data between them using different formats, as well as how to deploy the services and verify that their APIs work.

This chapter begins the third part of the book, dedicated to more advanced concepts of microservice development, including reliability, observability, maintainability, alerting, and monitoring. In this chapter, we will cover some practical aspects of microservice development that are important for ensuring your services can operate well under many conditions, including failure scenarios, changes in network traffic, and unexpected service shutdowns.

In this chapter, we will cover various techniques and processes that can help you increase the reliability of your services. We will cover the following topics:

- Reliability basics
- Achieving reliability through automation
- Achieving reliability through development processes and culture

Let's proceed to the first section of the chapter, which will help you to understand service reliability concepts better.

Technical requirements

To complete this chapter, you need Go 1.18 or above.

You can find the GitHub code for this chapter here:

<https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter11>

Reliability basics

While implementing new applications, services, or features, engineers often focus first on meeting various system requirements, such as implementing specific application features. The initial result of such work is usually some working code that correctly performs its job, such as handling some data processing task or serving network requests as an API endpoint. We can say that such code initially performs well in isolation – the implemented code produces expected outputs for the inputs we provide.

Things usually get more complex when we add more components to the system. Let's take our movie service from *Chapter 2 Scaffolding a Go Microservice* and assume that its API gets used by some external service that has millions of users. Our service can be implemented perfectly fine and produce the right results for various test inputs. Still, once we get requests from an external service, we can notice various issues. One of them is called **denial of service (DoS)** – an external service can overload our service by asking to process too many requests, to the extent that our service stops serving new requests. The outcome of such an issue can vary from minor system performance degradation to service crashes due to reaching CPU, file, or memory limits.

DoS is just one example of things that can go wrong in a microservice environment. Assume that you performed a fix that limits the number of incoming requests to your service, but the fix broke the services calling your API because they did not expect a sudden denial of serving their requests. An alternative scenario is a change in a service API that introduces a **backward-incompatible change**. This change is incompatible with one or multiple previously released versions of callers of your service API. As a result, services calling your API could experience various negative effects, up to the point that they would be unable to process any requests.

Let's define the quality of a service that can be resilient in the face of unexpected failures as **reliability** – the quality of operating expectedly and having explicitly defined limitations. The last clause in our definition of reliability makes a big difference to its meaning – it's not enough to perform a certain function well. It is equally important to be explicit about the service's limitations and what happens when these limitations are breached.

In our movie service example, we would need to be explicit about multiple things, such as the following:

- **System throughput:** How many requests the service can process (for example, maximal requests per second)
- **Congestion policy:** How we would handle scenarios when our service is overloaded

For example, if our service can't process more than 100 simultaneous requests per service instance, we could explicitly state this in the documentation for our API and reject all extra incoming requests by returning a special error code, such as `HTTP 429 Too Many Requests`. Such an indication of system limits and explicit communication of congestion issues would be a great step toward improving overall system reliability by making its behavior more deterministic and, hence, reliable.

In general, achieving a high degree of reliability is a continuous process and requires constant improvements in the following three categories:

- **Prevention:** An ability to prevent possible issues whenever possible
- **Detection:** An ability to detect possible issues as early as possible
- **Mitigation:** An ability to mitigate any issues as early as possible

Prevention, detection, and mitigation improvements can be made by performing two types of actions:

- Automating service responses to various types of failures
- Changing and improving service development processes

We will divide the rest of the chapter into two sections describing these two types of actions. Let's proceed to the first of these, covering the automation-related reliability work.

Achieving reliability through automation

In this section, we will talk about various automation techniques that can help you improve the reliability of your services.

First, let's get back to communication error handling, which we briefly covered earlier in *Chapter 5 Synchronous Communication*. Having the right communication error-handling logic in place is the first step toward achieving higher reliability of your services, so we will focus on multiple aspects of error handling that are equally important in microservice development.

Communication error handling

As we discussed in *Chapter 5 Synchronous Communication* of this book, when two components – such as a client and a server – communicate with each other, there are three possible resulting scenarios:

- **Successful response:** The server receives and successfully processes a request
- **Client error:** An error occurs, and it is not caused by the server (for example, the client sends an invalid request)
- **Server error:** An error occurs, and it is caused by the server (for example, due to an application crash or an unexpected error on the server side)

From the perspective of a client, there are two different classes of errors:

- **Retriable errors:** A client may retry the original request (for example, when a server is temporarily unavailable)
- **Non-retriable errors:** A client should not retry the request (for example, when the request itself is incorrect due to failing validation)

Differentiating between retriable and non-retriable errors is the responsibility of the client. However, it is a good practice to indicate this explicitly whenever possible. For example, a server can return specific codes indicating the types of errors (such as HTTP 404 Not Found) so that a client can recognize retriable errors and perform retries. Differentiation between client and server errors also helps to ensure that requests are not retried for non-retriable errors. It is important from the server's perspective because handling duplicate, invalid requests increases its load.

Let's illustrate how to handle retriable communication errors by implementing client request retries. Setting up automated responses to potential issues, such as communication errors, helps to make the system more resilient to transient failures, resulting in a better experience for all components in the system.

Implementing request retries

Let's illustrate how to implement request retries in microservice code. For this, let's review the metadata gRPC gateway code we implemented earlier in *Chapter 5 Synchronous Communication*. The Get function includes the actual call to the metadata service:

```
    resp, err := client.GetMetadata(ctx, &gen.GetMetadataRequest{MovieId:  
id})  
    if err != nil {  
        return nil, err  
    }
```

Let's now look at the implementation of the `GetMetadata` endpoint in the metadata service gRPC handler. The `GetMetadata` function includes the following code:

```
func (h *Handler) GetMetadata(ctx context.Context, req *gen.GetMetadataRequest) (*gen.GetMetadataResponse, error) {
    if req == nil || req.MovieId == "" {
        return nil, status.Errorf(codes.InvalidArgument, "nil req or empty id")
    }
    m, err := h.ctrl.Get(ctx, req.MovieId)
    if err != nil && errors.Is(err, metadata.ErrNotFound) {
        return nil, status.Errorf(codes.NotFound, err.Error())
    } else if err != nil {
        return nil, status.Errorf(codes.Internal, err.Error())
    }
    return &gen.GetMetadataResponse{Metadata: model.MetadataToProto(m)},
    nil
}
```

As we can see, the implementation of the `GetMetadata` endpoint includes three error cases, each having its own gRPC error code:

- `InvalidArgument`: The incoming request fails the validation
- `NotFound`: The record with the provided identifier is not found
- `Internal`: Internal server error

The `InvalidArgument` and `NotFound` errors are non-retriable – there is no point in retrying requests failing validation or trying to retrieve records that are not found. `Internal` errors may indicate a wide range of issues, such as bugs in the service code, so we can't certainly state that you should perform retries on them.

There are, however, some other types of gRPC error codes that indicate potentially retriable errors. Let's list some of them:

- `DeadlineExceeded`: This indicates a problem with processing a request within the configured interval of time.
- `ResourceExhausted`: The service processing the request is exhausted. This can indicate a problem with a lack of available resources (for example, the CPU, memory, or disk reaching its limits) or the client reaching its quota for accessing the service (for example, when a service does not allow more than a certain number of parallel requests).
- `Unavailable`: The service is currently unavailable.

Let's first implement a simple retry logic inside the metadata gRPC gateway by replacing the Get function with the following code:

```
// Get returns movie metadata by a movie id.
func (g *Gateway) Get(ctx context.Context, id string) (*model.Metadata,
error) {
    conn, err := grpcutil.ServiceConnection(ctx, "metadata", g.registry)
    if err != nil {
        return nil, err
    }
    defer conn.Close()
    client := gen.NewMetadataServiceClient(conn)
    var resp *model.Metadata
    const maxRetries = 5
    for i := 0; i < maxRetries; i++ {
        resp, err = client.GetMetadata(ctx, &gen.
GetMetadataRequest{MovieId: id})
        if err != nil {
            if shouldRetry(err) {
                continue
            }
            return nil, err
        }
        return model.MetadataFromProto(resp.Metadata), nil
    }
    return nil, err
}
```

Then, let's add a function that should help us to check whether a communication error is retriable:

```
func shouldRetry(err error) bool {
    e, ok := status.FromError(err)
    if !ok {
        return false
    }
    return e.Code() == codes.DeadlineExceeded || e.Code() == codes.
ResourceExhausted || e.Code() == codes.Unavailable
}
```

Note

Always keep in mind that request retries put more load on the underlying systems. The negative impact of request retries can be multiplied in the case of **stacked retries** – scenarios when multiple layers of logic perform retries independently. Carefully review your retry logic to make sure it is implemented correctly and does not produce unexpected load on other services and components.

We also need to import two extra packages for checking for specific gRPC error codes – `google.golang.org/grpc/codes` for accessing a list of error codes and `google.golang.org/grpc/status` for checking whether the communication error is a valid gRPC error.

Now, our metadata gRPC gateway can perform up to five retries of requests to the metadata service. The retry logic that we just added should help us minimize the impact of occasional errors, such as temporary server unavailability (for example, during an unexpected outage or temporary network issues). However, it introduces some additional challenges:

- **Extra requests to the server:** For every call to the `Get` function, the metadata service gRPC gateway now performs up to five calls instead of one for retriable errors
- **Request bursts:** The metadata gRPC gateway performs immediate retries on errors, which will generate bursts of requests to the server

The latter scenario may be especially challenging to the server due to uneven load distribution. Imagine that you are doing some work and getting some phone calls with extra tasks. If you responded to such calls and said that you were busy, you wouldn't want to get called again immediately and asked to perform the same tasks again – instead, you would want the caller to call back after some time. Similarly, immediate retries would be suboptimal to servers experiencing congestion issues, so we would need to perform additional modifications to our retry logic to introduce extra delays between the retries so that our server would not get overloaded with immediate retries.

The technique of adding extra delays between client request retries is called **backoff**. Different types of backoff are implemented by using different delay intervals between the retry requests:

- **Constant backoff:** Each retry is performed after a constant delay
- **Exponential backoff:** Each retry is performed after a delay that is exponentially higher than the previous one

An example of exponential backoff would be a sequence of calls where the first retry would be done after a 100 ms delay, the second one would take 400 ms, and the third retry delay would be 900 ms. Exponential backoff is usually a better solution than constant, because it performs the next retry much slower than the previous ones, allowing the server to recover in case of overloading. A popular Go library at <https://github.com/cenkalti/backoff> provides an implementation of exponential and other types of backoff algorithms.

Backoff delay can also be modified by introducing small, random changes to its duration. For example, the retry delay value on each step could be increased or decreased by up to 10% to better spread the load on the server. This optimization is called **jittering**. To illustrate the usefulness of jittering, assume multiple clients start calling the server simultaneously. If retries are performed with the same delays for each client, they will keep calling the server simultaneously, generating bursts of server requests. Adding pseudo-random offsets to retry delay intervals helps to distribute the load on a server more evenly, preventing possible traffic bursts from request retries.

Deadlines and timeouts

Let's now talk about another class of communication issues related to time. When a client performs a request to a server, multiple possible failures may result in either the client or the server not receiving enough data to consider the request as successfully processed. Possible failure scenarios include the following:

- The client request does not reach the server due to network issues
- The server gets overloaded and takes longer to respond to the client
- The server processes the request, but the response does not reach the client due to network issues

These failures can result in longer waiting times for a client. Imagine you are sending a letter to your relative and not getting a response back. Without additional information, you would continue waiting without knowing whether the letter got lost at any step or whether the relative simply hasn't responded.

For synchronous requests, there is a way to improve the client experience by setting a **request timeout** – an interval after which the request is considered as failed in the case of not receiving a response. Setting request timeouts is a good practice due to multiple reasons:

- **Elimination of unexpected waits:** If a request takes an unexpectedly long time, the client can stop it earlier and perform an optional retry.

- **Ability to estimate maximum request processing time:** When requests are performed with explicit timeouts, it is easier to calculate how long it can take until the operation returns a response or an error to the caller.
- **Ability to set longer timeouts for long-running operations:** Libraries used for performing network calls often set default request timeouts (for example, 30 seconds). Sometimes the clients want to set a higher value, knowing that the request may take longer to complete (for example, when uploading a large file to a server). Explicitly setting a higher timeout helps to prevent the situation of a request getting canceled due to exceeding the default timeout.

In Go, timeouts are usually propagated via the `context.Context` object. As we mentioned in *Chapter 1 Introduction to Microservices*, each I/O operation, such as a network call, accepts the `context` object as an argument, and we can set a timeout by calling the `context.WithTimeout` function, as shown in the following code snippet:

```
func TimeoutExample(ctx context.Context, args Args) {  
    const timeout = 10 * time.Second  
    ctx, cancel := context.WithTimeout(ctx, timeout)  
    defer cancel()  
    resp, err := SomeOperation(ctx, args)  
}
```

In the preceding example, we set the timeout for the `SomeOperation` function to 10 seconds, so it should not take more than 10 seconds to complete the operation.

Setting a timeout is not the only way to limit request processing time. An alternative solution to this is setting a **deadline** – the maximal time until which the request should get processed and not to be considered as failed. Unlike a timeout, which is set using the `time.Duration` structure (for example, having the value of 10 seconds), a deadline indicates the exact instance of time (for example, January 1, 2074, 00:00:00). Here's an example of using a deadline for the same operation as in the previous code example:

```
deadline := time.Parse(time.RFC3339, "2074-01-01T00:00:00Z")  
ctx, cancel := context.WithDeadline(ctx, deadline)  
defer cancel()  
resp, err := SomeOperation(ctx, args)
```

Technically, both a timeout and a deadline help us achieve the same goal – to set a time limit for a target operation. You are free to use either format, depending on your preferences.

Fallbacks

Let's now talk about another client-server communication failure scenario – when a client tries to operate and doesn't get a successful response even after a set of retries. In such a case, there are three possible options for the client:

- Return an error to the caller, if any (this scenario is generally called **fail close**)
- Panic, in case an error is fatal to the system
- Ignore an error if it is not critical (for example, if we can't identify whether we should block/rate limit the request or not – this scenario is called **fail open**)
- Perform an alternative backup operation, if it is possible

The last option is called a **fallback** – an alternative logic that can get executed if some operation can't be performed as expected.

Let's take our rating service as an example. In our service, we implemented the `GetAggregatedRating` endpoint by reading all ratings for a provided record from the rating repository. Now, let's consider a failure scenario when we can't retrieve the ratings due to some problem, such as MySQL database unavailability. Without a fallback logic, we would not be able to process an incoming request and would need to return an error to our caller.

An example of a fallback would be to use a **cache** – we could store the previously retrieved ratings in the memory of a service (for example, inside a map structure) and return them on database read errors. The following code snippet provides an example of such a fallback logic:

```
ratings, err := c.repo.Get(ctx, recordID, recordType)
if err != nil && err == repository.ErrNotFound {
    return 0, ErrNotFound
} else if err != nil {
    log.Printf("Failed to get ratings for %v %v: %v", recordID,
    recordType, err)
    log.Printf("Fallback: returning locally cached ratings for %v %v",
    recordID, recordType)
    return c.getCachedRatings(recordID, recordType)
}
```

Using fallbacks is an example of **graceful degradation** – a practice of handling application failures in a way that allows an application to still perform its operations in a limited mode. In our example, the movie service would continue processing requests for getting movie details even if the recommendation feature is unavailable, providing a limited but working functionality to its users.

When designing new services or features, ask yourself which operations could be replaced with fallbacks in case of failures. Additionally, check which features and operations are absolutely necessary and which ones can be turned off in case of any failure, such as system overload or losing a part of the system due to an outage. Also, a good practice is to emit additional useful information related to failures, such as logs and metrics, and make it explicit in the code that the fallback is intentional, as in the preceding example.

Rate limiting

As we discussed at the beginning of this chapter, there may be a situation when a microservice is overloaded and can't handle incoming requests anymore. How can we prevent or mitigate such issues?

In software engineering, a general mechanism for handling such situations is called **backpressure**. The main idea of backpressure is to signal the sender of the data (such as the service making requests) to slow down and reduce the rate of requests until the receiver keeps up with the load.

A popular way to implement backpressure in microservice communication is to set a hard limit on the number of requests to be processed in parallel. Such a technique is called **rate limiting** and can be applied on multiple levels:

- **Client level:** A client limits the number of simultaneous outgoing requests
- **Server level:** A server limits the number of simultaneous incoming requests
- **Network/intermediate level:** The number of requests between a server and its clients is controlled by some logic or an intermediate component between them (for example, by a load balancer)

When a client or a server exceeds the configured number of requests, the result of a request would be an error that should include a special code or message, indicating that a request has been rate limited.

An example of a rate-limiting indication in the HTTP is a built-in status code, `429 Too Many Requests`. When a client receives a response with such a code, it should take this into account by either reducing the call rate or waiting for some time until the server can process requests again.

Client- and server-level rate limiting are often done by each service instance separately – each instance keeps track of the current number of outgoing or incoming requests. The downside of these models is the inability to configure the limits on a global-service level. If you configure each service client instance to send no more than 100 requests per second, you may still receive 100,000 simultaneous requests if there are 1,000 client instances. Such a high number of simultaneous requests could easily overload your service.

Network-level rate limiting can potentially solve this problem – if rate limiting is performed in a centralized way (for example, by a load balancer that handles requests between the services), the component performing rate limiting can keep track of the total number of requests across all service instances.

While network-level rate limiters provide more flexibility in configuring the settings, they often require additional centralized components (such as load balancers). Because of this, we are going to demonstrate how to use a simpler approach, based on the client level.

There is a popular package that implements rate limiting in Go, called `golang.org/x/time/rate`. The package implements the **token bucket** algorithm – a limiting algorithm that initializes a bucket of some configured maximal size b , decrements its value by 1 on each request, and refills it at a configured rate of r elements per second. For example, for $b = 100$ and $r = 50$, the token bucket algorithm creates a bucket of size 100 and refills it at a rate of 50 per second. At any moment in time, it doesn't allow more than 100 simultaneous requests (the maximal number is controlled by the current bucket size).

Here is an example of using a token-bucket-based rate limiter in Go:

```
package main

import (
    "fmt"

    "golang.org/x/time/rate"
)

func main() {
    limit := 3
    burst := 3
    limiter := rate.NewLimiter(rate.Limit(limit), burst)
    for i := 0; i < 100; i++ {
        if limiter.Allow() {
            fmt.Println("allowed")
        } else {
            fmt.Println("not allowed")
        }
    }
}
```

This code prints `allowed` three times and then keeps printing `not allowed` 97 times unless it takes more than one second to run.

Let's illustrate how to use such a rate limiter in combination with a gRPC API handler, which we implemented in *Chapter 5 Synchronous Communication*. The gRPC protocol allows us to define **interceptors** – operations that are performed on each request and can modify the gRPC server's response to it. To add a gRPC rate limiter to the movie service gRPC handler, perform the following steps:

1. Open the `movie/cmd/main.go` file and add the following code to its imports:

```
"github.com/grpc-ecosystem/go-grpc-middleware/ratelimit"
```

2. Replace the line with a `grpc.NewServer` call with the following code:

```
const limit = 100
const burst = 100
l := newLimiter(100, 100)
srv := grpc.NewServer(grpc.UnaryInterceptor(ratelimit.
    UnaryServerInterceptor(l)))
```

3. Then, add the following structure definition to the file:

```
type limiter struct {
    l *rate.Limiter
}

func newLimiter(limit int, burst int) *limiter {
    return &limiter{rate.NewLimiter(rate.Limit(limit), burst)}
}

func (l *limiter) Limit() bool {
    return l.l.Allow()
}
```

Our rate limiter is using a rate-limiting gRPC server interceptor from the `github.com/grpc-ecosystem/go-grpc-middleware/ratelimit` package. Its interface is slightly different from our limiter from `golang.org/x/time/rate`, so we added a structure that links them together. Now, our gRPC server allows up to 100 requests per second and returns an error with a `ResourceExhausted` special code in case the limit is exceeded. This allows us to make sure the service does not get overloaded with a sudden spike of a large number of requests – if somebody

requests 1 million movie details at once from it, we are not going to make 1 million calls to our metadata service and overload its database.

Keep in mind that rate limiting is a powerful technique; however, it needs to be used with caution because setting the limit too low would make your system unnecessarily restrictive for users by rejecting too many requests. To calculate fair rate-limiting settings for your services, you need to periodically perform benchmarking, understanding the maximum throughput of their logic.

Let's move on to the next topic of automation-based reliability techniques, describing how to gracefully terminate the execution of your services.

Graceful shutdown

In this section, we are going to talk about the graceful handling of service shutdown events. Service shutdowns can be triggered by multiple events:

- Manual interruption of execution (for example, when a user types *Ctrl + C/Cmd + C* in a terminal that runs the service process, and the process receives a `SIGINT` signal from the operating system)
- Termination of execution by the operating system (for example, by `SIGTERM` or `SIGKILL` signals)
- Panic in service code

Generally, a sudden termination of the execution of a service may result in the following negative consequences:

- **Dropped requests:** Incoming API requests may be dropped before they get fully processed, resulting in errors for the callers of the service.
- **Connection issues:** Service network connections may not be properly closed during a shutdown, resulting in multiple negative effects. For example, not closing a database connection may result in a situation called a **connection leak**, when the database keeps the connection allocated to the service instead of allowing it to be reused by another instance.

To prevent these issues, you need to ensure that your service shuts down gracefully by performing a set of operations that minimize any negative consequences for the service and its components. In performing a **graceful shutdown**, the service would run some extra logic before the termination, such as the following:

- Completing as many unfinished operations, such as unprocessed requests, as possible
- Closing all open network connections and yielding any shared resources, such as network sockets

Graceful shutdown logic for Go services is usually implemented in the following way:

1. The service subscribes to shutdown events by calling a `Notify` function of an `os/signal` package.
2. When a service receives a `SIGINT` or `SIGTERM` event from the operating system, indicating that the service is about to be terminated, it performs a set of required operations for closing all open connections and completing all pending tasks.
3. Once all operations are completed, the service finishes the execution.

Here is a code example that you can add to the `main` function of any Go service, such as the ones that we implemented in *Chapter 2 Scaffolding a Go Microservice*:

```
sigChan := make(chan os.Signal, 1)
signal.Notify(sigChan, os.Interrupt, syscall.SIGTERM)
var wg sync.WaitGroup
wg.Add(1)
go func() {
    defer wg.Done()
    s := <-sigChan
    log.Printf("Received signal %v, attempting graceful shutdown", s)
    // Graceful shutdown logic.
}()
wg.Wait()
```

There is also a way to gracefully handle panics in Go code by using the built-in `recover` function. The following code snippet demonstrates how to handle a panic inside the `main` function and execute any custom logic, such as closing any open connections:

```
func main() {
    defer func() {
        if err := recover(); err != nil {
            log.Printf("Panic occurred, attempting graceful shutdown")
            // Graceful shutdown logic.
        }
    }()
    panic("panic example")
}
```

In our code, we check whether there is a service panic by calling the `recover` function and checking whether it returns a non-nil error. In case of a panic, we can perform any additional operations, such as saving any unsaved data or terminating any open connections.

To gracefully terminate the execution of a Go gRPC server, you need to call the `GracefulStop` function instead of `Stop`. Unlike the `Stop` function, `GracefulStop` would wait until all requests are processed, helping to reduce the negative impact of the shutdown on the clients.

If you have some long-running components, such as Kafka consumers or any background goroutines executing long-running tasks, you can communicate the service termination signal using the built-in `context.Context` structure. The `context.Context` structure provides a feature called **context cancellation** – an ability to notify different components about the cancellation of an execution by sending a specific event through the channel associated with the context.

Let's update our rating service code to illustrate how to implement context cancellation and a graceful shutdown of a gRPC server:

1. Open the `main.go` file of the rating service and find the line that performs a call to the `context.Background()` function. Replace it with the following code:

```
ctx, cancel := context.WithCancel(context.Background())
```

Our code creates an instance of a context and the `cancel` function, which we will be calling on service shutdown to notify our components, such as the service registry, about upcoming termination.

2. Immediately before the call to the `srv.Serve` function, add the following code:

```
sigChan := make(chan os.Signal, 1)
signal.Notify(sigChan, os.Interrupt, syscall.SIGTERM)
var wg sync.WaitGroup
wg.Add(1)
go func() {
    defer wg.Done()
    s := <-sigChan
    cancel()
    log.Printf("Received signal %v, attempting graceful
shutdown", s)
    srv GracefulStop()
    log.Println("Gracefully stopped the gRPC server")
}()
```

In our code, we let the rating service listen for process interruption and termination signals and start the background goroutine, which keeps listening for the relevant notifications. Once it receives either signal, it calls the `cancel` function that we obtained in the previous

step. The result of calling this function would be a notification that would be sent to the components initialized with our context, such as the service registry.

3. Let's add the final touch by adding the following line to the end of our `main` function:

```
wg.Wait()
```

Let's now test the code that we just implemented. Run the rating service and then terminate it by pressing *Ctrl + C/Cmd + C* (depending on your OS). You should see the following messages:

```
2022/10/13 08:55:05 Received signal interrupt, attempting graceful shutdown  
2022/10/13 08:55:05 Gracefully stopped the gRPC server
```

Communication of termination and interruption events is a common practice in Go microservice development and is an elegant way of implementing graceful shutdown logic. When designing and implementing your services, think in advance about possible resources that need to be closed or deinitialized upon the service termination, such as any network clients and connections. A graceful shutdown logic can prevent the negative effects of sudden service termination. It can also reduce the number of possible errors in your services and improve your operating experience.

At this point, we have reviewed some automation techniques to improve the reliability of your services and reduce the symptoms of various failure scenarios. Now, we can proceed to the next section of the chapter, covering another aspect of reliability work related to development processes and culture. Improvements to your development processes are essential to achieving high reliability in the long term, and the section should be useful to you by providing some valuable tips and ideas that you can utilize in microservice development.

Achieving reliability through development processes and culture

In this section, we are going to describe some techniques for achieving higher service reliability based on changes in the development processes and culture. You will learn how to establish the processes for improving and reviewing your service reliability, how to learn from any service-related issues and incidents efficiently, and how to measure your service reliability. We will cover the processes and practices that are widely used across the industry, outlining the most important ideas from each one. This section is going to be more theoretical than the previous one; however, it should be equally useful.

First, we are going to provide an overview of the on-call process essential for setting up a mechanism for monitoring issues with your services.

On-call process

When your services start handling production traffic or serving user requests, one of your first reliability goals should be to detect any issues or incidents as early as possible. Efficient detection should be automatic – a program would always be much more efficient than a human in detecting most issues. Each automatic detection should notify one or more engineers about the incident so that they can perform work in order to mitigate it.

The process for establishing a mechanism for notifying engineers about service incidents is called **on-call**. This process helps ensure that at any moment in time, service incidents are acknowledged and addressed by the engineers responsible for the service.

The main ideas behind the on-call process are the following:

- Engineers can get grouped into **on-call rotations**. Each engineer participating in the on-call process repeatedly gets assigned a continuous *shift* (often taking one week), during which they take responsibility for periodically handling notifications regarding service-level incidents.
- On-call rotation can have an *escalation policy* – a process of escalating incidents in case they remain unresolved. First, an incident gets reported to the *primary* on-call engineer of the rotation. If the primary engineer is unavailable, the incident gets reported to the *secondary* engineer, and so on.
- There can be a *shadow* role, commonly assigned to new engineers. This role does not require any response to the incident, but it can be used for getting familiar with the on-call process and subscribing to real-time incident notifications.
- Each incident triggers one or multiple notifications, notifying the on-call engineers about the issue. Each notification must be acknowledged by the responsible on-call engineer unless the incident self-resolves (for example, if a service stops receiving too many requests and starts operating normally).
- You can also set up an escalation policy for a rotation – a mechanism for escalating the incident notifications if the responsible on-call engineers don't acknowledge them within the configured time. Usually, an escalation policy follows the reporting chain of the engineering hierarchy – if no engineer acknowledges the incident, the incident first triggers a notification to the closest engineering manager, then to the person the manager is reporting to, and so on until it reaches the highest level (this can even be a CTO at some companies).

Having an on-call process is common to most technology companies and teams, and the on-call process is pretty similar in most companies across the industry. Some popular solutions provide mechanisms for triggering various types of notifications, such as SMS, emails, and even phone calls. You can also configure on-call rotations and assign them to different services. One of the most popular solutions to on-call management is **PagerDuty** – a platform providing a set of tools for automating on-call operations, as well as integrations with hundreds of services, including Slack, Zoom, and many more. PagerDuty provides all the features we listed earlier, allowing engineers to configure on-call rotations for their services and notifying them about incidents in different ways. Additionally, it provides an API that can be used for both accessing the incident data and triggering new incidents from the code.

We are not going to dive into the details of PagerDuty features and integrations in this chapter – I suggest you check the official PagerDuty documentation on their website at <https://developer.pagerduty.com/docs>. I also suggest you read *Chapter 13 Setting Up Service Alerting* before establishing an on-call process for your services. It will help you to learn more about possible incident detection mechanisms and tools you can utilize in your projects.

Let's discuss the common challenges of establishing an on-call process in a microservice environment:

- **Rotation ownership:** Different services may be maintained by different teams, so there may be multiple on-call rotations inside a single company. A good practice is to have an explicit mapping between each production service and the associated on-call rotation so that it would be clear which rotation each incident should be reported to. In *Chapter 13 Setting Up Service Alerting*, we will cover the ownership aspect of this.
- **Cross-service issues:** Some issues, such as database or network failures, can span multiple services, so it becomes important to have some centralized team(s) that would be able to help with any issues crossing the boundaries of individual services.

Some companies may have thousands of microservices, so centralized incident response teams become crucial. For example, Uber has a dedicated team of engineers called *RingO* that is responsible for addressing any widespread incidents and coordinating the mitigation of issues that span multiple teams. Having such a team helps to dramatically reduce incident mitigation time.

To better understand what happens next after incidents are detected and acknowledged by the engineers, we are now going to move on to the next topic: incident management.

Incident management

Once incidents are detected and acknowledged by the engineers, there are two other types of work necessary for improving the service or system reliability – mitigation and prevention. Mitigation is required for resolving an open issue unless it gets resolved by itself or due to some external changes (for example, an external API getting fixed by the owning team). Prevention work is useful for ensuring the issue does not happen again. Without a proper prevention response to the incident, you may keep fixing the same issue over and over again, spending your time and affecting the experience of your system's users.

To make the incident mitigation process quick and efficient, especially in a large team where engineers may have different levels of understanding of the system, there should be enough documentation describing which actions to perform in case of an incident. Such documentation is called a **runbook** and should be prepared for as many types of detectable incidents as possible. Whenever an on-call engineer gets an incident notification, it should be clear from the runbook which steps to take to mitigate it.

A good runbook should be short and concise and provide clear, actionable steps that are easy to understand for any engineer. Let's take this example:

```
rating_service_fd_limit_reached:  
    mitigation: Restart the service
```

If the incident mitigation requires further investigation, include any useful links, such as links to the relevant application logs and dashboards. You should aim for the lowest possible incident mitigation time – also called **time to repair (TTR)** – to increase the availability of your service and improve its overall health.

Once the incident is mitigated, focus on prevention work to ensure you take all actions to eliminate its causes, as well as to improve detection and mitigation mechanisms, if needed. Multiple companies across the industry use the process of writing documents called **incident postmortems** to organize the learnings around incidents and make sure each incident involves enough work related to its future prevention. An incident postmortem generally consists of the following data:

- Incident title and summary
- Authors
- When and how the incident was detected and mitigated

- Incident context in the form of a text or a set of diagrams that can help to understand it
- Root cause
- Incident impact
- Incident timeline
- Lessons learned
- Action items

A great example of a postmortem document is provided in the famous Google *Site Reliability Engineering (SRE)* book, and you can get familiar with it at the following link: <https://sre.google/sre-book/example-postmortem/>

To get to the root cause of the incident, you can use a technique called **Five Whys**. The idea of this technique is to keep asking what caused the previously mentioned problem until the root cause is found. Let's take the following **root cause analysis (RCA)** as an example to understand the technique:

Incident: The rating service returns internal errors to its API callers

Root cause analysis:

1. The rating service started returning internal errors to its API callers due to the rating database's unavailability.
2. The rating database became unavailable because of an unexpectedly high request load.
3. The unexpectedly high request load to the rating service was caused by an application bug in the movie service.

In this example, we kept finding the underlying cause of each previous issue by using the Five Whys technique until we got to the root cause of the incident in just three steps. This technique is very powerful and easy to use, and it can help you get to the root cause of even complex issues quite quickly.

Make sure you include and track action items for your incidents. Capturing the incident details and identifying the causes isn't enough to make sure incidents are prevented. Prioritizing the action items helps ensure that the most critical ones get addressed as early as possible.

Now, let's move on to the next reliability process based on periodic testing of your possible service-failure scenarios.

Reliability drills

As many system administrators know, it is not enough to have backups of your data to guarantee its durability. You also need to ensure that you can restore the data from the backups in case of any failure. The same principle applies to any part of your service infrastructure – to know that your services are resilient to particular failures, you need to perform periodic exercises, called **drills**.

You can perform many possible types of drills. As in the example with the database backups, if you have any persistent data stored in a database, you can periodically test its ability to back up and restore the data, verifying that your services are tolerant to database availability issues. Another example would be network drills. You can simulate network issues, such as connectivity loss, by updating service routing configuration or any other network settings to check how your services behave in case of network unavailability.

There are multiple benefits of performing reliability drills:

- **Detect unexpected service failures:** By performing failure drills, you can detect some unexpected service errors and panics that don't happen in the regular mode. Such issues will present themselves in a controlled environment, where engineers are ready to stop the drill at any moment and address the detected errors as early as possible.
- **Detect unexpected service dependencies:** Reliability drills often uncover unexpected dependencies between the services, such as **transitive dependencies** (service A depends on service B, which depends on service C) or even **circular dependencies** (two services require each other in order to operate).
- **Be able to mitigate future incidents quicker:** By knowing how the services operate in case of a failure and how they resolve related issues, you invest in improving future incident mitigation.

Drills are often performed as **planned incidents** – incidents that get announced in advance and follow the regular incident management process, including the work on the postmortem document. The drill postmortem document should include the same items as a regular incident, with a focus on improving the mitigation and prevention experience. Additionally, engineers should focus on reviewing and updating the service runbooks, making sure that the incident mitigation instructions are accurate and up-to-date.

Disaster recovery plans

For achieving reliability, it is essential to think ahead regarding all possible failures and define the exact steps for mitigating them. For this, it is useful to write and maintain explicit **disaster recovery plans** – documents describing different types of possible disasters and large-scale outages, such as loss of a database server or failures of large groups of services.

Disaster recovery plans are broader in scope than on-call runbooks and focus on high-level problems instead of low-level failures. Consider a case when a large group of microservices starts crashing, and it becomes apparent that certain services depend on others in order to become operational. In such a case, you would need to know the exact order of services and any other components, such as databases, that need to be repaired first to unblock other failing parts of the system.

Handling the results of such cascading failures without explicit disaster recovery plans can take much more time, resulting in prolonged system outages. Therefore, having clear and up-to-date sequences of actions, as well as getting ready to address critical system failures, could dramatically improve system availability, as well as reduce the time necessary to repair complex outages.

A good disaster recovery plan should include the following information:

- Dependencies between microservices
- Critical infrastructural components used by key services, such as databases or message brokers
- Potential failure scenarios and risk assessments
- Team roles and responsibilities during disaster recovery
- A communication plan (including backup communication channels)
- Backup recovery procedures

Set and track the reliability objectives of your services

It is hardly possible to achieve reliability in your microservices without the constant tracking of their health and performance. For this, there are several industry-wide practices that help to establish the reliability measures of various kinds of applications.

The basis of tracking service reliability is defining key metrics called **service-level indicators**, or **SLIs**. SLIs are quantitative metrics that measure the performance or reliability of some part of your system. Let's take some examples of SLIs:

- *Service API error rate*: Percentage of API errors compared to the total number of API requests for a given service
- *Service API request latency*: Amount of time it takes to process incoming API requests for a given service
- *Kafka topic producer throughput*: Number of messages produced to a Kafka topic per a specified unit of time (for example, per second)

We will cover different types of performance indicators of your services in the following three chapters of this book. However, the most common types of SLIs include the following:

- **Error rate**: Percentage of failed requests out of the total
- **Latency**: Amount of time it takes to execute an operation
- **Throughput**: Number of operations being processed per unit of time
- **Resource utilization**: Percentage of the resource (for example, CPU or memory) consumed by the system

Once defined, SLIs can be used for defining and tracking **service-level objectives (SLOs)** – target values for SLIs, such as the following:

- *Service <1% API error rate SLO*: The objective of having the value of service error rate SLI lower than 1% at any given moment of time (or, usually, interval, such as second/minute)
- *Service <100ms p99 API request latency*: The objective of having 99th percentile of service API requests with latency lower than 100 milliseconds
- *Kafka topic >0 producer throughput*: The objective of having a non-zero value of Kafka topic producer throughput for some topic

Note



Latency and some other indicators are often grouped into percentiles for measuring. You can read more on using percentiles for tracking latency and other indicators in this article: <https://gatling.io/blog/latency-percentiles-for-load-testing-analysis>

SLOs can be seen as reliability goals for your services or the entire system. For example, you might set an objective p99 API error rate of no more than 5% for any of your services, and no more than 1% for some selected services that can be considered critical to the system (for example, a payment processing service).

There might be cases when you need to establish an agreement with your customers that certain SLOs will be met, with some potential penalty scenarios in case of SLO violation. Such agreements are called **service-level agreements**, or **SLAs**, and are common when providing service access to external customers. A perfect example of SLAs is on the Amazon Web Services SLA page: <https://aws.amazon.com/compute/sla/>. The page states the current SLAs, as well as defining compensations to platform users in case of SLA violations (for example, receiving 100% of service credits spent on API usage if monthly API availability is less than 95%).

You likely don't need to define SLAs unless there are explicit business obligations (usually explicitly stated in contracts with customers); however, using SLOs is a good and important practice for tracking and improving the reliability of your microservices by setting and following reliability goals.

At this point, we have discussed the most important service reliability techniques. There are many more interesting topics to cover that are related to service reliability – some of them, related to incident detection, we are going to cover in *Chapter 13 Setting Up Service Alerting* of the book. If you are interested in the topics described in the section, I strongly encourage you to read the Google *Site Reliability Engineering (SRE)* book, which provides a comprehensive guide to various reliability-related techniques. You can find the online version of the book by going to the following link: <https://sre.google/sre-book/table-of-contents> The practices that are described in the book are applicable to any microservice, so you can always use it as a reference while working on building any type of system.

Summary

In this chapter, we covered the topic of reliability, describing a set of techniques and practices that can help you make your microservices more resilient to various types of failures. You have learned some useful techniques for automating error responses of your services and reducing the negative impact of various types of issues, such as service overloading and unexpected service shutdowns.

In the final part of the chapter, we discussed various reliability techniques based on changes in engineering processes and culture, such as introducing the on-call and incident management processes, performing periodic reliability drills, defining disaster recovery plans, and tracking reliability objectives for your services. The knowledge that you've gained from reading this chapter should help you establish a solid foundation for writing reliable microservices.

In the next chapter, we are going to continue our journey into the reliability topic and focus on collecting service telemetry data such as logs, metrics, and traces. Service telemetry data is the primary instrument for setting up service incident detection, and we will illustrate how to work with each type of telemetry data in your microservice code.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Timeouts, retries, and backoff with jitter: <https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter>
- Token bucket rate-limiting algorithm: https://en.wikipedia.org/wiki/Token_bucket
- PagerDuty documentation: <https://developer.pagerduty.com/docs>
- Incident postmortems: <https://www.pagerduty.com/resources/learn/incident-postmortem/>
- Google Site Reliability Engineering (SRE) website: <https://sre.google/>
- Google Site Reliability Engineering (SRE) book: <https://sre.google/sre-book/table-of-contents>
- The Evolution of SRE at Google: <https://www.usenix.org/publications/loginonline/evolution-sre-google>

12

Collecting Service Telemetry Data

In the previous chapter, we explored the topic of service reliability and described various techniques for making your services more resilient to different types of errors. You learned that reliability-related work consists of making constant improvements in incident detection, mitigation, and prevention techniques.

In this chapter, we are going to take a closer look at various types of service performance data, which is essential for setting up service health monitoring and debugging and automating service incident detection. You will learn how to collect service logs, metrics, and traces and how to visualize and debug communication between your microservices using the distributed tracing technique.

We will cover the following topics:

- Telemetry overview
- Collecting service logs
- Collecting service metrics
- Collecting service traces

Let's start with an overview of all the techniques that we are going to describe in this chapter.

Technical requirements

To complete this chapter, you will need Go 1.18 or above. You will also need the following tools:

- **grpcurl**: <https://github.com/fullstorydev/grpcurl>
- **Jaeger**: <https://www.jaegertracing.io/>

You can find the GitHub code for this chapter here: <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter12>.

Telemetry overview

In the introduction to this chapter, we mentioned that there are different types of service performance data, all of which are essential for service health monitoring and troubleshooting. These types of data are called **telemetry data** and include the following:

- **Logs**: Messages recorded by your services that provide insights into the operations they perform or errors they encounter
- **Metrics**: Metrics are numeric representations of system state over time, such as the number of registered users, API request error rate, or percentage of free disk space
- **Traces**: Data that shows how your services perform various operations, such as API requests, which other services they call, which internal operations they perform, and how long these operations take
- **Performance profiles**: Snapshots of the state of system resources, such as CPU and memory, that can be helpful for analyzing their utilization (for example, active threads, memory contents, and CPU flame graphs)

Telemetry data is *immutable*: it captures events that have already happened to the service and provides the results of various measurements, such as service API response latency. When different types of telemetry data are combined, they become a powerful source of information about service behavior.



Note

There is an OpenTelemetry project that aims to standardize different types of telemetry data and make them interchangeable between different tools and services. You can get familiar with the project on its website: <https://opentelemetry.io>.

In this chapter, we are going to describe how to collect service telemetry data to monitor the health of services. There are two types of service health and performance monitoring:

- **White-box monitoring:** Monitoring services while having access to different types of internally produced data. For example, you can monitor a server's CPU utilization by viewing it in the system monitoring application.
- **Black-box monitoring:** Monitoring services using only externally available data and indicators. In this case, you don't know or have access to data related to its structure or internal behavior. For example, if a service has a publicly available health check API, an external system can monitor its health by calling that API without having access to internal service data.

Both types of monitoring are powered by collecting and continuously analyzing service performance data. In general, the more types of data you collect from your application, the more opportunities you get for extracting various types of information about its health and behavior. Let's list some ways you can use information about your service performance:

- **Trend analysis:** Detect any trends in your service performance data:
 - Is your service health getting better or worse over time?
 - How does your API success rate change?
 - How many new users are you getting compared to the previous day/week/month?
- **Semantic graph capturing:** Capture the data on how your services communicate with each other and with any other components, such as databases, external APIs, and message brokers
- **Anomaly detection:** Automatically detect anomalies in your service behavior, such as sudden drops in API requests
- **Event correlation:** Detect the relationship between various types of events, such as unsuccessful deployments and service panics

While observability opens many such opportunities, it comes with the following challenges:

- **Collecting large datasets:** Real-time performance data often takes lots of space to store, especially if you have lots of services or if your services produce lots of data.
- **Need for specific tooling:** To collect, process, and visualize different types of data, such as logs, metrics, and traces, you need some extra tools. These tools often come at a price.
- **Complex setup:** Observability tooling and infrastructure are often difficult to configure. To access all data coming from multiple services, you need to set up proper data collection, aggregation, data retention policies, and many more strategies.

We are going to describe how to work with each type of telemetry data that you can collect in your microservices. For each type of data, we will provide some usage examples and describe common ways of setting up the tooling for working with it. First, let's proceed to service log collection.

Collecting service logs

Logging is a technique that involves collecting real-time application performance data in the form of a time-ordered set of messages called a **log**. Here is an example of a service log:

```
2022/06/06 23:00:00 Service started  
2022/06/06 23:00:01 Connecting to the database  
2022/06/06 23:00:11 Unable to connect to the database: timeout error
```

Logs can help us understand what was happening in the application at a particular moment in time. As you can see in the preceding example, the service started at 23:00:00 and began connecting to the database a second later, finally logging a timeout error 10 seconds later.

Logs can provide lots of valuable insights about the components that emitted them, such as the following:

- **Order of operations:** Logs can help us understand the logical sequence of operations performed by a service by showing when each operation took place
- **Failed operations:** One of the most useful applications of logs is the ability to see a list of errors recorded by a service
- **Panics:** If a service experiences an unexpected shutdown due to panic, a log can provide the relevant information, helping troubleshoot the issue
- **Debugging information:** Developers can log various types of additional information, such as request parameters or headers, that can help when debugging various issues
- **Warnings:** Logs can indicate various system-level warnings, such as low disk space, that can be used as notification mechanisms for preventing various types of errors

We used logs in the services that we created in *Chapter 2 Scaffolding a Go Microservice* – our services have been logging some important status messages via the built-in log library. Here's an example:

```
log.Printf("Starting the metadata service on port %d", port)
```

The built-in log library provides functionality for logging arbitrary text messages and panics. The output of the preceding operation would be as follows:

```
2022/07/01 13:05:21 Starting the metadata service on port 8081
```

By default, the `log` library records all logs to the `stdout` stream associated with the current process. However, it is possible to set the output destination by calling the `SetOutput` function. This way, you can write your logs to files or send them over the network.

Two types of functions are provided by the `log` library that can be used if a service experiences an unexpected or non-recoverable error:

- `fatal`: Functions with this prefix immediately stop the execution of the process after logging the message.
- `panic`: After logging the message, they call the Go `panic` function, writing the output of the associated error. The following is an example output of calling a `panic` function:

```
2022/11/10 23:00:00 network unavailable
panic: network unavailable
```

While the built-in `log` library provides a simple way of logging arbitrary text messages, it lacks some useful functionality that makes it easier to collect and process service logs. Among the missing features is the ability to log events in popular serialization formats, such as JSON, which would simplify how message data is parsed. Another issue is that it lacks `Error` and `Errorf` functions, which could be used for explicitly logging errors. Since the built-in logging library only provides a `Print` function, it's unclear by default if the logged message indicates an error, a warning, or neither.

In Go 1.21, the Go team introduced an additional logging library called `log/slog` that provides some extra features, including support for **structured logging** – an ability to collect log messages in the form of serialized structures, such as JSON records. A distinct feature of such structures, compared to arbitrary text strings, is that they can contain additional metadata in the form of fields – key-value records. This allows the service to represent the message metadata as any supported type, such as a number, string, or serialized record.

The following snippet includes an example of a JSON-encoded log structure:

```
{"level":"info", "time":"2022-09-04T20:10:10+1:00", "message":"Service
started", "service":"metadata"}
```

As you may have noticed, in addition to using JSON as the output format, there are two additional features of the preceding log format:

- **Log level:** There is a field called `level` that specifies the type of a log message
- **Additional message fields:** Our example includes a field called `service`, which is used to indicate the service that emitted the message

The output format described earlier allows us to decode the log messages much more easily. It also helps us interpret their contents based on the log level and the additional message fields. Additional metadata can also be used for searching: for example, we can search for messages that have a particular service field value.

Now, let's focus on log-level metadata from the previous example. First, let's review some common log levels:

- **Info:** Informational messages that do not indicate any error. An example of such a message is a log record indicating that the service successfully connected to the database.
- **Error:** Messages indicating errors, such as network timeouts.
- **Warning:** Messages indicating some potential issues, such as too many open files.
- **Fatal:** Messages indicating critical or non-recoverable errors, such as insufficient memory, that make executing the service further impossible.
- **Debug:** Messages that provide some additional context to the developers that can help troubleshoot various issues or get some additional insights for application performance. Collecting Debug messages is usually disabled by default, as they often generate a large amount of data.

Log levels also help us interpret log messages. Consider the following unstructured message produced by the built-in log library, which does not include any level information:

```
2022/06/06 23:00:00 Connection terminated
```

Can you tell if it's a regular informational message indicating regular behavior (for example, the service intentionally terminated a connection after performing some work), a warning, or an error? If this is an error, is it critical or not? Without the log level providing additional context, it is difficult to interpret this message.

Another advantage of explicitly using log levels is the ability to enable or disable that ability to log specific types of levels. For example, logging Debug messages can be disabled under normal service conditions and enabled during troubleshooting. Messages of the Debug type often include much more information than regular ones, requiring more disk space and making it harder to navigate other types of logs. Different logging libraries let us enable or disable specific levels, such as Debug or even Info, leaving only logs indicating warnings, errors, fatal errors, and panics.

Let's review some popular Go logging libraries and focus on choosing one that we would use in our microservices.

Choosing a logging library

In this section, we will describe some existing Go logging libraries and review their features. This section should help you choose the logging library that you will use in your microservices. When evaluating library performance, we will be using the logging library benchmark data: <https://github.com/uber-go/zap#performance>.

A list of popular Go logging libraries includes the following:

- **Built-in Go log package (<https://pkg.go.dev/log>):**
 - Officially supported by the Go development team, included in the Go SDK
 - Does not support structured logging and does not have built-in support for log levels
- **Built-in Go log/slog package (<https://pkg.go.dev/log/slog>):**
 - Officially supported by the Go development team, included in the Go SDK starting from Go 1.21
 - Supports structural logging and log levels
- **zap (<https://github.com/uber-go/zap>):**
 - Fastest performance among all logging libraries reviewed
 - Feature-rich: Supports two types of loggers – a minimalistic high-performance logger and a slightly slower but more customizable one
- **zerolog (<https://github.com/rs/zerolog>):**
 - Fast performance: Just slightly slower than zap
 - Simple and elegant API
- **go-kit/log (<https://github.com/go-kit/log>):**
 - Part of a larger go-kit toolkit for microservice development
 - Slightly slower than zerolog and zap, but faster than the other logging libraries
- **apex/log (<https://github.com/apex/log>):**
 - Has built-in support for various log storages, such as Elasticsearch, Graylog, and AWS Kinesis

The preceding list provides some high-level details about some popular Go logging libraries to help you choose the right one for your services. All libraries, except the built-in log library, provide the features that we need, including structured logging and log levels. Now, the question is, how do we select the best one among them?

My personal opinion is that the `zap` library provides the most flexible and yet most performant solution to service logging problems. It allows us to use two separate loggers, called `Logger` and `SugaredLogger`. The `Logger` logger can be used in high-performance applications, while `SugaredLogger` can be used when you need some extra features; we will review these features in the next section.

Using logging features

Let's start practicing and demonstrate how to use some features of the `zap` logging library that we picked in the previous section. First, let's start with the basics and illustrate how to log a simple message that has the `Info` level and an additional metadata field called `serviceName`. The complete Go code for this example is as follows:

```
package main

import "go.uber.org/zap"

func main() {
    logger, _ := zap.NewProduction()
    logger.Info("Started the service", zap.String("serviceName",
"metadata"))
}
```

We initialize the `logger` variable by calling the `zap.NewProduction` function, which returns a production-configured logger. This logger omits debug messages, uses JSON as the output format, and includes stack traces in the logs. Then, we create a structured log message by including a `serviceName` field by using the `zap.String` function, which can be used to log string data.

The output of the preceding example is as follows:

```
{"level": "info", "ts": 1257894000, "caller": "sandbox1575103092/prog.
go:11", "msg": "Started the service", "serviceName": "metadata"}
```

The `zap` library offers support for other types of Go primitives, such as `int`, `long`, `bool`, and many more. Corresponding functions for creating log field names follow the same naming format, such as `Int`, `Long`, and `Bool`. Additionally, `zap` includes a set of functions for the other built-in Go types, such as `time.Duration`. The following code snippet shows an example of a `time.Duration` field:

```
logger.Info("Request timed out", zap.Duration("timeout", 10*time.Second))
```

Let's illustrate how to log arbitrary objects, such as structures. In *Chapter 2 Scaffolding a Go Microservice*, we defined the `Metadata` structure:

```
// Metadata defines the movie metadata.  
type Metadata struct {  
    ID      string `json:"id"  
    Title   string `json:"title"  
    Description string `json:"description"  
    Director string `json:"director"  
}
```

Let's assume that we want to log the entire structure for debugging purposes. One way of doing so is to use the `zap.Stringer` field. This field allows us to log any structure or interface with the `String()` function. We can define a `String` function for our `Metadata` structure as follows:

```
func (m *Metadata) String() string {  
    return fmt.Sprintf("Metadata{id=%s, title=%s, description=%s,  
director=%s}", m.ID, m.Title, m.Description, m.Director)  
}
```

Now, we can log the `Metadata` structure as a log field:

```
logger.Debug("Retrieved movie metadata", zap.Stringer("metadata",  
metadata))
```

The output would look as follows:

```
{"level": "debug", "msg": "Retrieved movie  
metadata", "metadata": "Metadata{id=id, title=title,  
description=description, director=director}"}
```

Now, let's illustrate one more useful technique of using the `zap` library. If you want to include the same fields in multiple messages, you can re-initialize the logger by using the `With` function, as illustrated in the following example:

```
logger = logger.With(zap.String("endpoint", "PutRating"), zap.  
String("ratingId", ratingID))  
logger.Debug("Received a PutRating request")  
// endpoint logic  
logger.Debug("Processed a PutRating request")
```

The results of both calls to the Debug function will now include both the endpoint and ratingId fields.

You can also use this technique when you create new service components in your code. In the following example, we are creating a sub-logger inside the New function:

```
func New(logger *zap.Logger, ctrl *rating.Controller) *Handler {
    return &Handler{logger.With("component": "ratingController"), ctrl}
}
```

This way, the newly created instance of a Handler structure will be initialized with a logger that includes the component field with the ratingController value in each message.

Now that we have covered some primary service logging use cases, let's discuss how to store logs in a microservice environment.

Storing microservice logs

By default, logs of each service instance are written to the output stream of the process running it. This mechanism of log collection allows us to monitor service operations by continuously reading the data from the associated stream (stdout in most cases) on a host running a service instance. However, without any additional software, the log data would not be persisted, so you would not be able to read your previously recorded logs after a service restart or a sudden crash.

Various software solutions allow us to store and query log data in a multi-service environment. They help solve multiple other problems:

- **Distributed log collection:** If you have multiple services running on different hosts, you need to collect the service logs on each host independently and send them for further aggregation.
- **Centralized log storage:** To be able to query data that's emitted by different services, you need to store it in a centralized way – all logs across all services should be accessible during the query execution.
- **Data retention:** Logging data usually takes a lot of disk space, and it often becomes too expensive to store it indefinitely for all your services. To solve this problem, you need to establish the right data retention policies for your services that will allow you to configure how long you can store the data for each one.
- **Efficient indexing:** To be able to quickly query your logging data, logs need to be indexed and stored efficiently. Modern indexing software can help you query terabytes of log data in under 10 milliseconds.

Let's briefly review some popular log collection tools and solutions.

Elasticsearch

Elasticsearch is a popular open source search engine that gained popularity as a scalable system for indexing and querying different types of structured data. While the primary use case of Elasticsearch is a full-text search, it can be efficiently used for storing and querying various types of structured data, such as service logs. Elasticsearch is also a part of the toolkit called the **Elastic Stack**, also called **ELK**, which includes some other systems:

- **Logstash:** A data processing pipeline that can collect, aggregate, and transform various types of data, such as service logs
- **Kibana:** A user interface for accessing data in Elasticsearch, providing convenient visualization and querying features

One of the key advantages of the Elastic Stack is that most of its tools are available for free and are open source. It is well maintained and extremely popular in the developer community, making it easier to search for relevant documentation, get additional support, or find some additional tooling. It also has a set of libraries for all popular languages, allowing us to perform various types of queries and API calls to all components of the pipeline.

The Go library for using the Elasticsearch API is called `go-elasticsearch` and can be found on GitHub at <https://github.com/elastic/go-elasticsearch>.

We are not going to cover the Elastic Stack in detail as it's outside of the scope of this chapter, but you can get more familiar with the Elastic Stack by reading its official documentation (<https://www.elastic.co/guide/index.html>).

OpenSearch

OpenSearch is a fork of the Elastic Stack, including Elasticsearch and Kibana, that provides some additional features, including built-in security controls, **machine learning (ML)**-based anomaly detection, enhanced dashboards, and many more. OpenSearch has a less strict license that allows you to use it among a broader range of applications (you can find info on Elasticsearch license limitations at <https://www.elastic.co/pricing/faq/licensing>) and is supported by various cloud providers, including Amazon Web Services.

OpenTelemetry Collector

OpenTelemetry Collector is a relatively new but popular solution for collecting, aggregating, and storing telemetry data, including logs. It offers some strong benefits to its users:

- **Built-in support of multiple types of telemetry data:** In addition to logs, OpenTelemetry Collector supports other types of telemetry data, including metrics and traces, offering a unified telemetry data collection solution.
- **Log correlation:** OpenTelemetry tools can help to correlate logs with other types of telemetry data, simplifying service debugging: developers can seamlessly trace specific requests through logs and traces.
- **Rich set of integrations:** OpenTelemetry Collector does not require the use of any specific system for exporting and storing data. Instead, it supports a wide range of tools, including Elasticsearch, as well as cloud-based solutions, such as AWS CloudWatch.

There are OpenTelemetry libraries for multiple languages, including Go: <https://opentelemetry.io/docs/languages/go/>

Note that OpenTelemetry Collector is just an extra layer for collecting, aggregating, and routing telemetry data, and it requires the use of an additional data exporting tool for storing log data. You can choose from a list of supported solutions, including Elasticsearch, AWS CloudWatch, Amazon Simple Storage Service (S3), and many more.

You can read more on OpenTelemetry Collector on its website: <https://opentelemetry.io/docs/collector/>

Having covered some high-level details regarding some popular logging solutions, let's move on to the next topic: describing the best practices of logging.

Logging best practices

So far, we have covered the most important aspects of logging and described how to choose a logging library, as well as how to establish a logging infrastructure for collecting and analyzing data. Let's describe some best practices for logging service data:

- Avoiding using interpolated strings
- Standardizing your log messages
- Periodically reviewing your log data
- Setting up appropriate log retention
- Identifying the message source in logs

Let's now cover each practice in detail.

Avoiding using interpolated strings

One of the top logging anti-patterns is the usage of **interpolated strings** – messages that embed metadata inside text fields. Let's take the following snippet of code as an example:

```
logger.Infof("User %s successfully registered", userID)
```

The problem with this code is that it merges two types of data into a single text message: an operation name (user registration) and a user identifier. Such messages make it harder to search and process log metadata: each time you need to extract `userID` from a log message, you would need to parse a string that contains it.

Let's update our example by following the structured logging approach, where we log additional metadata as message fields:

```
logger.Infof("User successfully registered", zap.String("userId", userID))
```

The updated version makes a big difference when you want to query your data. Now, you can query all log events that have `User successfully registered` text messages and easily access all user identifiers associated with them. Avoiding interpolated messages helps keep your log data easy to query and parse, simplifying all operations with it.

Standardizing the format of your log messages

In this section, we covered the benefits of log centralization and the advantages of querying data across multiple services. However, I would like to emphasize how important it is to standardize the format of log messages in a microservice environment. Sometimes, it is useful to execute log queries that span multiple services, API endpoint handlers, or other components. For example, you may need to perform the following types of queries on your log data:

- Get the distribution of timeout errors across all services
- Get the daily count of errors for each API endpoint
- Get distinct error messages across all database repositories

If your services log data using different field names, you will not be able to easily gather such data using a common query function. On the opposite side, establishing common field names helps ensure log messages follow the same naming convention, simplifying any queries you write.

To make sure logs are emitted in the same way across all services and all components, you may follow these tips:

- Create a shared package that includes log field names as constants, as in the following example:

```
package logging

const (
    FieldService = "service"
    FieldEndpoint = "endpoint"
    ...
)
```

- To avoid forgetting to include some important field inside a certain structure, function, or a set of functions, re-initialize the logger by setting the field as early as possible, as in the following example:

```
func (h *Handler) PutRating(ctx context.Context, req
    *PutRatingRequest) (*PutRatingResponse, error) {
    logger := h.logger.With(logging.FieldEndpoint, "putRating")
    // Now we can make sure the endpoint field is set across all
    // handler logic.
    ...
}
```

- Additionally, ensure that the root logger of your service is setting the service name so that all your service components will automatically collect this field by default:

```
func main() {
    logger, _ := zap.NewProduction()
    logger = logger.With(logging.FieldService, "rating")
    // Pass the initialized logger to all service components.
```

The tips that we just provided should help you standardize the usage of common fields across all your service components, making it easier to query logged data and aggregate it in different ways.

Periodically reviewing your log data

Once you start collecting your service logs, it is important to periodically review them. Look out for the following cases:

- **Make sure there is no PII data in logs:** Personally identifiable information (PII), such as full names and SSNs, falls under many regulations and generally must not be stored in logs. Make sure that no component, such as an API handler or a repository component, emits any such data, even for debugging.
- **Check that your service doesn't emit extra debug data:** Sometimes, developers log some additional data, such as request fields, to debug various issues. Check that no service is continuously emitting too many debug messages during a prolonged period, polluting the logs and using too much disk space.

Setting up appropriate log retention

Log data often takes a lot of space to store. If it keeps growing in size without any additional actions being taken, you may end up using all your disk space and having to urgently clean up old records. To prevent this, various log storage solutions allow you to configure **retention policies** for your data. For example, you can configure your log storage to keep the logs for some services for up to a few years while limiting some other services to just a few days, depending on the requirements. Additionally, you can set some size constraints so that the logs of your services don't exceed a predefined size threshold.

Ensure you set retention policies for all types of your logs, avoiding situations when you need to clean up unneeded log records manually.

Identifying the message source in logs

Imagine that you are viewing your system logs and notice the following error event:

```
{"level": "error", "time": "2022-09-04T20:10:10+1:00", "message": "Request timed out"}
```

Can you understand the problem described in this event? The log record includes the Request timed out error message and has the error level, but it does not provide any meaningful context to us. Without any additional context, we can't easily understand the problem that caused the log event.

Providing the context of any log message is crucial for making it easy to work with the logs. This is especially important in a microservice environment, where similar operations can be performed by multiple services or components. It should always be easy to understand each message and have some reference to the component it is coming from. In this section, we already mentioned the practice of including some additional information, such as the name of the component, in a log event. Such metadata would generally include the following:

- Name of the service
- Name of the component emitting the event (for example, endpoint name)
- Name of the file (optional)

A more detailed version of the preceding log message looks like this:

```
{"level": "error", "time": "2022-09-04T20:10:10+1:00", "message": "Reque  
st timed out", "service": "rating", "component": "handler", "endpoint":  
"putRating", "file": "handler.go"}
```

At this point, we have discussed the main topics related to logging and can move on to the next section, which describes another type of telemetry data – metrics.

Collecting service metrics

In this section, we are going to describe another type of service telemetry data: **metrics**. To understand what metrics are and how they are different from log data, let's start with an example. Imagine that you have a set of services providing APIs to their users, and you want to know how many times per second each API endpoint is called. How would you do this?

One possible way of solving this problem is using logs. We could create a log event for each request, and then we would be able to count the number of events for each endpoint, aggregating them by seconds, minutes, or in any other possible way. Such a solution would work until we get too many requests per endpoint and can't log each one independently anymore. Let's assume there is a service that processes more than a million requests per second. If we used logs to measure its performance, we would need to produce more than a million log events every second, generating lots of data.

A more optimal solution to this problem would be to use some sort of value-based aggregation. Instead of storing the data representing each request separately, we could summarize the count of requests per second, minute, or hour, making the data more optimal for storing.

The problem that we just described is a perfect use case for using metrics – real-time quantitative measurements of system performance, such as request rate, latency, or cumulative counts. Like logs, metrics are time-based – each record includes a timestamp representing a unique instant of time in the past. However, unlike log events, metrics are primarily used for storing individual values. In our example, the value of an endpoint request rate metric would be the count of requests per second.

Metrics are generally represented as **time series** – sets of objects, called **data points**, containing the following data:

- Timestamp
- Value (most commonly, the value is numerical)
- An optional set of **tags**, defined as key-value pairs, that contain any additional metadata

To help you better understand the use cases of metrics, let's define some common metric types:

- **Counters:** These are time series representing counts of specific events over time. An example would be the counter of service requests – each data point would include a timestamp and the count of requests at that particular moment.
- **Gauges:** These are time series representing the changes of a single scalar value over time. An example of a gauge is a dataset that contains the amount of free disk space on a server at different moments: each data point contains a single numerical value.
- **Histograms:** These are time series representing the distribution of some value against a predefined set of value ranges, called **buckets**. An example of a histogram metric is a dataset, containing the number of users for different age groups.

Let's focus on each metric type to help you understand their differences and common use cases for each one.

Counter metrics are generally used for measuring two types of data:

- Cumulative value over time (for example, the total number of errors)
- Change of the cumulative value over time (for example, the number of newly registered users per hour)

The second use case is technically a different representation of the first one – if you know how many users you had at each moment in time, you can see how this value was changing. Because of this, counters are often used to measure the rates of various events, such as API requests, over time.

The following code snippet provides an example of a Counter interface in a tally metrics library (we'll review this library later in this chapter):

```
type Counter interface {
    // Inc increments the counter by a delta.
    Inc(delta int64)
}
```

Unlike counters, gauges are used for storing unique values of measurements, such as the service's available memory over time. Here is a gauge example from a tally library:

```
type Gauge interface {
    // Update sets the gauges absolute value.
    Update(value float64)
}
```

Some other gauge use cases include the following:

- Number of goroutines running by a service instance
- Number of active connections
- Number of open files

Histograms are slightly different from counters and gauges. They require us to define a set of ranges that will be used to store the subsets of recorded data. The following are some examples of using histogram metrics:

- **Latency tracking:** You can track how long it takes to perform a certain service operation by creating a set of buckets representing various duration ranges. For example, your buckets could be 0-100 ms, 100-200 ms, 200-300 ms, and so on.
- **Cohort tracking:** You can track statistical data, such as the number of records in each group of values. For example, you can track how many users of each age subscribed to your service.

Now that we have covered some high-level basics of metrics, let's provide an overview of storing metrics.

Storing metrics

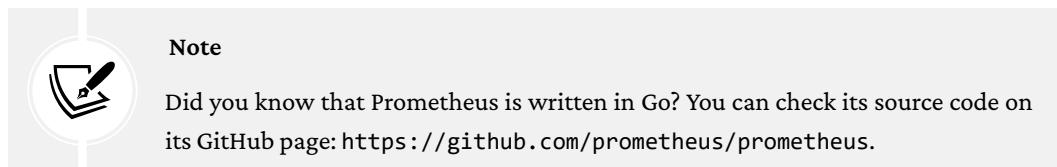
Similar to logs, storing metrics in a microservice environment brings some common challenges:

- **Collection and aggregation:** Metrics need to be collected from all service instances and sent for further aggregation and storage.
- **Aggregation:** Collected data needs to be aggregated, so various types of metrics, such as counters, would contain data coming from all service instances. For example, the counter measuring the total number of requests should summarize the data across all service instances.

Let's review some popular tools that provide such features.

Prometheus

Prometheus is a popular open source monitoring solution that provides mechanisms for collecting and querying service metrics, as well as setting up automated alerts for detecting various types of incidents. Prometheus gained popularity in the developer community due to its simple data model and being a very flexible model for data ingestion, which we are going to cover in this section.



Prometheus supports three types of metrics – counters, gauges, and histograms. It stores each metric as a time series, similar to the model that we described at the beginning of the *Collecting service metrics* section. Each metric contains the value, additional tags, called **labels**, and a name that can be used for identifying it.

Once the data gets into the Prometheus time series storage, it is available for querying via its query language, called **PromQL**. PromQL allows us to fetch time series data using various functions that allow us to easily filter or exclude certain name and label combinations. The following is an example of a PromQL query:

```
http_requests_total{environment="production",method!="GET"}
```

In this example, the query fetches a time series with the `http_requests_total` name, a label that contains the `environment` key and `production` value, and any value of a `method` label that is not equal to `GET`.

There is an official Prometheus Go client on GitHub that provides various mechanisms to get metrics data into Prometheus, as well as to execute PromQL queries. You can access it here: https://github.com/prometheus/client_golang.

The documentation for instrumenting Go applications for using Prometheus can be found at the following link: <https://prometheus.io/docs/guides/go-application>.

Graphite

Graphite is another popular monitoring tool that offers metric collection, aggregation, and querying functionality that is similar to Prometheus. Although it has been among the oldest service monitoring tools in the industry, it remains an extremely powerful instrument for working with service metric data.

A typical Graphite installation consists of three main components:

- **Carbon:** A service that listens for time series data
- **Whisper:** A time series database
- **Graphite-Web:** A web interface and an API for accessing metrics data

Graphite offers a quick integration with a data visualization tool called **Grafana**, which we are going to cover in *Chapter 13 Setting Up Service Alerting* of this book. You can read more details about Graphite on its website: <https://graphiteapp.org>.

OpenTelemetry Collector

We already mentioned OpenTelemetry Collector in the *Storing microservice logs* section of this chapter, as well as the fact that it supports other types of telemetry data, including metrics. If your service is already onboarded to OpenTelemetry, or you are considering standardizing the collection of different types of telemetry data using a single tool, OpenTelemetry is a great choice. Using OpenTelemetry Collector still requires you to choose a backend for storing metrics, and you can use a wide range of tools, including Prometheus, as well as managed cloud solutions, such as AWS CloudWatch.

Now, let's move on to the next section, where we will describe popular libraries for emitting service metrics.

Popular Go metrics libraries

There are some popular Go libraries for working with metrics that could help you ingest and query your time series metrics data. Let's provide a brief overview of some of them:

- `tally (https://github.com/uber-go/tally):`
 - A performant and minimalistic library for emitting service metrics
 - Built-in support for data ingestion in Prometheus and M3
- `rcrowley/go-metrics (https://github.com/rcrowley/go-metrics):`
 - The Go port of a popular Java metric library (<https://github.com/dropwizard/metrics>)
 - Has lots of integrations for exporting data to various observability systems, such as Datadog and Prometheus
- `go-kit/metrics (https://github.com/go-kit/kit/metrics):`
 - Part of the go-kit toolkit
 - Supports multiple metric storages, such as Prometheus and Graphite

We will leave the decision of picking a metrics library for your services to you, as each library provides some useful features that you can leverage when developing your microservices. I am going to use the `tally` library in the examples throughout this chapter as it provides a simple and minimalistic API that can help illustrate common metrics use cases. In the next section, we will review some use cases of metrics in Go microservice code.

Emitting service metrics

In this section, we will provide some examples of emitting and collecting service metrics while covering some common scenarios, such as measuring API request rates, operation latencies, and emitting gauge values. We will use the `tally` library in our examples, but you can implement this logic using all other popular metric libraries.

First, let's provide an example of how to initialize the `tally` library so that you can start using it in your service code. In the following example, we are initializing it using the Prometheus client (you can use any other tool to collect metrics):

```
reporter := prometheus.NewReporter(prometheus.Options{})  
scope, closer := tally.NewRootScope(tally.ScopeOptions{  
    Tags:      map[string]string{"service": "rating"},  
    Reporter: reporter,  
}, time.Second)
```

In this example, we are creating a `tally` reporter that will submit metrics to the data collector (Prometheus in our use case) and create a `scope` – an interface for reporting metrics data – that would automatically submit them for collection.

Scopes for the `tally` library are hierarchical: when we initialize the library, we create a `root scope`, which includes initial metadata in the form of key-value tags. All scopes that are created from it would include the parent metadata, preventing cases of missing tags during the metric's emission.

Once you get the scope, you can start reporting the metrics. The following example illustrates how to increment a counter metric by measuring the API request count, which would be automatically reported by `tally`:

```
counter := scope.Counter("request_count")
counter.Inc(1)
```

The `Inc` operation increments the value of the counter by 1, and the updated value of the metric gets collected by `tally` automatically in the background. This does not affect the performance of the function that performs the provided operations.

If you want to add some additional tags to the metric, you can use the `Tagged` function:

```
counter := scope.Tagged(map[string]string{"operation": "put"}).
Counter("request_count")
```

The following example illustrates how to update the gauge value. Let's say we have a function that calculates the number of active users in the system and we want to report this value to the metrics storage. We can achieve this by using a gauge metric in the following way:

```
gauge := scope.Gauge("active_user_count")
gauge.Update(userCount)
```

Similarly to counter metrics, gauges also support tags, so you can attach additional metadata to your metrics, such as operation names.

Now that we have provided some basic examples of emitting service metrics, let's look at best practices for working with metrics data.

Metrics best practices

In this section, we will describe some best practices related to metric data collection. The list is not exhaustive but should still be useful for setting up metric collection logic in your services.

Keeping tag cardinality in mind

When you emit your metrics and add extra tags to time series data, keep in mind that most time series databases are not designed to store highly dimensional data that contains lots of possible tag values. For example, the following types of data should not be generally included in service metrics:

- Object identifiers, such as movie or rating IDs
- Randomly generated data, such as UUIDs (for example, request UUIDs)

The reason for this is indexing: each tag combination must be indexed separately to make the time series searchable, and it becomes expensive to perform this when there are lots of distinct tag values.

You can still use some low-cardinality data in metric tags. The following are some possible examples:

- City ID
- Service name
- Endpoint name

Remember this tip to avoid reducing the throughput of your metrics pipeline and to ensure your services don't emit user identifiers and other types of high-cardinality metadata.

Standardizing metric and tag names

Imagine that you have hundreds of services, and each service follows different metric naming conventions. For example, one service could be using `api_errors` as the name of the API error counter metric, while the other could be using the `api_request_errors` name. If you wanted to compare the metrics for such services, you would need to remember which naming convention each service was using. Such metric discovery will always take time, reducing your ability to analyze your data.

A much better solution is to standardize the names of common metrics and tags across all your services. This way, you can easily search and compare various performance indicators, such as service client and server error rates, API throughput, and request latency. This also applies to other types of telemetry data: standardization is essential for making telemetry data queryable and interoperable across your microservices. In *Chapter 14*, we will review some common performance indicators that you can use to monitor the health of your services.

Setting the appropriate retention

Most time series databases are capable of storing large datasets of metrics due to efficient aggregation. Unlike logs, which require you to store each record independently, metrics can be aggregated into a smaller dataset. For example, if you store counter data, you can store the sums of the values instead of storing each value separately. Even with these optimizations, time series data can take a lot of disk space to store. Large companies can store terabytes of metrics data, so it becomes important to manage its size and set up data retention policies, similar to logs and other types of telemetry data.

Metric storages, such as Prometheus, have a default retention time of 15 days, allowing you to change it in the settings. For example, to set the data retention time in Prometheus to 60 days, you can use the following flag:

```
--storage.tsdb.retention.time=60d
```

Limiting storage retention time helps keep the size of the time series datasets under control, making it easier to manage storage capacity and plan infrastructure spending on data storage.

Now that we've discussed metrics data, let's move on to the next section, which covers a powerful technique: tracing.

Collecting service traces

So far, we have covered two common types of observability data – logs and metrics. Having logs and metrics data in place is often sufficient for service debugging and troubleshooting. However, there is another type of data that is useful for getting insights into microservice communication and data flows.

In this section, we are going to discuss **distributed tracing** – a technique that involves recording and analyzing interactions between different services and service components. The main idea behind distributed tracing is to automatically record all such interactions and provide a convenient way to visualize them. Let's look at the following example, which illustrates a distributed tracing use case known as call analysis:

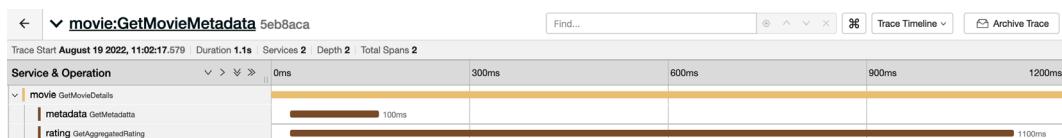


Figure 12.1 – Tracing visualization example

Here, you can see the execution of a single `GetMovieDetails` request for our movie service. The data provides some insights into the operation's execution:

- Soon after the request starts, two parallel calls come from the movie service: one to the metadata service and another to the rating service
- The call to the metadata service takes 100 milliseconds to complete
- The call to the rating service takes 1,100 milliseconds to complete, spanning almost the entire request processing time

The data that we just extracted provided us with lots of valuable information for analyzing the movie service's performance. First, it helped us understand how an individual request was handled by the movie service, as well as which sub-operations it performed. We could also see the duration of each operation and find out which one was slowing down the entire request. By using this data, we could troubleshoot endpoint performance issues, finding out which components make a significant impact on request processing.

In our example, we showed the interaction between just three services, but tracing tools allow us to analyze the behavior of systems that use tens and even hundreds of services simultaneously. The following are some other use cases that make tracing a powerful tool for production debugging:

- **Error analysis:** Tracing allows us to visualize errors on complex call paths, such as chains of calls spanning lots of different services.
- **Call path analysis:** Sometimes, you may investigate issues in systems you are not very familiar with. Tracing data helps you visualize the call path of various operations, helping you understand the logic of services without requiring you to analyze their code.
- **Operation performance breakdown:** Tracing allows us to see the duration of individual steps of a long-running operation.

Let's describe the tracing data model so that you can get familiar with its common terminology. The core element of tracing is a **span** – a record representing some logical operation, such as a service endpoint call. Each span has the following properties:

- Operation name
- Start time
- End time
- An optional set of tags providing some additional metadata related to the execution of the associated operation
- An optional set of associated logs

Spans can be grouped into hierarchies to model relationships between different operations. For example, in *Figure 12.1*, our `GetMovieDetails` span included two child spans, representing `GetMetadata` and `GetAggregatedRating` operations.

Let's explore how we can collect and use tracing data in our Go applications.

Tracing tools

There are various tools for distributed tracing in a microservice environment. Among the most popular ones is Jaeger, which we will review in this section.

Jaeger is an open source distributed tracing tool that provides mechanisms for collecting, aggregating, and visualizing tracing data. It offers a simple but highly flexible setup and a great user interface for accessing trace data. This is the reason why it quickly became one of the most popular observability tools across the industry.

Jaeger is compatible with the OpenTelemetry specification, so it can be used in combination with any clients implementing the tracing specification, such as the **Go SDK** (<https://opentelemetry.io/docs/instrumentation/go/>). The **OpenTelemetry SDK** is currently the recommended way of emitting trace data from applications, so we are going to use it in our examples throughout this section.

The general data flow for services using Jaeger looks like this:

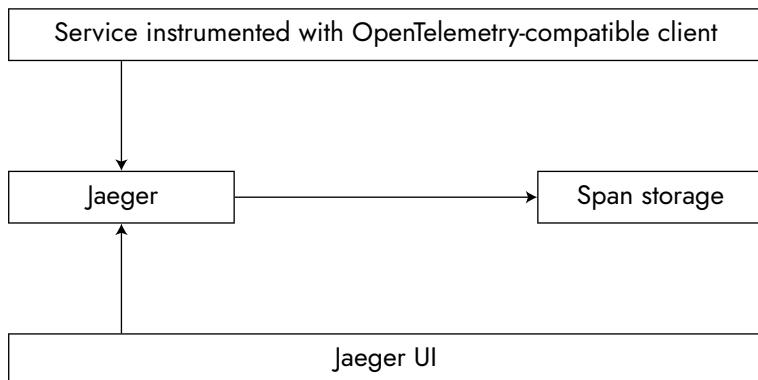


Figure 12.2 – Jaeger data flow

In the preceding diagram, a service using a Jaeger-compatible library is emitting traces to the Jaeger backend, which is storing them in the span storage. The data is accessible for querying and visualization via the Jaeger UI.

Note

Jaeger is another example of an observability tool written in Go. You can check out the Jaeger source code on its official GitHub page: <https://github.com/jaegertracing/jaeger>.

You can find more information about the Jaeger project on its website: <https://www.jaegertracing.io>.

Let's see some examples of instrumenting Go services to emit tracing data. We will use the OpenTelemetry SDK in our examples to make our code compatible with different tracing software.

Collecting tracing data with the OpenTelemetry SDK

In this section, we will show you how to emit tracing data in your service code.

As we mentioned at the beginning of the *Collecting service traces* section, the core benefit of distributed tracing is the ability to automatically capture data that shows how services and other network components communicate with each other. Unlike metrics, which measure the performance of individual operations, traces help to collect information on how each request or operation is handled across the entire network of nodes that report this data. To report traces, service instances need to be **instrumented** so that they perform two distinct roles:

- **Report the data on distributed operations:** For each **traceable operation** – an operation spanning multiple components, such as network requests or database queries – an instrumented service should report spans. The report should contain the operation name, start time, and end time.
- **Facilitate context propagation:** The service should explicitly propagate context throughout the execution (if you are confused by this, please read the following paragraphs – this is the main trick behind distributed tracing!).

We already defined a span earlier in this chapter, so let's move to the second requirement for service instrumentation. What is context propagation, and how do we perform it in Go microservice code?

Context propagation is a technique that involves explicitly passing an object, called a **context**, into other functions in the form of an argument. The context may contain arbitrary metadata, so passing it to another function helps propagate it further. That is, each function down the stream can either add new metadata to the context or access the metadata that already exists in it.

Let's illustrate context propagation via a diagram:



Figure 12.3 – Context propagation example

In the previous flow chart, there is an HTTP request coming from **Service A** to **Service B** to process a payment. **Service A** includes an additional HTTP header called `ctx.reqId` for the request to pass the request identifier. **Service B** calls **Service C** to check the user details to verify whether the user is eligible to make a payment. Then, **Service B** passes the `ctx.reqId` header further to **Service C** so that all services can record the identifier of the request, for which they perform the operations.

The example that we just provided illustrates context propagation between three services. This is achieved by including specific HTTP headers in requests, which provide additional metadata for request processing. There are multiple ways of propagating data when executing various operations. We will start by looking at regular Go function calls.

We covered Go context propagation in *Chapter 2 Scaffolding a Go Microservice* and mentioned the `context` package, which provides a type called `context.Context`. Passing the context between two Go functions of a single service is as easy as calling another function with an additional argument, as shown here:

```

func ProcessRequest(ctx context.Context, ...) {
    return ProcessAnotherRequest(ctx, ...)
}
  
```

In our example, we pass the context that we receive in our function into another one, propagating it throughout the execution chain. We can attach additional metadata to the context by using the `WithValue` function, as shown in the following code block:

```

func ProcessRequest(ctx context.Context, ...) {
    newCtx = context.WithValue(ctx, someKey, someValue)
    return ProcessAnotherRequest(newCtx, ...)
}
  
```

In this updated example, we are passing the modified context to the other function, which will include some additional tracing metadata.

Now, let's connect this knowledge with the core concept of tracing – a span. A span represents an individual operation, such as a network request, that can be related to some other operations, such as other network calls that are made during the request's execution. In our `getMovieDetails`

example, the original request would be represented as a **root span** or a parent span. Its child spans represent calls to the endpoints of metadata and rating services – both calls are made as a part of `getMovieDetails` request handling. To establish the relationship between the child and parent spans, we need to pass the identifier of the parent span to its children. We can do this by propagating it through the context of each function call, as we illustrated earlier.

To make this easier to understand, let's summarize the steps for collecting the trace data for a Go function:

1. For the original function being traced, we generate a new parent span object.
2. When the function makes calls to any other functions that need to be included in the trace (for example, network calls or database requests), we pass the parent span data to them as a part of the Go context argument.
3. When a function receives a context with some parent span metadata, we include the parent span ID in the span data associated with the function.
4. All the functions in the chain should follow the same steps and, at the end of each execution, report the captured span data.

Now, let's demonstrate how to use this technique in Go applications. We are going to use the OpenTelemetry Go SDK in our examples and use Jaeger as the data source of the tracing data:

1. Let's start with the configuration changes. Inside each service directory, update the `cmd/config.go` file to the following:

```
package main

type config struct {
    API           apiConfig      'yaml:"api"'
    ServiceDiscovery serviceDiscoveryConfig 'yaml:"serviceDiscovery"'
    Jaeger        jaegerConfig   'yaml:"jaeger"'
}

type apiConfig struct {
    Port int 'yaml:"port"'
}

type serviceDiscoveryConfig struct {
    Consul consulConfig 'yaml:"consul"'
}
```

```
type consulConfig struct {
    Address string `yaml:"address"`
}

type jaegerConfig struct {
    URL string `yaml:"url"`
}
```

The configuration that we just added will help us set the Jaeger URL for submitting the trace data.

2. The next step is to update the `configs/base.yaml` file for each service so that it includes the Jaeger API URL property. We can do this by adding the following code at the end:

```
jaeger:
  url: http://localhost:14268/api/traces
```

3. Let's create a shared function that can be used in each service to initialize the tracing data provider. This is going to submit our traces to Jaeger. In our root pkg directory, create a directory called `tracing` and add a `tracing.go` file with the following contents:

```
package tracing

import (
    "go.opentelemetry.io/otel/exporters/jaeger"
    "go.opentelemetry.io/otel/sdk/resource"
    tracesdk "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/semconv/v1.12.0"
)

// NewJaegerProvider returns a new jaeger-based tracing provider.
func NewJaegerProvider(url string, serviceName string) (*tracesdk.TracerProvider, error) {
    exp, err := jaeger.New(jaeger.WithCollectorEndpoint(jaeger.
        WithEndpoint(url)))
    if err != nil {
        return nil, err
    }
}
```

```
    tp := tracesdk.NewTracerProvider(
        tracesdk.WithBatcher(exp),
        tracesdk.WithResource(resource.NewWithAttributes(
            semconv.SchemaURL,
            semconv.ServiceNameKey.String(serviceName),
        )),
    )
    return tp, nil
}
```

Here, we are initializing the Jaeger client and using it to create an OpenTelemetry trace data provider. The provider will automatically submit the trace data that we will collect throughout the service execution.

4. The next step is to update the `main.go` file of each service. Add the `go.opentelemetry.io/otel` import to the `imports` block of the `main.go` file for each service, and add the following code block after the first `log.Printf` call:

```
    tp, err := tracing.NewJaegerProvider(cfg.Jaeger.URL,
    serviceName)
    if err != nil {
        logger.Fatal("Failed to initialize Jaeger provider", zap.
Error(err))
    }
    defer func() {
        if err := tp.Shutdown(ctx); err != nil {
            logger.Fatal("Failed to shut down Jaeger provider", zap.
Error(err))
        }
    }()
    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(propagation.TraceContext{})
```

The last two lines of the code set the global OpenTelemetry trace provider to our Jaeger-based version. These lines also enable context propagation, which will allow us to transfer the tracing data between the services.

5. To enable client-side context propagation, update the `internal/grpcutil/grpcutil.go` file to the following:

```
package grpcutil

import (
    "context"
    "math/rand"

    "go.opentelemetry.io/contrib/instrumentation/google.golang.org/
    grpc/otelgrpc"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
    "movieexample.com/pkg/discovery"
)

// ServiceConnection attempts to select a random service instance
and returns a gRPC connection to it.

func ServiceConnection(ctx context.Context, serviceName string,
    registry discovery.Registry) (*grpc.ClientConn, error) {
    addrs, err := registry.ServiceAddresses(ctx, serviceName)
    if err != nil {
        return nil, err
    }
    return grpc.Dial(
        addrs[rand.Intn(len(addrs))],
        grpc.WithTransportCredentials(insecure.NewCredentials()),
        grpc.WithUnaryInterceptor(otelgrpc.
    UnaryClientInterceptor()),
        )
}
```

Here, we added an OpenTelemetry-based interceptor that injects the tracing data into each request.

6. Inside the `main.go` file of each service, change the line containing the `grpc.NewServer()` call to the following one, to enable server-side context propagation:

```
srv := grpc.NewServer(grpc.UnaryInterceptor(otelgrpc.
    UnaryServerInterceptor()))
```

The change that we just made is similar to the previous step, just for server-side handling.

7. The last step is to make sure all the new libraries are included in our project by running the following command:

```
go mod tidy
```

With that, our services have been instrumented with tracing code and emit span data on each API request.

Let's test our newly added code by running our services and making some requests to them:

1. To be able to collect tracing data, you will need to run Jaeger locally. You can do this by running the following command:

```
docker run -d --name jaeger \
-e COLLECTOR_OTLP_ENABLED=true \
-p 6831:6831/udp \
-p 6832:6832/udp \
-p 5778:5778 \
-p 16686:16686 \
-p 4317:4317 \
-p 4318:4318 \
-p 14250:14250 \
-p 14268:14268 \
-p 14269:14269 \
-p 9411:9411 \
jaegertracing/all-in-one:1.37
```

2. Now, we can start all our services locally by executing the `go run *.go` command inside each `cmd` directory.
3. Let's make some requests to our movie service. In *Chapter 5 Synchronous Communication*, we mentioned the `grpcurl` tool. Let's use it again to make a manual gRPC query:

```
grpcurl -plaintext -d '{"movie_id":"1"}' localhost:8083
MovieService/GetMovieDetails
```

If everything was correct, we should get our trace in Jaeger. Let's check it in the Jaeger UI by going to <http://localhost:16686/>. You should see a similar page:

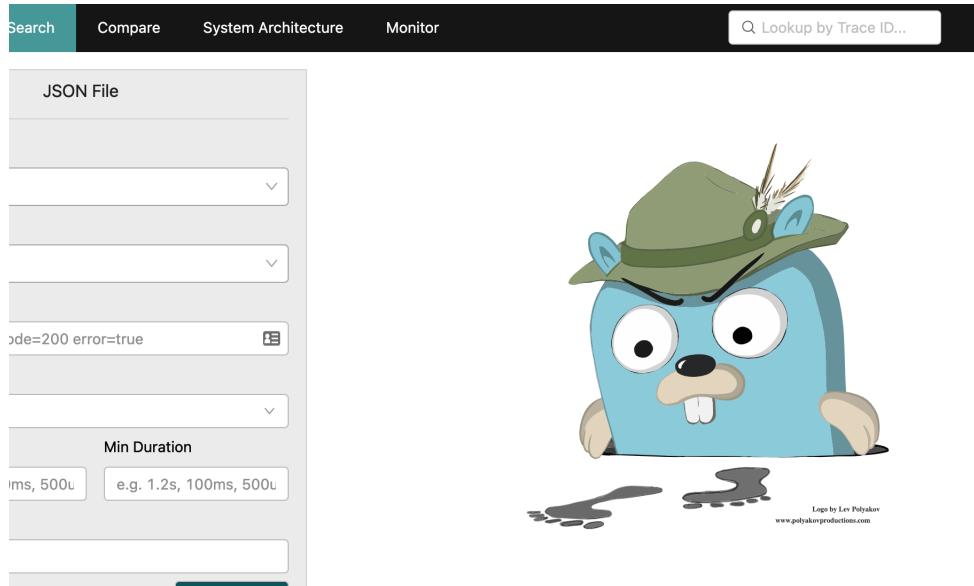


Figure 12.4 – Jaeger UI

- Select the **movie** service in the **Service** field and click **Find Traces**. You should see some trace results, as shown here:

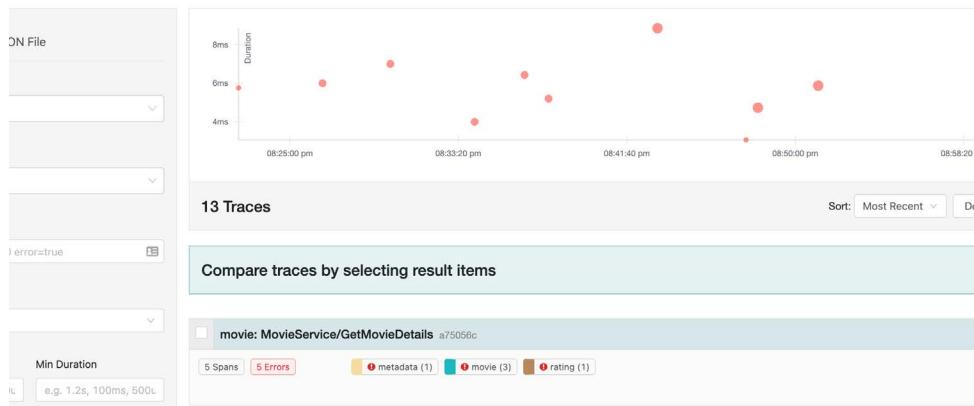


Figure 12.5 – Jaeger traces for the movie service

5. If you click on the trace, you will see its visualized view, as shown here:

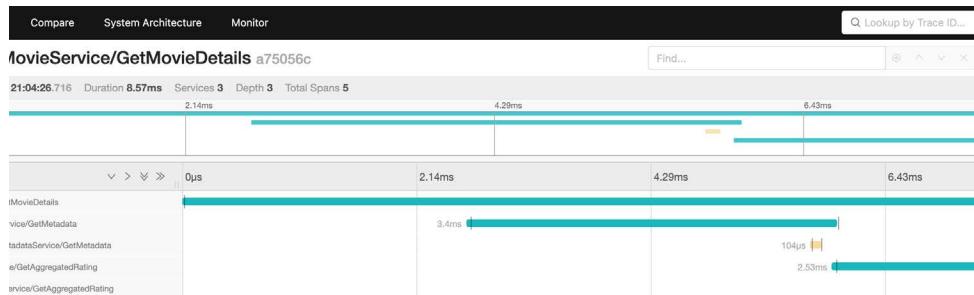


Figure 12.6 – Jaeger trace view for the GetMovieDetails endpoint call

On the left panel, you can see the request as a tree of spans, where the root span represents the MovieService/GetMovieDetails operation, which includes calls to the MetadataService/GetMetadata and RatingService/GetAggregatedRating endpoints. Congratulations – you have set up distributed tracing for your microservices using the OpenTelemetry SDK! All our gRPC calls are now traced automatically without any need to add any extra service logic. This provides us with a convenient mechanism for collecting valuable data on service communication.

As an extra step, let's illustrate how to add tracing for our database operations. As you can see from the trace view in the preceding screenshot, we currently don't have any database-related spans on our graph. This is because our database logic has not been instrumented yet. Let's demonstrate how to do this manually:

1. Open the `metadata/internal/repository/memory/memory.go` file and add `go.opentelemetry.io/otel` to its imports.
2. In the same file, add the following constant:

```
const tracerID = "metadata-repository-memory"
```

3. At the beginning of the `Get` function, add the following code:

```
_ , span := otel.Tracer(tracerID).Start(ctx, "Repository/Get")
defer span.End()
```

4. Add a similar code block at the beginning of the `Put` function:

```
_ , span := otel.Tracer(tracerID).Start(ctx, "Repository/Put")
defer span.End()
```

We just manually instrumented our in-memory metadata repository for emitting trace data on its primary operations, `Get` and `Put`. Now, each call to these functions should create a span in the captured trace, allowing us to see when and for how long each operation is being executed.

Let's test our newly added code. Restart the metadata service and make a new `grpcurl` request to the movie service provided previously. If you check for new traces in Jaeger, you should see the new one, with an additional span:

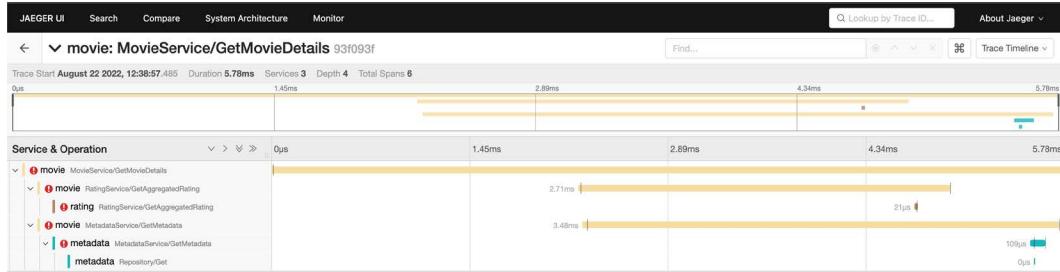


Figure 12.7 – Jaeger trace view with an additional repository span

Notice the last span in the trace view, representing the `Repository/Get` operation. It is the result of our change. Now, we can see the database operations on our traces. You can go ahead and update the rating service repository by including similar logic – follow the preceding instructions, and you should be able to make it work in the same way that we just did for the metadata service.

When should you manually add span data to your functions? I would suggest doing this for each operation involving network calls, I/O operations (such as writing and reading from files), database writes and reads, and any other calls that can take a substantial amount of time. I would personally say that any function that takes more than 50 milliseconds to complete is a good candidate for tracing.

At this point, we have provided a high-level overview of Go tracing techniques, and this marks an end to our journey into telemetry data. In the next few chapters, we will continue our explorations into other fields, such as dashboarding, system-level performance analysis, and some advanced observability techniques.

Summary

In this chapter, we covered observability by describing various techniques for analyzing the real-time performance of Go microservices and covering the main types of service telemetry data, such as logs, metrics, and traces. You learned about some best practices for performing logging, metric collection, and distributed tracing. We demonstrated how you can instrument your Go

services to collect telemetry data, as well as how to set up tooling for distributed tracing. We also provided some examples of tracing requests spanning three of the services that we implemented earlier in this book.

The knowledge that you gained in this chapter should help you debug various performance issues of your microservices, as well as enable monitoring of various types of telemetry data. In *Chapter 13 Setting Up Service Alerting*, we will demonstrate how to use the collected telemetry data to set up service alerting for detecting service-related incidents as quickly as possible.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Monitoring distributed systems: <https://sre.google/sre-book/monitoring-distributed-systems/>
- Effective troubleshooting: <https://sre.google/sre-book/effective-troubleshooting/>
- Mastering distributed tracing: <https://www.packtpub.com/product/mastering-distributed-tracing/9781788628464>
- Logging best practices: <https://devcenter.heroku.com/articles/writing-best-practices-for-application-logs>
- Ten commandments of logging: <https://www.dataset.com/blog/the-10-commandments-of-logging/>
- Microservice logging tips: <https://www.techtarget.com/searchapparchitecture/tip/5-essential-tips-for-logging-microservices>
- OpenTelemetry documentation: <https://opentelemetry.io/docs/>
- Beginner's guide to OpenTelemetry: <https://logz.io/learn/opentelemetry-guide/>
- Three pillars of observability: <https://iamondemand.com/blog/the-3-pillars-of-system-observability-logs-metrics-and-tracing/>
- What is observability?: <https://www.dynatrace.com/news/blog/what-is-observability-2/>
- What is telemetry?: <https://www.sumologic.com/insight/what-is-telemetry/>

13

Setting Up Service Alerting

In the previous chapter, we covered a service telemetry topic and described various types of telemetry data, such as logs, metrics, and traces. We also provided some examples of setting up telemetry data collection, allowing us to troubleshoot service performance issues and use the collected data to improve the reliability of our services.

In this chapter, we will illustrate how to use telemetry data to automatically detect incidents by setting up alerts for our microservices. You will learn which types of service metrics to collect, how to define the conditions for various incidents, and how to establish the complete alerting pipeline for your microservices using a popular monitoring and alerting tool, Prometheus.

We will cover the following topics:

- Alerting basics
- Introduction to Prometheus
- Setting up Prometheus alerting for our microservices
- Alerting best practices

Now, we are going to proceed to the overview of alerting basics.

Technical requirements

To complete this chapter, you will need Go 1.18 or above. You will also need the Docker tool, which you can download at <https://www.docker.com/>.

You can find the code examples for this chapter on GitHub: <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter13>.

Alerting basics

No microservice operates without issues; even if you have a stable, highly tested, and well-maintained service, it can still experience various types of issues, such as the following:

- **Resource constraints:** A host running the service may experience high CPU utilization or insufficient RAM or disk space.
- **Network congestion:** The service may experience a sudden increase in load or decreased performance in any of its dependencies. This could limit its ability to process incoming requests or operate at the expected performance level.
- **Dependency failures:** Other services or libraries that your service is depending on may experience various issues, affecting your service execution.

Such issues can be self-resolving. For example, a slower network throughput could be a transient issue caused by temporary maintenance or a network device being restarted. Many other types of issues, which we call incidents, require some actions from the engineers to be mitigated.

To mitigate an incident, first, we need to detect it. Once the issue is known, we can notify the engineers or perform automated actions, such as an automated deployment rollback or application restart. In this chapter, we will describe the **alerting technique** that combines incident detection and notification. This technique can be used to automate the incident response to various types of microservice issues.

The key principles behind alerting can be summarized in the following statements:

- To set up alerts, developers define the **alerting conditions**
- Alerting conditions are based on the telemetry data (most commonly, metrics) and are defined in the form of queries
- Each defined alerting condition is evaluated periodically, such as every minute
- If the alerting condition is met, the associated actions get executed (for example, an email or an SMS is sent to an engineer)

To illustrate how alerting works, imagine that one of your services is emitting a metric called `active_user_count` that reports the number of active users at a particular moment. Let's assume that we would like to get notified if the number of active users suddenly drops to zero. Such a situation would likely indicate some incident with our service unless we have too few users (for simplicity, we will assume our system should always have some active users).

Using pseudocode, we could define the alerting condition for our use case in the following way:

```
active_user_count == 0
```

Once the alerting condition has been met, the alerting software will check actions that should be triggered based on its configuration. Assuming that we have configured our alerts to trigger email notifications, it would send the emails and include any necessary metadata. The metadata would include information such as the values of metrics breaching thresholds and, if provided, the steps to mitigate it.

We will provide some examples of alerting configurations in the *Introduction to Prometheus* section of this chapter. For now, we will focus on some practical use cases, providing you with some ideas on setting up alerts for your services.

Alerting use cases

There are many use cases for which you would need to set up automated alerts. In this section, we will provide some common examples that can act as a reference point for you.

In the *Google SRE* book we mentioned earlier in *Chapter 11*, there was a definition of the **four golden signals** of monitoring, which can be used to monitor various types of applications, from microservices to data processing pipelines. These signals provide a great basis for service alerting, so let's review them and describe how you can use each one to increase your service reliability:

- **Latency:** Latency is a measure of processing time, such as the duration of processing an API request, a Kafka message, or any other operation. It is the main indicator of system performance – when it gets too high, the system starts affecting its callers, creating network congestion. You should generally track the latency of your primary operations, such as API endpoints providing the critical functionality.
- **Traffic:** Traffic measures the load on your system, such as the number of requests your microservices are getting at the current moment. An example of a traffic-based metric is an API request rate, measured as the number of requests per second. Measuring traffic is important to ensure you have enough capacity to handle the requests to your system, as well as to know when traffic suddenly drops below the expected rate.
- **Errors:** Errors are often measured as the **error rate** or the ratio between the failed and total operations. Measuring the error rate is critical for ensuring your services remain operational.
- **Saturation:** Saturation generally measures the utilization of your resources, such as RAM or disk usage, CPU, or I/O load. You should keep track of saturation to ensure your services don't fail unexpectedly due to resource insufficiency.

These four golden signals can help you establish monitoring and alerting for your services and critical operations, such as your primary API endpoints. Let's provide some practical examples to help you understand some common alerting use cases.

First, let's start with the common signals for API alerting that can be measured either across all services or service endpoints or on a per-endpoint basis:

- **API client error rate:** The ratio between the requests that fail due to client errors and all requests
- **API server error rate:** The ratio between the requests that fail due to server errors and all requests
- **API latency:** The time it takes to process requests

Note



When tracking latency, it's common to use **percentiles** (e.g., P95 latency, meaning 95% of requests were faster than this value) to focus on the typical user experience while excluding outliers – unusually high values caused by issues such as network delays or timeouts.

Now, let's provide some examples of signals for measuring system saturation:

- **CPU utilization:** This measures how much your CPUs are being used on a scale from 0% (unused/idle) to 100% (fully used, no extra capacity).
- **Memory utilization:** This is the ratio between the used and total memory.
- **Disk utilization:** This is the percentage of used disk space.
- **Open file descriptors:** File descriptors are often used to handle network requests, file writes and reads, and other I/O operations. There is usually a limit on the number of open file descriptors per process, so if your service reaches a critical limit (based on your OS settings), your service may fail to serve requests.

Let's also provide some examples of other signals to monitor:

- **Service panics:** The general recommendation is not to tolerate any service panics, as they often signal application bugs or issues such as out-of-memory errors
- **Failed deployments:** You can automate the detection of failed deployments and emit a metric indicating the failure, using it to create automated alerts

Now that we have covered some common alerting use cases, let's proceed to the overview of the Prometheus tool, which we will use to set up our microservice alerts.

Introduction to Prometheus

In *Chapter 12, Collecting Service Telemetry Data*, we mentioned a popular open source alerting and monitoring tool called Prometheus, which can collect service metrics and set up automated alerts based on the metric data. In this section, we will demonstrate how to use Prometheus to set up alerts for our microservices.

Let's summarize our learnings about Prometheus from *Chapter 12, Collecting Service Telemetry Data*:

- Prometheus allows us to collect and store service metrics in the form of a time series
- There are three types of metrics – counters, histograms, and gauges
- To query metrics data, Prometheus offers a query language called **PromQL**
- Service alerts can be configured using a tool called **Alertmanager**

Metrics can be imported from service instances into Prometheus in two different ways:

- **Scraping:** Prometheus reads metrics from service instances
- **Pushing:** The service instance sends metrics to Prometheus using a dedicated service, called **Prometheus Pushgateway**

Scraping is the recommended way of setting up metrics data ingestion in Prometheus. Each service instance needs to expose an endpoint to provide the metrics, and Prometheus takes care of pulling the data and storing it for further querying, as shown in the following diagram:

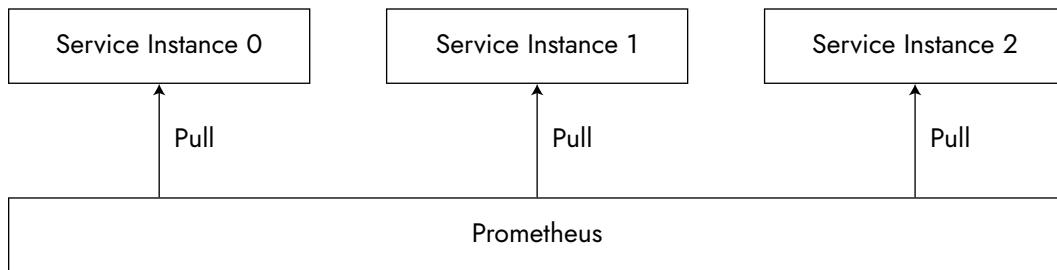


Figure 13.1 – Prometheus scraping model

Let's provide an example of a service instance response to a scraping request by Prometheus. Let's assume you add a separate HTTP API endpoint called `/metrics` and return the newest service instance metrics in the following format:

```
active_user_count 755
api_requests_total_count 18900
api_requests_getuser_count 500
```

In this example, the service instance reports three metrics in the form of key-value pairs, where the key defines a time series name and the value defines the value of the time series at the current moment. Once Prometheus calls the `/metrics` endpoint, the service instance should provide a new dataset containing only time series that have not been included in previous responses.

Once Prometheus collects the metrics, they become available for querying using a Prometheus-specific language called PromQL. PromQL-based queries can be used to analyze the time series data through the Prometheus UI or to set up automated alerts using Alertmanager. For example, the following query returns all values of the active user count time series, as well as their tags:

```
active_user_count
```

The result of this query will include a list of time series values for all distinct metric tag/label combinations:

```
active_user_count{instance="localhost:8083", job="prometheus",  
service="movie"} 87
```

You can use additional query filters, called **matchers**, to include only specific data points. For example, if multiple services emit the `active_user_count` metric, you can only request time series that have a particular tag value:

```
active_user_count{service="rating-ui"}
```

Alerting conditions are generally defined as expressions that return Boolean results. For example, to define the alerting condition when the active user count drops to zero, you would use the following PromQL query with the `==` operator:

```
active_user_count == 0
```

PromQL provides some other types of time series matchers, such as `quantile`, which can be used to perform various aggregations. The following query example can be used to check whether the median `api_request_latency` value exceeds 1:

```
api_request_latency{quantile="0.5"} > 1
```

Let's also provide an example of an alert for a **service-level agreement (SLA)**. Assume that you want to track when your service HTTP API availability drops below 95% and you have a `http_requests_total` metric with a `status` field representing HTTP status code. Then, you can define an SLA query in the following way:

```
(1 - (sum(rate(http_requests_total{status=~"5.."}[5m])) / sum(rate(http_  
requests_total[5m])))) * 100
```

Our SLA query tracks HTTP API availability over the last 5 minutes and produces the numeric value of availability success rate between 0 and 100 (with 100 meaning 100% available).

You can become familiar with the other use cases of PromQL by reading the official documentation on its website: <https://prometheus.io/docs/prometheus/latest/querying/basics/>. Now, let's explore how to set up alerts using the Prometheus alerting tool, Alertmanager.

Alertmanager is a separate component of Prometheus that allows us to configure alerts and notifications to detect various types of incidents. Alertmanager operates by reading the provided configuration and querying Prometheus time series data periodically. Let's provide an example of Alertmanager's configuration:

```
groups:
- name: Availability alerts
  rules:
    - alert: Rating service down
      expr: service_availability{service="rating"} == 0
      for: 3m
      labels:
        severity: page
      annotations:
        title: Rating service availability down
        description: No available instance of the rating service.
```

In our configuration example, we set an alert for when the value of the `service_availability` metric, which has a `service="rating"` tag, is equal to `0` for 3 minutes or more, triggering a PagerDuty incident to notify the on-call engineer about the issue.

Some other features of Alertmanager include notification grouping, notification retries, and alert suppression. To illustrate how Prometheus and Alertmanager work in practice, let's describe how to set them up for our example microservices from the previous chapters.

Setting up Prometheus alerting for our microservices

In this section, we will illustrate how to set up service alerting using Prometheus and its alerting extension, Alertmanager, for the services we created in the previous chapters. You will learn how to expose the service metrics for collection, how to set up Prometheus and Alertmanager to aggregate and store the metrics from multiple services, and how to define and process service alerts.

Our high-level approach is as follows:

1. Set up Prometheus metric reporting to our services.
2. Install Prometheus and configure it to scrape the data from the three example services that we created in the previous chapters.
3. Configure service availability alerts using Alertmanager.
4. Test our alerts by triggering an alerting condition and running Alertmanager.

Let's start by illustrating how to integrate our services with Prometheus. To do this, we need to add a metric collection to our services by exposing an endpoint that will provide the newest metrics to Prometheus:

1. First, we need to add Prometheus configuration to our services. In each service directory, update the config structure in the cmd/config.go file to the following:

```
type config struct {
    API           apiConfig      'yaml:"api"'
    ServiceDiscovery serviceDiscoveryConfig
    'yaml:"serviceDiscovery"'
    Jaeger        jaegerConfig   'yaml:"jaeger"'
    Prometheus    prometheusConfig 'yaml:"prometheus"'
}
```

2. Additionally, add the following code block to the end of each cmd/config.go file:

```
type prometheusConfig struct {
    MetricsPort int 'yaml:"metricsPort"'
}
```

3. Our new configuration allows us to specify the service port of the metric collection endpoint. Inside each configs/base.yaml file, add the following block:

```
prometheus:
  metricsPort: 8091
```

4. We are ready to update our services so that they can start reporting the metrics. Update the main.go file of each service by adding the following imports:

```
"github.com/uber-go/tally"
"github.com/uber-go/tally/prometheus"
```

5. In any part of the `main` function, add the following code:

```
reporter := prometheus.NewReporter(prometheus.Options{})
_, closer := tally.NewRootScope(tally.ScopeOptions{
    Tags:           map[string]string{"service": "metadata"},
    CachedReporter: reporter,
}, 10*time.Second)
defer closer.Close()
http.Handle("/metrics", reporter.HTTPHandler())
go func() {
    if err := http.ListenAndServe(fmt.Sprintf(":%d", cfg.
Prometheus.MetricsPort), nil); err != nil {
        logger.Fatal("Failed to start the metrics handler", zap.
Error(err))
    }
}()

counter := scope.Tagged(map[string]string{
    "service": "metadata",
}).Counter("service_started")
counter.Inc(1)
```

In the code we just added, we initialized the `tally` library to collect and report the metrics data, which we mentioned in *Chapter 12, Collecting Service Telemetry Data*. We used a built-in Prometheus reporter that implements metric data collection using the Prometheus time series format and exposed an HTTP endpoint to allow Prometheus to collect our data.

Let's test the newly added endpoint. Restart the metadata service and try accessing the new endpoint by opening `http://localhost:8091/metrics` in your browser. You should get a similar response:

```
# HELP go_gc_duration_seconds A summary of the pause duration of garbage
collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
...
```

The response of the metrics handler includes the Go runtime data, such as the number of goroutines at the current moment, the Go library version, and many other useful metrics.

Now, we are ready to set up Prometheus alerting. The steps are the following:

1. Inside the `src` directory of our project, create a directory called `configs` and add a `prometheus.yaml` file with the following content:

```
global:  
  scrape_interval: 15s  
  scrape_timeout: 10s  
  evaluation_interval: 15s  
alerting:  
  alertmanagers:  
    - follow_redirects: true  
      enable_http2: true  
      scheme: http  
      timeout: 10s  
      api_version: v2  
      static_configs:  
        - targets:  
          - host.docker.internal:9093
```

2. Additionally, add the following configuration to the file:

```
rule_files:  
  - alerts.rules  
scrape_configs:  
  - job_name: prometheus  
    honor_timestamps: true  
    scrape_interval: 15s  
    scrape_timeout: 10s  
    metrics_path: /metrics  
    scheme: http  
    follow_redirects: true  
    enable_http2: true  
    static_configs:  
      - targets:  
        - localhost:9090  
      - targets:  
        - host.docker.internal:8091
```

```
labels:  
  service: metadata  
- targets:  
  - host.docker.internal:8092  
labels:  
  service: rating  
- targets:  
  - host.docker.internal:8093  
labels:  
  service: movie
```

Let's describe the configuration that we just added. We set the scraping interval provided to 15 seconds and provided a set of targets to scrape the metrics data, which includes the address of each of our services. You may notice that we are using the `host.docker.internal` network address in each target definition – we will run Prometheus using Docker, and the `host.docker.internal` address will allow it to access our newly added endpoints running outside of Docker.



Note

To keep our setup simple, we are not using Kubernetes in this chapter. If you are interested in running Prometheus in Kubernetes, you might check this open source project providing the necessary configuration: <https://github.com/prometheus-operator/kube-prometheus>.

Note that we provided a static list of service addresses inside the `static_configs` block. We did this intentionally to illustrate the simplest scraping approach, which is when Prometheus knows the address of each service instance. In a dynamic environment, where service instances can be added or removed, you would need to use Prometheus with a service registry, such as Consul. Prometheus provides built-in support for scraping metrics from services registered with Consul; instead of `static_configs`, you could define the Consul scraping configuration:

```
consul_sd_configs:  
- server: host.docker.internal:8500  
services:  
- <SERVICE_NAME>
```

Next, we will demonstrate how to scrape a static list of service instances; you can try setting up Consul-based Prometheus scraping as an exercise after reading this chapter. Let's add alerting rules for our services. Inside the newly added `configs` directory, create the `alerts.rules` file and add the following to it:

```
groups:
- name: Service availability
  rules:
    - alert: Metadata service down
      expr: up{service="metadata"} == 0
      labels:
        severity: warning
      annotations:
        title: Metadata service is down
        description: Failed to scrape {{ $labels.service }}. Service possibly down.
    - alert: Rating service down
      expr: up{service="rating"} == 0
      labels:
        severity: warning
      annotations:
        title: Metadata service is down
        description: Failed to scrape {{ $labels.service }} service on {{ $labels.instance }}. Service possibly down.
    - alert: Movie service down
      expr: up{service="movie"} == 0
      labels:
        severity: warning
      annotations:
        title: Metadata service is down
        description: Failed to scrape {{ $labels.service }} service on {{ $labels.instance }}. Service possibly down.
```

The file we just added includes the alert definitions for each of our services. Each alert definition includes the expression Prometheus would check to evaluate whether an associated alert should be fired.

Now, we are ready to install and run Prometheus to test our alerting. Inside the `src` directory of our project, run the following command to run Prometheus using the newly created configuration:

```
docker run \
-p 9090:9090 \
-v configs:/etc/prometheus \
prom/prometheus
```

If everything is successful, you should be able to access the Prometheus UI by opening `http://localhost:9090/`. On the initial screen, you will see the search input you can use to access the Prometheus metrics emitted by our services. Type up in the search input and click **Execute** to access the metrics:

The screenshot shows the Prometheus Metrics Search interface. At the top, there is a navigation bar with links for Alerts, Graph, Status, and Help. Below the navigation bar are several configuration checkboxes: "Use local time" (unchecked), "Enable query history" (unchecked), "Enable autocomplete" (checked), "Enable highlighting" (checked), and "Enable linter" (checked). A search bar contains the text "up". To the right of the search bar are three icons: a gear, a crescent moon, and a circle. Below the search bar is a button labeled "Execute". Underneath the search bar, there are two tabs: "Table" (disabled) and "Graph" (selected). A date range selector shows "Evaluation time" with arrows for navigating between dates. To the right of the date range are statistics: "Load time: 20ms", "Resolution: 14s", and "Result series: 4". The main area displays a table of metric results:

up{instance="host.docker.internal:8091", job="prometheus", service="metadata"}	1
up{instance="host.docker.internal:8092", job="prometheus", service="rating"}	1
up{instance="host.docker.internal:8093", job="prometheus", service="movie"}	1
up{instance="localhost:9090", job="prometheus"}	1

At the bottom right of the search interface is a "Remove Panel" button.

Figure 13.2 – Prometheus metrics search

You can go to the **Alerts** tab to see the currently configured alerts that we defined in our `alerts.rules` file:

The screenshot shows the Prometheus Alerts view. At the top, there is a navigation bar with links for Alerts, Graph, Status, and Help. Below the navigation bar are filter buttons for "Inactive (3)", "Pending (0)", and "Firing (0)". There is also a search bar with the placeholder "Filter by name or labels" and a checkbox for "Show annotations". A "Show annotations" button is located at the top right. The main area displays a list of alerts under the heading "/etc/prometheus/alerts.rules > Service availability". Each alert is represented by a green box with an "inactive" status indicator at the top right. The alerts are:

- > Metadata service down (0 active)
- > Rating service down (0 active)
- > Movie service down (0 active)

Figure 13.3 – Prometheus Alerts view

If all three services are running, all three associated alerts should be marked as **inactive**. We will get back to the **Alerts** page shortly; for now, let's proceed and set up Alertmanager so that we can trigger some alerts for our services.

Inside our `configs` directory, including the Prometheus configuration, add a file called `alertmanager.yml` with the following content:

```
global:  
  resolve_timeout: 5m  
route:  
  repeat_interval: 1m  
  receiver: default  
receivers:  
  - name: default
```

Now, run the following command to start Alertmanager:

```
docker run -p 9093:9093 -v <PATH_TO_CONFIGS_DIR>:/etc/alertmanager prom/  
alertmanager --config.file=/etc/alertmanager/alertmanager.yml
```

Don't forget to replace the `<PATH_TO_CONFIGS_DIR>` placeholder with the full local path to the `configs` directory containing the newly added `alertmanager.yml` file.

Now, let's simulate the alerting condition by manually stopping the rating and movie services. Once you do this, open the **Alerts** page in the Prometheus UI; you should see that both alerts are marked as **firing**:

The screenshot shows the Prometheus UI with the 'Alerts' tab selected. At the top, there are filters for 'Inactive (1)', 'Pending (0)', and 'Firing (2)'. A search bar and a 'Show annotations' checkbox are also present. Below the filters, a breadcrumb navigation shows the path: '/etc/prometheus/alerts.rules > Service availability'. On the right, there are buttons for 'inactive' and 'firing (2)'. The main area displays three alert cards: 1. 'Metadata service down' (0 active) - green card. 2. 'Rating service down' (1 active) - pink card. 3. 'Movie service down' (1 active) - pink card. Each card has a link to its details.

Figure 13.4 – Firing Prometheus alerts

You can access the Alertmanager UI by going to <http://localhost:9093>.

If alerts are fired in Prometheus, you should also see them in the Alertmanager UI:

The screenshot shows the Alertmanager UI interface. At the top, there is a navigation bar with links for 'Alertmanager', 'Alerts', 'Silences', 'Status', and 'Help'. On the far right of the navigation bar is a 'New Silence' button. Below the navigation bar is a search bar with tabs for 'Filter' and 'Group'. To the right of the search bar are three status indicators: 'Receiver: All' (selected), 'Silenced' (disabled), and 'Inhibited' (disabled). Below the search bar is a text input field with placeholder text 'Custom matcher, e.g. env="production"'. To the right of the input field are a '+' button and a 'Silence' button with a crossed-out bell icon. Below the search area, there is a link to expand all groups ('Expand all groups'). Underneath, there are two alert entries. Each entry consists of a '+' button, a text box containing the alert name, another '+' button, and the number '1 alert'. The first alert is for 'alertname="Movie service down"' and the second for 'alertname="Rating service down"'. Both entries have a '+' button to their right.

Figure 13.5 – The Alertmanager UI

If everything worked well, congratulations, you have set up service alerting! We intentionally haven't covered many of Alertmanager features – it includes many configurable settings that are outside the scope of this chapter, including email and HTTP webhook notifications. If you are interested in learning more about it, check the official documentation at <https://prometheus.io/docs>.

Now, let's proceed to the next section, where we will provide some best practices for setting up service alerting that should help you increase your service reliability.

Alerting best practices

The knowledge you will gain by reading this section should be useful for establishing the new alerting process for your services. It will also help you improve the existing alerts if you are working with some established alerting processes.

Among the most valuable best practices, I would highlight the following ones:

- **Keep your alerts immediately actionable:** Alerting is a powerful technique to ensure any issues or incidents get acknowledged and addressed. However, you should not overuse it for the types of issues that do not require immediate attention. Some types of alerts, such as alerts indicating high saturation, are not necessarily actionable. For example, a sudden increase in CPU load may not indicate any immediately actionable issue, unless it remains high for some prolonged period (for example, CPU load not going below 85% for more than 10 minutes), and might just be a transient symptom of high service usage. When creating alerts, think about whether an engineer needs to perform any manual action as a result of a notification, and reduce any possible noise as much as possible (for example, specify how long the metric should be breaching the threshold before an alert gets fired by providing the `for` value in the rule configuration).
- **Standardize your alert definitions:** The more services you have, the more important it becomes to have consistent definitions of service availability and other reliability metrics. Standardizing the format of your service metrics and common queries used in your alerts helps to maintain the same reliability requirements across your microservices, as well as simplify the maintenance of your alert metadata.
- **Include the runbook references:** For each alert, ensure you have a runbook in place that provides clear instructions to the on-call engineers receiving it. Having an accurate and up-to-date runbook for each alert helps reduce the incident mitigation time and share relevant knowledge among all engineers.
- **Ensure the alerting configuration is reviewed periodically:** The best solution to ensure the alerting configuration is accurate is to make it easy to access and review. One of the easiest solutions is to make the alerting configuration a part of your code base so that all alert configurations are easily reviewable. Perform periodic checks of your alerts to ensure all important scenarios are covered, as well as to ensure no alerts are outdated.

This list contains just a handful of best practices to improve your service alerting. If you are interested in the topic, I strongly suggest that you read the relevant chapters of the *Google SRE* book, including the *Monitoring Distributed Systems* chapter from <https://sre.google/sre-book/monitoring-distributed-systems/>.

This provided a brief overview of service alerting. Now, let's summarize this chapter.

Summary

In this chapter, we covered one of the most important aspects of service reliability work – alerting. We learned how to set up the service metric collection using the Prometheus tool and the tally library, set up service alerts using the Alertmanager tool, and connect all these components to create an end-to-end service alerting pipeline.

The material in this chapter summarized our learnings from the reliability and service telemetry topics from *Chapter 11* and *Chapter 12*. By collecting the telemetry data and establishing the notification mechanisms using the alerting tools, we can quickly detect various service issues and get notified each time we need to mitigate them.

In the next chapter, we will cover additional aspects of Go performance monitoring, including system profiling.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Practical Alerting from Time-Series Data: <https://sre.google/sre-book/practical-alerting/>
- Monitoring Distributed Systems: <https://sre.google/sre-book/monitoring-distributed-systems/>
- Prometheus documentation: <https://prometheus.io/docs/introduction/overview/>
- Eliminating Toil: <https://sre.google/workbook/eliminating-toil/>

14

Performance Monitoring

In the previous chapters, we discussed how to collect various types of telemetry data and establish alerting mechanisms to ensure the reliability of our microservices. In this chapter, we are going to review additional mechanisms for collecting and visualizing service telemetry data to monitor and analyze their performance. We are going to cover the following topics:

- Creating dashboards to visualize service telemetry data
- Profiling Go services

By the end of the chapter, you will have learned how to visualize most parts of service telemetry data to track the health and performance of your microservices.

Let's proceed to the first section of this chapter, which covers service telemetry data monitoring using dashboards.

Technical requirements

To complete this chapter, you will need Go 1.18 or above. Additionally, you will need the following tools:

- **Graphviz:** <https://graphviz.org>
- **Docker:** <https://www.docker.com>

You also need to download the `flamegraph.pl` and `stackcollapse-go.pl` files from the <https://github.com/brendangregg/FlameGraph> repository. If you are using Windows, you will need a Perl interpreter, which you can download from <https://strawberryperl.com/>.

You can find the code examples for this chapter on GitHub: <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter14>

Creating dashboards to visualize service telemetry data

In this section, we will describe a useful technique for accessing the service telemetry data that can help you explore your data in the form of various graphical charts. The technique that we will cover is called **dashboarding**, and it is widely used to track performance metrics and events, including logs, traces, and metrics.

Introduction to dashboards

We can define a **dashboard** as a set of charts representing different types of telemetry data. The following figure shows the dashboard of a Go service containing some system-level metrics, such as the goroutine count, the number of Go threads, and the allocated memory size:



Figure 14.1 – Go process dashboard example from the Grafana tool

Among the most common use cases of using dashboards is visualizing service metrics, including counters (such as service panic counts), gauges (current resource utilization in percent), and histograms (distribution of request processing latency, such as <50 ms, 50-200 ms, 200-1000 ms, 1000+ ms). In *Figure 14.1*, there are four gauge metrics, representing current resource utilization by a single service instance.

Dashboards are useful for the following types of service performance analysis:

- **System health monitoring and debugging:** Visual representations of service metrics provide a convenient mechanism for analyzing the health of different services, as well as their high-level performance

- **Data correlation:** Having a side-by-side representation of multiple service performance charts helps us find related events, such as an increase in server errors or a sudden drop in available memory
- **Trend analysis:** Viewing service performance data on higher time ranges (for example, over the last 7 or 30 days) allows us to see various data patterns, such as daily or weekly activity/performance spikes

There are multiple ways to group various performance indicators into dashboards. In a microservice environment, there are two common types of dashboards:

- **Per-service dashboards:** Performance details for a specific service (for example, average CPU and memory utilization of service instances or service API error rate)
- **Central dashboards covering multiple services:** Performance details for a group of services (or all system services), such as the total number of Go errors and panics across all Go microservices

It's generally a good practice to have both types of dashboards to provide both granular per-service performance views and centralized views on the entire system. Per-service dashboards can be further enriched with service-specific details (for example, error rates for each service endpoint), while centralized cross-service dashboards can provide a "bird's-eye view" on everything that is happening across all services, helping to identify higher-level issues and data patterns (for example, infrequent errors in a specific library used by multiple services that are hard to detect while looking at a specific service dashboard due to low rate of occurrence).

Creating performance dashboards using the Grafana tool

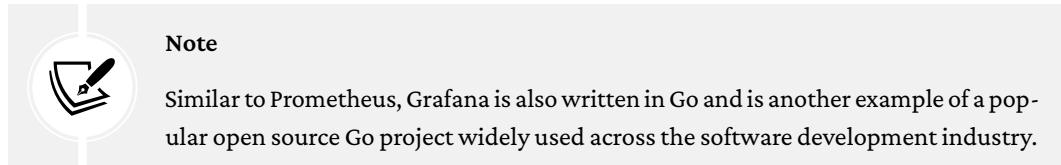
Let's demonstrate how to set up an example dashboard for the Prometheus data that we collected in *Chapter 13, Setting Up Service Alerting*. For this, we will use the open source tool called **Grafana**, which has built-in support for various types of time-series data and provides a convenient UI for setting up different dashboards.

The steps are the following:

1. Execute the following command to run the Grafana Docker image:

```
docker run -d -p 3000:3000 grafana/grafana-oss
```

This command should fetch and run the open source version of Grafana (Grafana also comes in an enterprise version, which we won't cover in this chapter) and expose port 3000 so that we can access it via HTTP.



2. Once you've run the preceding command, open `http://localhost:3000` in your browser. This will lead you to the Grafana login page. By default, the Docker-based version of Grafana includes a user with `admin` as both its username and password, so you can use these credentials to log in.
3. From the side menu, select **Data sources** within the **Connections** section:

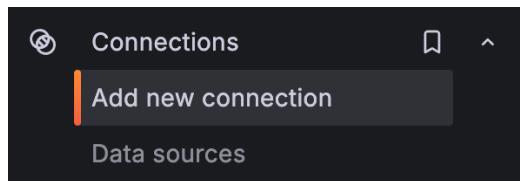


Figure 14.2 – Grafana data source configuration menu

4. On the **Data sources** page, click **Add data source** and choose **Prometheus** from the list of available data sources. Doing so will open a new page that displays Prometheus settings. In the **Connection** section, set **URL** to `http://host.docker.internal:9090`, as shown in the following screenshot:

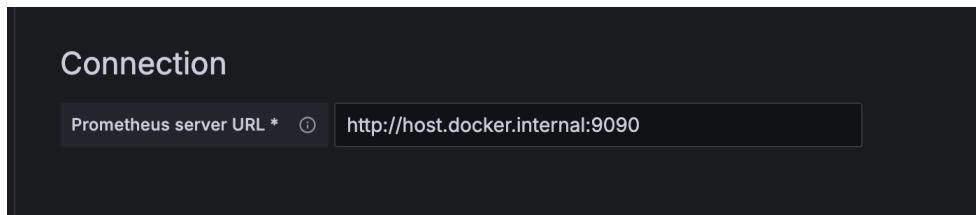


Figure 14.3 – Grafana configuration for a Prometheus data source

5. Now, you can click the **Save and test** button at the bottom of the page, which should let you know if the operation was successful. If you did everything well, Grafana should be ready to display your metrics from Prometheus.
6. From the side menu, click on **Dashboards**:

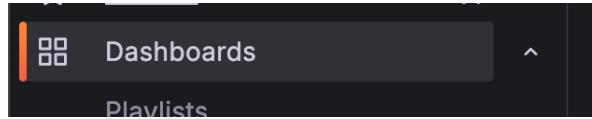


Figure 14.4 – Grafana new dashboard menu item for dashboard creation

7. On the **Dashboards** page, click on the **Create dashboard** button; you will be redirected to the visualization creation page, where you should click on the **Add visualization** button:

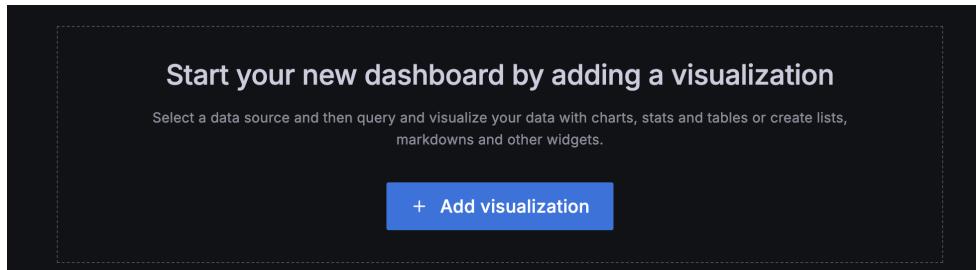


Figure 14.5 – Grafana empty dashboard page

Visualization is a core building block of a Grafana dashboard. To illustrate how to use it, let's select our Prometheus data source and some of the metrics that it already has. In the panel view, choose **Prometheus** as the data source and, in the **Metric** field, find the **process_open_fds** element and select it. Now, click on the **Run queries** button; you should see the following view:

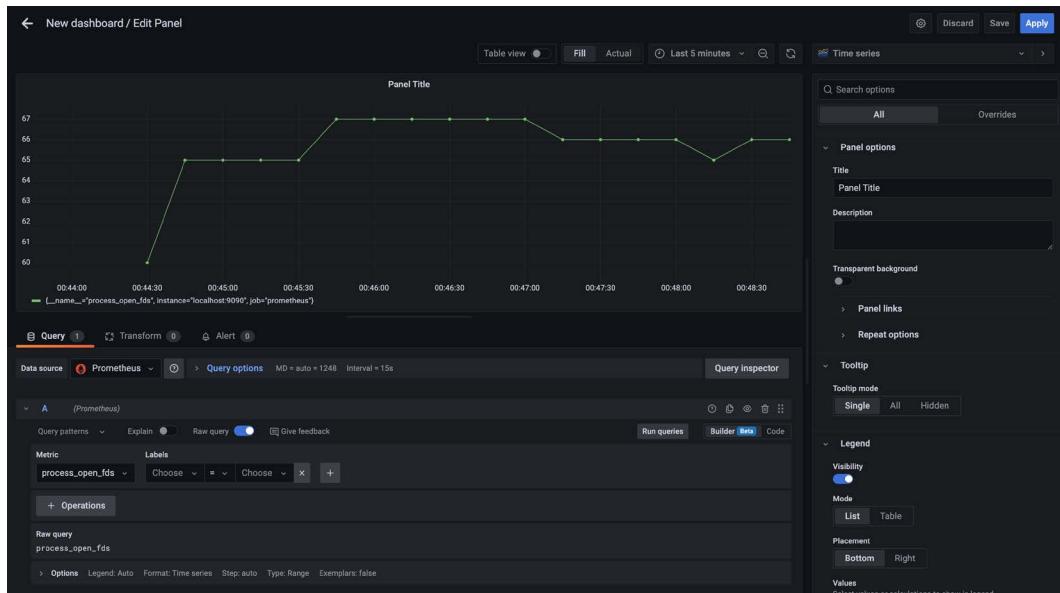


Figure 14.6 – Grafana panel view

We just configured the dashboard panel to display the process_open_fds time series stored in Prometheus. Each data point on the chart shows the value of the time series at a different time, displayed below the chart. On the right-hand panel, you can set the panel title to **Open fd count**. Now, save the dashboard by clicking the **Apply** button provided in the top menu. You will be redirected to the dashboard page.

In the top menu, you will find the **Add** button, which you can use to add a new visualization to your dashboard. If you follow the same steps that we did for the previous panel and choose the go_gc_duration_seconds metric, you will add a new panel to the dashboard that will visualize the go_gc_duration_seconds time series from Prometheus.

The resulting dashboard should look like this:



Figure 14.7 – Example Grafana dashboard

We just created an example dashboard that has two panels that display some existing Prometheus metrics. Let's now demonstrate how to visualize additional service performance metrics that are more closely related to service logic, such as its API endpoints.

Adding service-level metrics to dashboards

In *Chapter 13, Setting Up Service Alerting*, we mentioned the *four golden signals* (latency, errors, traffic, and saturation), which are commonly used for service performance analysis and monitoring. Examples of these signals are as follows:

- **Client error rate:** The ratio between client errors (such as invalid or unauthenticated requests) and all requests to the service
- **Server error rate:** The ratio between server errors (such as database write errors) and all requests to the service
- **API throughput:** Number of API requests per second/minute
- **API latency:** API request processing latency
- **CPU utilization:** Current usage of CPUs (100% means all CPUs are fully loaded)

- **Memory utilization:** Ratio between used and total memory
- **Network throughput:** Total amount of network write/read traffic per second/minute

We already demonstrated how to collect and visualize some latency and saturation metrics, such as Go garbage collection latency and file description count. Let's demonstrate how to add some service-level golden signals to our dashboard. For this, we are going to add some service-level metrics to the metadata service:

1. Open the `metadata/internal/handler/grpc/grpc.go` file and add "`github.com/uber-go/tally`" to the `imports` block of it.
2. Update the definition and the `New` function of our handler structure to the following code:

```
// Handler defines a movie metadata gRPC handler.
type Handler struct {
    gen.UnimplementedMetadataServiceServer
    ctrl *metadata.Controller
    getMetadataMetrics *EndpointMetrics
    putMetadataMetrics *EndpointMetrics
}

// New creates a new movie metadata gRPC handler.
func New(ctrl *metadata.Controller, scope tally.Scope) *Handler {
    return &Handler{
        ctrl:           ctrl,
        getMetadataMetrics: newEndpointMetrics(scope,
"GetMetadata"),
        putMetadataMetrics: newEndpointMetrics(scope,
"PutMetadata"),
    }
}
```

3. Add the following code below the `New` function inside the same file:

```
type EndpointMetrics struct {
    calls          tally.Counter
    invalidArgumentErrors tally.Counter
    notFoundErrors   tally.Counter
    internalErrors    tally.Counter
    successes       tally.Counter
}
```

```

func newEndpointMetrics(scope tally.Scope, endpoint string)
*EndpointMetrics {
    scope = scope.Tagged(map[string]string{"component": "handler",
"endpoint": endpoint})
    return &EndpointMetrics{
        calls: scope.Counter("call"),
        invalidArgumentErrors: scope.Tagged(map[string]string{
            "error": "invalid_argument"}, Counter("error")),
        notFoundErrors: scope.Tagged(map[string]string{
            "error": "not_found"}, Counter("error")),
        internalErrors: scope.Tagged(map[string]string{
            "error": "internal"}, Counter("error")),
        successes: scope.Counter("success"),
    }
}

```

- Let's update our `GetMetadata` API handler structure by adding extra metrics to it:

```

// GetMetadata returns movie metadata.
func (h *Handler) GetMetadata(ctx context.Context, req *gen.GetMetadataRequest) (*gen.GetMetadataResponse, error) {
    h.getMetadataMetrics.calls.Inc(1)
    if req == nil || req.MovieId == "" {
        h.getMetadataMetrics.invalidArgumentErrors.Inc(1)
        return nil, status.Errorf(codes.InvalidArgument, "nil req or
empty id")
    }
    m, err := h.ctrl.Get(ctx, req.MovieId)
    if err != nil && errors.Is(err, metadata.ErrNotFound) {
        h.getMetadataMetrics.notFoundErrors.Inc(1)
        return nil, status.Errorf(codes.NotFound, err.Error())
    } else if err != nil {
        h.getMetadataMetrics.internalErrors.Inc(1)
        return nil, status.Errorf(codes.Internal, err.Error())
    }
}

```

```
    h.getMetadataMetrics.successes.Inc(1)
    return &gen.GetMetadataResponse{Metadata: model.
        MetadataToProto(m)}, nil
}
```

5. Add similar changes to the PutMetadata endpoint so it increments the counter metrics depending on the result of the operation.
6. Update the `metadata/cmd/main.go` file by finding the line where we make a call to the `grpchandler.New` function and replacing it with the following code:

```
h := grpchandler.New(ctrl, scope)
```

We just added the emission of custom service-level API metrics to our metadata service: our metadata API handler now emits success and error metrics when processing each request. Based on this data, we can calculate the following types of golden metrics:

- **Client error rate:** Ratio between handler errors with the `invalid_argument` and `not_found` tags and total requests (collected as the `call` counter)
- **Server error rate:** Ratio between handler errors with an internal tag and total requests
- **API throughput:** Value of the call counter of the API handler

Note that we also introduced the `EndpointMetrics` function, which can be used to track endpoint-specific metrics for any other endpoints. You can optionally move it to some shared directory, such as `internal`, so the other endpoints can also use it to track the number of calls, as well as the number of successes and errors of different types.

Restart the metadata service and make some API requests to it:

```
grpcurl -plaintext -d '{}' localhost:8081 MetadataService/GetMetadata
grpcurl -plaintext -d '{"movie_id": "0"}' localhost:8081 MetadataService/
GetMetadata
```

Let's check whether we can access our metrics. Open the Prometheus UI by navigating to `http://localhost:9090`. Enter the call query and select **Execute** to check whether our data is there:

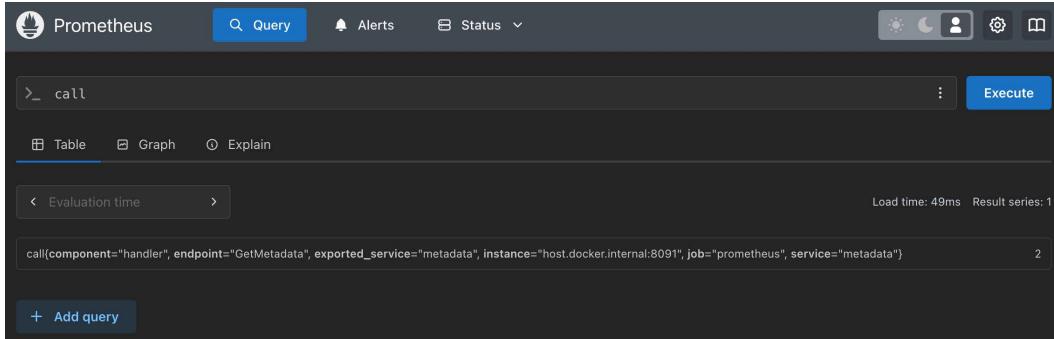


Figure 14.8 – Our service metrics in Prometheus

Note that it can take a few seconds for metrics to propagate to Prometheus. Once you see that our service metrics are there, go back to the Grafana dashboard edit view and add a visualization. On the visualization edit page, type **call** for the **Metric** name, change the panel title to **API throughput**, and click **Run queries** or **Refresh**. You should see our metric data:

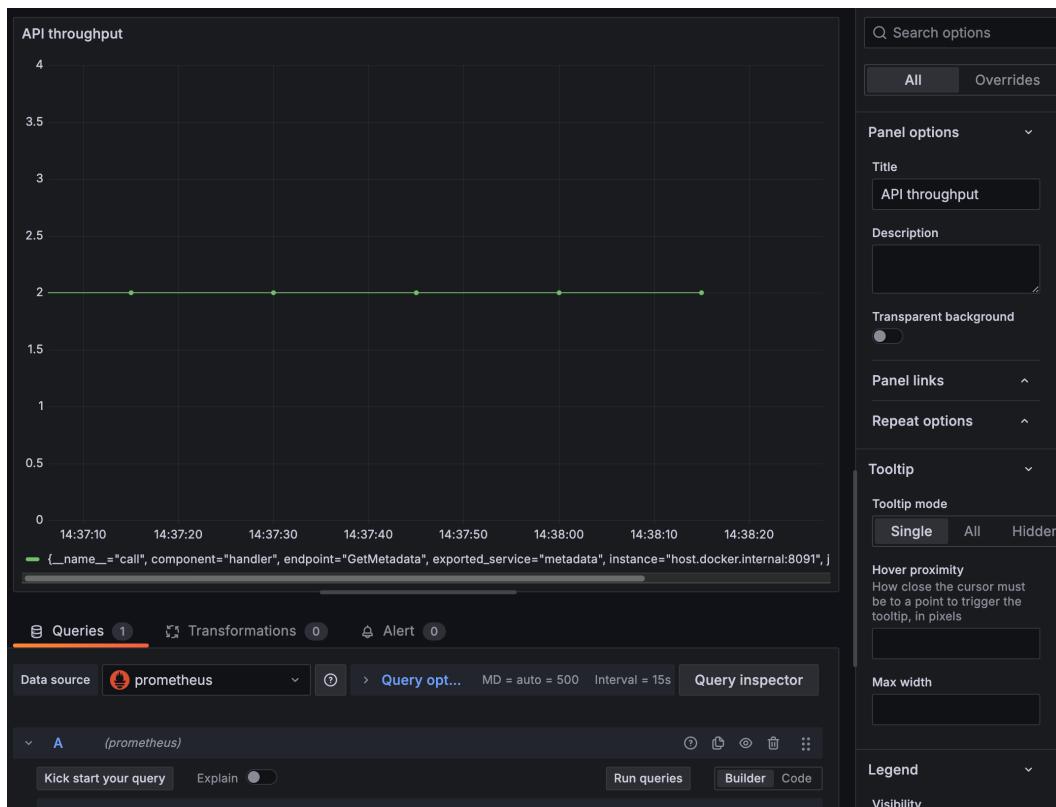


Figure 14.9 – API throughput metric panel in Grafana

We just added a new visualization panel to our dashboard that shows the total number of API requests (API throughput, or traffic from the golden metrics) to our dashboard. You can click **Back to dashboard** and see all the panels together:

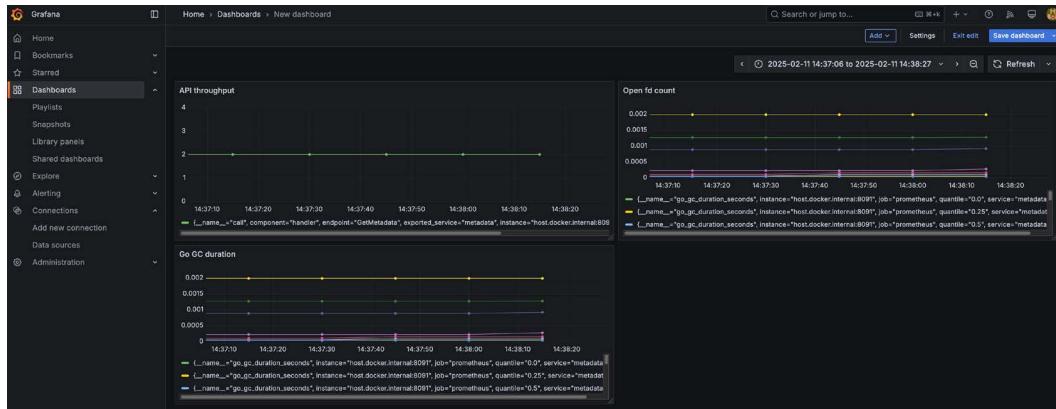


Figure 14.10 – Grafana dashboard with our metrics

You can re-arrange the dashboard visualization panels by dragging and dropping them to different parts of the screen similarly to our example. If you have done everything correctly, you just added a few service golden metrics to your dashboard! We won't illustrate how to add remaining metrics to it since this operation should be straightforward for you now – you can do this as an additional exercise instead.

Depending on the operations performed by your service (for example, database writes or reads, cache usage, Kafka consumption, or production), you may wish to include additional panels that will help you visualize your service performance. Make sure that you cover all the high-level functionality of the service so that you can visually notice any service malfunctions on your dashboards.

The Grafana tool, which we used in our example, also supports lots of different visualization options, such as displaying tables, heatmaps, numerical values, and much more. Additionally, Grafana also supports visualization of service logs and traces, providing a unified view across all service telemetry data. We will not cover these features in this chapter, but you can get familiar with them by reading the official documentation: <https://grafana.com/docs/>. Using the full power of Grafana will help you set up excellent dashboards for your services, simplifying your debugging and performance analysis.

Now that we have covered the basics of Go dashboard-based monitoring, let's move on to the next topic of this chapter, Go service profiling.

Profiling Go services

In this section, we are going to review a technique called **profiling**, which involves collecting real-time performance data of processes running your Go microservices. Profiling is a powerful technique that can help you analyze various types of resource usage data:

- **CPU usage:** Which operations used the most CPU power and what was the distribution of CPU usage among them?
- **Heap allocation:** Which operations used heap (dynamic memory allocated in Go applications) and what amount of memory was used?
- **Call graph:** In which order were service functions executed?

Profiling may help you in different situations:

- **Identifying CPU-intensive logic:** At some point, you may notice that your service is consuming most of your CPU power. To understand this problem, you can collect the CPU profile – a graph showing the CPU usage of various service components, such as individual functions. Components that consume too much CPU power may indicate various issues, such as inefficient implementations or code bugs.
- **Capturing the service memory footprint:** Similar to high CPU consumption, your service may be using too much memory (for example, to allocate too much data to the heap), resulting in occasional service crashes due to out-of-memory panics. Performing memory profiling may help you analyze the memory usage of various parts of your service and find components that have unexpectedly high memory usage.

In the following section, we are going to show how to profile Go applications using the **pprof** tool, which is a part of the Go SDK.

Profiling CPU usage using the pprof tool

Let's demonstrate first how to profile CPU usage of Go microservices using the **pprof** tool. To visualize the results of the tool, you will need to install the Graphviz library: <https://graphviz.org/>

We will use the metadata service that we implemented in the previous chapters as an example. To make our example more illustrative, we will add a function performing calculations that will increase CPU usage by our service. Then, using the **pprof** tool, we are going to find a function that causes high CPU utilization.

The steps are the following:

1. Open the `metadata/cmd/main.go` file and add the `flag`, `crypto/rand`, and `crypto/md5` packages to the `imports` block. Additionally, add the following `import` so our code supports profiling:

```
    _ "net/http/pprof"
```

2. Then, add the following code to the beginning of the main function, immediately after the logger initialization:

```
simulateCPUUpload := flag.Bool("simulatecpupload",
    false, "simulate CPU load for profiling")
flag.Parse()
if *simulateCPUUpload {
    go heavyOperation()
}

go func() {
    if err := http.ListenAndServe("localhost:6060", nil);
    err != nil {
        logger.Fatal("Failed to start profiler handler",
            zap.Error(err))
    }
}()
```

3. In the code we just added, we introduced an additional flag called `simulatecpupload` that will let us simulate a CPU-intensive operation for our profiling. We also started an HTTP handler that we will use to access the profiler data from the command line.
4. Let's add another function to the same file that will run a continuous loop and execute some CPU-intensive operations. For this, we will generate random 1,024-byte arrays and calculate their MD5 hashes (you can read about the MD5 operation in the comments of its Go package at <https://pkg.go.dev/crypto/md5>). Our selection of such logic is fully arbitrary; we could easily choose any other operation that would consume some visible part of the CPU load. Add the following code to the `main.go` file that we just updated:

```
func heavyOperation() {
    for {
        token := make([]byte, 1024)
        rand.Read(token)
        md5.New().Write(token)
    }
}
```

5. We are ready to test our profiling logic. Run the service with the `--simulatecpupload` argument:

```
go run *.go --simulatecpupload
```

6. Now, execute the following command:

```
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=5
```

The command should take five seconds to complete. If it executes successfully, the `pprof` tool will be running, as shown here:

```
Type: cpu
Time: Sep 13, 2022 at 5:37pm (+05)
Duration: 5.14s, Total samples = 4.42s (85.92%)
Entering interactive mode (type "help" for commands,
"o" for options)
(pprof)
```

Type `web` in the command prompt of the tool and press *Enter*. If everything worked well, you will be redirected to a browser window containing a CPU profile graph:

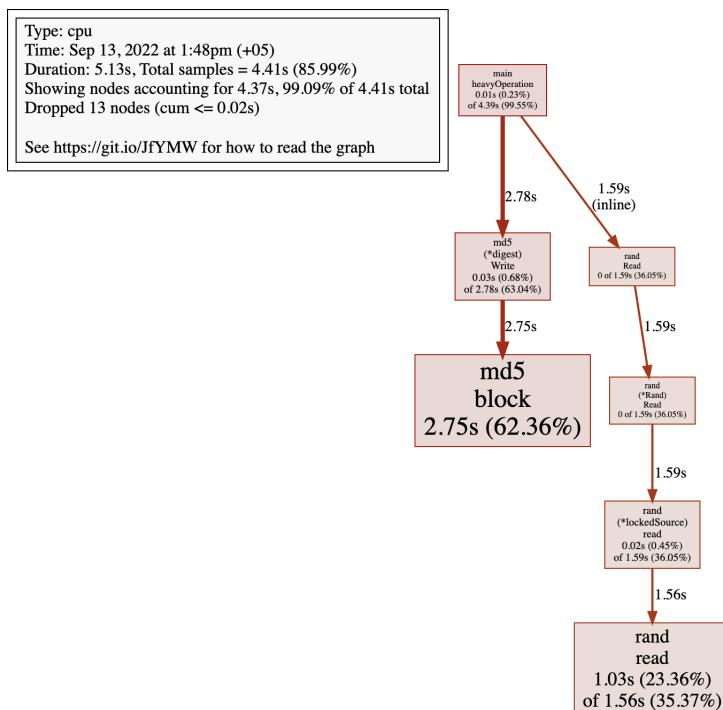


Figure 14.11 – Go CPU profile example

Let's walk through the data in the graph to understand how to interpret it. Each node on the graph includes the following data:

- Package name
- Function name
- Elapsed time and the total time of the execution

For example, the `heavyOperation` function took just 0.01 seconds, but all the operations that were executed in it (including all function calls inside it) took 4.39 seconds, taking most of the elapsed time.

If you walk through the graph, you will see the distribution of the elapsed time by sub-operations. In our case, `heavyOperation` executed two functions that were recorded by the CPU profiler: `md5.Write` and `rand.Read`. The `md5.Write` function took 2.78 seconds in total, while `rand.Read` took 1.59 seconds of the execution time. Level by level, you can analyze the calls and find the CPU-intensive functions.

When working with the CPU profiler data, notice the functions that take the most processing time. To help you find them, such functions are illustrated as larger rectangles on the profiler output graph. If you notice that some functions have unexpectedly high processing time, spend some time analyzing their code to see whether there is any opportunity to optimize them.

Now, let's also illustrate how to visualize **flame graphs** of CPU profiles. Flame graphs are visual structures that use different colors to indicate how frequently each code block is executed. To get our CPU flame graphs, perform the following steps:

1. Re-collect the CPU profile and save it into a separate file by running the following:

```
go tool pprof -raw -output=cpu.prof \
  'http://localhost:6060/debug/pprof/profile?seconds=10'
```

2. Copy the `flamegraph.pl` and `stackcollapse-go.pl` files mentioned in the *Technical requirements* section to the directory containing the captured `cpu.prof` profile and run the following command:

```
perl stackcollapse-go.pl cpu.prof | perl flamegraph.pl > flame.svg
```

If everything goes correctly, you should find the `flame.svg` file containing our CPU flame graph:

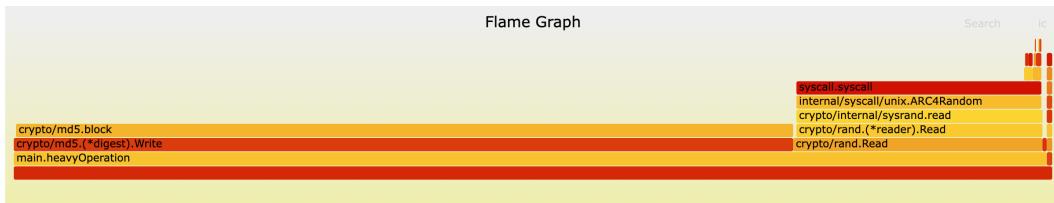


Figure 14.12 – CPU flame graph

It is easy to spot our `heavyOperation` call on the flame graph. We can also see that the most frequent operation it executes is a call to the `md5.Write` function. Notice how easy it is to work with flame graphs and visually spot all functions consuming the most CPU load. You can now leverage these CPU profiling techniques to optimize your microservices and ensure there are no unexpected code paths that perform unexpectedly high amounts of CPU operations.

Profiling heap memory usage using the pprof tool

Now, let's illustrate another example of profiler data. This time, we will be capturing a **heap profile** – a profile showing dynamic memory allocation by a Go process. Run the following command:

```
go tool pprof http://localhost:6060/debug/pprof/heap
```

Similar to the previous example, successfully executing this command should run the `pprof` tool, where we can execute a web command. The result will contain the following graph:

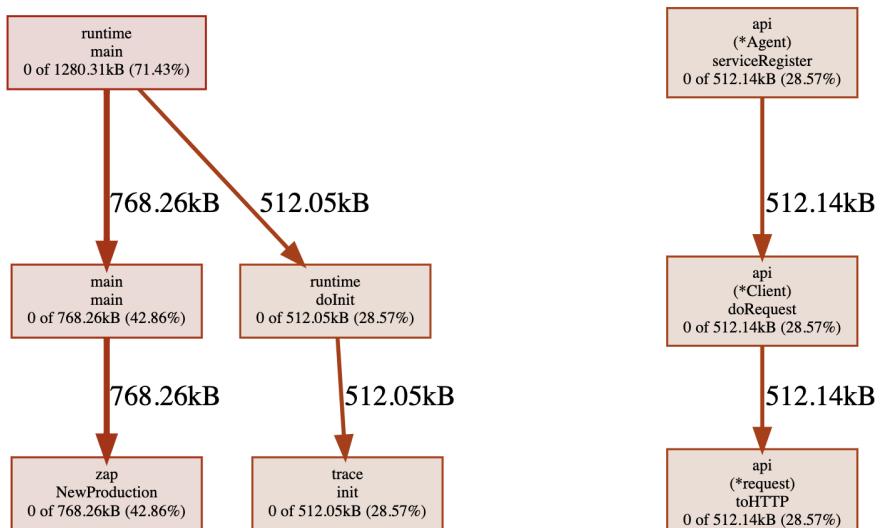


Figure 14.13 – Go heap profile example

This diagram is similar to the CPU profile. The last line inside each node shows the ratio between the memory used by the function and the total heap memory allocated by the process.

In our example, three high-level operations are consuming the heap memory:

- `api.serviceRegister`: A function that registers a service via the Consul API
- `zap.NewProduction`: Logger initialization via the `zap` library
- `trace.init`: Initializes the tracing logic

Looking at the heap profiler data, it's easy to find functions allocating an unexpectedly high amount of heap memory. Similar to CPU profiler graphs, heap profilers display the functions that have the highest heap allocation as larger rectangles, making it easier to visualize the most memory-consuming functions.

I suggest that you practice with the `pprof` tool and try the other operations it provides. Being able to profile Go applications is a highly valuable skill in production debugging that should help you optimize your services and solve different performance-related issues. The following are some other useful tips for profiling Go services:

You can profile Go tests without adding any extra logic to your code. Running the `go test` command with the `-cpuprofile` and `-memprofile` flags will capture the CPU and memory profiles of your logic, respectively.

The `top` command of the `pprof` tool is a convenient way of showing the top memory consumers. There is also the `top10` command, which shows the top 10 memory consumers.

Using the `rroutine` mode of the `pprof` tool, you can get a profile of all used goroutines, as well as their stack traces.

Summary

In this chapter, we discussed the key aspects of performance monitoring of microservices. First, you learned how to create service performance dashboards using a popular open source dashboarding tool, Grafana. Then, we demonstrated how to capture and analyze the CPU and memory profiles of Go microservices to identify which parts of the service logic consume the most CPU and memory resources. The knowledge you gained should help you establish mechanisms for monitoring the performance of your microservices to identify scalability bottlenecks, optimize resource usage, and ensure the reliability and scalability of your Go microservices.

This chapter ends *Part 3* of the book. In the next chapter, we are going to review some advanced topics of Go microservice development, such as common distributed system challenges that you might face when building more complex Go applications.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Profiling Go Programs: <https://go.dev/blog/pprof>
- Grafana documentation: <https://grafana.com/docs/>
- FlameGraph repository and documentation: <https://github.com/brendangregg/FlameGraph>

Part 4

Advanced Topics

This part covers some advanced topics of Go microservice development, including distributed system scenarios, data validation, static code analysis, and streaming communication. You will learn how to implement common distributed system scenarios, such as leader election, how to perform static code checks and store service ownership data, how to validate microservice data, and how to implement streaming API clients and servers. The chapter includes a variety of examples and best practices related to these topics and provides an overview of common tooling that might help you along the way.

This part of the book includes the following chapters:

- *Chapter 15, Implementing Distributed System Scenarios*
- *Chapter 16, Advanced Topics*

15

Implementing Distributed System Scenarios

So far, we have covered most high-level topics related to microservice development. You have learned how to use service discovery and deploy your services, establish synchronous and asynchronous communication, and collect service telemetry data and use it to drive the reliability of the system. In the last part of the book, we are going to dive into more advanced scenarios and problems of microservice development that may arise when building complex applications and systems. In this chapter, we will learn how microservices can coordinate with each other to solve some problems such as allocating tasks between themselves.

We are going to cover the following topics:

- Introduction to distributed system problems
- Distributed system tools
- Implementing leader election with HashiCorp Consul
- Distributed system best practices

We will use the microservices we created in the previous chapter to illustrate how to implement some scenarios covered in the chapter. Now, let's move on to the overview of common distributed system problems.

Technical requirements

To complete this chapter, you will need Go 1.18 or above. Additionally, you will need the following tools:

- **Docker:** <https://www.docker.com>

You can find the code examples for this chapter on GitHub: <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter15>

Introduction to distributed system problems

In microservice applications, there can be scenarios where multiple services need to coordinate their actions to perform some types of tasks. Let's imagine that we have a collection of large video files (for example, on a network drive within our system) and we want to build service logic for processing these files by converting them from one format to another. To avoid duplicate work, we want to coordinate the execution of processing tasks so that not more than one service instance processes the same data at each moment in time.

One way to tackle such a problem is to introduce some logic that would orchestrate the execution of such tasks (for example, a separate component that would select a random service instance and assign it some specific work to do). However, such an approach has some challenges:

- **Centralization and single point of failure:** Orchestration logic would become “centralized” within another component, which would create a single point of failure in our system: if the orchestrator component becomes unavailable, our services would not be able to assign work by themselves
- **Scalability bottleneck:** If the number of processing tasks or service instances gets too large, the centralized orchestrator may become a bottleneck, limiting the system’s ability to scale further
- **Latency overhead:** If all task assignments need go through the orchestrator, it can introduce additional processing latency

There is, however, a way to design such a system that would not require an extra component that orchestrates work execution. Such an approach would be based on coordination between our service instances themselves: by talking to each other, they would be able to assign the work between themselves without any overlaps, like members of a team dividing the work between themselves. The approach that we are talking about is based on a concept called **consensus**.

Consensus in distributed systems

Consensus can be defined as an agreement within a system, such as a set of microservices, to reach some specific goal. In our case with video processing tasks, services would need to form a consensus on assigning processing work between themselves, so each instance could know its role in the system: to process specific tasks or to remain idle, waiting for further instructions.

There are some real-life examples of achieving consensus between multiple entities. Imagine a team of workers allocating work between themselves. To do so, they can schedule a meeting and split tasks between each other, defining the clear responsibilities of each person. Another example is political elections: we can define specific rules (such as voting and counting the majority of votes) to elect defined candidates to new roles and positions.

The rules for reaching consensus are called **consensus protocols**. There are some popular software consensus protocols that have been used for decades for achieving consensus in various types of software, including service discovery tools and databases. Let's review some of the common protocols.

Paxos

Paxos is a family of consensus protocols that was introduced in 1989 but remains broadly used in modern-day distributed systems. Each of the Paxos protocols offers some trade-offs (for example, higher fault tolerance with high latency, or the opposite) and typically consists of a sequence of phases. A high-level, simplified view of the Paxos algorithm involves the following steps:

Leader proposal phase

- A node (called the **proposer**) wants to propose a value (for example, a proposal to allocate work to a specific instance)
- It sends a prepare request to a majority of nodes (**acceptors**)
- Acceptors respond with a promise to not accept proposals with earlier proposal ID numbers (which are always increasing)

Consensus phase

- If the proposer gets enough promises, it sends an accept request with its value
- Acceptors acknowledge if they haven't accepted a higher proposal
- Once a majority accepts, the value is committed and learned by all nodes

It is important to note that our version of the Paxos algorithm is very simplified, and in practice, it has lots of challenges and workarounds to make it work in distributed systems. There are multiple possible reasons for this:

- **Network partitions:** It is always possible that a part of the network will become temporarily unavailable, leaving your system without a few instances participating in the consensus.
- **Message loss and message delays:** Network messages can occasionally get delayed or even be lost in case of various network issues. Consensus protocols need to be able to handle such issues.
- **Node failures:** Services may become unavailable and leave the consensus process; however, the entire system must still remain operational.

Due to these reasons, achieving consensus in a distributed system is generally considered a hard problem, and developers have spent years and even decades making various optimizations and adjustments to the Paxos protocols. Due to these optimizations and improvements, the Paxos protocol family still remains one of the best choices for consensus management.

Raft

Raft is another consensus protocol that was designed for simplicity and understandability compared to Paxos, which has lots of complex details in its implementation. Unlike Paxos, which allows any node to propose data changes, the Raft protocol always assigns a single node to coordinate changes and drive the consensus process. This allows Raft to have a much simpler implementation, however, might result in quick downtimes if such nodes become unavailable.

Still, Raft is widely used across the industry, powering such tools as Kubernetes, HashiCorp Consul, and many more. Its implementations exist for multiple languages, which you can find in this repository: <https://raft.github.io/>

As we have reviewed some basics of distributed consensus, let's proceed to some practical use cases of consensus that you might face in your microservice applications.

Distributed locking and leader election

Among the most common use cases of distributed system consensus are **distributed locks** and **leader election**. Let's start with distributed locks first.

Distributed locks are data structures that help to ensure that only one process, such as a service instance, has access to a specified resource at a time. In our example with video processing logic, we may assume that only one instance can process a video file (or even a group of files, such as an entire directory) at a given moment of time, and associate with it a **lock**, that would allow only one instance to use it, as illustrated in the following diagram:

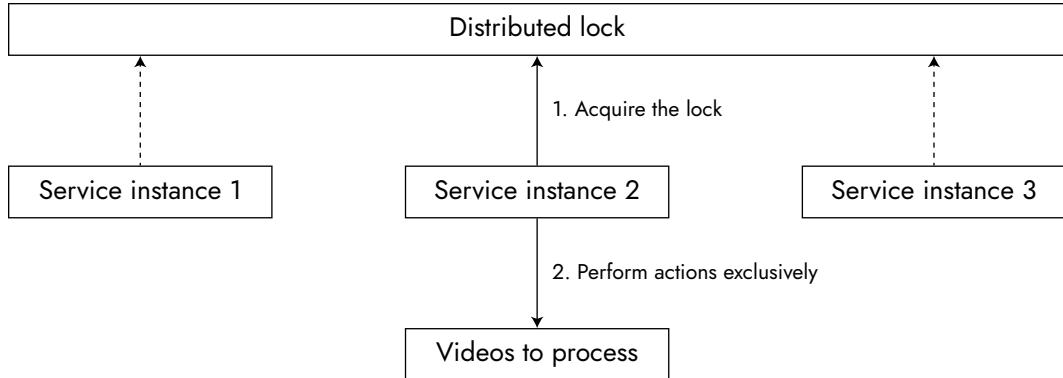


Figure 15.1 – Usage of a distributed lock to guard access to a single resource, such as videos to process. Only one service instance can acquire and keep the lock until it is released

In our scenario, all service instances would continuously try to acquire the lock (for example, by performing this operation every few seconds). Only the instance that acquires it would be able to perform specific operations, such as processing video files.

The solution that we just described can be fully decentralized: instead of requiring a component that would coordinate the work between service instances, our service instances could use a consensus protocol, such as Paxos or Raft, to coordinate work by themselves.

Logically, a distributed lock is just a shared value, or **shared state** between our instances. It can be represented as a simple key-value pair in pseudocode:

```
/locks/video.processing: "service-instance-2"
```

You don't necessarily need to lock the entire collection of resources, such as a set of video files across multiple directories. Instead, each directory might get its own lock, so service instances would monitor for a group of locks, as illustrated in the following diagram:

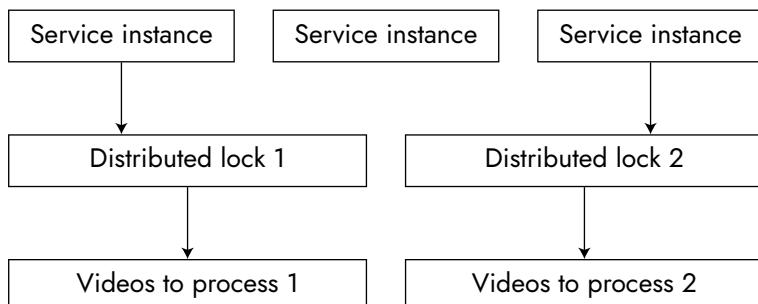


Figure 15.2 – Usage of multiple locks linked to logical groups of resources

In our diagram, there are three service instances competing for two shared resources associated with unique groups of videos to process. An additional service instance is kept for redundancy: if one of our service instances becomes unavailable, we would still be able to process all videos.

Distributed locks are closely related to the concept of **distributed leader election** – electing a leader among a group of entities, such as service instances. In general, both provide similar functionality: the ability to achieve consensus on what entities are able to access a shared resource or perform specific actions within the system, such as processing some tasks or performing coordination of them among the other instances. In the Raft protocol, nodes continuously elect a leader that coordinates updates across them; in many databases, there is a leader node that accepts all incoming updates and sends them to nodes called **followers**.

To solidify our learnings, let's proceed to the practical part of the chapter and review how we can use distributed locks in our Go microservices. Prior to the practical examples, we are going to briefly review the common tools that could provide us with consensus-based mechanisms that we could use in our logic.

Distributed system tools

It is generally recommended to use well-established libraries and tools to deal with consensus in distributed systems since there are many edge cases and error scenarios that can be challenging to implement and test. Fortunately, there are some open source solutions that can be used in a broad range of applications, such as microservices.

Apache Zookeeper

Apache Zookeeper is among the first widely used open source implementations of the Paxos protocol, which still powers lots of tools, including multiple versions of Apache Kafka. Written in Java and requiring Java runtime to execute, it still offers Go libraries, such as the official go-zookeeper library: <https://github.com/go-zookeeper/zk>

Apache Zookeeper can be seen as a form of a distributed file system, where records (called **znodes**) have keys (similar to file names) and associated values. Using this data with a combination of Paxos-based consensus logic, Apache Zookeeper offers the following high-level APIs:

- Distributed key-value storage
- Leader election and distributed locks
- Distributed transactions – atomic execution of all operations

Note

Apache Zookeeper has been well-maintained and stable for more than 15 years; however, there is a trend toward transitioning to Raft-based protocols. Apache Kafka has been transitioning to its own version of the Raft protocol (KRaft) since version 3.0.

etcd

etcd is another solution that uses the Raft protocol to achieve distributed consensus among its nodes. Similarly to Apache Zookeeper and HashiCorp Consul, it provides read, update, and watch APIs, as well as the ability to use distributed locks. Its API is leveraged by various tools, including Kubernetes.

Note

Due to the complex nature of consensus protocols and lots of interaction steps being required, you can practically store just gigabytes of data until you start experiencing a significant increase in latency. There are some systems that overcome such limitations, such as Google Spanner; however, there are various trade-offs that allow this. If you are interested, you can find useful information on this topic in these whitepapers: <https://cloud.google.com/spanner/docs/whitepapers>

HashiCorp Consul

HashiCorp Consul is another popular tool leveraging distributed consensus protocols. It uses the Raft protocol and provides features similar to Apache Zookeeper, including distributed locks and leader election, storing key-value metadata, and watching for its updates.

You can find more information on HashiCorp APIs in its documentation: <https://developer.hashicorp.com/consul/api-docs>

In the following section, we are going to demonstrate how to use HashiCorp Consul to implement distributed locking for our Go microservices. Technically, we could pick any of the preceding solutions, such as etcd; however, our services already use HashiCorp Consul, so it would be convenient to reuse it for yet another use case.

Implementing distributed locking with HashiCorp Consul

At the beginning of the chapter, we mentioned an example of a video data processing pipeline, where multiple service instances were attempting to process shared video files. In our example, the basic requirement was to avoid duplicate processing, so we used distributed locks to show how to limit the processing of the same files to one service instance at a time.

Let's illustrate how to implement this scenario using our movie service as an example. The steps are the following:

1. Inside our movie service code, create a directory called `internal/lock/consul`. This directory will contain the distributed lock implementation that will use the HashiCorp Consul API. Add the file called `consul.go` with the following contents:

```
package consul

import (
    "context"
    "errors"

    "github.com/hashicorp/consul/api"
    "go.uber.org/zap"
)

// LockProvider defines a Consul-based Lock provider.
type LockProvider struct {
    client *api.Client
    logger *zap.Logger
    sessionID string
}

// NewLockProvider creates a Consul-based Lock provider.
func NewLockProvider(logger *zap.Logger, addr string) (*LockProvider, error) {
    config := api.DefaultConfig()
    config.Address = addr
    client, err := api.NewClient(config)
    if err != nil {
```

```
        return nil, err
    }
    sessionID, _, err := p.client.Session().Create(&api.SessionEntry{
        TTL: "10s", // Lock expires if our processor instance gets
        unavailable after 10s.
    }, nil)
    if err != nil {
        return nil, err
    }
    return &LockProvider{client, logger, sessionID}, nil
}
```

2. After the code that we just added, add the following logic, which performs distributed lock acquisition:

```
// Acquire acquires a Lock for the given key. If successful, it
// returns a function that releases the lock.
func (p *LockProvider) Acquire(ctx context.Context, key string)
(bool, func(), error) {
    kvPair := &api.KVPair{
        Key:     key,
        Value:   []byte("locked"),
        Session: sessionID,
    }
    p.logger.Info("Acquiring a lock", zap.String("key", key), zap.
String("sessionID", sessionID))
    acquired, _, err := p.client.KV().Acquire(kvPair, nil)
    if err != nil {
        return false, nil, err
    }
    if !acquired {
        return false, nil, errors.New("failed to acquire lock")
    }
    return true, func() error {
        return releaseLock(p, key, sessionID, kvPair)
    }, nil
}
```

3. Finally, let's add the remaining functions for closing the session and releasing the lock:

```
// Close closes the Lock provider.
func (p *LockProvider) Close() error {
    p.logger.Info("Destroying a session", zap.String("sessionID",
    p.sessionID))
    if _, err := p.client.Session().Destroy(p.sessionID, nil); err
    != nil {
        return err
    }
    return nil
}

func releaseLock(p *LockProvider, key string, sessionID string,
kvPair *api.KVPair) error {
    p.logger.Info("Releasing a lock", zap.String("key", key), zap.
String("sessionID", sessionID))
    _, _, err := p.client.KV().Release(kvPair, nil)
    return err
}
```

4. Now, let's implement the logic that will use our locking functionality and perform any processing once the lock is acquired. For this, we are going to create a directory called internal/processor inside the movie service source directory and add the processor.go file with the following contents:

```
package processor

import (
    "context"
    "errors"
    "time"

    "go.uber.org/zap"
)

// LockProvider defines a distributed Lock provider.
type LockProvider interface {
    Acquire(ctx context.Context, key string) (bool, func() error,
error)
```

```
}

// Processor defines a movie processor.
type Processor struct {
    logger      *zap.Logger
    lockProvider LockProvider
}

// New creates a new movie processor.
func New(logger *zap.Logger, lockProvider LockProvider) *Processor {
    return &Processor{logger, lockProvider}
}
```

5. Add logic to the same file that starts our processor and performs lock acquisition:

```
// Start starts the movie processor.
func (p *Processor) Start(ctx context.Context) error {
    p.logger.Info("Starting the movie processor")
    for {
        select {
        case <-ctx.Done():
            return ctx.Err()
        default:
        }
        _, release, err := p.lockProvider.Acquire(ctx, "locks/
service/movie/processor")
        if err != nil {
            p.logger.Error("Unable to acquire lock, retrying in 10
seconds", zap.Error(err))
            time.Sleep(time.Second * 10)
            continue
        }
        p.logger.Info("Lock has been acquired, starting processing")
        if err := p.process(ctx); err != nil {
            p.logger.Error("Processing error", zap.Error(err))
        } else {
            p.logger.Info("Processing completed successfully")
        }
        p.logger.Info("Releasing the lock")
```

```

        if err := release(); err != nil {
            p.logger.Error("Failed to release the lock", zap.
Error(err))
        }
    }
}

```

6. Our file is missing a function for doing actual processing. For simplicity, we are not going to implement actual video conversion, but just simulate this in our logic (the code that could potentially perform video conversion is included in the comment):

```

func (p *Processor) process(ctx context.Context) error {
    const timeout = 5 * time.Minute
    ctx, cancel := context.WithTimeout(ctx, timeout)
    defer cancel()
    // Here we would process the movie videos.
    // We omit the implementation of movie processing logic for
    // simplicity and just use time.Sleep to simulate processing.
    // Example of some real processing logic would be:
    // return exec.Command("bash", "-c", 'for file in *.mp4; do
    ffmpeg -i "$file" "${file%.mp4}.mp3"; done').Run()
    time.Sleep(time.Second * 10)
    return nil
}

```

7. Let's plug our processor into the `main.go` file of the movie service. Add the following import to its `imports` block:

```
consullock "movieexample.com/movie/internal/lock/consul"
```

8. Finally, after the `registry` definition inside the same file, add the following code:

```

lockProvider, err := consullock.NewLockProvider(logger, cfg.
ServiceDiscovery.Consul.Address)
if err != nil {
    logger.Fatal("Failed to create lock provider", zap.
Error(err))
}
defer lockProvider.Close()
processor := processor.New(logger, lockProvider)
go processor.Start(ctx)

```

Let's review the changes that we just made:

1. First, we implemented a Consul-based lock provider that provides a distributed lock and a function to release it inside the `Acquire` function. In our code, we first created a Consul session – a required step to perform further locking. Then, we called the Consul `Acquire` API, which performed actual lock acquisition. In the case of successful lock acquisition, our code also returns a function that allows the lock to be released once processing is done.
 2. After the Consul-based lock implementation, we added a `Processor` structure that performs a continuous loop of the following operations:
 - a. Attempt to acquire the lock or return if the parent context is canceled (for example, when the application is shut down)
 - b. If the lock is acquired, call the `process` function that simulates the actual execution of the processing logic. Release the lock once processing is done and go back to the first step.
 3. Finally, we plugged our processor into the `main.go` file of the movie service and scheduled the execution of its `Start` function in a separate goroutine, so processing would happen in parallel to the rest of the `main` function.



Note

It is useful to set an upper time limit for any actions performed on locked resources, such as video processing in our example. This prevents situations where the processing logic gets stuck and other service instances are unable to help because the lock is not released.

Each movie service instance will run its own processor and compete for a distributed lock that will be shared among them via HashiCorp Consul. You might try running multiple movie service instances to test our logic. If you run a single movie service instance with the new code, you will see the following logs being written continuously:

```
{"level":"info","ts":1738917402,"caller":"consul/consul.  
go:42","msg":"Acquiring a lock","key":"locks/service/movie/  
processor","sessionID":"1ce80eef-4536-7dc5-c8ee-6ad673715b81"}  
{ "level": "info", "ts": 1738917402, "caller": "processor/processor.  
go:40", "msg": "Lock has been acquired, starting processing" }  
{ "level": "info", "ts": 1738917402, "caller": "processor/processor.  
go:40", "msg": "Processing movie: Star Wars: Episode IV - A New Hope" }
```

```
go:44", "msg": "Processing completed successfully"}  
{"level": "info", "ts": 1738917402, "caller": "processor/processor.  
go:46", "msg": "Releasing the lock"}  
{"level": "info", "ts": 1738917402, "caller": "consul/consul.  
go:51", "msg": "Releasing a lock", "key": "locks/service/movie/  
processor", "sessionID": "1ce80eef-4536-7dc5-c8ee-6ad673715b81"}
```

If you run a second instance of the movie service, you will see similar logs:

```
{"level": "info", "ts": 1738924039, "caller": "consul/consul.  
go:42", "msg": "Acquiring a lock", "key": "locks/service/movie/  
processor", "sessionID": "05fd7b3a-18d4-92e1-60c6-1510ec91852d"}  
{"level": "error", "ts": 1738924039, "caller": "processor/  
processor.go:41", "msg": "Unable to acquire lock, retrying in 10  
seconds", "error": "failed to acquire lock"}
```

We just demonstrated how multiple service instances could coordinate via HashiCorp Consul-based distributed locks to avoid duplicate execution of the same task: only one service instance was able to acquire the lock and run processing logic at any given moment of time. We used HashiCorp Consul to save our time implementing distributed consensus logic for distributed locking; however, it is possible to use a dedicated Raft library such as <https://github.com/hashicorp/raft>, to avoid using an additional component, such as Consul.

Note



It is strongly recommended not to implement your own consensus-based logic or use low-level Raft or Paxos implementations unless absolutely required: high-level systems such as Consul and etcd will save you weeks and months of work by providing well-established and maintained libraries without any need to configure them for myriads of edge cases and failure scenarios.

We have completed the practical part of this chapter and are now ready to review the best practices for working with distributed systems in the context of Go microservices.

Distributed system best practices

In this section, we are going to briefly review some of the best practices dealing with distributed system tools and solutions in microservice development. As with any technology, distributed system tooling comes at a cost, and it is important to understand the trade-offs when designing your microservice applications.

Build systems with minimal necessary coordination between components

Earlier in the chapter, we discussed some possible challenges and limitations of building consensus-based mechanisms for microservice applications. Each of the popular consensus protocols, such as Paxos and Raft, involves a lot of communication between system nodes, as well as adding extra complexity to the overall system. Like a group of people trying to reach a consensus, a distributed system might be prone to communication failures, as well as taking a much longer time to get to the consensus state.

When designing your systems, you should always ask yourself whether the level of complexity introduced by your solutions is justifiable enough for the problem you are trying to solve. In some cases, centralized orchestration can be much simpler to implement and maintain than decentralized consensus-based coordination. The decision of whether to use distributed system protocols or tools should be based on the complexity and scale of your problems.

One effective approach to reducing coordination overhead is to use idempotent actions – actions, that are safe to perform concurrently or repeatedly, reducing the reliance on distributed locks or leader election. In our example, video processing operations could be generally idempotent: the worst-case scenario would be just duplicate processing of the same file in parallel. If the possibility of doing duplicate work, such as processing the same video files, is low enough (for example, if video processing operations happen rarely), it might be tolerable to let services occasionally perform duplicate work while minimizing such a possibility as much as possible (for example, executing processing logic by different instances at random moments in time or with random intervals, so they won't overlap as much as possible).

Another strategy is to delegate coordination to external services where appropriate. Maintaining your own distributed coordination logic might be challenging due to various performance and debugging challenges, so if you face a problem that requires a distributed consensus, consider searching for specific tooling (such as dedicated databases or services such as Google Spanner).

Ultimately, minimizing coordination leads to more scalable, resilient, and maintainable micro-service architectures.

Avoid distributed transactions whenever possible and find the right level of consistency for your data

You should be careful when using other complex features provided by distributed system tools, such as **distributed transactions**. Distributed transactions can help to update multiple values atomically (for example, changing the user's balance in one system and gaining access to the movie when a user makes a purchase in another system); however, they come at a cost of much higher latency and system complexity.

As an alternative, there are often ways to simplify the logic behind your operations. In the movie purchase example that we just described, the entire flow of operations can be instead re-modeled as a sequence of independent actions:

1. Charge the user's account for the movie. In case of success, emit a success event (for example, a Kafka message).
2. Upon receiving a success event for step 1, grant access to the movie.

Such a pattern is called **Saga** and allows you to perform changes to multiple entities without the need for low distributed transactions. Separate changes of different types of data might not be fully atomic, but this might be tolerable depending on your system requirements. If it takes just a few seconds between a user being charged for a purchase and gaining access to a purchased movie, such a trade-off might be completely reasonable.



Note

For multi-step scenarios, it is important to keep in mind operation error handling logic (for example, undo the payment operation or issue a refund if the operation can't be completed in one of the following steps due to any non-resolvable issues).

The data consistency model that we just described is called **eventual consistency** and is widely used in different distributed systems to achieve higher system performance without adding extra complexity, such as complex distributed transactions.

Summary

In this chapter, we reviewed common distributed system challenges and solutions, such as distributed locking, leader election, and distributed transactions. You learned about common protocols for establishing consensus in distributed systems, some of their limitations, as well as popular tools for implementing distributed system scenarios in your microservices. At the end of the chapter, we reviewed some of the best practices for working with distributed system tools in Go microservices.

In the next chapter, we are going to cover a broad range of other advanced topics of Go microservice development, which should solidify your skills and prepare you for solving lots of interesting real-world microservice development challenges.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Awesome Distributed Systems: <https://github.com/FedericoPonzi/awesome-distributed-systems>
- Testing Distributed Systems: <https://asatarin.github.io/testing-distributed-systems/>

16

Advanced Topics

If you are reading this chapter – congratulations, you have reached the very final chapter of this book! We have discussed many topics related to microservice development, but some remain that are important to cover. The topics in this chapter span many areas, from static code analysis and validation to frameworks and service ownership metadata. You may find these topics useful at various points in time: some of them will be helpful once you have working services serving production traffic, while others will be useful while your services are still in active development.

In this chapter, we will cover the following topics:

- Static analysis of Go service code
- Implementing data validation
- Implementing streaming APIs
- Frameworks
- Storing microservice ownership data

Let's proceed to the first section of this chapter, which covers service profiling.

Technical requirements

To complete this chapter, you will need Go 1.18+ or above. Additionally, you will need the following tools:

- **Graphviz:** <https://graphviz.org>
- **Docker:** <https://www.docker.com>

You can find the code examples for this chapter on GitHub: <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter16>

Static analysis of Go service code

In previous chapters, we covered many ways of testing and monitoring Go services to ensure high reliability; however, we still haven't used a powerful technique, called **static analysis**. Static analysis is a process of examining code to identify potential issues in it, including (but not limited to) the following:

- Improper uses of defer statements
- Invalid error handling (for example, invalid use of the `errors.As` function)
- Mistakes using HTTP responses
- Invalid `Printf` formatting
- Incorrect structured logging calls
- Unreachable code blocks

The power of static analysis lies in its ability to find potential problems in Go code even before executing it or running unit and integration tests. Static code analysis checks all code structures and function calls and identifies issues that would often be hard to detect otherwise (for example, not using the `defer` statement correctly, causing memory leaks sometime after the code execution).

There are some existing tools that offer static analysis of Go code. First, let's take a look at the `go vet` tool, which is a part of the official Go SDK: <https://pkg.go.dev/cmd/vet>. This tool includes about 30 different code checks, including the ones that we mentioned earlier. To run the tool, navigate to the `src` directory of our code, and simply run the following command:

```
go vet ./...
```

If you correctly followed the steps in the previous chapters, the output of the first run should be empty – there are no issues in our code that the `go vet` tool detected.

Let's check that the tool can identify potential problems by temporarily adding some issues to our code. For this, let's open the `movie/internal/gateway/metadata/http/metadata.go` file and intentionally use incorrect formatting by skipping the `print` argument:

```
log.Printf("Calling metadata service. Request: GET %s")
```

Now, when we run the `go vet` tool again, we should get the following output:

```
# movieexample.com/movie/internal/gateway/metadata/http  
movie/internal/gateway/metadata/http/metadata.go:33:2: log.Printf format  
%s reads arg #1, but call has 0 args
```

We did confirm that the static code analysis tool worked – the issue we just found would be hard to notice with any type of test because it wasn't critical to code execution and would just affect the way the message was logged.

Let's remove the change that we just made and proceed to explore a more powerful static analysis tool, called **staticcheck**: <https://staticcheck.dev/>. The staticcheck tool was created with the goal of supporting a wider set of code checks and currently includes more than 150 different rules – more than 5 times bigger than go vet.

First, let's install the staticcheck tool locally:

```
go install honnef.co/go/tools/cmd/staticcheck@latest
```

Once you run this command, try executing it by running the following command in your terminal:

```
staticcheck
```

If your terminal is not able to locate the newly installed tool, you might need to include the Go path. On Linux and Mac, the command would be the following:

```
PATH=$HOME/go/bin:$PATH
```

Once you verify that the staticcheck tool works, let's run it inside the `src` directory of our code:

```
staticcheck ./...
```

Unlike our first go vet execution, staticcheck should be able to identify a wider set of issues, so you should see a similar set of messages in the output:

```
internal/grpcutil/grpcutil.go:19:9: grpc.Dial is deprecated: use  
NewClient instead. Will be supported throughout 1.x. (SA1019)  
internal/grpcutil/grpcutil.go:22:29: otelgrpc.UnaryClientInterceptor is  
deprecated: Use [NewClientHandler] instead. (SA1019)  
metadata/cmd/main.go:38:3: rand.Read has been deprecated since Go 1.20  
because it shouldn't be used: For almost all use cases, [crypto/rand.Read]  
is more appropriate. If a deterministic source is required, use [math/  
rand/v2.ChaCha8.Read]. (SA1019)  
metadata/cmd/main.go:128:46: otelgrpc.UnaryServerInterceptor is  
deprecated: Use [NewServerHandler] instead. (SA1019)  
movie/cmd/main.go:87:46: otelgrpc.UnaryServerInterceptor is deprecated:  
Use [NewServerHandler] instead. (SA1019)  
movie/cmd/main.go:95:6: type limiter is unused (U1000)  
movie/cmd/main.go:99:6: func newLimiter is unused (U1000)
```

```
movie/cmd/main.go:103:19: func (*limiter).Limit is unused (U1000)
pkg/tracing/tracing.go:4:2: "go.opentelemetry.io/otel/exporters/jaeger"
is deprecated: This module is no longer supported. OpenTelemetry dropped
support for Jaeger exporter in July 2023. Jaeger officially accepts and
recommends using OTLP. Use [go.opentelemetry.io/otel/exporters/otlp/
otlptrace/otlptracehttp] or [go.opentelemetry.io/otel/exporters/otlp/
otlptrace/otlptracegrpc] instead. (SA1019)
rating/cmd/main.go:83:46: otelgrpc.UnaryServerInterceptor is deprecated:
Use [NewServerHandler] instead. (SA1019)
test/integration/main.go:46:23: grpc.Dial is deprecated: use NewClient
instead. Will be supported throughout 1.x. (SA1019)
```

The staticcheck tool did a much better job of finding possible issues: it helped us to find deprecated functions that are going to be removed in future versions of libraries, as well as finding some unused code blocks.

Each message in the staticcheck output includes a check code, such as SA1019. You can run the following command to get more details on a particular check:

```
staticcheck -explain SA1019
```

The output of the command should include some additional information, as well as a link to its online documentation:

Using a deprecated function, variable, constant or field

Available since

2017.1

Online documentation

<https://staticcheck.dev/docs/checks#SA1019>

We won't cover each found issue in this chapter, but I suggest you go through the list of identified issues and resolve them, using additional documentation for each check if necessary. You can also find documentation for all types of supported static checks on this page: <https://staticcheck.dev/docs/checks/>

A good practice is to perform static code checks on a recurring basis. You might include this step in your regular coding workflow (for example, execute the tool on each commit, or once every few days/weeks). A better approach is to automate static analysis, so it is executed on each code change. There is an official article on setting up static checks in continuous integration mode:

<https://staticcheck.dev/docs/running-staticcheck/ci/github-actions/>. Note that the provided example uses GitHub Actions; however, it would be possible to set up a similar process using any other tools, depending on your preferred setup.

Now, let's proceed to the next section of the chapter, which covers the topic of service data validation.

Implementing data validation

Validation is a common problem in all kinds of applications: when accepting requests or incoming data, we want to make sure the provided values are valid (for example, user emails, date of birth values, or even movie scores).

In microservice applications, validation often becomes a non-trivial task due to multiple reasons:

- **Consistency of validation across services:** Different microservices might work with the same types of records, while possibly enforcing different validation rules (for example, if each service has its own validation logic)
- **Lack of standardization:** If services are implemented and maintained by separate teams, the format and coding style of validation rules might vary dramatically, making maintenance harder
- **Asynchronous event validation:** Validation rules should work not only in API handlers but also in asynchronous event handlers, such as Kafka consumers
- **Client versus server-side validation mismatch:** Both clients and servers might be using different validation libraries and rules

Solving such problems involves standardizing development processes across multiple services and teams, as well as unifying the way validation is implemented in each microservice.

There are some existing libraries that help to simplify data validation, as well as providing reusable functions to perform some common types of validations, such as validating email addresses, numbers, and dates. Using such functions might help to standardize validation logic across your microservices.

Among the most popular Go validation libraries is the <https://github.com/go-playground/validator> library. This library offers an annotation-based approach to structure validation: inside the Go structures, you can include validation annotations in each field. To illustrate this approach, let's briefly take a look at the `rating/pkg/model/rating.go` file, which includes the definitions of our `Rating` and `RatingEvent` types:

```
// Rating defines an individual rating created by a user for some record.
type Rating struct {
    RecordID string      'json:"recordId"'
    RecordType string     'json:"recordType"'
    UserID    UserID      'json:"userId"'
    Value     RatingValue 'json:"value"'
}

// RatingEvent defines an event containing rating information.
type RatingEvent struct {
    UserID    UserID      'json:"userId"'
    RecordID RecordID    'json:"recordId"'
    RecordType RecordType 'json:"recordType"'
    Value     RatingValue 'json:"value"'
    ProviderID string     'json:"providerId"'
    EventType  RatingEventType 'json:"eventType"'
}
```

Each structure has existing JSON annotations for each field; however, we can add additional validation annotations for a validator library:

```
type Rating struct {
    RecordID string      'json:"recordId" validate:"required"'
    RecordType string     'json:"recordType" validate:"required"'
    UserID    UserID      'json:"userId" validate:"required"'
    Value     RatingValue 'json:"value" validate:"gte=1,lte=5"'
}

// RatingEvent defines an event containing rating information.
type RatingEvent struct {
    UserID    UserID      'json:"userId" validate:"required"'
    RecordID RecordID    'json:"recordId" validate:"required"'
    RecordType RecordType 'json:"recordType" validate:"required"'
    Value     RatingValue 'json:"value" validate:"gte=1,lte=5"'
    ProviderID string     'json:"providerId" validate:"required"'
    EventType  RatingEventType 'json:"eventType" validate:"required"'
}
```

In our example, we use three different validation rules: `required`, `lte`, and `gte`. The last two rules allow us to specify the range of valid values of rating records, so the validator library can perform the value checks for us without the need to implement separate logic for this.

Let's demonstrate how to run such validations. For this, let's open the `rating/internal/controller/rating/controller.go` file and look at the `PutRating` function:

```
// PutRating writes a rating for a given record.  
func (c *Controller) PutRating(ctx context.Context, recordID model.  
RecordID, recordType model.RecordType, rating *model.Rating) error {  
    return c.repo.Put(ctx, recordID, recordType, rating)  
}
```

The existing function doesn't perform any validation, so it's a great candidate for plugging our validation logic. The steps for this are the following:

1. Add an import of the `github.com/go-playground/validator/v10` package to the `imports` block of the file.
2. Add the following line after the `ErrNotFound` definition:

```
var validate = validator.New()
```

3. Update the code of the `PutRating` function to the following:

```
// PutRating writes a rating for a given record.  
func (c *Controller) PutRating(ctx context.Context, recordID model.  
RecordID, recordType model.RecordType, rating *model.Rating) error {  
    if err := validate.Struct(rating); err != nil {  
        return fmt.Errorf("validation failed: %w", err)  
    }  
    return c.repo.Put(ctx, recordID, recordType, rating)  
}
```

Let's also reuse these validation rules in our API handler. Inside the `rating/internal/handler/grpc/grpc.go` file, add similar changes as in steps 1 and 2, and update the `PutRating` function to the following:

```
// PutRating writes a rating for a given record.
func (h *Handler) PutRating(ctx context.Context, req *gen.PutRatingRequest) (*gen.PutRatingResponse, error) {
    if req == nil {
        return nil, status.Errorf(codes.InvalidArgument, "nil req")
    }
    rating := &model.Rating{UserID: model.UserID(req.UserId), Value:
        model.RatingValue(req.RatingValue)}
    if err := validate.Struct(rating); err != nil {
        return nil, status.Errorf(codes.InvalidArgument, "validation
failed: %v", err)
    }
    if err := h.ctrl.PutRating(ctx, model.RecordID(req.RecordId), model.
        RecordType(req.RecordType), rating); err != nil {
        return nil, err
    }
    return &gen.PutRatingResponse{}, nil
}
```

We just defined our annotation-based validation rules, which will now be reused by multiple components across our rating service. In addition to this, any other code that is using the `model.Rating` structure can now use the validator library to perform structure validations (for example, the `ratingproducer` package, which you can update yourself as an exercise). Now, all our components will be synchronized in the way they validate the incoming data, preventing any logical inconsistencies (for example, if we use different validations when accepting requests and producing Kafka messages).

In addition to the basic validations that we used, the validator library offers some additional rules that might be useful in various cases:

- `eq`: Only accept values equal to the provided one
- `ne`: Not equal
- `min`: Value must contain at least N characters
- `oneof`: Allow only provided values
- `email`: Value must be a valid email address
- `url`: URL string

You can find the complete set of supported rules on the GitHub page of the library: <https://github.com/go-playground/validator>

Implement streaming APIs

In *Chapter 5, Synchronous Communication*, we covered different types of synchronous communication and mentioned that there are client- and server-side streaming options that can be used for sending sequences of messages. So far, we haven't used any streaming APIs in the book; however, let's illustrate how to implement client-side streaming so you get a better picture of how to use streaming APIs in your microservices.

To demonstrate streaming, we are going to modify the movie service by adding a function that will handle streaming file uploads: a client will send a sequence of chunks to a server, and a server will append them to a file with a provided name. The steps are the following:

1. Let's open the `api/movie.proto` file and add an `UploadFile` function to it, as well as our request and response structures:

```
service MovieService {
    rpc GetMovieDetails(GetMovieDetailsRequest) returns
    (GetMovieDetailsResponse);
    rpc UploadFile(stream UploadRequest) returns (UploadResponse);
}

message UploadRequest {
    string filename = 1;
    bytes chunk = 2;
}

message UploadResponse {
    string message = 1;
}
```

Our handler accepts a stream of `UploadRequest` messages – a sequence of messages that will be sent by the client to our server.

2. Let's now re-generate the proto files for our movie service. Inside the `src` directory, run the `protoc` command:

```
protoc -I=api --go_out=. --go-grpc_out=. movie.proto
```

3. Now, we are ready to update our movie service gRPC handler. Open the `movie/internal/handler/grpc/grpc.go` file and add the following function to it:

```
// UploadFile handles streaming file uploads.  
func (h *Handler) UploadFile(stream gen.MovieService_  
UploadFileServer) error {  
    var filename string  
    var file *os.File  
    for {  
        req, err := stream.Recv()  
        if err == io.EOF {  
            return stream.SendAndClose(&gen.UploadResponse{Message:  
fmt.Sprintf("File %s uploaded successfully", filename)})  
        }  
        if err != nil {  
            return err  
        }  
  
        if file == nil {  
            file, err = os.Create(filename)  
            if err != nil {  
                return err  
            }  
            defer file.Close()  
        }  
        if _, err := file.Write(req.Chunk); err != nil {  
            return err  
        }  
    }  
}
```

4. Add the missing `os`, `io`, and `fmt` imports to this file and save it.

Our server implementation that accepts a stream of incoming file upload messages is completed. Note how we receive new stream messages by calling a `Recv()` function of it and check if the stream is out of incoming messages by checking if we got an `io.EOF` error from it.

Now, let's demonstrate how to create a streaming client. Inside our top-level cmd directory, create an uploadclient package and add a `main.go` file with the following content:

```
package main

import (
    "context"
    "fmt"
    "io"
    "log"
    "os"
    "time"

    "google.golang.org/grpc"
    "movieexample.com/gen"
)

func main() {
    conn, err := grpc.NewClient("localhost:8083")
    if err != nil {
        log.Fatalf("Failed to connect: %v", err)
    }
    defer conn.Close()

    client := gen.NewMovieServiceClient(conn)

    filePath := "upload.txt"
    if err := uploadFile(client, filePath); err != nil {
        log.Fatalf("Failed to upload file: %v", err)
    }
}
```

Now, let's add the function that will upload the data to our server using a stream of 1 KB messages:

```
func uploadFile(client gen.MovieServiceClient, filePath string) error {
    file, err := os.Open(filePath)
    if err != nil {
        return fmt.Errorf("failed to open file: %w", err)
    }
```

```
defer file.Close()

stream, err := client.UploadFile(context.Background())
if err != nil {
    return fmt.Errorf("failed to create upload stream: %w", err)
}

buf := make([]byte, 1024)
for {
    n, err := file.Read(buf)
    if err == io.EOF {
        break
    }
    if err != nil {
        return fmt.Errorf("failed to read file: %w", err)
    }

    if err := stream.Send(&gen.UploadRequest{
        Filename: filePath,
        Chunk:    buf[:n],
    }); err != nil {
        return fmt.Errorf("failed to send chunk: %w", err)
    }
}

resp, err := stream.CloseAndRecv()
if err != nil {
    return fmt.Errorf("error receiving response: %w", err)
}

fmt.Println("Server response:", resp.Message)
return nil
}
```

Our client code is ready. Let's highlight some implementation details of it:

- To open a stream, we called our `UploadFile` function on a client, without passing any data yet
- In a loop, we read data from our input file in 1024-byte chunks, and sent it to our server via a `Send` function
- To complete the operation, we called the `CloseAndRecv` function of a stream
- If the `CloseAndRecv` call returns an error, the client needs to re-perform the operation since there is no guarantee that the server will complete it successfully

To test the streaming implementation, you will need to start the movie service, create an input file called `upload.txt` and add some content to it, then start our new client command that will run the upload. If everything executes successfully, you should be able to see the server output and see the successful execution of the operation.

Streaming APIs can be used for a variety of use cases, including file uploads, video streaming, and continuous communication between a client and a server (for example, in games or chat applications). Using the principles that we just discussed, you will be able to implement such applications without any issues.

Frameworks

In *Chapter 2, Scaffolding a Go Microservice*, we covered the topic of the Go project structure, as well as some common patterns for organizing your Go code. The code organization principles that we described are generally based on conventions – written agreements or statements that define specific rules for naming and placing Go files. Some of the conventions that we followed were proposed by the authors of the Go language, while the others are commonly used and proposed by authors of various Go libraries.

While conventions play an important role in establishing the common principles of organizing Go code, there are other ways of enforcing specific code structures. One such way is using frameworks, which we are going to cover in this section.

Generally speaking, **frameworks** are tools that establish a structure for various components of your code. Let's take the following code snippet as an example:

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/echo",
        func(w http.ResponseWriter, _ *http.Request) {
            fmt.Fprintf(w, "Hi!")
        })
    if err := http.ListenAndServe(":8080", nil);
        err != nil {
            panic(err)
    }
}
```

Here, we are registering an HTTP handler function and letting it handle HTTP requests on the `localhost:8080/echo` endpoint. The code for our example is extremely simple, yet it does a lot of background work (you can check the source of the `net/http` package to see how complex the internal part of the HTTP handling logic is) to start an HTTP server, accept all incoming requests, and respond to them by executing the function provided by us. Most importantly, our code allows us to add additional HTTP handlers by following the same format of calling the `http.HandleFunc` function and passing handler functions to it. The `net/http` library that we used in our example established a structure for handling HTTP calls to various endpoints, acting as a framework for our Go application.

The authors of the `net/http` package were able to add additional HTTP endpoint handlers (provided by the `http.HandleFunc` function) by following a pattern called **Inversion of Control (IoC)**. IoC is a way of organizing code in which some component (in our case, the `net/http` package) takes control of the execution flow by calling the other components of it (in our case, the function provided as an argument to `http.HandleFunc`). In our example, the moment we call the `http.ListenAndServe` function, the `net/http` package takes control of executing the HTTP handler functions: each time the HTTP server receives an incoming request, our function is called automatically.

IoC is a primary mechanism of most frameworks that allows them to establish a foundation for various parts of application code. In general, most frameworks work by taking control of an application, or a part of it, and handling some routing operations, such as resource management (opening and closing incoming connections, writing and reading files, and so on), serialization and deserialization, and many more.

What are the primary use cases for using Go frameworks? We can list some of the most common ones:

- **Writing web servers:** Similar to our example of an HTTP server, there can be other types of web servers handling requests to different endpoints using different protocols, such as Apache Thrift or gRPC
- **Async event processing:** There are libraries for various asynchronous communication tools, such as Apache Kafka, that help organize code in an IoC way by passing handler functions for various types of events (such as Kafka messages belonging to different topics), which get called automatically each time there is a new unprocessed message

It is important to note that frameworks have some significant downsides:

- **Harder to debug and understand the execution flow:** In addition to taking control of the execution flow, frameworks also perform lots of background work that is hidden from developers. Because of this, it is usually much harder to understand how your code is being executed, as well as to debug various issues, such as initialization errors (you can find more information on this in the following article: <https://www.baeldung.com/cs/framework-vs-library>).
- **Steeper learning curve:** Frameworks generally require a good understanding of the logic and abstractions they provide. This requires developers to spend more time reading the related documentation or learning some key lessons in practice.
- **Harder to catch some trivial bugs via static checks:** Frameworks often use dynamic code invocation libraries, such as reflect (<https://pkg.go.dev/reflect>). Such operations are performed when executing a program, making it hard to catch various types of issues, such as the incorrect implementation of interfaces or invalid naming.

When deciding on using a specific framework, you should do some analysis and compare the advantages it provides to you with the downsides it brings, especially in the long term. Many developers underestimate the complexity that frameworks bring to them or the other developers in their organizations: most frameworks perform a fair amount of *magic* to provide a convenient code structure to application developers. In general, you should always start from a simpler

option (in our case, not using a particular framework) and only decide to use it if its benefits outweigh its downsides.

Now that we have discussed the topic of frameworks, let's move on to the next section, where we will describe the different aspects of microservice ownership.

Storing microservice ownership data

One of the key benefits of using microservice architectures is the ability to distribute their development: each service can be developed and maintained by a separate team, and teams can be distributed across the globe. While the distributed development model helps different teams build various parts of their systems independently, it brings some new challenges, such as service ownership.

To illustrate the problem of service ownership, imagine that you are working in a company with thousands of microservices. One day, the security engineers of your company find out that there is a critical security vulnerability in a popular Go library that is used in most of the company's services. How can you communicate with the right teams and find out who would be responsible for making the changes in each service?

There are numerous companies with thousands of microservices. In such companies, it becomes impossible to remember which team and which developers are responsible for each of them. In such companies, it becomes crucial to find a solution to the service ownership problem.

Note



While we are discussing the ownership problem for microservices, the same principles apply to many other types of technological assets, such as Kafka topics and database tables.

How can we define service ownership?

There are many different ways of doing this, each of which is equally important for some specific use cases:

- **Accountability:** Which person/entity is accountable for the service and who can act as the primary point of contact or the main authority for it?
- **Support:** Who is going to provide support for the service, such as a service bug, feature request, or user question?

- **On-call:** Who is currently on-call for the service? Who can we contact in case of an emergency issue?

As you can see, there are many ways of interpreting the word ownership, depending on the use case. Let's look at some ways to define each role, starting with accountability: who should be accountable, or liable, for a service?

In most organizations, liability is attributed to engineering managers: every engineering manager acts as an accountable individual for some unique domain. If you define a mapping between your services and the engineering managers who are responsible for them, you can solve the service accountability problem by allowing them to easily find the relevant point of contact, such as an engineering manager who is liable for it.

An alternative way of defining service accountability is to associate services with teams. However, there can be multiple issues with this:

- **Shared accountability does not always work:** If you have multiple people who are responsible for a service, it becomes unclear who the final authority among them is
- **A team is a loosely defined concept in many organizations:** Unless you have a single, well-defined registry of teams in your company, it's better to avoid referencing team names in your systems

Now, let's discuss the support aspect of ownership. Ideally, each service should have a mechanism for reporting any issues or bugs. Such a mechanism can take one of the following forms:

- **Support channel:** The identifier or URL of a messaging channel for leaving support requests, such as a link to the relevant Google group, Slack channel, or any other similar tool.
- **Ticketing system URL:** The URL to a system/page that allows you to create a support request ticket. Developers often use Atlassian Jira for this purpose.

If you provide such metadata for all your services, you will significantly simplify user support: all service users, such as other developers, will always know how to request support for them or report any bugs or other issues.

Let's move to the on-call ownership metadata. An easy solution to this is to link each service to its on-call rotation. If you use PagerDuty, you can store the relationships between service names and their corresponding PagerDuty rotation identifiers.

An example of the ownership metadata that we just described is as follows:

```
ownership:  
  rating-service:  
    accountable: example@somecompany.com  
    support:  
      slack: rating-service-support-group  
      oncall:  
        pagerduty_rotation:SOME_ROTATION_ID
```

Our example is defined in YAML format, though it may be preferable to store this data in some system that would allow us to query or modify it via an API. This way, you can automatically submit new ownership changes (for example, when people leave the company and you want to reassign the ownership automatically). I would also suggest making the ownership data mandatory for all services. To enforce this, you can establish a service creation process that will request the ownership data before developers provision new services.

Summary

With that, we have finished the last chapter of this book by reviewing some additional microservice development topics that were not included in the previous chapters. You learned how to profile Go services, create microservice dashboards so that you can monitor their performance, define and store microservice ownership data, and secure microservice communication with JWTs. I hope that you have found lots of interesting tips in this chapter that will help you build scalable, highly performant, and secure microservices.

The Go language keeps evolving, as does the tooling for it. Each day, developers release new libraries and tools that can solve various microservice development problems that we described in this book. While this book provided you with lots of tips on Go microservice development, you should keep improving your skills and make your services simpler and easier to maintain.

I also want to thank you for reading this book. I hope you enjoyed it and gained lots of useful experience that will help you in mastering the art of Go microservice development. Let your Go microservices be highly performant, secure, and easy to maintain!

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Top Go Frameworks: <https://github.com/mingrammer/go-web-framework-stars>
- Go Cookbook: <https://go-cookbook.com>

We suggest that you follow these resources to stay up to date with the latest news related to Go microservice development:

- The Go blog: <https://go.dev/blog/>
- Microservice architecture: <https://microservices.io>
- A curated list of awesome Go software: <https://github.com/avelino/awesome-go>



www.packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

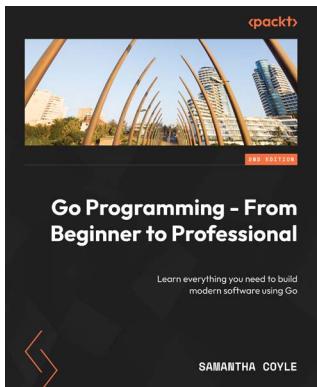
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

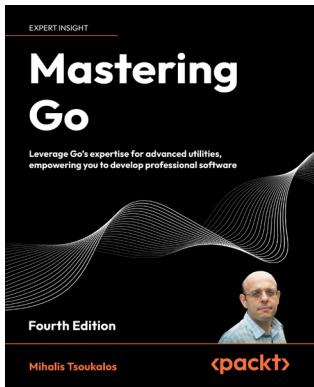


Go Programming - From Beginner to Professional

Samantha Coyle

ISBN: 978-1-80324-305-4

- Understand the Go syntax and apply it proficiently to handle data and write functions
- Debug your Go code to troubleshoot development problems
- Safely handle errors and recover from panics
- Implement polymorphism using interfaces and gain insight into generics
- Work with files and connect to popular external databases
- Create an HTTP client and server and work with a RESTful web API
- Use concurrency to design efficient software
- Use Go tools to simplify development and improve your code



Mastering Go

Mihalis Tsoukalos

ISBN: 978-1-80512-714-7

- Learn Go data types, error handling, constants, pointers, and array and slice manipulations through practical exercises
- Create generic functions, define data types, explore constraints, and grasp interfaces and reflections
- Grasp advanced concepts like packages, modules, functions, and database interaction
- Create concurrent RESTful servers, and build TCP/IP clients and servers
- Learn testing, profiling, and efficient coding for high-performance applications
- Develop an SQLite package, explore Docker integration, and embrace workspaces

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Microservices with Go - Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from. Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

access control 217

ACID properties

atomicity 148

consistency 148

durability 148

isolation 148

alerting

best practices 325, 326

Alpine 173

Amazon Web Services (AWS) 163

annotations 82

anomaly detection 275

Apache Avro 86

Apache Kafka

adopting, for microservices 128-138

basics 126, 127

URL 127

used, for messaging 126

Apache Thrift 85, 100

benefits 86

limitations 86

Apache ZooKeeper 63, 354

apex/log

reference link 279

application code

preparing, for deployment 164

application configuration 165-169

application security 235

asynchronous communication

basics 122

benefits 122, 123

challenges 123, 124

message broker 124

publisher-subscriber model 125, 126

techniques 124

authentication and access control

implementing, with JWT 225

authorization 217

automated rollbacks 179

AWS Elastic Kubernetes Service 170

B

backoff 253

constant backoff 253

exponential backoff 253

backoff delay 254

backpressure 257

backward-incompatible change 248

best practices, asynchronous communication
error handling 142
explicit message acknowledgment, using 140, 141
partitioning, leveraging 139, 140
versioning 138, 139

black boxes 196

black-box monitoring 275

black box testing 196

blob databases 149

buckets 289

built-in Go log package 279

built-in Go log/slog package 279

C

cache 158, 256

California Consumer Privacy Act (CCPA) 242

call analysis 296

canary deployments 180

canary mode 165

centralized secret management tools 239

Certificate Authorities (CAs) 217

circular dependencies 268

client error 250

clients
implementing 105

client-side service discovery 60

cluster 171

cmp library 210
reference link 209

cold start time 12

comments
principles 18
uses 18

Common Weakness Enumeration (CWE) 240

communication error handling 250
deadline, setting 255
fallbacks 256
rate limiting 257, 258
request retries, implementing 250-254
request timeout, setting 254

communication protocols 85

compliance 241
regulations for software development 241, 242

congestion policy 249

connection leak 260

connection string 153

constant backoff 253

consul-based service discovery implementation 70-72

consumer 127

consumer offset 135

container image 172

containers 171

context 20, 299
best practices 21

context cancellation 262
implementing 262, 263

context leakage 21

context propagation 20
example 300

continuous deployment (CD) 180

counters 289

CPU usage
profiling, with pprof tool 340-344

D

- dashboarding** 330
- dashboards** 330
 - benefits 330
 - creating, to visualize service telemetry data 330, 331
 - service-level metrics, adding 334-339
- databases** 146
 - durability 146
 - instances 147
- database types**
 - blob databases 149
 - document databases 149
 - graph databases 149
 - key-value databases 148
 - relational databases 149
- data caching**
 - implementing 158-160
- data caching, solutions**
 - Memcached 159
 - Redis 158
- data locality** 140
- data points** 289
- data schema** 89
 - benefits 89
- data validation**
 - implementing 371-374
- dead letter queue (DLQ)** 142
- deadline** 255
- denial of service (DoS)** 248
- deployment** 163, 164
 - application code, preparing 164
 - build process 164
 - rollout process 164
- deployment, best practices** 179
 - automated rollbacks 179
 - canary deployments 180
 - continuous deployment (CD) 180
 - feature flags, using to decouple deployments from feature releases 181
- detection** 249
- digital certificates** 217
- discovery logic**
 - implementing 66
- distributed leader election** 354
- distributed locks**
 - implementing, with HashiCorp Consul 356-362
- distributed system best practices** 362
 - build systems 363
 - distributed transactions, avoiding 364
- distributed system problems**
 - distributed locks 354
 - leader election 354
- distributed system tools** 354
 - Apache Zookeeper 354
 - etcd 355
 - HashiCorp Consul 355
- distributed tracing** 296
- distributed transactions** 364
- Docker**
 - URL 77
- Docker Swarm** 169
 - reference link 170
- document databases** 149
- dynamic configuration** 181

E**Effective Go** 16

URL 16

Elasticsearch 283**Elastic Stack (ELK)** 283**environments**

local/development 165

production 165

staging 165

errors

recommendations 19, 20

etcd 62, 355**event correlation** 275**eventual consistency** 364**example application**

code structure 30

controller 30

gateway 30

handler 30

movie metadata service 31

movie service 46

repository 30

exponential backoff 253

example 254

F**fail close** 256**fail open** 256**fallbacks** 256**feature flags** 181using, to decouple deployments from
feature releases 181**Five Whys technique** 267**flaky tests**

detecting 210

fixing 210

frameworks 379-381

disadvantages 381

G**gateways**

implementing 105

**General Data Protection Regulation
(GDPR)** 241**generated model** 106**Git hooks** 180**Go** 16

advantages 11

context 20

core principles 16, 17

idiomatic Go code, writing 17

interfaces 20

tests 20

Go code review comments

reference link 16

go-elasticsearch

reference link 283

Go frameworks

use cases 381

Go function

trace data, collecting for 301

go-kit

reference link 279

Go logging libraries

apex/logs 279

built-in Go log package 279

built-in Go log/slog package 279

go-kit 279

zap 279

zerolog 279

Go metrics libraries 292, 293

gomock library 192, 193

Go MySQL driver 153

Google Cloud Platform (GCP) 179

Google Kubernetes Engine 170

Go SDK

reference link 298

gosec tool

using, to automate security analysis 236, 237

Go service code

static analysis 368-371

Go services

gosec, using to automate security analysis 236, 237

profiling 340-345

security analysis 235

Go testing

capabilities 184

Skip function 187, 188

subtests 186, 187

go vet tool

URL 368

govulncheck 239

Go vulnerabilities

URL 239

graceful degradation 256

graceful shutdown

performing 260, 261

Grafana 292

URL 339

used, for creating performance dashboards 331-334

graph databases 149

Graphite

components 292

Graphviz

URL 340

gRPC 100

adoption 101

bi-directional streaming 101

client streaming 101

features 101

server streaming 101

Unary 101

guages 289

H

HashiCorp Consul 62, 355

reference link 355

URL 62

used, for implementing distributed locks 356-362

HashiCorp Nomad 170

health checks 61

Health Insurance Portability and Accountability Act (HIPAA) 242

heap memory usage

profiling, with pprof tool 344, 345

heap profile

example 344

histograms 289

cohort tracking 290

latency tracking 290

HTTP protocol

client error 99

headers 99

request body 99

response body 99

server error 99

URL parameters 99

I

idempotency keys 119
idiomatic Go code
 writing 17
idiomatic Go code, guidelines
 comments 18
 error 19
 naming principles 17
 naming recommendations 18
incident management 266, 267
incident postmortems 266
in-memory service discovery implementation 67-70
integration tests 196
 challenges, with using existing persistent databases 206
 implementing 198-206
 plan, writing for 197
 structure 197
 writing, suggestions 207
interceptors 259
interfaces
 principles 20
internal model 106
interpolated strings 285
inversion of control (IoC) 380

J

Jenkins
 URL 207
jittering 254
JSON 83
 benefits 83
 limitations 83

JSON Web Tokens (JWT)

 basics 225, 226
 issuance and validation, implementing in microservices 226-235
 URL 235
 used, for implementing authentication and access control 225

K

key-value databases 148
Kibana 283
Kubernetes 62
 benefits 172
 deploying via 171
 deployment model 171
Kubernetes deployments
 microservices, setting up for 173-178

L

labels 291
load balancer 60
logging 276
 best practices 284-288
 using 280, 281
logging library
 selecting 279
log levels
 debug 278
 error 278
 fatal 278
 info 278
 warning 278
log library
 fatal function 277
 panic function 277

- logs** 274
 debugging information 276
 failed operations 276
 order of operations 276
 panics 276
 warnings 276
- Logstash** 283
- M**
- machine learning (ML)** 283
- man-in-the-middle attacks** 218
- mapping logic** 106
- matchers** 316
- Memcached** 159
- message batching** 124
- message broker** 124
 lossless 125
 lossy 125
- message delivery** 124, 125
- message routing** 124
- message transformation** 124
- metadata database**
 data replication 148
 query capabilities 148
 transaction support 148
- metadata service** 146
 gateways and clients,
 implementing 105-110
- MetadataService** 86
- metric data collection**
 best practices 294-296
- metrics** 288
 challenges, of storing 291
 counters 289
- guages 289
 histograms 289
- microservice architecture**
 role, of Go 11, 12
 using 9, 10
- microservice logs**
 storing 282
- microservice ownership data**
 storing 382-384
- microservices** 3, 4
 Apache Kafka, adopting for 128-138
 benefits 4-8
 best practices 10
 issues 8, 9
 JWT issuance and validation, implementing
 226-235
 motivation 4-6
 Prometheus alerting, setting up
 for 317-325
 setting up, for Kubernetes
 deployments 173-178
- Microsoft Azure** 163
- mitigation** 249
- mocking** 190
- mocks** 190
 example 190-193
- monolithic applications (monoliths)** 4
- movie metadata service** 31
 controller 34, 35
 data schema, reviewing 151
 handler 36, 37
 main file 37, 38
 model 32
 repository 32-34
- movie service** 46
 controller 51-53

directory structure 46
gateways 47-51
gateways and clients,
 implementing 112-117
handler 53, 54
main file 54, 55
model 46

Mutual TLS (mTLS) 219

MySQL 150
used, for storing service data 150-157

N

naming 17
principles 17
recommendations 18
nodes 171
non-retriable errors 250

O

observability
challenges 275
offset 127
offset commits 141
on-call process 264, 265
 challenges 265
on-call rotations 264
OpenSearch 283
OpenTelemetry Collector 292
 benefits 284
 reference link 284
OpenTelemetry project
 reference link 274
OpenTelemetry SDK
 tracing data, collecting 300-308

P

PagerDuty 265
reference link 265
Payment Card Industry Data Security Standard (PCI DSS) 242
performance dashboards
 creating, with Grafana tool 331-334
personally identifiable information (PII) 287
planned incidents 268
pods 171
pprof tool
 used, for profiling CPU usage 340-344
 used, for profiling heap memory
 usage 344-345
prevention 249
principles 16
private key 217, 218
producer 127
profiling 340
 benefits 340
 Go services 341-345
project structure 21
 best practices 24
 common files 23
 directories 23
 executable packages 23
 private packages 22
 public packages 22
Prometheus 291, 292, 315-317
 reference link 325
Prometheus alerting
 setting up, for microservices 317-325
PromQL 291, 316
 reference link 317

Protocol Buffers 81, 87

- benefits 87
 - service API, defining with 101-105
 - using 87-89
- proto compiler 101**
- public key 218**
- publisher-subscriber model 125, 126**

R**race conditions**

- detecting 211, 212

Raft library

- reference link 362

rate limiter

- using 258-260

rate limiting 257, 258

- client level 257
- network/intermediate level 257, 258
- server level 257

ratings 150**rating service 38**

- controller 41-43
- gateways and clients,
implementing 110-112
- handler 43-45
- main file 45
- model 39
- repository 40, 41

read-through cache 160**Redis 158****relational databases 149****reliability**

- basics 248, 249
- reliability, achieving through
automation 249**
- communication error handling 250

graceful shutdown 260, 261

**reliability, achieving through development
processes and culture 263**

- disaster recovery plans 269
- incident management 266, 267
- objectives, tracking 269-271
- on-call process 264
- reliability drills 268

reliability drills 268

- benefits 268

remote procedure call (RPC) 99

- features 100

replicas 148**request retries**

- implementing 250-254

request timeout 254

- benefits 254, 255

retention policies 287**retryable errors 250****root cause analysis (RCA) 267****root scope 294****root span 301****RPC frameworks**

- Apache Thrift 100
- gRPC 100

runbook 266**S****Saga 364****Sarbanes-Oxley Act (SOX) 242****scope 294****secure secret management 238, 239****secure service communication 218, 219**

- implementing, with TLS protocol 219-224
- mechanisms 218, 219

TLS protocol 218

-
- secure service communication, with TLS protocol**
 - additional improvements 224
 - security 216**
 - authentication and access control 217, 218
 - secure service communication 218, 219
 - security analysis**
 - automating, with gosec tool 236, 237
 - of Go services 235
 - security, best practices 237**
 - automated vulnerability scanning, using 239, 240
 - periodic threat modeling exercises, performing 240, 241
 - secure secret management 238, 239
 - semantic graph capturing 275**
 - serialization 81**
 - basics 82, 83
 - best practices 94
 - use cases 82
 - serialization format 82-84**
 - Apache Avro 86
 - Apache Thrift 85, 86
 - benchmark, performing 89-93
 - Protocol Buffers 87
 - XML 84
 - YAML 84
 - server authentication 217**
 - server error 250**
 - server-side service discovery 60, 61**
 - service**
 - configuring, ways 166
 - service API**
 - defining, with Protocol Buffers 101-105
 - service data**
 - storing, MySQL used 150-158
 - service, deploying in Kubernetes**
 - steps 172
 - service deployment solutions**
 - Docker Swarm 169
 - service discovery**
 - adopting 63-64
 - health monitoring 61
 - limitations 58
 - metadata service 58
 - models 59
 - movie service 58
 - overview 58
 - rating service 58
 - registry 59
 - service discovery, adoption**
 - application, preparing 63-66
 - logic, implementing 66-79
 - logic, using 72-75
 - service discovery models 59**
 - client-side service discovery 60
 - server-side service discovery 60, 61
 - service discovery solutions**
 - Apache ZooKeeper 63
 - etcd 62
 - HashiCorp Consul 62
 - Kubernetes 62
 - service health monitoring 61**
 - pull model 61
 - push model 61
 - service-level agreement (SLA) 316**
 - example link 271
 - service level indicators (SLIs) 270**
 - service level objectives (SLOs) 270**
 - service logs**
 - collecting 276
 - example 276

- service metrics**
 - collecting 288, 289
 - emitting 293, 294
 - service ownership** 382
 - service registry** 59
 - example 59
 - features 59
 - service traces**
 - collecting 296, 297
 - session key** 219
 - short mode** 188
 - Simple Storage Service (S3)** 284
 - Skip function** 187, 188
 - software security**
 - areas 216
 - spaghettification** 22
 - span** 300
 - SQL**
 - reference link 148
 - stacked retries** 253
 - static analysis**
 - of Go service code 368-371
 - staticcheck**
 - URL 369
 - streaming APIs**
 - implementing 375-379
 - streams** 98
 - structured logging** 277
 - subtests** 186, 187
 - successful response** 250
 - synchronous communication**
 - protocols 98
 - synchronous communication, best practices**
 - correct error codes, returning 118, 119
 - excessive request validation,
 - performing 118
 - idempotency, ensuring 119, 120
 - system throughput** 249
- T**
- table-driven tests** 185
 - tags** 289
 - telemetry data** 274
 - logs 274
 - metrics 274
 - traces 274
 - testing** 183
 - testing, best practices**
 - cmp library, used for comparing metadata structure 209, 210
 - code coverage, tracking 212
 - Fatal function usage, avoiding in logs 208
 - flaky tests, detecting 210, 211
 - helpful messages, using 207, 208
 - race conditions, detecting 211, 212
 - tests**
 - suggestions 20
 - threat modeling** 241
 - Thrift services** 85
 - time series** 289
 - time to repair (TTR)** 266
 - TLS protocol**
 - used, for implementing secure service communication 219-224
 - token bucket algorithm** 258
 - topics** 127
 - traceable operation** 299
 - trace data**
 - collecting, for Go function 301

tracing

example 296
tools 298, 299
use cases 273-309

tracing data

collecting, with OpenTelemetry
SDK 299-308

transitive dependencies 268**trend analysis** 275**two-factor authentication** 217**type embedding** 130**U****unique identifier (UID)** 177**unit tests** 189, 190

implementing 193-195

user authentication 217**V****versioning technique** 138, 139**W****white-box monitoring** 275**X****XML** 84**Y****YAML** 84**Z****zap**

reference link 279

zerolog

reference link 279

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781836207337>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.