BUILD SYSTEMS with GO & save the world

Everything a Gopher must know

Concurrency, reflection, testing, modules, benchmarking, protocol buffers, gRPC, loggers, CLI, SQL/NoSQL, Cassandra, Kafka, and more...

With more than 200 examples

v0.1.0

Juan M. Tirado

Build systems with Go

Everything a Gopher must know

Juan M. Tirado

Build systems with Go

by Juan M. Tirado

Copyright ©2021

Independently published

Cover by Juan M.Tirado

Gopher Gotham image by Egon Elbre (@egonelbre)

This book has been entirely written using LaTeX.

EPub 3.0 conversion was done using tex4ebook:

https://github.com/michal-h21/tex4ebook

Revision History:

- **v0.1.0**: 2021-03-29 First version

PREFACE

Welcome and thank you for reading these lines.

Since I started programming in Go, I have always enjoyed its extraordinary commitment to simplicity. It is difficult to find another language that can make complex things so easily. That is the beauty of this language. Years have passed by and Go is no longer the new kid on the block, it has already become a mature language surrounded by a rich ecosystem of libraries, projects, and tools. Talking about Go is no longer talking about that fancy language that makes your life easier. Go is the gravity centre of a continuously growing ecosystem of amazing solutions maintained by a devoted community of developers.

Go was originally designed to simplify the building of complex systems. However, when a developer decides to learn Go most of the learning resources simply explain the language. This book goes one step further by exploring tools, libraries, and projects from the Go ecosystem you can use to build ready-for-production systems. Everything a gopher must know in a single book.

I hope you find this book useful.

WHO SHOULD READ THIS BOOK?

This book is oriented to new Go adopters and developers with programming experience in other languages. The first part of this book covers the Go language from its basics to more advanced concepts. The second part assumes these concepts to be known by the reader and explores how to use them with other tools to build systems. If you are new to Go you can start from the beginning. However, if you have some experience you can start with the second part and revisit any basic concept if needed. Or you can simply go and check the chapters at your convenience.

STRUCTURE OF THIS BOOK

This book is structured to easily find those pieces you may find more interesting for your work. However, if you are an absolute beginner or you do not feel very comfortable with all the concepts explained in this book you can always start from the beginning. Whatever your use case is, these are the contents of this book.

- **Part I: The GO language**

  The first part explores the language from the very basics to advanced tools offered by the standard library.
- **Chapter 1: First steps with Go**

  This Chapter is specifically written to motivate newbies to run their first Go program.
- **Chapter 2: The basics**

  This Chapter explains all the Go basics including syntax, variables, types, pointers, functions, and execution flow.
- **Chapter 3: Arrays, slices, and maps**

  Go includes powerful native data structures such as arrays and maps. This Chapter extends previous concepts and shows the reader how to write her first data processing solutions.
- **Chapter 4: Structs, methods, and interfaces**

  This Chapter describes how Go defines advanced data structures, their associated methods, and interfaces.
- **Chapter 5: Reflection**

  By exploring how Go uses reflection, the reader can understand the many possibilities of manipulating in-memory data structures.
- **Chapter 6: Concurrency**

  Concurrency is not an easy topic. However, this Chapter demonstrates how Go help developers to design complex solutions effortless. This Chapter covers goroutines, channels, concurrency statements, contexts and more.
- **Chapter 7: Input/Output**

  Any program requires to write or read data to and from different sources. This Chapter explains through examples how Go provides I/O support.

- **Chapter 8: Encodings**

  The Go standard library offers by default solutions to work with encodings such as CSV, JSON or XML. This Chapter, explains how to use these encodings and others not available by default.
- **Chapter 9: HTTP**

  This Chapter explains how we can implement our own HTTP clients and servers, and how to deal with requests, cookies, headers or middleware.
- **Chapter 10: Templates**

  Templates are pieces of data than can be filled programmatically. This Chapter explains how to define, customize, and use them.
- **Chapter 11: Testing**

  This Chapter will show the reader how simple it is to execute testing routines and benchmarks in Go. Additionally, it will introduce the reader how to run coverage tests and execution profiles.
- **Chapter 12: Modules and documentation**

  This Chapter explains how to manage dependencies in Go and how to document code.
- **Part II: Building systems**

  The second part of the book is oriented to those readers who feel comfortable with the language and want to explore solutions from the Go ecosystem that can be used to build sophisticated systems.
- **Chapter 13: Protocol buffers**

  This Chapter reviews what is the protocol buffer serialization format and how to use it with Go.
- **Chapter 14: gRPC**

  Read this Chapter if you need of a fast, modular, and easy-to-deploy message protocol in your system. This Chapter explains how to define services, servers, clients, streaming, and interceptors.
- **Chapter 15: Logging with Zerolog**

This Chapter shows the reader how to log a program using the powerful Zerolog library.

- **Chapter 16: Command Line Interface**

  Complex programs require complex command line interfaces. This Chapters, shows the developer how to define and integrate the Cobra library in their projects to obtain professional CLIs with minimal effort.
- **Chapter 17: Relational databases**

  This Chapter introduces how the standard library can be used to manipulate and query data from SQL databases. Additionally, it explores how to use the GORM library for ORM solutions.
- **Chapter 18: NoSQL databases**

  NoSQL database solutions are quite common and the Go ecosystem offers solutions to work with them. This Chapter, explains how to operate with Apache Cassandra using the GoCQL client.
- **Chapter 19: Kafka**

  This Chapter reviews the basics of Apache Kafka and overviews three different solutions to interact with this streaming platform.

CONVENTIONS

This book is built around self-contained examples. These examples are minimalist pieces of code that help the reader becoming familiar with the explained concepts. Examples are small enough to bring the reader an idea of how a real program looks like. Some examples may print something to help the reader, in that case, the expected output is shown for the reader's convenience.

This is how an example looks like.

: Title of this example.

```
1 In the left side

2 of this box,

3 you can find
```

```
the code for
this example
```

```
The output goes here.
```

Additional tips and notes can be found across the book.

This is a warning note.

This is a curiosity or tip with additional information.

THE CODE

This book contains a large number of examples fully available at the author’s GitHub repository under the Apache license:

https://github.com/juanmanuel-tirado/savetheworldwithgo

Feel free to fork the repository at your convenience. If you find any issue or have any comment regarding the code, please let the author know.

ABOUT THE AUTHOR

Juan M. Tirado has been programming half of his life. He holds a Ph. D. in computer science and has been a researcher at the UC3M, INRIA, and the University of Cambridge. He is interested in how data can be leveraged to enhance large scale distributed systems. With a background between a systems architect and a data scientist, he helps companies to design and implement data-driven solutions. In his free time, he enjoys music, mountaineering, and tapas.

You can follow the author at:

- Website: https://jmtirado.net/
- LinkedIn: https://www.linkedin.com/in/juan-tirado
- Medium: https://juanmanuel-tirado.medium.com/
- Twitter: @jmtirado

SOME WORDS OF GRATITUDE

This book is a one-person project carried out with a lot of effort and great illusion. If you have found this book useful, the author would appreciate you spread the word and tell your friends and colleagues. Your comments and/or suggestions are always welcome to help in improving this book.

Part I

The GO language

CHAPTER 1

FIRST STEPS WITH GO

This chapter will show you how to write, compile and execute your first program in Go. For this, you need a working Go installation. Follow the steps for your platform described in the official documentation[1]. Next, take any plain text editor of your choice: NotePad, Nano, Vim, etc. You will need one of them to write down the code. If you prefer to use more sophisticated tools such as GoLand, Atom or Visual Studio Code the following examples still apply. However, I recommend you follow the current explanation if this is your first time with Go.

1.1 SAVE THE WORLD WITH GO!!!

If you are familiar with any computer language you already know what comes next: a *Hello World!* program. This is just a program that will print a message in your console output. Traditionally this is the first approach to any programming language. And this is still the case although we have changed the message.

Example 1.1: Save the world with Go!!!

```
1 package main
2
3 import “fmt”
4
5 func main() {
6     fmt.Println(“Save the world with Go!!!”)
7 }
```

```
Save the world with
Go!!!
```

The above code has the basic components of a Go program. First, we set the name of the package that contains our code (line 1). In line 2, we import the library required to invoke our `Println` function. The logic of our program is contained between brackets in a function called `main` between lines 5 and 7. The statement in line 6 prints our message using the standard output.

Go must be compiled before execution. This is, we need to run our code through a compiler to generate executable code for our platform. The result from the compilation process is an executable file. Depending on your platform this file will be different. To compile our program, we only need to write down the code above in any text editor, save it as *main.go* and compile it. To compile the code only run the *go build* command.

Example 1.2: Compilation with go build.

```
>> go build main.go

>> ls

main main.go

>> ./main

Save the world with Go!!!
```

If you run the code above in a Unix-compatible terminal you should get the same result. As you can see, the process is straight forward for this example. The `go build` command generates an executable file named `main`. This file can be executed (notice that ./ runs any executable file) displaying our message.

1.2 PASSING ARGUMENTS TO OUR PROGRAM

Now that we already know how to print a message, it would be nice if we could add some information from the outside. For example, what about computing the sum of two numbers? The idea is to pass two numbers to our program and tell the user what is the resulting sum.

First, we need to know how we can pass arguments to our program. This can be done using the `os.Args` variable. The example below is taken from here[2].

Example 1.3: Passing arguments.

```
package main

import (
    “fmt”
    “os”
)

func main() {

     argsWithProg := os.Args
     argsWithoutProg := os.Args[1:]

     arg := os.Args[3]

     fmt.Println(argsWithProg)
     fmt.Println(argsWithoutProg)
     fmt.Println(arg)
 }
```

There is a bunch of interesting things in this code. We have declared and initialized three variables called `argsWithProg`,`argsWithoutProg`, and `arg`. These variables contain all the arguments passed to our program, the arguments without the program name, and the argument in the third position respectively. If we compile and run the program like shown in the previous example we can understand how arguments passing works.

Example 1.4: Passing arguments output

```
>>> ./main Save the world with Go

[./main Save the world with Go]

[Save the world with Go]

world
```

The `os.Args` method returns an array (do not worry, this is explained in Chapter 3) containing all the arguments passed to the program including the name of the executable file. The variable `ArgsWithoutProg` has our input message (Save the world with Go). We removed the name of the program with the index `os.Args[1:]`. As mentioned before, this will be explained in more detail in the corresponding Chapter. In Go, arrays are indexed from 0 to $n-1$ with n the array length. Finally, in `arg` we get the argument at position 3 returning the word `world`.

Now that we explored how we can pass arguments to a program, we can do something with these parameters.

Example 1.5: Sum two numbers passed by arguments.

```
package main

import (
    “fmt”
    “os”
    “strconv”
)

func main() {

     argsWithProg := os.Args

     numA,  err := strconv.Atoi(argsWithProg[1])
     if err != nil {
       fmt.Println(err)
       os.Exit(2)
     }
     numB, err := strconv.Atoi(argsWithProg[2])
     if err != nil {
       fmt.Println(err)
       os.Exit(2)
     }
     result := numA + numB
     fmt.Printf(“%d + %d = %d\n”, numA, numB, result)
}
```

We can only run mathematical operations with numbers. This is a problem because arguments are passed as strings of characters. Fortunately, we can use the `strconv.Atoi` function to convert an integer number into a string representation. This may result in some conversion errors. For example:

- “42” 42

- “-33”   -33

- “4.2”   This is a conversion error because we are not expecting floating numbers.

- “thirteen” This is a conversion error because this is a textual representation of a number.

Is for this reason that `strconv.Atoi` returns two parameters. The first one is the integer number we can extract from the string. The second one is an error variable that will be filled in case there is an error. To know if there was a problem during the conversion process we can check if the error variable was filled or not. This is done in lines 14 and 19 with `if` statements. If the `err` variable contains some value (`!=nil`), we print the error and exit the program with `os.Exit(2)`.

If everything is correct, we compute the sum of `numA` and `numB` variables and print the result. To make it more appealing, we add some additional formatting to our output in line 24. You do not need to fully understand the meaning of `fmt.Printf` but you can guess that we are filling a string with `numA`, `numB`, and `result` values.

Now we can compile it and run like we did before:

Example 1.6: Sum numbers output.

```
>>> ./sum 2 2

2 + 2 = 4

>>> ./sum 42 -2

42 + -2 = 40

>>> ./sum 2 2.2

strconv.Atoi: parsing “2.2”: invalid syntax

>>> ./sum 2 two

strconv.Atoi: parsing “two”: invalid syntax
```

And voilà! our little calculator is ready. We can sum two numbers and detect when the input cannot be converted into an integer. However, there is one potential issue? What happens if we do not have any arguments? Consider this as an improvement exercise.

1.3 SUMMARY

In this Chapter, we showed how to write a Go program, compile it, and execute it. Additionally, we extended a basic example to include arguments passing and perform some mathematical operations, error control, and mathematical operations. If you feel comfortable with the content of this Chapter, consider exploring the basics of Go as presented in Chapter 2 .

CHAPTER 2 THE BASICS

This Chapter introduces the basics of Go. Like any programming language, Go uses variables, control loops, and data structures to create programs. You may find this Chapter not long enough to cover all the basics of a programming language. This is one of the greatest advantages of Go, its simplicity. The content of this Chapter reviews all the concepts a Go adopter must know to dive into the language.

2.1 PACKAGES AND IMPORTS

If you have already read Chapter 1 you will have noticed that every piece of code starts with a `package` statement. Go programs are organized into packages. A `package` is a group of one or more source files which code is accessible from the same `package`. Additionally, a `package` can be exported and used in other packages.

The `package main` is a special case that informs the Go compiler to consider that package as the entry point for an executable file. Actually, the `package main` is expected to have a `main` function in order to be compiled.

A `package` can be imported into other packages using the keyword `import`. The line `import “fmt”` makes the `fmt` package available to the source file. When importing a package, Go checks the `GOPATH` and `GOROOT` environment variables. The `GOPATH` points to the Go workspace and it is defined during the installation[3]. Similarly, `GOROOT` points to a custom Go installation. This variable should not be required unless a custom installation is done. The Go compiler will first check the `GOROOT` and then the `GOPATH` when importing a package.

2.1.1 Import third-party packages

Programs may require additional packages that are developed by third-parties. For example, the implementation of a database driver. Go is dramatically different importing third-party code when compared to other languages. Go forces code transparency by only compiling source code. This means that in order to import third-party packages, the source code must be locally available. Before import any third-party package you can use the Go command-line tool to download the code.

For example, to get the Mongo driver which is available at `http://go.mongodb.org/mongo-driver` we execute:

Example 2.1: Third-party package download using `go get`.

```
>>> go get -v go.mongodb.org/mongo-driver

 get “go.mongodb.org/mongo-driver”: found meta tag
get.metaImport{Prefix:”go.mongodb.org/mongo-driver”, VCS:”git”,
RepoRoot:”https://github.com/mongodb/mongo-go-driver.git”} at //go.mongodb.org/mongo-
driver?go-get=1

 go.mongodb.org/mongo-driver (download)

 package go.mongodb.org/mongo-driver: no Go files in
/XXXX/nalej_workspace/src/go.mongodb.org/mongo-driver
```

This downloads the source code from the external repository into our environment. Afterwards, we can import the code using `import “go.mongodb.org/mongo-driver/mongo”`. In some cases, the package name may not be very convenient. If required we can use an alias like shown below:

```
package main



import (

    myalias “go.mongodb.org/mongo-driver/mongo”

)

//…

client, err := myalias.NewClient(…))

//….
```

❓
Using `go get` to download third-party code is not a scalable solution. Fortunately, Go modules facilitate the acquisition of packages and their versioning for any project. Go modules are explained in Chapter 12 .

2.2 VARIABLES, CONSTANTS, AND ENUMS

Variables are the cornerstone of any programming language. This Section explores Go variables and special cases such as constants and enums.

2.2.1 Variables

Go is a strong statically typed language. This means that the type of the variable must be fixed at compilation time. Go syntax permits different alternatives when declaring variables as shown in Example 2.2.

Example 2.2: Declaration of variables.

```
package main

import “fmt”

func main() {

    var a int
    a = 42

    var aa int = 100

    b := -42

    c := “this is a string”

    var d, e string
    d, e = “var d”, “var e”

    f, g := true, false

    fmt.Println(a)
    fmt.Println(aa)
```

```
42
100
-42
this is a string
var d
var e
true
false
```

```
    fmt.Println(b)
    fmt.Println(c)
    fmt.Println(d)
    fmt.Println(e)
    fmt.Println(f)
    fmt.Println(g)

}
```

The basic construction of a variable is formed by the reserved word `var` followed by the variable name and its type. For example, `var a int` declares variable `a` of type `int`. Notice that this declares a variable with no value. The type and the value can be set in one line as shown in the example with `var aa int = 100`. Similarly, using the `:=` we can declare and assign a value to the variable. However, the type will be inferred by the compiler. In our example, `b := -42` has type `int` while `c := “this is a string”` is a string. Finally, we can declare and assign values to several variables in one line like in `f, g := true, false`.

2.2.2 Basic types

Go comes with the set of basic types described in Table 2.1.

Type	Description
`bool`	Boolean
`string`	String of characters
`int, int8, int16, int32, int64`	Signed integers
`uint, uint8, uint16, uint32, uint64, uintptr`	Unsigned integers
`byte`	Byte, similar to uint8
`rune`	Unicode code point
`float32, float64`	Floating numbers
`complex64, complex128`	Complex numbers

Table 2.1: Basic types

Integer numbers `int` and `uint` are platform-dependant and may vary from 32 to 64 bits. Using types such as `uint8` or `int16` set the variable size. For floating and complex numbers it is required to set the type size.

If you are familiar with other languages you may find the `rune` type something weird. This type is simply a character represented using UTF-8 which requires 32 bits instead of the classic 8 bits used in ASCII. Actually, `rune` is simply an alias for `int32`[4].

Example 2.3: Variables declaration

```go
package main

import “fmt”

func main() {
    var aBool bool = true
    var aString string = “yXXXy”
    var aComplex complex64 = 5i
    var aRune rune = ’€’

     fmt.Println(aBool)
     fmt.Println(aString)
     fmt.Println(aComplex)
     fmt.Println(aRune)
     fmt.Printf(”%U\n”,aRune)
     fmt.Printf(”%c\n”,aRune)
}
```

```
true
yXXXy
(0+5i)
8364

U+20AC
€
```

Example 2.3 shows how variables from different types are declared, assigned and printed. Running the code prints the variable values. The `rune` type requires special attention. By simply printing the variable we get the integer value 8364. However, the UTF-8 representation is U+20AC (format using `\%U`). A printable representation of the Euro symbol (€) is obtained with the `\%c` format.

2.2.3 Constants

A constant is a value defined at compilation time that cannot be changed. Apart from the impossibility of setting new values, constants are similar to variables.

Example 2.4: Constants declaration

```
package main

import (
    “fmt”
    “reflect”
)

const (
    Pi = 3.14
    Avogadro float32 = 6.022e23
)

func main() {
    fmt.Println(”What is the value of Pi? Pi is”, Pi)
    fmt.Println(reflect.TypeOf(Pi))
    fmt.Println(”Avogadro’s Number value is”, Avogadro)
```

```
What is the value of Pi? Pi is 3.14
float64
Avogadro’s Number value is 6.022e+23
float32
```

```
    fmt.Println(reflect.TypeOf(Avogadro))
}
```

π

Example 2.4 defines and Avogadro’s number. A constant can be defined in the same places a variable can be defined. Like variables, the type of a constant can be inferred. In our example, we defined `Pi` constant without type and `Avogadro` as `float32`. By default Go will select the largest available type. Is for this reason that `Pi` is a `float64` number even when a `float32` would be large enough[5].

2.2.4 Enums

Enums (enumerates) is a data type consisting of constant values. Classic examples are the days of the week, the months of the year, the states of a system, etc. Enums are intrinsically related to the `iota` keyword.

Example 2.5: Enums declaration

```
package main

import “fmt”

type DayOfTheWeek uint8

const(
    Monday DayOfTheWeek = iota
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    Sunday
```

```
Monday is 0
Wednesday is 2
Friday is 4
```

```
)


func main() {

    fmt.Printf(”Monday is %d\n”, Monday)
    fmt.Printf(”Wednesday is %d\n”, Wednesday)
    fmt.Printf(”Friday is %d\n”, Friday)

}
```

The code above defines an enum with the days of the week from Monday to Sunday. We have declared a `type` called `DayOfTheWeek` which is represented using `uint8` (an unsigned byte). Items from the same enumerate are expected to have consecutive values. In our example Monday is zero, Tuesday is one, Wednesday is two, etc. This is what `iota` does. It assigns consecutive values starting from 0 to the items of the enum. Notice that after the `iota` statement all the variables belong to the same type (`DayOfTheWeek`).

2.3 FUNCTIONS

Functions are a basic concept in Go. A function encapsulates a piece of code that performs certain operations or logic that is going to be required by other sections of the code. A function is the most basic solution to reuse code.

A function receives none or several parameters and returns none or several values. Functions are defined by keyword `func`, the arguments with their types, and the types of the returned values.

In example 2.6, the `sum` function returns the sum of two `int` arguments `a` and `b`.

Example 2.6: Function with two arguments.

```
package main

import “fmt”

func sum(a int, b int) int {
    return a + b
}

func main() {
    result := sum(2,2)
    fmt.Println(result)
}
```

```
4
```

It is possible to return multiple values like shown in example 2.7.

Example 2.7: Function returning several values.

```
package main

import “fmt”

func ops(a int, b int) (int, int) {
    return a + b, a - b
}

func main() {
    sum, subs := ops(2,2)
    fmt.Println(”2+2=”,sum, “2-2=”,subs)
    b, _ := ops(10,2)
    fmt.Println(”10+2=”,b)
```

```
2+2= 4 2-2=
0

10+2= 12
```

```
}
```

Functions can receive an undetermined number of arguments. These are called variadic functions. Example 2.8 declares a function to compute the sum of several numbers. Variadic arguments are identified with … before the type. These arguments must have the same type and can be treated as an array. How to iterate arrays is explained in more detail in section 3.2.2.

Example 2.8: Variadic function

```
package main

import “fmt”

func sum(nums …int) int {
    total := 0
    for _, a := range(nums) {
        total = total + a
    }
    return total
}

func main(){
    total := sum(1,2,3,4,5)
    fmt.Println(”The first five numbers sum is”,total)
}
```

```
The first five numbers sum is
15
```

Functions can receive other functions as arguments. In Example 2.9, the function `doit` expects a function and two integers as parameters. Notice that the `operator` argument is a function where we specify the type of its arguments and returned values. When using the `doit` function we can modify its behavior changing the `operator` argument. In this case, we can sum and multiply numbers using the corresponding functions.

Example 2.9: Functions as arguments.

```
package main

import “fmt”

func doit(operator func(int,int) int, a int, b int) int {
    return operator(a,b)
}

func sum(a int, b int) int {
    return a + b
}

func multiply(a int, b int) int {
    return a * b
}

func main() {
    c := doit(sum, 2, 3)
    fmt.Println(”2+3=”, c)
    d := doit(multiply, 2, 3)
    fmt.Println(”2*3=”, d)
}
```

```
2+3= 5
2*3=
6
```

Go permits anonymous functions, and these functions can be closures. A closure is a function that can refer to variables outside its body. This can be particularly useful to define inline functions or to solve complex problems like those that require recursion.

In Example 2.10, the function `accumulator` defines a closure function that is bounded to variable `i`. Statements `a := accumulator(1)` and `b := accumulator(2)` create two functions with different starting `i` variables. For this reason, for the same number of iterations outputs for `a` and `b` differ.

Example 2.10: Functions closure.

```
package main

import “fmt”

func accumulator(increment int) func() int {
    i:=0
    return func() int {
        i = i + increment
        return i
     }
}

func main() {

     a := accumulator(1)
     b := accumulator(2)

     fmt.Println(”a”,”b”)
     for i:=0;i<5;i++ {
         fmt.Println(a(),b())
     }
```

```
a b
1 2
2 4
3 6
4 8
5 10
```

```
}
```

2.4 POINTERS

Go works with arguments as values or references. When working with references we talk about pointers. A pointer addresses a memory location instead of a value. In Go pointers are identified following the C notation with a star. For a type `T`, `*T` indicates a pointer to a value of type `T`.

Example 2.11, has two functions `a` and `b` that set an incoming argument to zero. The code in the `main` function simply declares a variable `x` and call these functions. Notice that `a` does not change `x` value because it receives values as arguments. This is `a` works with a copy of variable `x`. However, function `b` sets `x` to zero because it receives a pointer to the variable. The operator `&` returns the pointer to the variable, which is of type `*int`. See how this operator returns the memory address of variable `x` with `fmt.Println(&x)`.

Example 2.11: Passing values and references to a function.

```
package main

import “fmt”

func a (i int){
    i = 0
}

func b (i *int) {
    *i = 0
}

func main() {
```

```
100
0

0xc00001c0a0
```

```
    x := 100

    a(x)
    fmt.Println(x)
    b(&x)
    fmt.Println(x)

    fmt.Println(&x)
}
```

How to decide when to use a pointer or a value depends on the use case. If a value is intended to be modified in different parts of the code, passing pointers seems reasonable.

2.5 NIL AND ZERO VALUES

A really important concept in Go is the zero value. When a variable is created and not initialized, the compiler automatically assigns it a default value. This value depends on the variable type. The keyword `nil` specifies a particular value for every non-initialized type. Notice that `nil` is not an undefined value like in other programming languages, `nil` is a value itself.

The output from Example 2.12 shows the zero value for various types. In general zero will be the zero value for numeric types such as `int` or `float`. Something similar occurs to `bool` although, 0 is considered to be `false`. In the case of `string` the empty string (`""`) is the zero value, not the numeric zero (`0`). For pointers, functions and other types `nil` is the default value.

Example 2.12: Zero values during initialization.

```
package main

import "fmt"
```

```
0
```

```

func main() {

    var a int
    fmt.Println(a)

    var b *int
    fmt.Println(b)

    var c bool
    fmt.Println(c)

    var d func()
    fmt.Println(d)

    var e string
    fmt.Printf(”[%s]”,e)
}
```

```
<nil>
false

<nil>
[]
```

2.6 LOOPS AND BRANCHES

No program is complete without control flow. Like in any other programming language, Go offers a set of constructions to define loops and branches.

2.6.1 If/else

The `if` statement permits the definition of branch executions using boolean expressions with the following construction.

```
if condition {
    // …
} else if condition {
    // …
} else {
   // …
}
```

Compared with other more verbose programming languages Go does not require parenthesis around the defined condition, only the braces.

Example 2.13 emulates tossing a coin and tell us if we get head or tail.

Example 2.13: `if/else` example.

```
package main

import (
    “fmt”
    “math/rand”
    “time”
)

func main() {
    rand.Seed(time.Now().UnixNano())
    x := rand.Float32()

    if x < 0.5 {
        fmt.Println(”head”)
    } else {
        fmt.Println(”tail”)
```

0

<nil>

false

<nil>

[]

```
17     }
18 }
```

In the example, we generate a random number using the `rand.Float32()` function. The `x` variable goes from 0 to 1 then, when `x` is less than 0.5 we get head, otherwise tail. The code in line 10 is just an initialization of the random generator. We set the initial random seed with the CPU time to get a different `x` value in every execution.

If you are familiar with other programming languages you may be wondering how is the ternary operator. Something like the one you can find in C: `condition ? statement : statement`. There is no such thing in Go. Those operators do not follow the simplicity concepts from Go.

2.6.2 Switch

The `switch` operator is particularly useful when you want to take action for several values of the same variable.

For example, the program described in Example 2.14 takes a number and prints the name of the corresponding finger. This is done by enumerating the values we want to control and adding the corresponding statement before we define the next value that requires a different statement. Finally, we set the special case `default` that is reached when the value is not controlled by any case.

Example 2.14: `switch` example.

```
package main

import “fmt”

func main() {
```

Index

```
    var finger int = 1

    switch finger {
    case 0:
         fmt.Println(”Thumb”)
     case 1:
         fmt.Println(”Index”)
     case 2:
         fmt.Println(”Middle”)
     case 3:
         fmt.Println(”Ring”)
     case 4:
         fmt.Println(”Pinkie”)
     default:
         fmt.Println(”Humans usually have no more than five fingers”)
     }
}
```

Branching with `switch` is very versatile. You do not always need to define the variable you want to check. You can simply start checking your cases and even you can use conditionals instead of constants.

Example 2.15 prints the quartile a random number belongs to. Compared with the previous example, we use conditions instead of constant values for the cases so we do not specify the variable to be observed in the `switch` statement. Notice that this is an alternative way to use the `if/else` logic.

Example 2.15: `switch`.

```
package main

```

```
import (
    “fmt”
    “math/rand”
    “time”
)

func main() {

     rand.Seed(time.Now().UnixNano())
     x := rand.Float32()

     switch {
     case x < 0.25:
         fmt.Println(”Q1”)
     case x < 0.5:
         fmt.Println(”Q2”)
     case x < 0.75:
         fmt.Println(”Q3”)
     default:
         fmt.Println(”Q4”)
     }
}
```

```
Q3
```

When several cases share the same logic, they can be stacked.

```
switch {
case 0:
case 1:
   // statement for 0 and 1
default:
   // default statement
}
```

Using `default` in every `switch` is a good practice to avoid errors and unexpected behaviours.

2.6.3 For loops

In Go the `for` construct permits to iterate depending on conditions and data structures. Iterations through structures are explained with further detail in Chapter 3.

Example 2.16: `for` loop example.

```
package main

import (
    "fmt"

)

func main() {

    x := 5

    counter := x

    for counter > 0 {
        fmt.Println(counter)
```

```
5
4
3
2
1
01234
5 is odd
Never
stop
```

```
16         counter–
17     }
18
19     for i:=0; i < x; i++ {
20         fmt.Print(i)
21     }
22     fmt.Println()
23
24     for {
25         if x % 2 != 0 {
26             fmt.Printf(”%d is odd\n”, x)
27             x++
28             continue
29         }
30         break
31     }
32
33     for {
34         fmt.Println(”Never stop”)
35         break
36     }
37
38 }
```

Example 2.16, shows four ways to construct a `for` loop. The first (line 14), iterates while a condition is satisfied. The loop prints all the numbers from `x` to 1 (5, 4, 3, 2, 1). The second loop (line 19) is very similar to a for loop declared in C/C++. We declare a variable `i`, a condition to be satisfied, and how the variable is going to be modified in every iteration. The output contains the numbers between 0 and `x` with zero included.

The third loop (line 24), has no conditions. In order to stop the loop we can use `break` and `continue`. They stop the loop and jump to the next iteration respectively. This particular loop will stop the execution if `x` is an even number by skipping the `if` branch. If the number is odd, `x` is modified and the `continue` statement jumps to the next iteration skipping the final `break` that will be reached in the next iteration. Consider that this is an artefact to demonstrate different flavours of a `for` construction and may not have a real practical sense.

Finally, in line 33 there is an infinite `for` loop. These loops are used to declare processes or operations that must be active for an undefined period (e.g. servers waiting for connections). The final `break` was added to avoid confusion during testing executions.

Go has no *while* loops. Actually, all the logic of a *while* construct can be achieved using `for`.

2.7 ERRORS

All error handling operations in Go are based on the type `error`. An `error` variable stores a message with some information. In situations where an error can occur, the usual way to proceed is to return a filled `error` informing about its cause. This can be done using the `errors.New` function.

Assume that there is a situation that may lead to an error, for example accessing a non-indexed item from a collection. In Example 2.17 , the function `GetMusketeer` returns the name of one of the Four Musketeers. Unfortunately, we cannot control the requested musketeer. If the `id` argument is outside the limits of the collection, we have an error. Notice that the signature function returns `(string, error)` types. The usual way to proceed in these situations is to fill the error with some information and assign the zero value to the return value.

Example 2.17: Example of error handling.

```
package main

import (
    “errors”
    “fmt”
    “math/rand”
    “time”
)

var Musketeers = []string{
    “Athos”, “Porthos”, “Aramis”, “D’Artagnan”,
}

func GetMusketeer(id int) (string, error){
    if id < 0 || id >= len(Musketeers) {
        return “”, errors.New(
            fmt.Sprintf(”Invalid id [%d]”,id))
    }
    return Musketeers[id], nil
}

func main() {
    rand.Seed(time.Now().UnixNano())
    id := rand.Int() % 6

    mosq, err := GetMusketeer(id)
    if err == nil {
        fmt.Printf(”[%d] %s”,id, mosq)
```

```
Invalid id [4]
…
[3]
D’Artagnan
…
[0] Athos
```

```
    } else {
        fmt.Println(err)
    }
}
```

Go has not `try/catch/except` idiom. According to the Go Faq, this was decided to remove convoluted code expressions.

2.8 DEFER, PANIC, AND RECOVER

The `defer` statement pushes a function onto a list. This list of functions is executed when the surrounding function ends. This statement is specially designed to ensure the correctness of the execution after the function ends. In particular, `defer` is useful to clean up resources allocated to a function.

Notice how in Example 2.18 the messages generated by `defer` statements are printed only after the main function ends. As can be extracted from the output, deferred functions are executed in inverse order as they were declared.

Example 2.18: `defer`.

```
package main

import “fmt”

func CloseMsg() {
    fmt.Println(”Closed!!!”)
}

func main() {
    defer CloseMsg()
```

```
Doing something…
Doing something else…
Certainly closed!!!
Closed!!!
```

```go

    fmt.Println("Doing something…")
    defer fmt.Println("Certainly closed!!!")
    fmt.Println("Doing something else…")

}
```

The built-in function `panic` stops the execution flow, executes deferred functions and returns control to the calling function. This occurs for all functions until the program crashes. A call to `panic` indicates a situation that goes beyond the control of the program. Example 2.19 , calls `panic` in the middle of a loop. The first deferred function to be executed is from `something`, then from `main()`. Observe that the panic message is printed in the last step and the last statement from `main` is never reached.

Example 2.19: `panic`.

```go
package main

import "fmt"

func something() {
    defer fmt.Println("closed something")
    for i:=0;i<5;i++ {
        fmt.Println(i)
        if i > 2 {
            panic("Panic was called")
        }
    }
}

```

```
0
1
2
3
closed something
closed main
panic: Panic was
called
```

```
func main () {
    defer fmt.Println(”closed main”)
    something()
    fmt.Println(”Something was finished”)
}
```

It may occur that under certain conditions when `panic` is invoked, the control flow can be restored. The `recover` built-in function used inside a deferred function can be used to resume normal execution. The scenario presented in Example 2.20 recovers the execution control after the panic inside function `something`. Calling `panic(i)` executes the deferred function where the recover is different from `nil`. The returned value is the parameter of the `panic` function. Observe that in this case the `main` function finished and we could print the final message.

Example 2.20: `recover`.

```
package main

import “fmt”

func something() {
   defer func() {
       r := recover()
       if r != nil{
           fmt.Println(”No need to panic if i=”,r)
        }
    }()
    for i:=0;i<5;i++ {
        fmt.Println(i)
        if i > 2 {
```

```
0
1
2
3
No need to panic if i=
3
Main was finished
closed main
```

```
            panic(i)
        }
    }
    fmt.Println("Closed something  normally")
}

func main () {
    defer fmt.Println("closed main")

    something()
    fmt.Println("Main was finished")
}
```

2.9 INIT FUNCTIONS

Now that we understand what are variable, functions, and imports we can better understand how Go starts a program execution. We have mentioned that every program in Go must have a `main` package with a `main` function to be executed. However, this imposes some limitations for certain solutions such as libraries. Imagine we import a library into our code. A library is not designed to be executed, it offers data structures, methods, functions, etc. Libraries probably do not even have a `main` package. If this library requires some initial configuration before invoked (initialize variables, detect the operating system, etc.) it looks impossible.

Go defines `init` functions that are executed once per package. When we import a package the Go runtime follows this order:

1. Initialize imported packages recursively.
2. Initialize and assign values to variables.
3. Execute `init` functions.

The output from Example 2.21 shows how the initialization follows the order described above. The `xSetter` function is invoked first, followed by `init`, and the `main` function.

Example 2.21: Go runtime initialization order.

```
package main

import “fmt”

var x = xSetter()

func xSetter() int{
    fmt.Println(”xSetter”)
    return 42
}

func init() {
    fmt.Println(”Init function”)
}

func main() {
    fmt.Println(”This is the main”)
}
```

```
xSetter
Init function
This is the main
```

The `init` function has no arguments neither returns any value. A package can have several `init` functions and they cannot be invoked from any part of the code.

Go does not allow importing a package if this is not used inside the code. However, we may only be interested in running the `init` functions of a package. This is what Go calls the side effects of a package. This is usually done in packages that perform some bootstrapping or registration operation. The special `import _` statement only calls the `init` functions of a package not requiring it to be used inside the code.

Example 2.22 imports package `a` to use its side effects. Observe that this package has two `init` functions that are executed before the `init` of the importing package.

Example 2.22: Main using `import _`

```
package main

import (
    “fmt”
    _ “a”
)

func init() {
    fmt.Println(”Init from my program”)
}

func main() {
     fmt.Println(”My program”)
}
```

Example 2.23: Package with `init` functions.

```
package a

import “fmt”

```

```
func init() {
    fmt.Println(”Init 1 from package a”)
}

func init() {
    fmt.Println(”Init 2 from package a”)
}
```

```
Init 1 from package a
Init 2 from package a
Init from my program
My program
```

2.10 SUMMARY

This Chapter overviews Go basics by introducing concepts such as packages, variables or errors. First, we explain how to use owned and third-party packages. We provide an overview of variables, constants and enums to explain how they are used inside of functions. Flow control is explained and how to manage errors with special control functions. Understanding this Chapter is necessary to continue with the next Chapter 3 where we explore advanced data structures.

CHAPTER 3

ARRAYS, SLICES, AND MAPS

So far we have introduced the basics about variables and how to define a workflow using branches and loops. However, we have not explored data structures. Go offers other powerful data structures that are extensively used: arrays, slices, and maps. Arrays and maps are common to other programming languages. Slices are a particular Go construct. In this Chapter we explore these data structures and provide examples of how they can be used.

3.1 ARRAYS

By definition, an array is an indexed sequence of elements with a given length. Like any other Go variable, arrays are typed and their size is fixed.

Example 3.1 shows different ways to declare arrays. Variable `a` (line 7) declares an array of integers of size 5. By default, this array is filled with zeros. Notice that printing the array returns the sequence of numbers between brackets. We can assign values to the array in a single line like in the case of `b`. Similarly, `c` is declared as a 5 integers array with only three values assigned.

Example 3.1: Arrays declaration

```
1 package main
2
3 import “fmt”
4
5 func main() {
6
7     var a[5] int
8     fmt.Println(a)
```

```
[0 0 0 0 0]
[0 1 2 3 4]
[0 1 2 0 0]
```

```go

    b := [5]int{0,1,2,3,4}
    fmt.Println(b)

    c := [5]int{0,1,2}
    fmt.Println(c)

}
```

Every array has a `len` function that returns the array length. For example, `len(a)` will return 5 for the previous example.

3.2 SLICES

We have said that arrays have a fixed size. This makes them not very flexible in certain scenarios. Go offers a type called `slice` defined as a "descriptor for a contiguous segment of an underlying array and provides access to a numbered sequence elements from that array"[6]. In other words, a `slice` is a reference to an array. The `slice` itself does not store any data but offers a view of it. Table 3.1 describes available ways to select slices.

Index	Selected element(s)
`a[0]`	Element at position 0
`a[3:5]`	Elements from position 3 to 4
`a[3:]`	All the elements starting at position 3
`a[:3]`	All the elements from the start till position 2
`a[:]`	All the elements

Table 3.1: Examples of slices selections

Arrays and slices items are indexed from to with the length of the array.

Example 3.2 and its output shows how to obtain different slices from a given array.

Example 3.2: Slices indexing.

```go
package main

import (
    “fmt”
)

func main(){
    a := [5]string{”a”,”b”,”c”,”d”,”e”}

     fmt.Println(a)
     fmt.Println(a[:])
     fmt.Println(a[0])
     fmt.Println(a[0],a[1],a[2],a[3],a[4])
     fmt.Println(a[0:2])
     fmt.Println(a[1:4])
     fmt.Println(a[:2])
     fmt.Println(a[2:])
}
```

```
[a b c d e]
[a b c d e]
a
a b c d e
[a b]
[b c d]
[a b]
[c d e]
```

In Go most of the time, we work with `slice` rather than arrays. Working with arrays or slices is very similar and does not require additional effort from the programmer. Example 3.3 uses `reflect.TypeOf` to print the type of different objects.

Example 3.3: Type differences between array, slice, and item.

```go
package main

import (
    “fmt”
    “reflect”
)

func main() {
    a := [5]string{”a”,”b”,”c”,”d”,”e”}

     fmt.Println(reflect.TypeOf(a))
     fmt.Println(reflect.TypeOf(a[0:3]))
     fmt.Println(reflect.TypeOf(a[0]))
}
```

```
[5]string
[]string
string
```

You can check below that the output from this program differs for every statement. The difference is subtle but very important. `[5]string` is an array with a fixed size of five elements. However, `[]string` is a `slice` without a defined size.

3.2.1 Length and capacity

We have mentioned that a `slice` has no fixed size because it is a view of an undergoing storage array. An important difference between an array and a `slice` is the concept of capacity. While an array allocates a fixed amount of memory that is directly related to its length, a `slice` can reserve a larger amount of memory that does not necessarily have to be filled. The filled memory corresponds to its length, and all the available memory is the capacity. Both values are accessible using functions `len` and `cap`.

Example 3.4: Differences between length and capacity

```go
package main

import “fmt”

func main() {

    a := []int{0,1,2,3,4}

    fmt.Println(a, len(a), cap(a))

     b := append(a,5)
     fmt.Println(b, len(b), cap(b))
     b = append(b,6)
     fmt.Println(b, len(b), cap(b))

     c := b[1:4]
     fmt.Println(c, len(c), cap(c))

     d := make([]int,5,10)
     fmt.Println(d, len(d), cap(d))
     // d[6]=5 —> This will fail

}
```

```
[0 1 2 3 4] 5 5
[0 1 2 3 4 5] 6 10
[0 1 2 3 4 5 6] 7
10
[1 2 3] 3 9
[0 0 0 0 0] 5 10
```

Example 3.4 prints the length and capacity of various variables. Let this example serve to introduce `make` and `append` built-in functions. Variable `a` is an array and therefore, its length and capacity are the same. However, `b` which is an slice built by appending number 5 to `a` has different length and capacity. If a second `append` is done length changes, but capacity does not change.

Variable `c` shows how creating a new slice from an existing one, does not necessarily inherit its length and capacity. `c` only has three elements with capacity 9. This occurs because we are selecting elements of slice `b` starting at index 1 which results in a `slice` with the original capacity minus one. Finally, variable `d` is built using the `make` function. This function takes a slice type, its length and capacity as arguments. If no capacity is set, this will be the same as the length.

The last statement from the example (line 21) illustrates a situation that triggers a runtime error. The element at position 6 is requested in a slice with capacity 10 and length 5. Any element with a position equal to or greater than length cannot be accessed independently of the slice capacity.

3.2.2 Iteration

The most common operation you can find in a `slice` is the iteration through its items. Any `for` loop is a good candidate to do this. Go simplifies iterations through collections with the `range` clause to.

Example 3.5: `slice` iteration using the `range` clause.

```
package main

import “fmt”

func main(){
    names := []string{
        “Jeremy”, “John”, “Joseph”,
    }

     for i:=0;i<len(names);i++{
         fmt.Println(i,names[i])
     }
```

```
0 Jeremy
1 John
2 Joseph
0 Jeremy
1 John
2
Joseph
```

```
13
14     for position, name := range names {
15         fmt.Println(position,name)
16     }
17 }
```

Example3.5 prints all the items of a collection of strings using two approaches. The first one (line 10), declares an index variable `i` to increment the index of the item to be retrieved. The second approach (line 14) using `range` prints the same output in a less verbose way. For arrays and slices, `range` returns the index of the item and the item itself for every iteration of the loop. The iteration is always done incrementally from index 0 to index n-1.

Notice that the item from the `slice` returned by `range` is a copy of the item. Therefore, modifying this variable inside the loop will not modify the iterated array. The next piece of code shows how when modifying the item returned by `range` we cannot modify the `slice` we are iterating through. A correct approach to modify the iterated `slice` is to access the original variable with the corresponding index.

```
names := []string{"Jeremy", "John", "Joseph"}


for _, name := range(names){
    // name is a copy
    name = name + "_changed"
}
fmt.Println(names)


for position, name := range(names){
    // this modifies the original value
    names[position] = name + "_changed"
}

fmt.Println(names)
```

3.3 MAPS

A `map` is a construct that maps a key with a value. Keys are intended to be unique and can be of any type that implements `==` and `!=` operators.

Example 3.6 shows how to instantiate a map that stores string keys and integer values. Maps are defined by `map[K]V` where `K` is the key type and `V` is the value type. By default, uninitialized maps are `nil`.

Notice that the statement `ages[”Jesus”]=33` (line 10) which is intended to set the value `33` for the key `“Jesus”` is intended to fail. This is because any `map` needs to be instantiated. This can be done using the `make` builtin function (line 12). For `map` the `make` function expects the map type (`map[K]V`) and optionally an initial size. This size is optional as the size can be modified during runtime. Finally, maps can also be initialized by indicating key-value pairs as shown in line 16.

Example 3.6: `map` creation.

```
1 package main

2

3 import “fmt”

4

5 func main() {

6     var ages map[string]int

7     fmt.Println(ages)

8

9     // This fails, ages was not initialized

10     // ages[“Jesus“] = 33
```

```
map[]

0 false

33 true

map[Jesus:33
Mathusalem:969]
```

```

    ages = make(map[string]int,5)
    // Now it works because it was initialized
    ages[”Jesus”] = 33

    ages = map[string]int{
        “Jesus”: 33,
        “Mathusalem”: 969,
    }
    fmt.Println(ages)
}
```

Items from a `map` can be retrieved using any key type value like appears on Example 3.7 (`birthdays[”Jesus”]`). Actually, this operation returns two items, one with the expected value and `true` if the item was found. In case, the key was not found, the value would be `nil`.

Example 3.7: `map` access operations.

```
package main

import “fmt”

func main() {
    birthdays := map[string]string{
        “Jesus”: “12-25-0000”,
        “Budha”: “563 BEC”,
    }
     fmt.Println(birthdays,len(birthdays))

```

```
map[Budha:563 BEC Jesus:12-25-0000]
2
12-25-0000 true
map[Budha:563 BEC] 1
Did we find when its Xmas? false
map[Budha:563 BEC Jesus:12-25-
0000]
```

```
    xmas, found := birthdays[”Jesus”]
    fmt.Println(xmas, found)

    delete(birthdays, “Jesus”)
    fmt.Println(birthdays,len(birthdays))

    _, found = birthdays[”Jesus”]
    fmt.Println(”Did we find when its Xmas?”, found)

    birthdays[”Jesus”]=”12-25-0000”
    fmt.Println(birthdays)

}
```

New items can be added like in `birthdays[”Jesus”]=”12-25-0000”`. Additionally, items can be removed using the built-in function `delete`.

3.3.1 Iteration

To iterate a `map` we would require the collection of keys. Fortunately, the `range` built-in function offers a simple solution to iterate through all the key-value pair of any `map`. The rules explained for slices apply in the case of maps. For every iteration `range` returns the current key and value.

Example 3.8: `map` iteration using range.

```
package main

import “fmt”

func main(){
```

Month	Sales
Jan	34345
Feb	11823

```go
    sales := map[string]int {
        “Jan”: 34345,
        “Feb”: 11823,
        “Mar”: 8838,
         “Apr”: 33,
     }

     fmt.Println(”Month\tSales”)
     for month, sale := range sales {
         fmt.Printf(”%s\t\t%d\n”,month,sale)
     }
}
```

```
Mar    8838
Apr    33
```

⚠ The function `range` does not guarantee the same order for consecutive iterations.

3.4 SUMMARY

This Chapter shows how to declare, manipulate, and iterate through arrays, slices, and maps. These three native structures are extremely versatile and widely used in any Go program. The next Chapter exposes the tools Go offers to define their own data structures.

CHAPTER 4

STRUCTS, METHODS, AND INTERFACES

In 1976 Niklaus Wirth published “Algorithms + Data Structures = Programs” [12]. It became a seminal book and its title is the best summary of what a computer program can be. Previous chapters explained the necessary items to define algorithms (branches, loops, variables, etc.). In this Chapter, we dig into how Go works with data structures.

4.1 STRUCTS

If you are familiar with languages such as C/C++ you will find structs a relatively known construct. In a few words, a `struct` is a sequence of elements named fields. Each field has a name and a type.

In Example 4.1 we have a struct named `Rectangle` which represents the geometric figure with `Height` and `Width` fields. A `struct` can be instantiated in different ways. Not passing any argument (line 11) initializes every field in the `struct` with the zero value. Initial values can be set passing them as arguments in the same order they were declared (line 14). Additionally, we can set what value corresponds to what field using the field name (line 17). In this way, we do not need to set all the fields (line 20). Missing fields are set to their default value.

Example 4.1: Structure definition for a rectangle.

```
1 package main

2

3 import “fmt”

4

5 type Rectangle struct{
```

```
{0 0}
{4 4}
{3
10}
```

```
    Height int
    Width  int
}

func main() {
    a := Rectangle{}
    fmt.Println(a)

    b := Rectangle{4,4}
    fmt.Println(b)

    c := Rectangle{Width: 10, Height: 3}
    fmt.Println(c)

    d := Rectangle{Width: 7}
    fmt.Println(d)
}
```

```
{0
7}
```

This flexibility creating structs can be inconvenient. In Example 4.2, `a` is a rectangle with no `Width` field value. In this case, it does not make any sense to a rectangle with a width of zero. One possible solution is to define a `NewRectangle` function that requires all the necessary arguments to create this `struct`. Notice, that this function returns a pointer to the `struct` instead of a value.

Example 4.2: Struct constructor.

```
package main

import “fmt”

```

```
{7 0}
&{2
```

```go
type Rectangle struct{
    Height int
    Width  int
}

func NewRectangle(height int, width int) *Rectangle {
    return &Rectangle{height, width}
}

func main() {
    a := Rectangle{Height: 7}
    fmt.Println(a)

    r := NewRectangle(2,3)
    fmt.Println(r)
    fmt.Println(*r)
}
```

```
3}
{2 3}
```

In Go it does not exist the concept of a constructor like in other languages. A `struct` is a very flexible construct that can be defined in many ways. When working with structs it is very important to take into consideration fields zero values and how these values may impact the code. In many cases, it is a good practice to define constructors, especially when certain values are not valid.

```go
func NewRectangle(height int, width int) (*Rectangle,error) {
    if height <= 0 || width <= 0 {
        return nil, errors.New(”params must be greater than zero”)
    }
    return &Rectangle{height, width},nil
}
```

```
…

r, err := NewRectangle(2,0)

if err != nil {

    …

}
```

4.1.1 Anonymous structs

Go permits the definition of anonymous structs like the one shown in Example 4.3 (line 15). Compared with a regular struct like `Circle` printing the struct brings a similar result. However, we cannot print its name as we do with type `Circle`. The fields from the anonymous function can be modified like done with regular structs. Notice that these anonymous structures can be compared with other structures if and only if they have the same fields (line 26).

Example 4.3: Anonymous struct.

```
1 package main

2

3 import (

4     “fmt”

5     “reflect”

6 )

7

8 type Circle struct {

9     x int

10     y int

11     radius int

12 }

13
```

```
{x:1 y:2 radius:3}

struct { x int; y int; radius int }

{x:10 y:10 radius:3}

main.Circle

{x:3 y:2 radius:3}

{x:10 y:10 radius:3}

struct { x int; y int; radius int
}
```

```
func main() {
    ac := struct{x int; y int; radius int}{1,2,3}
    c := Circle{10,10,3}

    fmt.Printf(”%+v\n”, ac)
    fmt.Println(reflect.TypeOf(ac))
    fmt.Printf(”%+v\n”,c)
    fmt.Println(reflect.TypeOf(c))

    ac.x=3
    fmt.Printf(”%+v\n”, ac)

    ac = c
    fmt.Printf(”%+v\n”, ac)
    fmt.Println(reflect.TypeOf(ac))
}
```

4.1.2 Nested structs

Structs can be nested to incorporate other structs definition. Example 4.4 defines a `Circle` type using a `Coordinates` type. Obviously, instantiating a `Circle` requires the instantiation of a `Coordinate` type.

Example 4.4: Nested structs.

```
package main

import “fmt”

type Coordinates struct {
```

```
{center:{x:1 y:2}
radius:3}
```

```
    x int
    y int
}

type Circle struct {
    center Coordinates
    radius int
}

func main() {
    c := Circle{Coordinates{1, 2}, 3}
    fmt.Printf(”%+v\n”, c)
}
```

4.1.3 Embedded structs

To embed a struct in other structs, this has to be declared as a nameless field. In Example 4.5 by embedding the `Coordinates` struct in the `Circle` type we make fields `x` and `y` directly accessible. The coordinates instance can also be accessed like the `Coordinates` field.

Example 4.5: Embedded structs.

```
package main

import “fmt”

type Coordinates struct {
    x int
    y int
}
```

```
{Coordinates:{x:1 y:2}
radius:3}

{x:1 y:2}

1 2
```

```go

type Circle struct {
    Coordinates
    radius int
}

func main() {
    c := Circle{Coordinates{1, 2}, 3}
    fmt.Printf("%+v\n", c)
    fmt.Printf("%+v\n", c.Coordinates)
    fmt.Println(c.x, c.y)
}
```

Embedding structs can be done only if the compiler find no ambiguities. Considering the following code:

```go
// …
type A struct { fieldA int }


type B struct { fieldA int }


type C struct {
    A
    B
}
// …
a := A{10}
b := B{20}
c := C{a,b}
```

```
// —> Ambiguos access

// fmt.Println(c.fieldA)

fmt.Println(c.A.fieldA,c.B.fieldA)
```

Because `fieldA` may belong to different structs, this access is ambiguous triggering an error during compilation. We have to specify which struct this field belongs to.

4.2 METHODS

In the object-oriented world, a method is defined as a procedure associated with a class. In Go there is not such a thing as classes. However, Go defines methods as a special function with a receiver. The receiver sets the type that can invoke that function.

Assume we work with the `Rectangle` type and we want to add some operations such as computing the surface. In Example 4.6 , the method `Surface() int` is a function with the receiver `(r Rectangle)`. This means that any type `Rectangle` can invoke this method. Inside the method, the fields `Height` and `Width` from the current instance `R` are accessible.

Example 4.6: Definition of methods for a `Rectangle` type.

```
package main

import “fmt”

type Rectangle struct{
    Height int
    Width  int
}

```

```
rectangle {2 3} has surface
6
```

```
func (r Rectangle) Surface() int {
    return r.Height * r.Width
}

func main() {
    r := Rectangle{2,3}
    fmt.Printf(”rectangle %v has surface %d”,r, r.Surface())
}
```

Receivers are very important because they define the type “receiving” the logic inside the method. In Example 4.7 , we define two methods with the same logic. Method `Enlarge` receives a value of type `Rectangle` and method `EnlargeP` receives a type `*Rectangle`. If you follow the output, you can see how `Enlarge` does not modify any field of the original variable, while `EnlargeP` does. This happens because the `EnlargeP` receives the pointer to `rect` whereas, `Enlarge` receives a copy.

Example 4.7: Value and pointer receivers.

```
package main

import “fmt”

type Rectangle struct{
    Height int
    Width  int
}

func (r Rectangle) Enlarge(factor int) {
    r.Height = r.Height * factor
    r.Width = r.Width * factor
```

```
{2 2}
{2 2}
{4
4}
```

```
}

func (r *Rectangle) EnlargeP(factor int) {
    r.Height = r.Height * factor
    r.Width = r.Width * factor
}

func main() {
    rect := Rectangle{2,2}
    fmt.Println(rect)

    rect.Enlarge(2)
    fmt.Println(rect)

    rect.EnlargeP(2)
    fmt.Println(rect)
}
```

In Example 4.7, the `EnlargeP` method requires a pointer. However, we invoke the method with `rect.EnlargeP(2)` and `rect` is not a pointer. This is possible because the Go interpreter translates this into `(&rect).EnlargeP(2)`.

If a method can have value or pointer receivers, which one should you use? Generally, using pointers is more efficient because it reduces the number of copies. However, in some situations you may be more comfortable with value receivers. In any case, you should be consistent and do not mix them.

4.2.1 Embedded methods

When a struct is embedded into other structs its methods are made available to the second one. This acts as some sort of inheritance in Go. In Example 4.8, the type `Box` embeds the type `Rectangle`. Observe how the method `Volume` can directly invoke the `Surface` method from `Rectangle` to compute the volume of the box.

Example 4.8: Embedded methods.

```
package main

import “fmt”

type Rectangle struct {
    Height int
    Width int
}

func (r Rectangle) Surface() int {
    return r.Height * r.Width
}

type Box struct {
    Rectangle
    depth int
}

func (b Box) Volume() int {
    return b.Surface() * b.depth
}

```

```
{Rectangle:{Height:3 Width:3}
depth:3}

Volume 27
```

```
func main() {
    b := Box{Rectangle{3,3}, 3}
    fmt.Printf(”%+v\n”,b)
    fmt.Println(”Volume”, b.Volume())
}
```

Remember that embedded methods only work if there is no ambiguity in its definition. Consider the following example:

```
type A struct {}
func (a A) Hi() string {
    return “A says Hi”
}

type B struct {}
func (b B) Hi() string {
    return “B says Hi”
}

type Greeter struct{
    A
    B
}

func (g Greeter) Speak() string {
    // return g.Hi() –> This method belongs to A or B?
    return g.A.Hi() + g.B.Hi()
}
```

Invoking method `Hi` in `Greeter` is not possible because the compiler cannot determine which type `A` or `B` is the owner. This has to be solved by specifying the method owner.

4.3 INTERFACES

An `interface` is a collection of methods with any signature. Interfaces do not define any logic or value. They simply define a collection of methods to be implemented. A type `A` implements an interface `I` if and only if all the methods from `I` are implemented in `A`.

Example 4.9: Interface declaration

```
package main

import “fmt”

type Animal interface {
    Roar() string
    Run() string
}

type Dog struct {}

func (d Dog) Roar() string {
    return “woof”
}

func (d Dog) Run() string {
    return “run like a dog”
```

```
woof and run like a dog
meow and run like a
cat
```

```
}

type Cat struct {}

func (c *Cat) Roar() string {
    return "meow"
}

func (c *Cat) Run() string {
    return "run like a cat"
}

func RoarAndRun(a Animal) {
    fmt.Printf("%s and %s\n", a.Roar(), a.Run())
}

func main() {
    myDog := Dog{}
    myCat := Cat{}

    RoarAndRun(myDog)
    RoarAndRun(&myCat)
}
```

Example 4.9 declares the `Animal interface` with two methods `Roar` and `Run`. Next we have `Dog` and `Cat` types that define these methods. Automatically both types are considered to implement interface `Animal`. Function `RoarAndRun` receives an `Animal` type, so we can invoke the `Roar` and `Run` methods independently of the final argument type.

Notice that method receivers from `Dog` and `Cat` are different. Because all the methods of the `interface` must be implemented in order to consider a type to implement `interface` of type `Animal`, certain combinations in the example can fail. For example:

```
RoarAndRun(myDog) // –> It works

RoarAndRun(&myDog) // –> It does not work
```

However, if we try to invoke `RoarAndRun` for a `Cat` type (not a pointer) we find that it fails.

```
RoarAndRun(&myCat) // –> It works

RoarAndRun(myCat) // –> It does not work
```

`RoarAndRun(myCat)` does not work because the receivers of the methods in the `Cat` type are pointers while we pass an argument by value. In case the method assigns a new value to any field, this cannot be reflected in the original caller because it is a copy. This difference may have an impact on your code. We have already seen that methods with pointer receivers can modify the values in the invoking struct (Example 4.7).

We can see this in Example 4.10 where `Person` implements `Greeter`. However, instantiating `p{}` does not return a `Greeter` type. Why? Because the method `SayHello` has a pointer receiver. This limits this method to `*Person` type. This may have an impact on your code if those types that implement interfaces are not consistently defined using pointers or values.

Example 4.10: Interface declaration

```
package main

import “fmt”

type Greeter interface {
    SayHello() string
```

```
Hi! This is me
John
```

```
}

type Person struct{
    name string
}

func (p *Person) SayHello() string {
    return “Hi! This is me “+ p.name
}

func main() {

    var g Greeter

    p := Person{”John”}
    // g = p —> Does not work
    g = &p
    fmt.Println(g.SayHello())
}
```

You may consider implementing an interface using methods with pointer and value receivers simultaneously to be able to work with both flavors. Something like

```
func (p *Person) SayHello() string {
    return “Hi! This is me “+ p.name
}
// …
func (p Person) SayHello() string {
    return “Hi! This is me “+ p.name
}
```

This is some sort of method overloading and is not allowed in Go.

4.3.1 Empty interface

A special case of `interface` is the empty interface `interface{}`. This interface has no methods and it is implemented by any type. This `interface` is particularly useful for those situations where we cannot know the data type beforehand. As shown in Example 4.11 an empty `interface` can be assigned any value.

Example 4.11: Using the empty interface.

```
package main

import “fmt”

func main() {
    var aux interface{}

    fmt.Println(aux)

    aux = 10
    fmt.Println(aux)

    aux = “hello”
    fmt.Println(aux)
}
```

```
<nil>
10
hello
```

The empty interface is a very ambiguous context for a typed language like

Go. In many situations, it is necessary to know the underlying data type of the interface. Otherwise, it is not possible to know how to proceed with that data. A practical way to find the variable type is using a `switch` statement. Example 4.12 fills a slice of empty interfaces with an integer, a string, and a boolean. The `switch` statement can extract the type of value in runtime. This can be used to define how to operate with the value. In the example, we modify a print statement accordingly to the type. Notice that `“%T”` in the print statement gets the name of the value type.

Example 4.12: Explore the type of an empty interface.

```
package main

import “fmt”

func main() {

    aux := []interface{}{42, “hello”, true}

    for _, i := range aux {
         switch t := i.(type) {
         default:
             fmt.Printf(”%T –> %s\n”, t, i)
         case int:
             fmt.Printf(”%T –> %d\n”, t, i)
         case string:
             fmt.Printf(”%T –> %s\n”, t, i)
         case bool:
             fmt.Printf(”%T –> %v\n”, t, i)
         }
     }
```

```
int —> 42
string —>
hello
bool —> true
```

```
}
```

Go cannot be fully considered an object-oriented language. Actually, concepts such as classes or hierarchy of classes do not exist. Similar functionality can indeed be obtained using the current language definition. However, we cannot say Go is an object-oriented language.

4.4 SUMMARY

This Chapter introduces the concepts of struct, methods, and interfaces used in Go. These concepts are fundamental pieces of the Go language and will appear across different sections of this book. Additionally, we explained certain situations that may seem weird to early adopters such as the difference between value and pointer receivers in methods.

CHAPTER 5

REFLECTION

In programming, reflection is the ability of a program to examine, introspect and modify its structure and behavior [9]. In other words, this is a form of metaprogramming. Reflection is an extremely powerful tool for developers and extends the horizon of any programming language. Unfortunately, this comes with additional complexity.

In this Chapter, we introduce the `reflect`[7] package and explain through examples how to explore, introspect and modify our code in run-time. We split this explanation into two sections, according to how Go organizes the reflection package. Finally, we introduce how Go uses tags to enhance fields information.

5.1 REFLECT.TYPE

A starting point to understand reflection in Go, is to remember the concept of empty interface `interface{}` seen in Section 4.3.1. The empty interface can contain whatever type. For example:

```
unknown := interface{}

a := 16

unknown = a
```

This code works because `interface{}` accepts everything, therefore we can say that everything is an empty interface. This is very important because the function `reflect.TypeOf` is the main entrypoint for code introspection and receives and empty interface as argument. Observe Example 5.1 and how it obtains the type of variables `a` and `b`.

Example 5.1: `reflect.TypeOf` with basic types.

```
package main

import (
```

```
int32
```

```
    “fmt”
    “reflect”
)

func main() {
    var a int32 = 42
    var b string = “forty two”

    typeA := reflect.TypeOf(a)
    fmt.Println(typeA)

    typeB := reflect.TypeOf(b)
    fmt.Println(typeB)
}
```

```
string
```

The function `TypeOf` returns type `Type` with a set of methods to for code introspection. Example 5.2 explores a struct with two fields. Notice that we can navigate through the fields of any type, accessing its name and type.

Example 5.2: `reflect.TypeOf` with structs.

```
package main

import (
    “fmt”
    “reflect”
)

type T struct {
    A int32
```

```
main.T
A int32
B
string
```

```
    B string
}

func main() {
    t := T{42, “forty two”}

    typeT := reflect.TypeOf(t)
    fmt.Println(typeT)

    for i:=0;i<typeT.NumField();i++{
        field := typeT.Field(i)
        fmt.Println(field.Name,field.Type)
    }
}
```

Beyond type exploration, we can check if a type implements an interface. This can be done using the `Implements` method as shown in Example 5.3. This is a good example of how interfaces work in Go. The method `Add` has a pointer receiver (`*Calculator`) for that reason, the `main.Calculator` type does not implement the `Adder` interface.

Example 5.3: `reflect.TypeOf` with structs.

```
package main

import (
   “fmt”
   “reflect”
)

```

```
main.Calculator false

*main.Calculator true
```

```
type Adder interface{
    Add (int, int) int
}

type Calculator struct{}

func(c *Calculator) Add(a int, b int) int {
     return a + b
}

func main() {

    var ptrAdder *Adder
    adderType := reflect.TypeOf(ptrAdder).Elem()

    c := Calculator{}

    calcType := reflect.TypeOf(c)
    calcTypePtr := reflect.TypeOf(&c)

    fmt.Println(calcType, calcType.Implements(adderType))
    fmt.Println(calcTypePtr, calcTypePtr.Implements(adderType))
}
```

Using `reflect.Type` we can explore any kind of struct, with any number of fields and types. Example 5.4 uses a recursive type inspector that prints the structure of any given type. The inspector iterates through the struct fields even if they are other structs. This is done obtaining the `Kind` of the field and comparing it with a struct (f.Type.Kind() == reflect.Struct). You can check how this code, does not skip unexported fields.

Example 5.4: Recursive struct inspector.

```
package main

import (
    “fmt”
    “reflect”
    “strings”
)

type T struct {
    B int
    C string
}

type S struct {
    C string
    D T
    E map[string]int
}

func printerReflect(offset int, typeOfX reflect.Type) {
    indent := strings.Repeat(” ”,offset)
    fmt.Printf(“%s %s: %s {\n”,indent, typeOfX.Name(), typeOfX.Kind())
    if typeOfX.Kind() == reflect.Struct {
        for i:=0;i<typeOfX.NumField();i++{
            innerIndent := strings.Repeat(” ”,offset+4)
            f := typeOfX.Field(i)
            if f.Type.Kind() == reflect.Struct {
```

```
S: struct {
  C: string
  T: struct {
    B: int
    C: string
  }
  E:
map[string]int
}
```

```
                printerReflect(offset+4, f.Type)
            } else {
                fmt.Printf(“%s %s: %s\n”,innerIndent, f.Name, f.Type)
            }
        }
    }
    fmt.Printf(“%s }\n”, indent)

}

func main() {

    x := S{“root”,
        T{42, “forty two”},
        make(map[string]int),
    }

    printerReflect(0, reflect.TypeOf(x))
}
```

5.2 REFLECT.VALUE

We have seen how `reflect.Type` permits code introspection. However, the `reflect.Type` type cannot access field values. This functionality is reserved to the `reflect.Value` type. Actually, from a `reflect.Value` type we can access its `reflect.Type` type. Example 5.5 uses the same variables from Example 5.1 . Notice that in this case we can print the variables current value.

Example 5.5: `reflect.ValueOf`.

```
package main
```

```

import (
    “fmt”
    “reflect”
)

func main() {
    var a int32 = 42
     var b string = “forty two”

     valueOfA := reflect.ValueOf(a)
     fmt.Println(valueOfA.Interface())

     valueOfB := reflect.ValueOf(b)
     fmt.Println(valueOfB.Interface())
}
```

```
42
forty
two
```

In order to know what type implements a value, we can be compare it with the `Kind` type returned by method `Kind()`. The type `Kind` is a number representing one of the types available in Go (`int32`, `string`, `map`, etc.). This can be combined with a `switch` statement as shown in Example 5.6 to identify what type are we working with.

Example 5.6: `switch` using `reflect.Kind`.

```
package main

import (
    “fmt”
    “reflect”
)
```

```
Int32 with value 42
String with value forty
two
```

```

func ValuePrint(i interface{}) {
    v := reflect.ValueOf(i)
     switch v.Kind() {
     case reflect.Int32:
         fmt.Println(“Int32 with value”, v.Int())
     case reflect.String:
         fmt.Println(“String with value”, v.String())
     default:
         fmt.Println(“unknown type”)
     }
}

func main() {
     var a int32 = 42
     var b string = “forty two”

     ValuePrint(a)
     ValuePrint(b)
}
```

Example 5.7 uses `reflect.Value` with a struct to print the field values. The reflected value of variable `t` is correctly printed. Similarly, we can print the value of every field in the struct. Notice the difference between printing `field.String()` and `field`. For numeric values `field.String()` returns a string like `<int32 Value>`. The string informs that there is an integer value in that field. However, `fmt.Println(field)` works as expected. This occurs because the function prints the corresponding value when it receives `Value` types.

Example 5.7: `reflect.ValueOf` with structs.

```
package main

import (
    “fmt”
    “reflect”
)

type T struct {
    A int32
    B string
    C float32
}

func main() {
    t := T{42, “forty two”, 3.14}

    valueT := reflect.ValueOf(t)
    fmt.Println(valueT.Kind(), valueT)

    for i:=0;i<valueT.NumField();i++{
        field := valueT.Field(i)
        fmt.Println(field.Kind(), field.String(), field.Interface())
    }
}
```

```
struct {42 forty two 3.14}
int32 <int32 Value> 42
string forty two forty two
float32 <float32 Value> 3.14
```

5.2.1 Setting values

Using `reflect.Value` we can set values on run-time. Every `Value` has methods `SetInt32`, `SetFloat32`, `SetString`, etc. that set the field to a `int32`, `float32`, `string`, etc.

value. Example 5.8 sets the string fields from a struct to uppercase.

Example 5.8: Setting values using reflection.

```
package main

import (
    “fmt”
    “reflect”
    “strings”
)

type T struct {
    A string
    B int32
    C string
}

func main() {
    t := T{“hello”,42,“bye”}
    fmt.Println(t)

    valueOfT := reflect.ValueOf(&t).Elem()
    for i:=0; i< valueOfT.NumField(); i++ {
        f := valueOfT.Field(i)
        if f.Kind() == reflect.String {
            current := f.String()
            f.SetString(strings.ToUpper(current))
        }
```

```
{hello 42 bye}
{HELLO 42
BYE}
```

```
    }
    fmt.Println(t)
}
```

If the set operation is not valid, the operation will panic. For example, setting fields to a different type or trying to set unexported fields. In Example 5.9, the field `c` in unexported. Additional checking must be done using the `canSet()` method. Using this method we can skip unexported fields. Observe that the output has not modified `c` value.

Example 5.9: Setting values using reflection considering unexported fields.

```
package main

import (
   “fmt”
   “reflect”
   “strings”
)

type T struct {
    A string
    B int32
    c string
}

func main() {
    t := T{“hello”,42,“bye”}
    fmt.Println(t)

```

```
{hello 42 bye}
{HELLO 42
bye}
```

```
    valueOfT := reflect.ValueOf(&t).Elem()
    for i:=0; i< valueOfT.NumField(); i++ {
        f := valueOfT.Field(i)
        if f.Kind() == reflect.String {
            if f.CanSet() {
                current := f.String()
                f.SetString(strings.ToUpper(current))
            }
        }
    }

    fmt.Println(t)
}
```

5.3 CREATING FUNCTIONS ON THE FLY

In previous sections, we have explored how to inspect fields and modify values. Additionally, the `reflect` package permits the creation of new entities such as functions on the fly. This offers certain functionalities available in other programming languages. For example, the lack of generics in Go imposes some limitations although there is already a proposal at the moment of this writing[8].

Assume we want to write a function that generalizes the add operation. This function must sum numbers (integers and floats) and append strings. Given the current language definition, this is not possible. Check the code below. Go does not permit any kind of function overload. Every function must have a unique name. Similarly, the lack of generics makes it impossible to reuse the same code using the add operator (`+`) defined for every type.

```
func Sum(a int, b int) int {…}


func Sum(a float32, b float32) float32 {…} // Not unique.
```

```go
func Sum(a string, b string) string {…} // Not unique.
```

One interesting workaround is to use the `reflect.MakeFunc` function to build our own functions with different signatures. Example 5.10 builds an add function factory in `BuildAdder()`. This function receives the pointer to a function with two arguments of the same type and one output. The `MakeFunc` receives a function type and a function with a variable number of `Value` types inside an array, and returns an array of `Value`. We fill this function with a `switch` statement that implements the addition between the two arguments according to its type. Finally, we set this function to the original function (`fn.Set(newF)`.

Example 5.10: Using `reflect.Makefunc` to create functions on run-time.

```go
package main

import (
    “fmt”
    “reflect”
)


func BuildAdder (i interface{}) {
     fn := reflect.ValueOf(i).Elem()

     newF := reflect.MakeFunc(fn.Type(), func(in []reflect.Value)[]reflect.Value{

         if len(in) > 2 {
             return []reflect.Value{}
         }

```

```
3
5.423
hello
go
```

```go
        a, b := in[0], in[1]

        if a.Kind() != b.Kind() {
            return []reflect.Value{}
        }

        var result reflect.Value

        switch a.Kind() {
        case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32, reflect.Int64:
            result = reflect.ValueOf(a.Int() + b.Int())
        case reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32, reflect.Uint64:
            result = reflect.ValueOf(a.Uint() + b.Uint())
        case reflect.Float32, reflect.Float64:
            result = reflect.ValueOf(a.Float() + b.Float())
        case reflect.String:
            result = reflect.ValueOf(a.String() + b.String())
        default:
            result = reflect.ValueOf(interface{}(nil))
        }
        return []reflect.Value{result}
    })
    fn.Set(newF)
}

func main() {
    var intAdder func(int64,int64) int64
    var floatAdder func(float64, float64) float64
    var strAdder func(string, string) string

    BuildAdder(&intAdder)
    BuildAdder(&floatAdder)
    BuildAdder(&strAdder)

    fmt.Println(intAdder(1,2))
    fmt.Println(floatAdder(3.0,2.423))
    fmt.Println(strAdder(“hello”,” go”))
}
```

5.4 TAGS

Go provides powerful and versatile structs enrichment using tags. These tags can be interpreted on run-time using reflection which adds valuable information that can be employed for different purposes.

```
type User struct {
    UserId string `tagA:“valueA1” tagB: “valueA2”`
    Email string `tagB:“value”`
    Password string `tagC: “v1 v2”`
    Others string `“something a b”`
}
```

The `struct` above declares four fields with different tags. Every tag becomes an attribute of the field that can be accessed later. Go permits tags to declare raw string literals like `“something a b”`. However, by convention tags follow a key-value schema separated by spaces. For example, the string `tagA:“valueA1”tagB:“valueA2”`, declares two tags `tagA` and `tagB` with values `valueA1` and `valueA2` respectively.

Example 5.11, uses `reflect.TypeOf` to access all the declared fields of the struct `User`. The type `Type` returned by `TypeOf` has functions to check the type name, number of fields, size, etc. Field information is stored in a type `StructField` that can be accessed using `Field()` and `FieldByName()` functions. For every `StructField` tags are stored in a `StructTag` type (`fieldUserId.Tag`). A `StructTag` contains all the available tags of a field.

Example 5.11: Access to field tags using `reflect`.

```
package main

import (
    “fmt”
    “reflect”
)

type User struct {
    UserId string `tagA:“valueA1” tagB: “valueA2”`
    Email string `tagB:“value”`
    Password string `tagC:“v1 v2”`
}

func main() {
    T := reflect.TypeOf(User{})

    fieldUserId := T.Field(0)
    t := fieldUserId.Tag
    fmt.Println(“StructTag is:”,t)
    v, _ := t.Lookup(“tagA”)
    fmt.Printf(“tagA: %s\n”, v)
```

```
StructTag is: tagA:“valueA1”
tagB: “valueA2”
tagA: valueA1
email tagB: value
Password tags: [tagC:“v1
v2”]
v1 v2
```

```

    fieldEmail, _ := T.FieldByName(“Email”)
    vEmail := fieldEmail.Tag.Get(“tagB”)
    fmt.Println(“email tagB:”, vEmail)

    fieldPassword, _ := T.FieldByName(“Password”)
    fmt.Printf(“Password tags: [%s]\n”,fieldPassword.Tag)
    fmt.Println(fieldPassword.Tag.Get(“tagC”))
}
```

By convention tags must be declared like `key:“value”` strings. Notice that blank spaces in the string do not follow the convention. E.g.: `key:␣“value”` is not a valid declaration.

5.5 THE THREE LAWS OF REFLECTION

At this point, we have explored some actions we can carry out using reflection. Reflection is an extremely powerful tool although it can become extremely convoluted. Rob Pike enumerated the three laws of reflection [10] that govern Go. We reproduce and explain these three laws in this Section.

5.5.1 The first law of reflection

Reflection goes from interface value to reflection object.

We have explained that function `reflect.TypeOf` inspects any type. The signature of this function receives an empty interface by argument (`TypeOf(i interface{})`) . However, we have already seen that print any type returned by this function like in the code below.

```
var a int32 = 42

fmt.Println(reflect.TypeOf(a)) // —> Prints int32
```

How is this possible? Variable `a` is stored into an empty interface before `TypeOf` is called. A similar process is done with `ValueOf`. The empty interface stores the underlying type and points to the corresponding value. That is why we can resolve the variable type or value.

5.5.2 The second law of reflection

Reflection goes from reflection object to interface value.

If the first law defines how we go from the interface to the value, the second law defines the inverse. For a `Value` type we can get the original interface using `func (v Value) Interface interface{}`. The code below will print 42 twice.

```
var a int32 = 42

v := reflect.ValueOf(a)

fmt.Println(v) // –> Prints 42

fmt.Println(v.Interface()) // –> Prints 42
```

What is the difference between printing `v` and `v.Interface()` if both outputs are the same? In the first case, we print a `Value`. In the second case, we print an `int32` variable. The reason why the first case prints the same is that `Println` states that in the case of `reflect.Value` types the output is the concrete value that this type holds. Then, why use `v.Interface()`? Simply because `v` is not the real value although some operations like `Println` can reach it.

5.5.3 The third law of reflection

To modify a reflection object, the value must be settable.

The third law may sound evident, you cannot set something if it is not settable. However, this is something difficult to see. The example below tries to set an integer.

```
var a int32 = 42
```

```
v := reflect.ValueOf(a)

v.SetInt(16) // <– panic
```

The last instruction will panic. This is due to the settability of `v`. The value contained by `v` is not the original one, it is a copy. Notice that `reflect.ValueOf(a)` uses a copy of `a`. The value does not point to the original place where the 42 is stored, we need a pointer. However, the following will fail.

```
var a int32 = 42

v := reflect.ValueOf(&a)

v.SetInt(16) // –> panic
```

Now `v` is a pointer to an integer. If we set this value, we are trying to modify the pointer. What we are looking for is the content that is been pointed by this pointer. This is where we use the `Elem()` method.

```
var a int32 = 42

v := reflect.ValueOf(&a).Elem()

v.SetInt(16)

fmt.Println(v.Interface())
```

The settability of a field can be checked using the method `CanSet()` as discussed in Example 5.9 .

5.6 SUMMARY

In this Chapter, we explored how to use reflection in Go and how to use types, values, and tags. We enumerated the three laws of reflection and showed examples for each of the rules. Reflection is a powerful tool in advanced programming projects and will appear in different Chapters of this book. However, the developer must consider that code introspection comes with a cost in terms of performance, code readability, and maintenance. Is for this reason, that reflection is expected to be used in certain scenarios where it is the only solution for a given problem.

CHAPTER 6 CONCURRENCY

The title of this Chapter may sound intimidating. Concurrency is a tough topic in computer science that causes developer headaches. Fortunately, Go was designed with simplicity in mind. It facilitates the creation of concurrent programs, inter-thread communication, and other topics that in other languages require deep tech knowledge. At the end of this Chapter you will understand concepts such as goroutines, channels, and how they can be used to design sophisticated concurrent programs.

6.1 GOROUTINES

A Goroutine is a lightweight thread managed by the Go runtime. Goroutines are declared using the `go` statement followed by the function to be executed. Example 6.1 launches the `ShowIt` function in a goroutine that runs concurrently with the `main` function. Both functions print a message after sleeping (`time.Sleep`). The `main` function sleeps half the time of `ShowIt` that is why we have a ratio of two messages from one function versus the other. It is important to notice that although the loop in `ShowIt` is endless, the program execution terminates when the `main` function finishes. No goroutine will remain running when the `main` function finishes.

Example 6.1: Creation of goroutines.

```
package main

import (
    “fmt”
    “time”
)

func ShowIt() {
    for {
```

```
Where is it?
Where is it?
Here it is!!!
Where is it?
Where is it?
Here it is!!!
Where is it?
```

```
        time.Sleep(time.Millisecond * 100)
        fmt.Println(”Here it is!!!”)
    }
}

func main() {

    go ShowIt()

    for i := 0; i < 5;  i++ {
        time.Sleep(time.Millisecond * 50)
        fmt.Println(”Where is it?”)
    }
}
```

Goroutines are very lightweight with a very small memory demand (only 2Kb) when compared with a thread. We can expect to have several goroutines concurrently running. This can as easy as invoking the `go` statement when required. Example 6.2 creates three goroutines that print a number after sleeping for a given time. The output shows the proportion of messages we expect depending on the sleeping time. Observe that multiple executions of this program may not return the same output. Why? Because the goroutines initialization time may vary and because we are using the console output which may is a single output for multiple routines.

Example 6.2: Creation of multiple goroutines.

```
package main

import (
    “time”
    “fmt”
```

```
1
10
1
```

```
)

func ShowIt(t time.Duration, num int){
    for {
        time.Sleep(t)
        fmt.Println(num)
    }
}

func main() {
    go ShowIt(time.Second, 100)
    go ShowIt(time.Millisecond*500,10)
    go ShowIt(time.Millisecond*250,1)

    time.Sleep(time.Millisecond*1200)
}
```

```
1

100
10
1
```

6.2 CHANNELS

Channels are a mechanism that provides communication for concurrently running functions. A channel can send or receive elements of a specified type.

Channels are instantiated using the `make` built-in function. Example 6.3 `make(chan int)` instantiates a channel that can send or receive integers. In this particular example, the `generator` function runs in a goroutine that computes the sum of the first five integers and sends it through the `channel`. Meanwhile, the main function waits until something is sent through the channel with `result := <- ch`. Notice that this last operation is blocking and will not be completed until something is sent through the channel.

Example 6.3: Goroutine using reading channels.

```go
package main

import (
    “fmt”
    “time”
)

func generator(ch chan int) {
    sum := 0
    for i:=0;i<5;i++ {
        time.Sleep(time.Millisecond * 500)
        sum = sum + i
    }
    ch <- sum
}

func main() {

    ch := make(chan int)

    go generator(ch)

    fmt.Println(”main waits for result…”)
    result := <- ch

    fmt.Println(result)
}
```

```
main waits for
result…
10
```

We can enhance this example using the channel in both directions:

reading and writing. In Example 6.4, `generator` receives the number of elements to iterate through the channel. The function will be blocked until the number is received. You can observe this by manipulating the sleep time in the main function before sending the `n` value through the channel.

Example 6.4: Goroutine using read/write channels.

```
package main

import (
    “fmt”
    “time”
)

func generator(ch chan int) {
    fmt.Println(”generator waits for n”)
    n := <- ch
    fmt.Println(”n is”,n)
    sum := 0
    for i:=0;i<n;i++ {
        sum = sum + i
    }
    ch <- sum
}

func main() {

    ch := make(chan int)

    go generator(ch)
```

```
main waits for
result…
generator waits
n is 5
10
```

```go

    fmt.Println(”main waits for result…”)
    time.Sleep(time.Second)
    ch <- 5
    result := <- ch

    fmt.Println(result)
}
```

Channels can be used to read or write. However, observe that the arrow statement `<-` always goes from the right to the left.

```go
ch <- 5 // send 5 through channel
n := <- ch // initialize n with value from channel
<- ch // wait until something is sent through ch
```

6.2.1 Buffered channels

Channels can be buffered or unbuffered. The statement `make(chan int)` generates an unbuffered channel. Unbuffered channels have no capacity, this means that both sides of the channel must be ready to send and receive data. On the other side, buffered channels can be declared with `make(chan int, 10)` where `10` is the buffer size. In this case, the channel can store values independently of the readiness of sender and receiver.

In Example 6.5, two functions and the `main` send data to the same channel. Due to the code workflow, the main writes to the channel when nobody is listening which triggers the error `fatal error: all goroutines are asleep - deadlock!`. In this case, a buffered channel can store messages until the receivers are ready to consume the messages. This example only needs a one-element buffer. However, you can check how removing the size value in the `make` statement returns an unbuffered channel.

Example 6.5: Channel buffering.

```go
package main

import (
    “fmt”
    “time”
)

func MrA(ch chan string) {
    time.Sleep(time.Millisecond*500)
    ch <- “This is MrA”
}

func MrB(ch chan string) {
    time.Sleep(time.Millisecond*200)
    ch <- “This is MrB”
}

func main() {
    //ch := make(chan string)
    ch := make(chan string,1)

    ch <- “This is main”

    go MrA(ch)
    go MrB(ch)

    fmt.Println(<-ch)
    fmt.Println(<-ch)
```

```
This is main
This is MrB
This is
MrA
```

```
    fmt.Println(<-ch)
}
```

Like slices, buffered channels have `len` and `cap` functions. We could think of using these functions to avoid sending data to full channels.

```
if len(ch) == cap(ch) {
    // channel was full, now we don’t know
} else {
    // channel was free, now we don’t know
}
```

However, this code is not very reliable because the checked condition was true before the current statement. In these situations, a `select` clause is more convenient as explained below.

6.2.2 Close

When a channel is not going to be used anymore, it can be terminated with the built-in function `close`. If a closed channel is used, it causes a runtime panic. Receivers can know if a channel was closed using a multi-value receive operation (`x, ok := <- ch`). The first returned is a value sent through the channel, the second is a boolean indicating whether the reception was correct or not. This second value can be used to identify closed channels.

In Example 6.6, a goroutine sends numbers through a channel. Once it has finished, the channel is closed. The receiver (`main` function) runs an endless loop consuming the elements sent through the channel. When the channel is closed, the `found` variable becomes `false` and we know that the channel was closed.

Example 6.6: `close` function in channels.

```go
package main

import (
    “fmt”
    “time”
)

func sender(out chan int) {
    for i:=0;i<5;i++ {
        time.Sleep(time.Millisecond * 500)
        out <- i
    }
    close(out)
    fmt.Println(”sender finished”)
}

func main() {

    ch := make(chan int)

    go sender(ch)

    for {
        num, found := <- ch
        if found {
            fmt.Println(num)
        } else {
            fmt.Println(”finished”)
```

```
0
1
2
3
4

finished
```

```
            break
        }
    }
}
```

6.2.3 Consuming data with range

Normally, we cannot know beforehand the number of values that are going to be sent through a channel. We can use to block the execution and wait for values until the channel is closed. Check how in Example 6.7 the loop in the main function consumes data from the channel until this is closed independently of the number of generated values.

Example 6.7: Channel consumption using `range`.

```
package main

import “fmt”
func generator(ch chan int) {
    for i:=0;i<5;i++{
        ch <- i
    }
    close(ch)
}

func main() {
    ch := make(chan int)

    go generator(ch)

    for x := range(ch) {
```

```
0
1
2
3
4

Done
```

```
        fmt.Println(x)
    }
    fmt.Println("Done")
}
```

6.2.4 Channels direction

The data flow direction in a channel can be specified.

```
ch := make(chan int) // sender and receiver
ch := make(<- chan int) // receiver
ch := make(chan <- int) // sender
```

This provides better type-safety and permits better utilization of channels. In Example 6.8, we define `sender` and `receiver` functions to send and receive values respectively. Because channel directions are set, if `receiver` tries to send data using the channel an error at compilation time will appear. And similarly, `sender` cannot read data from the channel.

Example 6.8: Channels direction.

```
package main

import (
    "fmt"
    "time"
)

func receiver(input <- chan int) {
    for i := range input {
        fmt.Println(i)
    }
```

```
0
1
4
9

Done
```

```
}

func sender(output chan <- int, n int) {
    for i:=0;i<n;i++ {
        time.Sleep(time.Millisecond * 500)
        output <- i * i
    }
    close(output)
}


func main() {

    ch := make(chan int)

    go sender(ch, 4)
    go receiver(ch)

    time.Sleep(time.Second*5)
    fmt.Println(”Done”)
}
```

6.3 SELECT

The `select` statement is somehow similar to the `switch` statement. From a set of send/receive operations, it blocks the execution until one operation is ready. Example 6.9, executes two different goroutines that send values through two different channels. See how the `select` statement can group the receive operations in a single statement. Inside the `select`, cases are evaluated in the same order they are defined. If two or more cases are ready then, a pseudorandom uniform selection chooses the next one. The `select` statement is executed only for a single operation, to keep on waiting for more messages it has to be iteratively executed like in a loop.

Example 6.9: `select`.

```
package main

import (
    “fmt”
    “strconv”
    “time”
)

func sendNumbers(out chan int) {
    for i:=0; i < 5; i++ {
        time.Sleep(time.Millisecond * 500)
        out <- i
    }
    fmt.Println(”no more numbers”)
}

func sendMsgs(out chan string) {
    for i:=0; i < 5; i++ {
        time.Sleep(time.Millisecond * 300)
        out <- strconv.Itoa(i)
    }
    fmt.Println(”no more msgs”)
}
```

```
msg 0
number 0
msg 1
msg 2
number 1
msg 3
number 2
msg 4
no more
msgs
number 3
number 4
```

```

func main() {
    numbers := make(chan int)
    msgs := make(chan string)

    go sendNumbers(numbers)
    go sendMsgs(msgs)

    for i:=0;i<10;i++ {
        select {
        case num := <- numbers:
            fmt.Printf(”number %d\n”, num)
        case msg := <- msgs:
            fmt.Printf(”msg %s\n”, msg)
        }
    }
}
```

`select` accepts multi-value reception (`x, ok := <- ch`). This feature can be used to know if a channel is ready for reception or not. Example 6.10, extends the previous example to stop data reception when both channels are closed. Remember that in multi-value reception, the second value turns `false` when the channel is closed. Compared with the previous example, now we do not need to know beforehand the number of messages to be received. We can wait until the channels are closed.

Example 6.10: `select` with multi values.

```
package main

import (
```

```
msg 0
```

```
    “fmt”
    “strconv”
    “time”
)

func sendNumbers(out chan int) {
    for i:=0; i < 5; i++ {
        time.Sleep(time.Millisecond * 500)
        out <- i
    }
    fmt.Println(”no more numbers”)
    close(out)
}

func sendMsgs(out chan string) {
    for i:=0; i < 5; i++ {
        time.Sleep(time.Millisecond * 300)
        out <- strconv.Itoa(i)
    }
    fmt.Println(”no more msgs”)
    close(out)
}

func main() {
    numbers := make(chan int)
    msgs := make(chan string)

    go sendNumbers(numbers)
```

```
number 0
msg 1
msg 2
number 1
msg 3
number 2
msg 4
no more msgs
number 3
number 4
no more
numbers
```

```
    go sendMsgs(msgs)

    closedNums, closedMsgs := false, false

    for !closedNums || !closedMsgs {
        select {
        case num, ok := <- numbers:
            if ok {
                fmt.Printf(”number %d\n”, num)
            } else {
                closedNums = true
            }
        case msg, ok := <- msgs:
            if ok {
                fmt.Printf(”msg %s\n”, msg)
            } else {
                closedMsgs = true
            }
        }
    }
}
```

The `select` statement is blocking. The execution will be blocked until at least one of the declared communications is ready. This can be changed by adding a `default` case which is executed when none of the communications is ready. It can be used to avoid errors when channels are not ready. In Example 6.11, we use an unbuffered channel that is never going to be ready. Using the `default` case, we can control this situation and run without panic. As an exercise, you can use a buffered channel to check how the execution changes.

Example 6.11: Non-blocking `select` using `default` cases.

```
package main

import “fmt”

func main() {
    ch := make(chan int)
    //ch := make(chan int,1)

    select {
    case i := <-ch:
        fmt.Println(”Received”, i)
    default:
        fmt.Println(”Nothing received”)
    }

    select {
    case ch <- 42:
        fmt.Println(”Send 42”)
    default:
        fmt.Println(”Nothing sent”)
    }

    select {
    case i := <-ch:
        fmt.Println(”Received”, i)
    default:
```

```
Nothing received
Nothing sent
Nothing
received
```

```
        fmt.Println(”Nothing received”)
    }
}
```

6.4 WAITGROUP

Normally, when working with several goroutines we have to wait until their completion. In order to facilitate this, Go offers the `WaitGroup` type in the `sync` package. This type has three methods `Add`, `Done`, and `Wait`. A `WaitGroup` works like a counter with the number of goroutines we are waiting to be finished. Every time a goroutine finishes, the `Done` method decreases the counter. The `Wait` method blocks the execution until the counter reaches zero.

Example 6.12, defines an example of consumer/producer with a function generating tasks (random numbers) that are consumed by other goroutines. The `WaitGroup` variable `wg` is instantiated and incremented to 3 elements (`wg.Add(3)`. Notifying `wg` about completion is up to the goroutines. For this, we pass the `wg` by reference so they can notify their completion. Notice, that this is done with `defer wg.Done()`) to ensure that the notification is sent. Finally, `wg.Wait()` waits until all the goroutines are finished.

Example 6.12: `WaitGroup` and several goroutines.

```
package main

import (
    “fmt”
    “math/rand”
    “sync”
    “time”
)

func generator(ch chan int, wg *sync.WaitGroup) {
```

```
2 -
task[3440579354231278675]
2 -
task[5571782338101878760]
1 - task[608747136543856411]
2 -
task[1926012586526624009]
Generator done
Consumer 2 done
1 - task[404153945743547657]
```

```
    defer wg.Done()
    for i:=0;i<5;i++ {
        time.Sleep(time.Millisecond*200)
        ch <- rand.Int()
    }
    close(ch)
    fmt.Println(”Generator done”)
}

func consumer(id int, sleep time.Duration,
    ch chan int, wg *sync.WaitGroup) {
    defer wg.Done()
    for task := range(ch) {
        time.Sleep(time.Millisecond*sleep)
        fmt.Printf(”%d - task[%d]\n”,id,task)
    }
    fmt.Printf(”Consumer %d done\n”,id)
}

func main() {
    rand.Seed(42)

    ch := make(chan int,10)
    var wg sync.WaitGroup
    wg.Add(3)

    go generator(ch,&wg)
    go consumer(1,400,ch,&wg)
```

```
Consumer 1 done
```

```go
    go consumer(2,100,ch,&wg)

    wg.Wait()
}
```

Similar behaviour can be obtained using channels. However, this approach is highly recommended in scenarios where several goroutines block the execution flow and their termination can be determined by the goroutines themselves.

6.5 TIMERS, TICKERS, AND TIMEOUTS

In concurrent scenarios time management becomes really important. During the execution several events may occur with different periods or timelines. For single events to occur in the future the `time` package offers the `Timer` type. A `Timer` has a channel `C` that triggers a signal after a given time. Similarly, the `Ticker` type triggers a signal for channel `C` for a given period.

Example 6.13: `Timer` and `Ticker`.

```go
package main

import (
   “fmt”
   “time”
)

func worker(x *int) {
   for {
        time.Sleep(time.Millisecond * 500)
        *x = *x + 1
    }
```

```
ticker -> 2
ticker -> 3
ticker -> 5
ticker -> 7
timer -> 9
ticker -> 9
ticker ->
11
```

```go
}

func main() {
    timer := time.NewTimer(time.Second * 5)
    ticker := time.NewTicker(time.Second)

    x := 0
    go worker(&x)

    for {
        select {
        case <- timer.C:
            fmt.Printf(”timer -> %d\n”, x)
        case <- ticker.C:
            fmt.Printf(”ticker -> %d\n”, x)
        }
        if x>=10 {
            break
        }
    }
}
```

Example 6.13, checks the value of `x` over time using a `Timer` and a `Ticker`. The `worker` increases `x` by one every 500 milliseconds. The `select` statement can be used here to react when `timer` and `ticker` send an event. `ticker` will wake up every second finding `x` to be increased by two. Notice that `worker` sleeps for half second in every iteration. For `timer` there will be a single operation after five seconds. Notice, that `worker` could be implemented using a `Ticker` instead of `time.Sleep`.

Example 6.14: `Timer` and `Ticker` management.

```
package main

import (
    “fmt”
    “time”
)

func reaction(t *time.Ticker) {
    for {
        select {
        case x := <-t.C:
            fmt.Println(”quick”,x)
        }
    }
}

func slowReaction(t *time.Timer) {
    select {
    case x := <-t.C:
        fmt.Println(”slow”, x)
    }
}

func main() {
    quick := time.NewTicker(time.Second)
    slow := time.NewTimer(time.Second * 5)
    stopper := time.NewTimer(time.Second * 4)
    go reaction(quick)
    go slowReaction(slow)

    <- stopper.C
    quick.Stop()

    stopped := slow.Stop()
    fmt.Println(”Stopped before the event?”,stopped)
}
```

```
quick 2021-01-13 19:56:59.2428 +0100 CET m=+1.004374708
quick 2021-01-13 19:57:00.240984 +0100 CET m=+2.002541186
quick 2021-01-13 19:57:01.240728 +0100 CET m=+3.002267097
quick 2021-01-13 19:57:02.241683 +0100 CET m=+4.003202851
Stopped before the event? true
```

Timers and tickers can be managed using `Stop()` and `Reset()` methods. Invoking the `Stop` method closes the channel and terminates the triggering of new events. In the case of timers, the method returns `true` if the invocation was triggered before the event was triggered.

In Example 6.14 we define two events, `quick` using a `Ticker` every second, and `slow` using a `Timer` triggered after 5 seconds. The third `Timer stopper` is set to 4 seconds. When the `stopper` timer is reached we stop `quick` and `slow`. Observe that `quick.Stop` has no returned value. In the case of `slow.Stop`, `true` value is returned as the timer was not triggered yet.

6.6 CONTEXT

Working with APIs or between processes requires additional control logic such as cancellations, timeouts or deadlines. If we send a request to an API, we wait a while for completion. Afterwards, we can assume the request expired. The `Context` type from the `context` package[9], offers constructions to deal with these situations.

The `Context` interface defines the elements of a common context in Go:

```
type Context interface {

    Deadline() (deadline time.Time, ok bool)

    Done() <-chan struct{}

    Err() error

    Value(key interface{}) interface{}

}
```

The `Deadline()` method returns the deadline for the context and `false` if no deadline was set. The `Done()` method returns a channel that is closed when the operations in the context are completed. This channel is closed depending on a `cancel` function, a deadline or a timeout. `Err()` returns and `error` if any. Finally, `Value()` stores key-value pairs of elements that belong to the context.

A `Context` can be built using an empty context (`context.Background()`) or a previously defined context. Go provides four types of contexts with their corresponding initialization functions. These contexts are `WithCancel()`, `WithTimeout()`, `WithDeadline()`, and `WithValue()`.

6.6.1 WithCancel

The function `context.WithCancel()` returns a context, and a `CancelFunc` type. This type, forces the context to be done. In Example 6.15, function `setter` is run simultaneously in several goroutines to increase a shared counter variable.

This function receives a context by argument. In the `main` function the `cancel` function from the context is invoked informing about its termination. All the goroutines receive the message by checking the `Done()` method.

Example 6.15: Context `WithCancel`.

```
package main

import (
    “context”
    “fmt”
    “sync/atomic”
    “time”
)

func setter(id int, c *int32, ctx context.Context) {
    t := time.NewTicker(time.Millisecond*300)
    for {
        select {
        case <- ctx.Done():
            fmt.Println(”Done”, id)
            return
        case <- t.C:
            atomic.AddInt32(c, 1)
        }
    }
}

func main() {
```

```
Final
check:  15
Done 1
Done 3
Done 2
Done 4
Done 0
```

```
    ctx, cancel := context.WithCancel(context.Background())

    var c int32 = 0
    for i:=0;i<5;i++ {
        go setter(i, &c, ctx)
    }

    time.Sleep(time.Second * 1)
    fmt.Println(”Final check: “, c)

    cancel()
    time.Sleep(time.Second)
}
```

Normally, the `cancel()` function is executed using `defer` to ensure that the context is terminated.

6.6.2 WithTimeout

Timeouts are a very common approach to avoid allocating resources for a long time. The `context.WithTimeout` function receives a `time.Duration` argument setting how much time to wait until the context is done. Example 6.16 iteratively executes a goroutine that takes an incrementally longer time to finish. The context is finished before we can run the fifth iteration. The `select` statement blocks the execution until we receive a new number or the context is done. Observe that in the case the context reaches the timeout, the `Err()` method is filled with the corresponding message.

Example 6.16: Context `WithTimeout`.

```
package main

```

```
Received 0
```

```
import (
    “context”
    “fmt”
    “time”
)

func work(i int, info chan int) {
    t := time.Duration(i*100)*time.Millisecond
    time.Sleep(t)
    info <- i
}

func main() {
    d := time.Millisecond * 300

    ch := make(chan int)
    i:=0
    for {
        ctx, cancel := context.WithTimeout(context.Background(), d)
        go work(i, ch)
        select {
        case x := <- ch:
            fmt.Println(”Received”,x)
        case <- ctx.Done():
            fmt.Println(”Done!!”)
        }
        if ctx.Err()!=nil{
            fmt.Println(ctx.Err())
```

```
Received 1
Received 2
Received 3
Done!!
context deadline
exceeded
```

```
            return
        }
        cancel()
        i++
    }
}
```

When working with contexts inside loops, cancel functions should not be deferred. Remember that `defer` executes when the function returns, therefore all the resources allocated are not released until then. If there is a number of contexts declared inside a loop

```
for … {
    ctx, cancel = context.WithCancel(context.Backgroun())
    // defer cancel() –> release when the function returns
    …
    cancel()
}
```

6.6.3 WithDeadline

Similar to a timeout, this kind of context has an absolute termination time set. When the deadline is reached, the context is finished. Example 6.17 , uses various goroutines to modify a shared variable. The context is set to expire after three seconds. Notice that this approach is similar to use a context with a timeout. We block the execution until the context is done (`<-ctx.Done()`). From the output, we can observe how the goroutines are informed about the context termination. Not all the done messages are printed because the program terminates before the goroutines have time to print the message.

Example 6.17: Context `WithDeadline`.

```
package main

import (
    “context”
    “fmt”
    “sync/atomic”
    “time”
)

func accum(c *uint32, ctx context.Context) {
    t := time.NewTicker(time.Millisecond*250)
    for {
        select {
        case <- t.C:
            atomic.AddUint32(c, 1)
        case <- ctx.Done():
            fmt.Println(”Done context”)
            return
        }
    }
}

func main() {
    d := time.Now().Add(time.Second*3)
    ctx, cancel := context.WithDeadline(context.Background(), d)
    defer cancel()

    var counter uint32 = 0
```

```
Done context
counter is: 57
Done context
Done
context
```

```

    for i:=0;i<5;i++ {
        go accum(&counter, ctx)
    }

    <- ctx.Done()
    fmt.Println(”counter is:”, counter)
}
```

6.6.4 WithValue

A context can be defined by the information it contains. The `context.WithValue` function receives key and value arguments not returning a cancel function. In Example 6.18, we use a context with key `“action”` to define the action to be performed by a calculator function.

Example 6.18: Context `WithValue`.

```
package main

import (
   “context”
   “errors”
   “fmt”
)

func main() {

    f := func(ctx context.Context, a int, b int) (int,error) {

        switch ctx.Value(”action”) {
```

```
42
<nil>
2
<nil>
```

```
        case “+”:
            return a + b,nil
        case “-”:
            return a - b,nil
        default:
            return 0, errors.New(”unknown action”)
        }
    }

    ctx := context.WithValue(context.Background(), “action”, “+”)
    r, err := f(ctx,22,20)
    fmt.Println(r,err)
    ctx2 := context.WithValue(context.Background(), “action”, “-”)
    r, err = f(ctx2,22,20)
    fmt.Println(r,err)
}
```

As it can be extracted from the example, these contexts are not attached to temporal restrictions and their termination can be determined by other factors such as an invalid authentication token.

6.6.5 Parent contexts

When creating a new context this can use the empty `context.Background` or it can use another existing context. This can be used to stack different restrictions and generate more complex contexts. Example 6.19 combines a timeout context with a context with values. While the value selects what action to be performed, the timeout sets the time to wait until completion. Check that for the `“slow”` action the message is not printed because it takes longer than the timeout limit.

Example 6.19: Context `WithValue`.

```
package main

import (
    “context”
    “fmt”
    “time”
)

func calc(ctx context.Context) {
    switch ctx.Value(”action”) {
    case “quick”:
        fmt.Println(”quick answer”)
    case “slow”:
        time.Sleep(time.Millisecond*500)
        fmt.Println(”slow answer”)
    case <- ctx.Done():
        fmt.Println(”Done!!!”)
    default:
        panic(”unknown action”)
    }
}

func main() {
    t := time.Millisecond*300
    ctx, cancel := context.WithTimeout(context.Background(), t)
    qCtx := context.WithValue(ctx, “action”, “quick”)
    defer cancel()

```

```
quick
answer

Finished
```

```
    go calc(qCtx)
    <-qCtx.Done()

    ctx2, cancel2 := context.WithTimeout(context.Background(), t)
    sCtx := context.WithValue(ctx2, “action”, “slow”)
    defer cancel2()

    go calc(sCtx)
    <-sCtx.Done()
    fmt.Println(”Finished”)
}
```

6.7 ONCE

In certain scenarios, it only makes sense to execute certain operations once. If several goroutines can execute these operations, the `sync.Once` type ensures that they are only run once. Its method `Do` receives a function by an argument that is executed a single time. This type is very useful to use with initialization functions.

Example 6.20 starts five goroutines that try to initialize a value. However, the initialization time for each one is random. Using `Once` we can register the first goroutine to be started. When the `Do` method is invoked, the other goroutines simply continue the program flow.

Example 6.20: Single actionable variable using `Once`.

```
package main

import (
   “fmt”
   “math/rand”
```

```
2 Done
3 Done
1 Done
4 Done
```

```
   “sync”
   “time”
)

var first int

func setter(i int, ch chan bool, once *sync.Once) {
    t := rand.Uint32() % 300
    time.Sleep(time.Duration(t)*time.Millisecond)
    once.Do(func(){
        first = i
    })
    ch <- true
    fmt.Println(i,”Done”)
}

func main() {
    rand.Seed(time.Now().UnixNano())

    var once sync.Once

    ch := make(chan bool)
    for i:=0;i<10;i++ {
       go setter(i, ch, &once)
    }

    for i:=0;i<10;i++{
        <- ch
```

```
0 Done

The first was
2
```

```
    }
    fmt.Println(”The first was”, first)
}
```

6.8 MUTEXES

Race conditions occur when a variable is accessed by two or several goroutines concurrently. To ensure the correctness of the program, we can set mutual exclusion areas that force goroutines to wait until no goroutine is operating in that area. The type `Mutex` and its methods `Lock()` and `Unlock()` restrict the access to code regions. When a goroutine attempts to operate in a code region it must acquire the mutex by invoking the `Lock` method. After the operations in the mutual exclusion finish, the `Unlock` method returns the control of the region and other goroutines can enter.

Example 6.21 shows a common scenario with three goroutines accessing a common variable `x`. The writer fills the map variable `x` by multiplying values from the previous index by a given factor. Observe that the two writers modify `x` with different frequencies. To make it concurrent safe a mutual exclusion area is defined around `x[i]=x[i-1]*factor`. The output shows how when the reader prints the current `x` value, sometimes the values correspond to modifications done by the first or second goroutine.

Example 6.21: `Mutex`.

```
package main

import (
    “fmt”
    “sync”
    “time”
)

```

```
map[0:1]
map[0:1 1:2]
map[0:1 1:3]
map[0:1 1:3 2:6]
map[0:1 1:3 2:9]
map[0:1 1:3 2:9 3:18]
map[0:1 1:3 2:9 3:27]
```

```go
func writer(x map[int]int, factor int, m *sync.Mutex) {
    i := 1
    for {
        time.Sleep(time.Second)
        m.Lock()
        x[i] = x[i-1] * factor
        m.Unlock()
        i++
    }
}

func reader(x map[int]int, m *sync.Mutex) {
    for {
        time.Sleep(time.Millisecond*500)
        m.Lock()
        fmt.Println(x)
        m.Unlock()
    }
}

func main() {
    x := make(map[int]int)
    x[0]=1

    m := sync.Mutex{}
    go writer(x, 2, &m)
    go reader(x, &m)

```

```
map[0:1 1:3 2:9 3:27
4:54]
```

```
    time.Sleep(time.Millisecond * 300)
    go writer(x, 3, &m)

    time.Sleep(time.Second*4)
}
```

The `Lock` method blocks the thread execution. This means that the longer the exclusion area, the longer is the wait. The size of the mutual exclusion area must be small enough to permit the correctness of the execution.

6.9 ATOMICS

The package `sync/atomic` defines a set of functions for atomic operations. Atomics use low-level atomic memory primitives outperforming mutexes when used correctly. However, as stated in the Go reference, these functions must be used with care and only for low-level applications. For other applications, the `sync` package offers better tools (`WaitGroup`, `Mutex`, etc.).

Atomics offer functions for specific native types. See Table 6.1 for a description of the available functions for type `int32`. Similar functions are available for `uint32`, `int64`, `uint64`, and `uintptr`.

Function	I
`AddInt32(addr *int32, delta int32) (new int32)`	A
`CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)`	I
`LoadInt32(addr *int32) (val int32)`	S
`StoreInt32(addr *int32, val int32)`	S
`SwapInt32(addr *int32, new int32) (old int32)`	S

Table 6.1: Atomic operations available for the `int32` type.

Working with atomics is similar to work with mutexes, we ensure safe reading and writing operations for shared variables. Example 6.22 shows how two goroutines concurrently modify the shared `counter` variable and how this is accessed to be printed. Notice how `atomic.AddInt32` ensures safe concurrent writings without the need for a mutual exclusion area. On the other side, to ensure a safe read we use `atomic.LoadInt32`.

Example 6.22: Atomic access to variable.

```go
package main

import (
    “fmt”
    “sync/atomic”
    “time”
)

func increaser(counter *int32) {
    for {
        atomic.AddInt32(counter,2)
        time.Sleep(time.Millisecond*500)
    }
}

func decreaser(counter *int32) {
    for {
        atomic.AddInt32(counter, -1)
        time.Sleep(time.Millisecond*250)
    }
}

func main() {
```

```
0
1
2
0
2

2
```

```
    var counter int32 = 0

    go increaser(&counter)
    go decreaser(&counter)

    for i:=0;i<5;i++{
        time.Sleep(time.Millisecond*500)
        fmt.Println(atomic.LoadInt32(&counter))
    }
    fmt.Println(atomic.LoadInt32(&counter))
}
```

Atomic operations are designed only for the native types shown in Table 6.1. Fortunately, Go offers the type `Value` that can load and store any type by using the empty interface `interface{}`. Example 6.23 shows a case with a shared variable of type struct. The `Value` type comes with the `Load()` and `Store()` functions that permit to safely read and write our `struct`. In this example, `updater` sets new values to the shared variable and one `observer` checks its content. By invoking `monitor.Load`, the observer loads the latest stored version of the struct. To ensure concurrent writing, a mutual exclusion region must be defined. Observe that the fields of the `Monitor` type are not modified atomically and this may lead to concurrency problems if no mutex is used.

Example 6.23: Atomic access to `Value` type.

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
```

```
&{200 600}
&{300 900}
&{500 1500}
&{700
2100}
```

```
    “time”
)

type Monitor struct {
    ActiveUsers int
    Requests int
}

func updater(monitor atomic.Value, m *sync.Mutex) {
    for {
        time.Sleep(time.Millisecond*500)
        m.Lock()
        current := monitor.Load().(*Monitor)
        current.ActiveUsers += 100
        current.Requests += 300
        monitor.Store(current)
        m.Unlock()
    }
}

func observe(monitor atomic.Value) {
    for {
        time.Sleep(time.Second)
        current := monitor.Load()
        fmt.Printf(”%v\n”, current)
    }
}

```

```
func main() {
    var monitor atomic.Value
    monitor.Store(&Monitor{0,0})
    m := sync.Mutex{}

    go updater(monitor, &m)
    go observe(monitor)

    time.Sleep(time.Second * 5)
}
```

6.10 SUMMARY

This Chapter describes the set of tools Go offers to tackle concurrent problems. The simple but yet powerful design of Go makes it possible to easily design highly concurrent solutions in just a few lines of code. We explain how concurrent goroutines can synchronize each other using channels with various examples. Components from `sync` package such as `Once`, `Mutex`, and `atomic` are detailed and must be understood by newcomers. Finally, this Chapter makes an exhaustive explanation of different context types and how and where they can be used.

CHAPTER 7 INPUT/OUTPUT

Data is not isolated into programs, it has to flow from and to other programs, systems or devices. Input/Output operations are expected to be present in most programs. Go provides basic interfaces in the `io` package that are extended by other packages such as `ioutils` or `os`. This Chapter explains the basics of I/O operations and provides examples of how to use them in Go.

7.1 READERS AND WRITERS

I/O operations can be summarized by readers and writers. Go defines the `Reader` and `Writer` interfaces in the `io` package[10]. A `Reader` must implement a read operation that receives a target byte array where the read data is stored, it returns the number of bytes read and error if any. A `Writer` takes a byte array to be written and returns the number of written bytes and error if any.

```
type Reader interface {

  Read(p []byte) (n int, err error)

}
```

```
type Writer interface {

  Write(p []byte) (n int, err error)

}
```

Example 7.1 shows a `Reader` implementation for strings. This reader fills the array `p` with as many characters as `len(p)` starting from the last read position stored in `from`. The first part in the `Read` method adds some error control. If all the characters from the string were processed, it returns `EOF` to indicate that no more characters are available. Notice that the `target` array is reused so in those iterations where the number of read characters is smaller than the length of the array it will contain characters from previous calls.

Example 7.1: Implementation of a `Reader` interface.

```
package main

import (
    “errors”
    “fmt”
    “io”
)

type MyReader struct {
     data string
     from int
}

func(r *MyReader) Read(p []byte) (int, error) {
     if p == nil {
         return -1, errors.New(”nil target array”)
     }
     if len(r.data) <= 0 || r.from == len(r.data){
         return 0, io.EOF
     }
     n := len(r.data) - r.from
     if len(p) < n {
         n = len(p)
     }
     for i:=0;i < n; i++ {
         b := byte(r.data[r.from])
         p[i] = b
         r.from++
```

```
Read 0: Error: EOF
Read 5: Error: <nil> -> Save
Read 5: Error: <nil> -> the w
Read 5: Error: <nil> -> orld
Read 5: Error: <nil> -> with
Read 5: Error: EOF -> Go!!!
```

```
    }
    if r.from == len(r.data) {
        return n, io.EOF
    }
    return n, nil
}

func main() {
    target := make([]byte,5)
    empty := MyReader{}
    n, err := empty.Read(target)
    fmt.Printf("Read %d: Error: %v\n",n,err)
    mr := MyReader{"Save the world with Go!!!",0}
    n, err = mr.Read(target)
    for err == nil {
        fmt.Printf("Read %d: Error: %v -> %s\n",n,err, target)
        n, err = mr.Read(target)
    }
    fmt.Printf("Read %d: Error: %v -> %s\n",n,err, target)
}
```

Implementing a writer is similar to implementing a reader. Example 7.2 implements a writer designed with a limiting number of bytes to be written in each call. Go specifies that when the number of written bytes is smaller than the size of `p` an error must be filled. Additionally, we consider that an empty `p` corresponds to an `EOF`. Our writer will add the content of `p` to the current data string in batches. Observe that all iterations except the last one return an error due to the size limit.

Example 7.2: Implementation of a `Writer` interface.

```go
package main

import (
    “errors”
    “io”
    “fmt”
)

type MyWriter struct {
    data string
    size int
}

func (mw *MyWriter) Write(p []byte) (int, error) {
    if len(p) == 0 {
        return 0, io.EOF
    }
    n := mw.size
    var err error = nil
    if len(p) < mw.size {
        n = len(p)
    } else {
        err = errors.New(”p larger than size”)
    }
    mw.data = mw.data + string(p[0:n])

    return n, err
}
```

```

func main() {
    msg := []byte("the world with Go!!!")

    mw := MyWriter{"Save ",6}
    i := 0
    var err error
    for err == nil && i < len(msg) {
        n, err := mw.Write(msg[i:])
        fmt.Printf("Written %d error %v -> %s\n", n, err, mw.data)
        i = i + n
    }
}
```

```
Written 6 error p larger than size -> Save the wo
Written 6 error p larger than size -> Save the world wi
Written 6 error p larger than size -> Save the world with Go!
Written 2 error <nil> -> Save the world with Go!!!
```

These examples are shown just to demonstrate how to work with these interfaces. Actually, the `io` package has more interfaces such as `ByteReader`, `ByteWriter`, `PipeReader`, `ReadSeeker`, etc. Before defining your interfaces, check the Go reference for existing ones or other types implementing these interfaces.

7.2 READING AND WRITING FILES

We mentioned that the `io` package groups basic I/O operations and their interfaces. The package `io/util`[11] has implementations of these interfaces ready to be used. Example 7.3 writes a message into a file and then reads it. The function `ioutil.WriteFile` requires the file path, the content of the file to be written, and the file permissions. Notice that the file content is intended to be a byte array. If using strings, the casting is straight forward. On the other side, the `ioutil.ReadFile` function returns a byte of arrays with the content of the file. It is important to highlight that both functions may return errors and these have to be controlled.

Example 7.3: File writing and reading with `ioutil`.

```
package main

import (
    “fmt”
    “io/ioutil”
)

func main() {
    msg := “Save the world with Go!!!”
    filePath := “/tmp/msg”

    err := ioutil.WriteFile(filePath,
        []byte(msg), 0644)
    if err != nil {
        panic(err)
    }

    read, err := ioutil.ReadFile(filePath)
    if err != nil{
        panic(err)
    }

    fmt.Printf(”%s\n”, read)
```

```
Save the world with
Go!!!
```

```
}
```

The `ioutil` simplifies the steps to be carried out when working with files. It heavily uses the `os` package that provides an operating system independent interface. This package is closer to the operating system including additional entities such as file descriptors. This makes possible file manipulation at a lower level.

Example 7.4 writes the items of an array to a file using a loop instead of a single statement like in the previous example. First, we create a file at the desired path with `os.Create`. This function returns an open file descriptor with a large number of available methods[12]. Next, we can use any of the writing available methods to write our content. Finally, we to close the file to release the descriptor (`defer f.Close()`).

Example 7.4: File writing with `os`.

```
package main

import “os”

func main() {
    filePath := “/tmp/msg”
    msg := []string{
        “Rule”, “the”, “world”, “with”, “Go!!!”}

    f, err := os.Create(filePath)
    if err != nil {
        panic(err)
    }
    defer f.Close()

    for _, s := range msg {
        f.WriteString(s+”\n”)
    }
}
```

Using low-level functions we have better control of read and write operations. For example, we can read and write portions of files using the `Seek` method from the `File` type. This method indicates the offset to be used when writing or reading a file. In Example 7.5 `Seek` is used to modify given certain positions in the content of a file and then reads the modified content. After using `Seek`, we have to set the pointer to the first position of the file with `file.Seek(0,0)`, otherwise the read content would start at the last modified position.

Example 7.5: Utilization of file descriptors with `os`.

```
package main

import (
    “fmt”
    “os”
)

func main() {
    tmp := os.TempDir()
    file, err := os.Create(tmp+”/myfile”)
    if err != nil {
        panic(err)
    }
    defer file.Close()

    msg := “Save the world with Go!!!”

    _, err = file.WriteString(msg)
    if err != nil {
        panic(err)
    }

    positions := []int{4, 10, 20}
    for _, i := range positions {
        _, err := file.Seek(int64(i),0)
        if err != nil {
            panic(err)
        }
        file.Write([]byte(”X”))
    }
    // Reset
    file.Seek(0,0)
    // Read the result
    result := make([]byte,len(msg))
    _, err = file.Read(result)
    if err != nil {
        panic(err)
    }
    fmt.Printf(”%s\n”,result)
}
```

7.3 STANDARD I/O

The standard I/O follows the same principles of writers and readers. The main difference is the utilization of `os.Stdin`, `os.Stdout`, and `os.Stderr` variables. These variables are open file descriptors to standard input, output, and error. Because they are variables of type `File` they offer some of the methods explained in the previous section.

Example 7.6 writes a message to the standard output using the `os.Stdout` variable. The result is similar to use `fmt.Print`. However, because we are using a file descriptor we can get the number of written bytes. Notice that the end of line `"\n"` is also a character.

Example 7.6: Writing to standard output with `os.Stdout`

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    msg := []byte("Save the world with Go!!!\n")
     n, err := os.Stdout.Write(msg)
     if err != nil {
         panic(err)
     }
     fmt.Printf("Written %d characters\n",n)
}
```

```
Save the world with
Go!!!

Written 26 characters
```

Reading from standard input may look complicated because of the interaction with the keyboard. However, from the point of view of the operating system, this is like writing to a file. Example 7.7 reads a message from standard input and prints the same message in upper case. By pressing enter, we insert the `EOF` character that finishes the stream. This example uses a fixed size target array. At the time of printing the result, we have to select a slice as long as the number of read characters. The reader can check how by removing this limitation unreadable characters will be printed because the array has not been initialized.

Example 7.7: Reading from standard input with `os.Stdin`

```
package main

import (
    “fmt”
    “os”
    “strings”
)

func main() {
     target := make([]byte,50)
     n, err := os.Stdin.Read(target)
     if err != nil {
         panic(err)
     }
     msg := string(target[:n])
     fmt.Println(n,strings.ToUpper(msg))
}
```

```
>>> save the world with go!!!
26 SAVE THE WORLD WITH GO!!!
```

In the previous example, the size of the target array limits the amount of data that can be received from the standard input. When the input is larger than the available size the remainder will be lost. This can be solved using buffers. The `bufio`[13] package implements buffered readers and writers that are very useful when the amount of incoming or outgoing data is unknown or exceeds a reasonable size for in-memory solutions.

The Example 7.8 reads a string from the standard input and returns it transformed to uppercase like in the previous example. Buffered readers come with various helping functions and methods. The used `ReadString` returns the string from the beginning until the argument delimiter is found.

Example 7.8: Standard input reading using `bufio`.

```go
package main

import (
    “bufio”
    “fmt”
    “os”
    “strings”
)

func main() {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(”>>> What do you have to say?\n”)
    fmt.Print(”<<< “)
    text, err := reader.ReadString(’\n’)
    if err != nil {
        panic(err)
    }
    fmt.Println(”>>> You’re right!!!”)
    fmt.Println(strings.ToUpper(text))
}
```

```
>>> What do you have to say?
<<< go rules
>>> You’re right!!!
GO RULES
```

The `Scanner` type can be particularly useful if we have a stream that has to be split into certain pieces. Example 7.9 uses a scanner to read lines from the standard input until the total accumulated length of the input strings exceeds 15 characters. The delimiter can be customized defining a split function[14].

Example 7.9: Standard input reading using `bufio` scanners.

```
package main

import (
    “bufio”
    “fmt”
    “os”
)

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    fmt.Println(”>>> What do you have to say?\n”)
    counter := 0
    for scanner.Scan() {
        text := scanner.Text()
        counter = counter + len(text)
        if counter > 15 {
            break
        }
    }
    fmt.Println(”that’s enough”)
}
```

```
>>> What do you have to
say?

Rule the world
with
that’s enough
```

Example 7.10 demonstrate how to use a `bufio.NewWriter` with the standard output. The program emulates a typing machine by printing the characters of a string with a temporal sleep between each character. Notice that the `Flush()` method has to be invoked to force the buffer to be printed.

Example 7.10: Standard output writing.

```
package main

import (
    “bufio”
    “os”
    “time”
)

func main() {
    writer := bufio.NewWriter(os.Stdout)

    msg := “Rule the world with Golang!!!”
    for _, letter := range msg {
        time.Sleep(time.Millisecond*300)
        writer.WriteByte(byte(letter))
        writer.Flush()
    }
}
```

Save the world with Go!!!

7.4 SUMMARY

This Chapter explores the simple and efficient approach Go follows to tackle input/output operations. We explain and demonstrate how writers and readers are the cornerstones of I/O and how `ioutils` and `os` packages offer solutions at a high or low level.

CHAPTER 8

ENCODINGS

We can find popular formats such as CSV, JSON, XML or YAML that are used to represent data from a byte level to a human-readable format. Other formats such as base64 or PEM data serialization are oriented to facilitate machine to machine interaction. In Go, the package `encoding` offers a set of subpackages that facilitate the conversion from Go types to these formats and the other way around. Many concepts explained in this Chapter were already introduced in Chapter 7 when we explained readers and writers. In this Chapter, we explore CSV, JSON, and XML manipulation techniques. Additionally, we present how to work with YAML using the GO-yaml third-party package.

8.1 CSV

Comma Separated Values [7] (CSV) is a widely used format to represent tabular data. Go provides CSV read/write operators in the package `encoding/csv`. Every line from a CSV is defined as a CSV record and every record contains the items separated by commas.

In order to read CSV data we use a `Reader` type that converts CSV lines into CSV records. Example 8.1 shows how a `csv.Reader` processes a string with CSV content. Notice that CSV are partially typed and in the case of strings they have to quoted like in variable `in`. The `Read` method returns the next CSV record or error if any. Like in any other `Reader` an `EOF` error is reached when no more records are available. Every record is represented using a `[]string`.

Example 8.1: CSV reading.

```
package main

import (
    “encoding/csv”
    “fmt”
    “io”
```

```
[user_id score password]
[Gopher 1000 admin]
[BigJ 10 1234]
[GGBoom  1111]
```

```
    “log”
    “strings”
)


func main() {
    in := ‘user_id,score,password
“Gopher”,1000,“admin”
“BigJ”,10,“1234”
“GGBoom”,,“1111”
‘
    r := csv.NewReader(strings.NewReader(in))

    for {
        record, err := r.Read()
        if err == io.EOF {
            break
        }
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println(record)
    }
}
```

Example 8.2 writes CSV records to standard output. Every CSV record is a string array that has to be passed to the `Write` method. Finally, the `Flush` method ensures that we send all the buffered data to the standard output.

The writing process follows a similar approach as shown in Example 8.2.

Instead of converting from strings to CSV records, the writer works oppositely.

Example 8.2: CSV writing.

```go
package main

import (
    “encoding/csv”
    “os”
)

func main() {
    out := [][]string{
        {“user_id”,“score”,“password”},
        {“Gopher”,“1000”,“admin”},
        {“BigJ”,“10”,“1234”},
        {“GGBoom”,””,“1111”},
    }
    writer := csv.NewWriter(os.Stdout)
    for _, rec := range out {
        err := writer.Write(rec)
        if err != nil {
            panic(err)
        }
    }
    writer.Flush()

}
```

```
user_id,score,password
Gopher,1000,admin
BigJ,10,1234
GGBoom,,1111
```

8.2 JSON

The JavaScript Object Notation [4] (JSON) is a light-weight data interchange format defined by ECMA in 1999. It has turned extremely popular because it is human readable and easy to parse. JSON processing operators are available at `encoding/json`. This package provides `Marhsal` and `Unmarshal` functions that convert types to a JSON representation in `[]byte`, and vice versa.

The `Marshal` function converts booleans, integers, floats, strings, arrays, and slices into its corresponding JSON representation. Example 8.3 shows the output generated after marshalling various types. Notice that the `Marshal` function returns a `[]byte` with the JSON representation of the input and `error` in case of failure.

Example 8.3: JSON marshalling.

```
package main

import (
“encoding/json”
“fmt”
)

func main() {
    number, err := json.Marshal(42)
    if err!=nil{
        panic(err)
    }
    fmt.Println(string(number))

    float, _ := json.Marshal(3.14)
    fmt.Println(string(float))
```

```
42
3.14
“This is a msg!!!”
[1,1,2,3,5,8]

{“one”:1,“two”:2}
```

```

    msg, _ := json.Marshal(“This is a msg!!!”)
    fmt.Println(string(msg))

    numbers, _ := json.Marshal([]int{1,1,2,3,5,8})
    fmt.Println(string(numbers))

    aMap, _ := json.Marshal(map[string]int{“one”:1,“two”:2})
    fmt.Println(string(aMap))
}
```

If a JSON is correctly formed, the `Unmarshal` function can convert it to a previously known type. Example 8.4 recovers JSON representations of `int` and `map[string]int`. The unmarshalling process requires a pointer to a type compatible with the data to be unmarshalled and returns an error if the process fails. Observe that the output from this example is the string representation of Go types, not JSON representations.

Example 8.4: JSON unmarshalling.

```
package main

import (
“encoding/json”
“fmt”
)

func main() {

    aNumber, _ := json.Marshal(42)

```

```
42
map[one:1
two:2]
```

```
    var recoveredNumber int = -1
    err := json.Unmarshal(aNumber, &recoveredNumber)
    if err!= nil {
        panic(err)
    }
    fmt.Println(recoveredNumber)


    aMap, _ := json.Marshal(map[string]int{“one”:1,“two”:2})

    recoveredMap := make(map[string]int)
    err = json.Unmarshal(aMap, &recoveredMap)
    if err != nil {
        panic(err)
    }
    fmt.Println(recoveredMap)
}
```

Structs can also be used with `Marshal` and `Unmarshal` functions. Actually, Go provides special tags to help in this task. Fields can be tagged to indicate how they have to be transformed to JSON.

In Example 8.5, we define a database of users where users are represented with a `User` value. Every field is tagged with an expression that consists of

```
FieldName type `json:“JSONfieldName,omitempty”`
```

This tag defines how to name the field in the JSON object, and whether to omit it if the zero value is found. In our example, `userC` has no `Score`. When printing the marshalled representation (`dbJson`), notice that one score is missing. Furthermore, the `password` field is always missing. This happens because the field visibility does not allow exporting values. In this case, we could consider this a good practice to hide user passwords.

Example 8.5: Custom `struct` JSON marshalling.

```
package main

import (
    “bytes”
    “encoding/json”
    “fmt”
)

type User struct {
    UserId string `json:“userId,omitempty”`
    Score int `json:“score,omitempty”`
    password string `json:“password,omitempty”`
}

func main() {

    userA := User{“Gopher”, 1000, “admin”}
    userB := User{“BigJ”, 10, “1234”}
    userC := User{UserId: “GGBoom”, password: “1111”}

    db := []User{userA, userB, userC}
    dbJson, err := json.Marshal(&db)
    if err != nil {
        panic(err)
    }
```

```go

    var recovered []User
    err = json.Unmarshal(dbJson, &recovered)
    if err != nil{
        panic(err)
    }
    fmt.Println(recovered)

    var indented bytes.Buffer
    err = json.Indent(&indented, dbJson,"","    ")
    if err != nil {
        panic(err)
    }
    fmt.Println(indented.String())
}
```

```
[{"userId":"Gopher","score":1000},{"userId":"BigJ","score":10},{"userId":"GGBoom"}]
[
   {
      "userId": "Gopher",
      "score": 1000
   },
   {
      "userId": "BigJ",
      "score": 10
   },
   {
      "userId": "GGBoom"
   }
]

[{Gopher 1000 } {BigJ 10 } {GGBoom 0 }]
```

To improve the readability of our JSON output, this can be unmarshalled with indentation. The `Indent` function adds indentation and sends the output to a buffered type (line 35). Finally, the example shows the unmarshalled struct. In this case, the `score` is displayed even when it is the zero value. Observe, that the field omission only works during marshalling. In the unmarshalling process, omitted fields are simply set to the zero value.

8.3 XML

The Extensible Markup Language[11] (XML) is a markup language developed by the W3C back in 1998. Like JSON, it is human-readable and easy to parse. However, it is very verbose which limits its applicability in certain scenarios. The package `encoding/xml` provides functions to work with this format. The package works similarly to the `encoding/json` package explained in the previous section. They share the same interfaces from the `encoding` package. However, there are certain limitations for XML we will explain in the examples below.

Example 8.6 transforms various variables into their XML representation. The `Marhsal` function returns a `[]byte` with the representation and an error if any. The process is similar to the one described for the JSON format. A more detailed explanation about how the different types are converted into XML can be found in the package documentation [15].

Example 8.6: XML marshalling.

```
package main

import (
    “encoding/xml”
    “fmt”
)

func main() {
    number, err := xml.Marshal(42)
     if err!=nil{
         panic(err)
     }
     fmt.Println(string(number))

     float, _ := xml.Marshal(3.14)
     fmt.Println(string(float))

     msg, _ := xml.Marshal(“This is a msg!!!”)
     fmt.Println(string(msg))

     numbers, _ := xml.Marshal([]int{1,2,2,3,5,8})
     fmt.Println(string(numbers))

     aMap, err := xml.Marshal(map[string]int{“one”:1,“two”:2})
     fmt.Println(err)
     fmt.Println(“-“,string(aMap),”-“)
 }
```

```
<int>42</int>
<float64>3.14</float64>
<string>This is a msg!!!</string>
<int>1</int><int>2</int><int>2</int><int>3</int><int>5</int><int>8</int>
xml: unsupported type: map[string]int
-  -
```

Notice that in the case of marshalling `map[string]int` we get an error. Unlike in Example 8.3 where we could marshal the same type into a properly formed JSON, we cannot directly do the same in XML. This is because there is not a single way of representing a map into XML. For this reason, the package implementation excludes the map `map` type from marshalling.

For this case, we have to define our marshaller. To do this, we can create a type that implements methods `MarshalXML` and `UnmarshalXML`. These methods will be invoked during marshal and unmarshal operations so we can control the XML representation of any type[16]. Example 8.7 defines the conversion for a `map[string]string` type using keys as element tags and values as data elements. The code does not control all the scenarios, but it may serve as a starting point for the reader to understand custom XML marshallers.

Example 8.7: XML unmarshalling.

```
package main

import (
    “encoding/xml”
    “errors”
    “fmt”
)

type MyMap map[string]string

func (s MyMap) MarshalXML(e *xml.Encoder, start xml.StartElement) error {
    tokens := []xml.Token{start}

    for key, value := range s {
        t := xml.StartElement{Name: xml.Name{“”, key}}
        tokens = append(tokens, t, xml.CharData(value), xml.EndElement{t.Name})
    }

    tokens = append(tokens, xml.EndElement{start.Name})

    for _, t := range tokens {
        err := e.EncodeToken(t)
        if err != nil {
            return err
        }
    }

    return e.Flush()
}

func (a MyMap) UnmarshalXML(d *xml.Decoder, start xml.StartElement) error {

    key := ””
    val := ””

    for {

        t, _ := d.Token()
        switch tt := t.(type) {

        case xml.StartElement:
            key = tt.Name.Local
        case xml.CharData:
            val = string(tt)
        case xml.EndElement:
            if len(key) != 0{
                a[key] = val
                key,val = ””, ””
            }
            if tt.Name == start.Name {
                return nil
            }

        default:
            return errors.New(fmt.Sprintf(“uknown %T”,t))
        }
    }

   return nil
}


func main() {

    var theMap MyMap = map[string]string{“one”: “1”,“two”:“2”,“three”:“3”}
    aMap, _ := xml.MarshalIndent(&theMap, ””, ”    ”)
    fmt.Println(string(aMap))

    var recoveredMap MyMap = make(map[string]string)

    err := xml.Unmarshal(aMap, &recoveredMap)
    if err != nil {
        panic(err)
    }

    fmt.Println(recoveredMap)
}
```

```
<MyMap>
   <one>1</one>
   <two>2</two>
   <three>3</three>
</MyMap>
map[one:1 three:3 two:2]
```

Excluding situations like the one described in the previous example, type fields can be tagged to facilitate its XML conversion. Example 8.8 is very similar to Example 8.5. In both cases, we use the same `User` type with the same omit options. However, we have to create an additional type `UsersArray` which contains the array of users. As an exercise, check what happens when a `[]User` type is directly passed to the marshal function.

Example 8.8: XML struct marshalling.

```
package main

import (
    “encoding/xml”
    “fmt”
)

type User struct {
    UserId string `xml:“userId,omitempty”`
    Score int `xml:“score,omitempty”`
    password string `xml:“password,omitempty”`
}

type UsersArray struct {
    Users []User `xml:“users,omitempty”`
}

func main() {

    userA := User{“Gopher”, 1000, “admin”}
    userB := User{“BigJ”, 10, “1234”}
    userC := User{UserId: “GGBoom”, password: “1111”}

    db := UsersArray{[]User{userA, userB, userC}}
    dbXML, err := xml.Marshal(&db)
    if err != nil {
        panic(err)
    }

    var recovered UsersArray
    err = xml.Unmarshal(dbXML, &recovered)
    if err != nil{
        panic(err)
    }
    fmt.Println(recovered)

    var indented []byte
    indented, err = xml.MarshalIndent(recovered,"","    ")
    if err != nil {
        panic(err)
    }
    fmt.Println(string(indented))
}
```

```
{[{Gopher 1000 } {BigJ 10 } {GGBoom 0 }]}
<UsersArray>
   <users>
      <userId>Gopher</userId>
      <score>1000</score>
   </users>
   <users>
      <userId>BigJ</userId>
      <score>10</score>
   </users>
   <users>
      <userId>GGBoom</userId>
   </users>
</UsersArray>
```

8.4 YAML

YAML [14] (Yet Another Markup Language) is a data serialization language that is human-readable and easy to parse. It is a superset of JSON [13] although it uses indentation to indicate nested entities. This format has gained popularity as a default format for configuration files. Go does not offer any YAML support in the standard library. However, we can find third-party modules with all the necessary tools to use this format.

This section covers the utilization of go-yaml [17] to marshal and unmarshal YAML content. Before running this code in your environment remember to download the go-yaml code to your machine. Execute `go get gopkg.in/yaml.v2` to get the code. Refer to Section 2.1.1 for more details about importing third-party code or Chapter 12 to use go modules.

Go-yaml follows the same `marshal`/`unmarshal` approach we have already seen for JSON and XML formats. Example 8.9 prints some YAML encodings after using the `Marshal` function.

Example 8.9: YAML marshalling.

```go
package main

import (
    “gopkg.in/yaml.v2”
    “fmt”
)

func main() {
    number, err := yaml.Marshal(42)
     if err!=nil{
         panic(err)
     }
     fmt.Println(string(number))

     float, _ := yaml.Marshal(3.14)
     fmt.Println(string(float))

     msg, _ := yaml.Marshal(“This is a msg!!!”)
     fmt.Println(string(msg))
```

```
42

3.14

This is a
msg!!!

- 1
- 1
- 2
- 3
- 5
- 8

one: 1
two: 2
```

```

    numbers, _ := yaml.Marshal([]int{1,1,2,3,5,8})
    fmt.Println(string(numbers))

    aMap, _ := yaml.Marshal(map[string]int{“one”:1,“two”:2})
    fmt.Println(string(aMap))
}
```

```
42
map[one:1
two:2]
```

YAML can be decoded using the `Unmarshal` function as shown in Example 8.10.

Example 8.10: YAML unmarshalling.

```
package main

import (
    “fmt”
    “gopkg.in/yaml.v2”
)

func main() {

    aNumber, _ := yaml.Marshal(42)

    var recoveredNumber int = -1
    err := yaml.Unmarshal(aNumber, &recoveredNumber)
    if err!= nil {
        panic(err)
    }
    fmt.Println(recoveredNumber)
```

```go


    aMap, _ := yaml.Marshal(map[string]int{“one”:1,“two”:2})

    recoveredMap := make(map[string]int)
    err = yaml.Unmarshal(aMap, &recoveredMap)
    if err != nil {
        panic(err)
    }
    fmt.Println(recoveredMap)
}
```

Go-yaml accepts field tags following the same rules we have seen so far. The output from Example 8.11 shows the representation of the users’ array after been encoded and the value of the filled structure.

Example 8.11: YAML struct marshalling.

```go
package main

import (
    “fmt”
    “gopkg.in/yaml.v2”
)

type User struct {
    UserId string `yaml:“userId,omitempty”`
    Score int `yaml:“score,omitempty”`
    password string `yaml:“password,omitempty”`
}
```

```
- userId: Gopher
  score: 1000
- userId: BigJ
  score: 10
- userId: GGBoom

[{Gopher 1000 } {BigJ 10 } {GGBoom 0 }]
```

```

func main() {

    userA := User{“Gopher”, 1000, “admin”}
    userB := User{“BigJ”, 10, “1234”}
    userC := User{UserId: “GGBoom”, password: “1111”}

    db := []User{userA, userB, userC}
    dbYaml, err := yaml.Marshal(&db)
    if err != nil {
        panic(err)
    }
    fmt.Println(string(dbYaml))

    var recovered []User
    err = yaml.Unmarshal(dbYaml, &recovered)
    if err != nil{
        panic(err)
    }
    fmt.Println(recovered)
}
```

8.5 TAGS AND ENCODING

In Section 5.4, we explained how Go provides field tags to enrich structs and how reflection can be used to analyse these tags. The encodings presented in this Chapter use tags to define how to marshal/unmarshal structs. Similarly, we can define our own tags to define new encoding solutions.

To show how we can leverage tags to define our encoding, let’s assume that we have developed a solution that only works with strings. For some reason, this solution makes a strong distinction between lowercase and uppercase strings. Due to the nature of the problem, we have to constantly use the functions `strings.ToUpper` and `strings.ToLower` in our code. To eliminate redundant code, we decide to use an encoding solution to automatically convert our strings to uppercase or lowercase.

```
type User struct {

    UserId string `pretty:“upper”`

    Email string `pretty:“lower”`

    password string `pretty:“lower”v

}


type Record struct {

    Name string `pretty:“lower”`

    Surname string `pretty:“upper”`

    Age int `pretty:“other”`

}
```

For the structs above, we defined the tag key `pretty`, with values `upper` or `lower` that transform a string into uppercase or lowercase respectively. Our encoding is defined as follows:

1. Unexported fields such as `password` are ignored.
2. Only `lower` and `upper` are valid tag values.
3. Only strings are candidates to be encoded.
4. Every field is identified by its field name, and separated from the new value by a colon.
5. The fields are surrounded by brackets.
6. No recursion, or collections will be encoded.

With this definition we have that `User:{“John”, “John@Gmail.com”, “admin”}` is encoded as `{UserId:JOHN, Email:john@gmail.com}` and `Record{“John”, “Johnson”,33}` as `{Name:john, Surname:JOHNSON}`. As you can see, the output is a subset of JSON that only accepts strings.

Following the common interface for encodings we have already seen in this Chapter, we can define a custom function for our encoding. Example 8.12 , defines how to process the field tags from any `interface` to return our encoding. What we do is to iterate through the fields of the `interface` and encode them when possible using the auxiliary function `encodeField`. The encoded fields are written to a buffer that contains the final output. Notice from the definition above, that not all the situations are permitted and this may generate errors.

Example 8.12: `Marshal` function for custom encoding using field tags (excerpt).

```
package main

import (
    “bytes”
    “encoding/json”
    “errors”
    “fmt”
    “reflect”
    “strings”
)

func Marshal(input interface{}) ([]byte, error) {
    var buffer bytes.Buffer
    t := reflect.TypeOf(input)
    v := reflect.ValueOf(input)

    buffer.WriteString(“{“)
    for i:=0;i < t.NumField();i++ {
        encodedField,err := encodeField(t.Field(i),v.Field(i))

        if err != nil {
            return nil, err
        }
        if len(encodedField) != 0 {
            if i >0 && i<= t.NumField()-1 {
                buffer.WriteString(“, ”)
            }
            buffer.WriteString(encodedField)
        }
    }
    buffer.WriteString(“}”)
    return buffer.Bytes(),nil
}

func encodeField(f reflect.StructField, v reflect.Value) (string, error) {

    if f.PkgPath!=””{
        return ””,nil
    }

    if f.Type.Kind() != reflect.String {
        return ””, nil
    }

    tag, found := f.Tag.Lookup(“pretty”)
    if !found {
        return ””, nil
    }

    result := f.Name+”:”
    var err error = nil
    switch tag {
    case “upper”:
        result = result + strings.ToUpper(v.String())
    case “lower”:
        result = result + strings.ToLower(v.String())
    default:
        err = errors.New(“invalid tag value”)
    }
    if err != nil {
        return ””, err
    }

    return result, nil
}
```

The `encodeField` function uses the `reflect` package to inspect the tags and field values. If we find the `pretty` tag and one of its values, we return the encoded value of the field. To do this there are some previous checks. The function `f.PkgPath` is empty for exported fields, therefore if non empty this is an unexported field and must not be encoded. With `f.Type.Kind()` we check the field type. `f.Tag.Lookup(“pretty”)` checks if this field has a `pretty` tag. Finally, a `switch` statement transforms the field value according to its tag value.

Example 8.13 continues the previous code and shows how our `Marshal` function can be used with different structs. Notice how fields such as `password` or `Age` are ignored in the encoding. We can combine our tags with other encoding formats such as JSON and check the different outputs.

Example 8.13: `Marshal` function for custom encoding using field tags (continues 8.12).

```

type User struct {
    UserId string `pretty:“upper”`
    Email string `pretty:“lower”`
    password string `pretty:“lower”`
}

type Record struct {
    Name string `pretty:“lower” json:“name”`
     Surname string `pretty:“upper” json:“surname”`
     Age int `pretty:“other” json:“age”`
}


func main() {
     u := User{“John”, “John@Gmail.com”, “admin”}

     marSer, _ := Marshal(u)
     fmt.Println(“pretty user”, string(marSer))

     r := Record{“John”, “Johnson”,33}
     marRec, _:= Marshal(r)
     fmt.Println(“pretty rec”, string(marRec))

     jsonRec, _ := json.Marshal(r)
     fmt.Println(“json rec”,string(jsonRec))
}
```

```
pretty user {UserId:JOHN, Email:john@gmail.com}

pretty rec {Name:john, Surname:JOHNSON}

json rec {”name”:”John”,”surname”:”Johnson”,”age”:33}
```

8.6 SUMMARY

In this Chapter we explore different encoding formats available in Go. In particular, we explore CSV, JSON, XML and YAML. The simplicity of the marshal/unmarshal methods together with the utilization of fields tagging makes it possible to define how a struct has to be converted to different formats. Finally, we showed a use-case that permits the definition of a custom encoding using fields tagging.

CHAPTER 9

HTTP

Since Tim Berners Lee came up with the idea of the World Wide Web, HTTP has been its foundation. A good understanding of HTTP is basic to exchange data, manipulate APIs or crawl the web. This Chapter, details the tools that Go provides all the necessary elements to manage HTTP constructs such as requests, cookies or headers. In particular, we explore the `net/http` package. This Chapter assumes you are familiar with HTTP and how it works.

9.1 REQUESTS

HTTP requests are actions to performed by an HTTP server at a given URL following the client-server paradigm. Go simplifies HTTP requests sending with functions `Get`, `Post`, and `PostForm`. By simply invoking these functions an HTTP request is sent.

9.1.1 GET

The simplest manner to send an `GET` request is using the `http.Get` function. This function returns an `http.Response` type with a filled error y any. Like defined in the HTTP protocol, the response has a `Header` type and a body encoded into a `Reader` type.

Example 9.1: GET request.

```
package main

import (
    “bufio”
    “fmt”
    “net/http”
)

```

```
func main() {

    resp, err := http.Get(“https://httpbin.org/get”)
    if err != nil {
        panic(err)
    }

    fmt.Println(resp.Status)
    fmt.Println(resp.Header[“Content-Type”])

    defer resp.Body.Close()
    buf := bufio.NewScanner(resp.Body)

    for buf.Scan() {
        fmt.Println(buf.Text())
    }
}
```

```
200 OK
[application/json]
{
  “args”: {},
  “headers”: {
    “Accept-Encoding”: “gzip”,
    “Host”: “httpbin.org”,
    “User-Agent”: “Go-http-client/2.0”,
    “X-Amzn-Trace-Id”: “Root=1-6006ab94-3e51e02b509a1d3433bb59c1”
  },
  “origin”: “111.111.111.111”,
  "url": "https://httpbin.org/get"
}
```

Example 9.1, shows how to send a `GET` request to `https://httpbin.org/get`. This URL will return a basic response we can use to test our program. The response status is one of the values defined at the RFC7231 [8] (200 if everything went right). The header field is a map, therefore we can access values using the name of the header we are looking for like in `resp.Header["Content-Type"]`. The body of the response is a reader that has to be consumed. If you are not familiar with I/O operations check Chapter 7. A convenient way to consume this reader is using any of the functions from the `bufio` package.

Do not forget to close the reader when accessing the body (`resp.Body.Close()`).

9.1.2 POST

Sending a POST follows the same principles of `GET` requests. However, we are expected to send a body with the request and set the Content-type attribute in the header. Again, Go has a `Post` function that simplifies this operation as shown in Example 9.2 . Notice that the POST body is expected to be an `io.Reader` type. In this example, we send a JSON body to the `https://httpbin.org/post`which simply returns the body we have sent. Notice that the second argument of `http.Post` indicates the content-type and the format or our body. Accessing the fields from the `Response` type can be done as explained in the previous example.

Example 9.2: POST request.

```
package main

import (
    "bufio"
    “bytes”
    “fmt”
    “net/http”
)

 func main() {
     bodyRequest := []byte(‘“user”: “john”,“email”:“john@gmail.com”‘)
     bufferBody := bytes.NewBuffer(bodyRequest)

     url := “https://httpbin.org/post”

     resp, err := http.Post(url, “application/json”, bufferBody)
     if err != nil {
         panic(err)
     }
     defer resp.Body.Close()

     fmt.Println(resp.Status)
     bodyAnswer := bufio.NewScanner(resp.Body)
     for bodyAnswer.Scan() {
         fmt.Println(bodyAnswer.Text())
     }
 }
```

```
200 OK
{
  “args”: {},
  “data”: “{\”user\”: \”john\”,\”email\”:\”john@gmail.com\”}”,
  “files”: {},
  “form”: {},
  “headers”: {
    “Accept-Encoding”: “gzip”,
    “Content-Length”: “41”,
    “Content-Type”: “application/json”,
    “Host”: “httpbin.org”,
    “User-Agent”: “Go-http-client/2.0”,
    “X-Amzn-Trace-Id”: “Root=1-6006b032-6cbe50a13751bc03798b9e0b”
  },
  “json”: {
    “email”: “john@gmail.com”,
    “user”: “john”
  },
  “origin”: “111.111.111.111”,
  “url”: “https://httpbin.org/post”
}
```

The `http.Post` is generic and admits any content-type. For those post methods using `application/x-www-form-urlencoded` the `PostForm` function gets rid off the content-type specification and directly admits and encodes the body values into the `url-encoded` format.

Example 9.3 sends url-encoded data to the URL from the previous example. The approach only differs in the utilization of the `url.Values` type to define the url-encoded values. Observe that the returned response contains the sent data in the `form` field, not in the `data` field like in the previous example.

Example 9.3: POST request using `PostForm`

```go
package main

import (
    “bufio”
    “fmt”
    “net/http”
    “net/url”
)

 func main() {

     target := “https://httpbin.org/post”

     resp, err := http.PostForm(target,
         url.Values{“user”: {“john”}, “email”: {“john@gmail.com”}})
     if err != nil {
         panic(err)
     }
     defer resp.Body.Close()

     fmt.Println(resp.Status)
     bodyAnswer := bufio.NewScanner(resp.Body)
     for bodyAnswer.Scan() {
         fmt.Println(bodyAnswer.Text())
     }
 }
```

```
200 OK

{
  “args”: {},
  “data”: “”,
  “files”: {},
  “form”: {
    “email”: “john@gmail.com”,
    “user”: “john”
  },
  “headers”: {
    “Accept-Encoding”: “gzip”,
    “Content-Length”: “32”,
    “Content-Type”: “application/x-www-form-urlencoded”,
    “Host”: “httpbin.org”,
    “User-Agent”: “Go-http-client/2.0”,
    “X-Amzn-Trace-Id”: “Root=1-602ce931-3ad4aabf7cba926306f53fd2”
  },
  “json”: null,
  “origin”: “139.47.90.49”,
  “url”: “https://httpbin.org/post”
}
```

9.1.3 Headers, clients, and other methods

We have already mentioned that the most common HTTP methods are GET and POST. However, there are more methods such as `DELETE`, `UPDATE` or `PATCH`. Additionally, headers contain valuable fields that can be required to successfully interact with servers and their methods. All these elements contained in a `http.Request` type can be customised. However, when a request is created from scratch it has to be sent using an `http.Client`. This type defines a configurable client that permits a more controlled utilization of resources.

Example 9.4 shows how to use a client with a customised request to use a `PUT` method with body. This example is easily extensible to other HTTP methods. First, we create a body content and a `http.Header` with the required. A new request is filled indicating the HTTP method, URL and body content. Observe that the header is set after we have created the request (`req.Header = header`). Next we instantiate a `http.Client` with a request timeout of five seconds. We invoke the `Do` method to send our request. From here, we follow the same steps from the previous examples.

Example 9.4: Other HTTP requests.

```
package main

import (
    “bufio”
    “bytes”
    “fmt”
    “net/http”
    “time”
)

func main() {
    bodyRequest := []byte(‘{“user”: “john”,“email”:“john@gmail.com”}‘)
    bufferBody := bytes.NewBuffer(bodyRequest)

    url := “https://httpbin.org/put”

    header := http.Header{}
    header.Add(“Content-type”, “application/json”)
    header.Add(“X-Custom-Header”, “somevalue”)
    header.Add(“User-Agent”, “safe-the-world-with-go”)

    req, err := http.NewRequest(http.MethodPut, url, bufferBody)

    if err != nil {
        panic(err)
    }

    req.Header = header

    client := http.Client{
        Timeout: time.Second * 5,
        }

    resp, err := client.Do(req)
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()

    fmt.Println(resp.Status)
    bodyAnswer := bufio.NewScanner(resp.Body)
    for bodyAnswer.Scan() {
        fmt.Println(bodyAnswer.Text())
    }
}
```

```
200 OK
{
  “args”: {},
  “data”: “{\”user\”: \”john\”,\”email\”:\”john@gmail.com\”}”,
  “files”: {},
  “form”: {},
  “headers”: {
    "Accept-Encoding": "gzip",
    "Content-Length": "41",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "safe-the-world-with-go",
    "X-Amzn-Trace-Id": "Root=1-6006b2a7-37b6eb882f50162e167aa0d8",
    "X-Custom-Header": "somevalue"
  },
  "json": {
    "email": "john@gmail.com",
    "user": "john"
  },
  "origin": "111.111.111.111",
  "url": "https://httpbin.org/put"
}
```

The response contains the `X-Custom-Header` we sent with the request and returns the body. The `Client` has additional fields such as a timeout, a redirection policy or a cookie jar. Check the documentation for more details[18].

9.2 HTTP SERVER

The `net/http` package defines a `ServerMux` type that implements a HTTP request multiplexer. The server matches incoming URL requests with a list of configured patterns. These patterns have an associated handler that is invoked to serve the request.

Example 9.5 registers the function `info` as the handler for the `/info` URL. Any function to be registered as a handler must follow the `func(ResponseWriter, *Request)` signature. In this example, the server returns a body with a message and prints the headers by standard output. Our handler does not make any distinction between HTTP methods and will respond to any request. To serve requests, the `http.ListenAndServe` function blocks the execution flow and waits forever for incoming requests.

Example 9.5: HTTP server using `http.HandleFunc`.

```
package main

import (
    “fmt”
    “net/http”
)

func info(w http.ResponseWriter, r *http.Request){
    for name, headers := range r.Header {
         fmt.Println(name,headers)
     }
     w.Write([]byte(“Perfect!!!”))
     return
}

func main() {
     http.HandleFunc(“/info”, info)
     panic(http.ListenAndServe(“:8090”, nil))
}
```

```
>>> curl -H “Header1: Value1” :8090/info
Perfect!!!
…
User-Agent [curl/7.64.1]
Accept [*/*]
```

```
Header1 [Value1]
```

To test this code we have to run the server and then make the request from an HTTP client. We could write our own client as explained before. However, we can use an already implemented client to check that our server is compatible with the standards and works as expected. In this case, we used `curl` to send a `GET` request with header `Header1: Value1`[19]. Notice that we print additional headers that the `curl` client adds to our request.

The `http.HandleFunc` registers handlers and a URI. We can directly set a handler to be invoked independently of the URI. This forces the handler to be invoked in every request. However, this can be useful depending on the use-case.

Example 9.6 implements a `Handler` type with struct `MyHandler`. To implement the `Handler` interface the `ServeHTTP` function has to be defined. Any request will be sent to this handler, therefore we can do things like URI selection. In the example `/hello` and `/goodbye` return different messages.

Example 9.6: HTTP server and handler.

```
package main

import “net/http”

type MyHandler struct {}

func(c *MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    switch r.RequestURI {
    case ”/hello”:
         w.WriteHeader(http.StatusOK)
         w.Write([]byte(“goodbye\n”))
    case ”/goodbye”:
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(“hello\n”))
    default:
        w.WriteHeader(http.StatusBadRequest)
    }
}

func main() {
    handler := MyHandler{}
    http.ListenAndServe(“:8090”, &handler)
}
```

```
>>> curl :8090/goodbye
hello
>>> curl :8090/hello
goodbye
```

9.3 COOKIES

The `Cookie`type[20] is a representation of the cookies you can find in a `Set-Cookie` header. They can be added to a `Request` and extracted from a `Response` type.

Example 9.7 instantiates a server that expects a cookie with a counter. The value received in the cookie is sent in the response incremented. The example is self-contained, the server is executed in a goroutine and the client sends requests to the server. After instantiating the cookie, we add it to the current request with `req.AddCookie(&c)`. Cookies are accessible in key-value pairs at both requests and responses. On the server-side, the function `r.Cookie(“counter”)` gets that cookie from the request if it was found. Similarly, we can set the cookie in the response using the `http.SetCookie` function. In the output, we capture the headers from the request and the response. The response contains the `Set-Cookie` header with the new value for the counter.

Example 9.7: Adding cookies to requests and responses.

```
package main

import (
    “fmt”
    “net/http”
    “strconv”
    “time”
)

func cookieSetter(w http.ResponseWriter, r *http.Request) {
    counter, err := r.Cookie(“counter”)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    value, err := strconv.Atoi(counter.Value)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    value = value + 1
    newCookie := http.Cookie{
        Name: “counter”,
        Value: strconv.Itoa(value),
    }
    http.SetCookie(w, &newCookie)
    w.WriteHeader(http.StatusOK)
    return
}

func main() {
    http.HandleFunc(“/cookie”, cookieSetter)
    go http.ListenAndServe(“:8090”, nil)

    url := “http://localhost:8090/cookie”
    req, err := http.NewRequest(“GET”,url,nil)
    if err != nil {
        panic(err)
    }

    client := http.Client{}

    c := http.Cookie{
        Name:“counter”, Value:“1”, Domain: “127.0.0.1”,
        Path: ”/”, Expires: time.Now().AddDate(1,0,0)}
    req.AddCookie(&c)

    fmt.Println(“—>”, req.Header)

    resp, err := client.Do(req)
    if err != nil {
        panic(err)
    }
    fmt.Println("<—",resp.Header)
}
```

```
—> map[Cookie:[counter=1]]
<— map[Content-Length:[0] Date:[Tue, 19 Jan 2021 20:12:59 GMT]
Set-Cookie:[counter=2]]
```

9.3.1 CookieJar

The previous example is a basic description of how to manipulate cookies. One drawback of our example is that we have to manually set the new cookie for future requests. This could be easily done considering that we only have one cookie. However, when interacting with web applications it is common to use many and it would be more efficient to have a non-supervised approach to update these cookies.

The [21] type from the package `net/http/cookiejar` implements an in-memory storage solution for cookies that follow the `CookieJar` interface from `net/http`. Example 9.8 extends Example 9.7 with a client using a `CookieJar`. The `SetCookies` method associate an array of cookies with an URL. Now every time the client operates with that URL, the `CookieJar` will update the corresponding cookies. This enhancement is only required at the client side. Finally, we iteratively send requests to the server which returns the updated cookie. Notice that without the `CookieJar` we would have to manually update the cookie for the next request.

This can be done adding the cookies to the `CookieJar` (`jar.SetCookies`) and setting the jar field in the `http.Client`. After every request cookies are automatically updated and we can check their values.

Example 9.8: Use of `CookieJar` to set cookie values.

```
package main
```

```go

import (
    “fmt”
    “net/http”
    “net/http/cookiejar”
    url2 “net/url”
    “strconv”
)

func cookieSetter(w http.ResponseWriter, r *http.Request) {
    counter, err := r.Cookie(“counter”)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    value, err := strconv.Atoi(counter.Value)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        return
    }
    value = value + 1
    newCookie := http.Cookie{
        Name: “counter”,
        Value: strconv.Itoa(value),
    }
    http.SetCookie(w, &newCookie)
    w.WriteHeader(http.StatusOK)
}
```

```

func main() {
    http.HandleFunc(“/cookie”, cookieSetter)
    go http.ListenAndServe(“:8090”, nil)

    jar, err := cookiejar.New(nil)
    if err != nil {
        panic(err)
    }
    cookies := []*http.Cookie{
        &http.Cookie{Name:“counter”,Value:“1”},
    }

    url := “http://localhost:8090/cookie”
    u, _ := url2.Parse(url)
    jar.SetCookies(u, cookies)

    client := http.Client{Jar: jar}

    for i:=0; i<5; i++ {
        _, err := client.Get(url)
        if err != nil {
            panic(err)
        }
        fmt.Println(“Client cookie”,jar.Cookies(u))
    }
}
```

```
Client cookie [counter=2]
Client cookie [counter=3]
Client cookie [counter=4]
Client cookie [counter=5]
Client cookie [counter=6]
```

9.4 MIDDLEWARE

Imagine an HTTP API that requires users to be authenticated in the system. A simple approach is to implement a basic authentication header checker to determine if a request must be processed or not. Additionally, certain operations are restricted to some user roles. These operations have to be done in every request. From the point of view of the implementation, this is a repetitive task that should be implemented once and reutilized accordingly. This is a clear example of middleware.

This middleware should run before any handler is invoked to process the request. The `net/http` package does not provide any tool to manage middleware solutions. However, this can be easily done by implementing handlers. The idea is to concatenate specialized middleware handlers that provide additional features.

```
func Middleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request){
        // Do something before the next handler is invoked
        next.ServeHTTP(w, r)
        // Do something when the next handler has finished
    })
}
// …
http.ListenAndServe(“:8090”,Middleware(Middleware(Handler)))
```

The code above summarizes the definition of a middleware handler. This handler receives another handler that has to be invoked with the same `Request` and `ResposeWriter`. This handler can modify both elements depending on its logic before and after the next handler is invoked. To apply the middleware to every request, we concatenate it with other handlers. Because the middleware returns a `Handler` we can concatenate other middlewares until finally invoke the target handler with the expected logic.

Example 9.9 implements a basic authorization mechanism for a server. Basic authorization is based on a header like `Authorization:`Basic credential where credentials is a string encoded in base64. The `AuthMiddleware` function checks if this header exists, decodes the credential and authorizes the request if it matches Open Sesame. If any of the steps fails, the middleware sets the request as non authorized (error 401) and returns.

Example 9.9: Basic authorization middleware handler.

```
package main

import (
    “encoding/base64”
    “net/http”
    “strings”
    “fmt”
)

type MyHandler struct{}

func (mh *MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request){
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(“Perfect!!!”))
    return
```

```
}

func AuthMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request){
        header := r.Header.Get(“Authorization”)
        if header == ”” {
            w.WriteHeader(http.StatusUnauthorized)
            return
        }

        authType := strings.Split(header,” ”)
        fmt.Println(authType)
        if len(authType) != 2 || authType[0] != “Basic” {
            w.WriteHeader(http.StatusUnauthorized)
            return
        }
        credentials,err := base64.StdEncoding.DecodeString(authType[1])
        if err != nil {
            w.WriteHeader(http.StatusUnauthorized)
            return
        }
        if string(credentials) == “Open Sesame” {
            next.ServeHTTP(w, r)
        }
    })
}

func main() {
    targetHandler := MyHandler{}
    panic(http.ListenAndServe(“:8090”,AuthMiddleware(&targetHandler)))
}
```

```
>>> auth=$(echo -n “Open Sesame” | base64);
>>> echo $auth
T3BlbiBTZXNhbWU=
>>> curl :8090 -w “%{http_code}”
401
>>> curl :8090 -w “%{http_code}” -H “Authorization: Basic Hello”
401
>>> curl :8090 -w “%{http_code}” -H “Authorization: Basic $auth”
Perfect!!!200
```

Now requests will be unauthorized until the authorization middleware finds the correct authorization header. To obtain the base64 encoding you can use any online encoder or the commands shown below if you use a Unix-like environment.

Now that we have seen how we can define a middleware handler a practical question is how to easily concatenate all the middleware. And this is an interesting question because we can expect applications to deal with several handlers and sophisticated configurations. The main idea is to replace `AuthMiddleware(&targetHandler)` for something that can be programmatically executed.

Example 9.10 defines a type `Middleware` and the `ApplyMiddleware` function to help the concatenation of multiple handlers. The idea is that `ApplyMiddleware` returns a handler that results from applying a collection of middlewares to the handler passed by argument. When registering handlers for our server, we simply invoke the function with the final handler and the middleware items. In our case, we register `/three` and `/one` URIs that apply three and one times the `SimpleMiddleware`. This middleware checks the simple header and adds a tick.

The number of ticks in the response header will the same of these middlewares executed when serving the request.

Example 9.10: Concatenation of several middleware handlers.

```
package main

import “net/http”

type MyHandler struct{}

func (mh *MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request){
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(“Perfect!!!”))
     return
}

type Middleware func(http.Handler) http.Handler

func ApplyMiddleware(h http.Handler, middleware … Middleware) http.Handler {
    for _, next := range middleware {
        h = next(h)
    }
    return h
}

func SimpleMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request){
        value := w.Header().Get(“simple”)
        if value == ”” {
            value = “X”
        } else {
            value = value + “X”
        }
        w.Header().Set(“simple”, value)
        next.ServeHTTP(w,r)
    })
}

func main() {

    h := &MyHandler{}

    http.Handle(“/three”, ApplyMiddleware(
        h, SimpleMiddleware, SimpleMiddleware, SimpleMiddleware))
    http.Handle(“/one”, ApplyMiddleware(
        h, SimpleMiddleware))

    panic(http.ListenAndServe(“:8090”, nil))
}
```

Using `curl` we can see how `/three` returns the header `Simple: XXX` while `/one` returns `Simple: X`. Obviously these middleware function only has demonstration purposes. However, applying other more realistic or sophisticated solutions can use exactly the same ideas.

```
>>> curl :8090/three  -i
HTTP/1.1 200 OK
Simple: XXX
```

```
Date: Wed, 17 Feb 2021 20:49:55 GMT
Content-Length: 10
Content-Type: text/plain; charset=utf-8

Perfect!!!%
>>> curl :8090/one  -i
HTTP/1.1 200 OK
Simple: X
Date: Wed, 17 Feb 2021 16:50:01 GMT
Content-Length: 10
Content-Type: text/plain; charset=utf-8


Perfect!!!%
```

9.5 SUMMARY

HTTP is widely used and offers many possibilities not only to retrieve web content but also to access remote APIs. This Chapter exposes how Go works with the main elements of HTTP including requests, responses, headers, and cookies. Extra material is explained regarding middleware and how easy it can be to define sophisticated pipelines to process HTTP requests.

CHAPTER 10

TEMPLATES

While you read these lines, there are thousands of millions of reports, web pages and other structured and non-structured data collections being generated. Many of them share a common template that is filled with information extracted from users and processed accordingly to a pre-defined logic. For example, the email sent to your account to reset your forgotten password is always the same, except for small chunks containing data you should be aware of.

From a practical perspective, these templates should be easy to change and modify without necessarily requiring to modify any source code. Go provides data-driven templates for generating textual output. These templates are executed by applying data structures that contain the data to be represented in the `template` package. This Chapter explores how to use these templates with some examples using the `text/template` and `html/template` packages for text and HTML output respectively.

10.1 FILLING TEMPLATES WITH STRUCTS

The `text/template`[22] package defines the syntax to generate textual templates. Our goal is to fill the gaps of a template with incoming data. Let's consider the string below.

```
Dear {{.Name}},

You were registered with id {{.UserId}}

and e-mail {{.Email}}.
```

This string defines a template that is expected to be filled with a `struct` with fields `Name`, `UserId`, and `Email`. Example 10.1 uses this template in conjunction with a `User` type to generate personalized messages.

Example 10.1: Fill template with a `struct`.

```
package main

```

```
import (
    “text/template”
    “os”
)

type User struct{
    Name string
    UserId string
    Email string
}

const Msg = ‘Dear {{.Name}},
You were registered with id {{.UserId}}
and e-mail {{.Email}}.
‘

func main() {
    u := User{”John”, “John33”, “john@gmail.com”}

    t := template.Must(template.New(”msg”).Parse(Msg))
    err := t.Execute(os.Stdout, u)
    if err != nil {
        panic(err)
    }
}
```

```
Dear John,
You were registered with id John33
and e-mail john@gmail.com.
```

Templates are allocated with the `New` function which receives a name for the template. The `Parse` method analyzes a template string definition. This method can return an error if the template parsing was not possible. The `template.Must` is a function wrapper that helps the template definition as shown in line 22. Finally, the method `Execute` applies a template to an `interface{}` type and writes the resulting output to an `io.Writer`. The resulting output is our template filled with the initialized `u` variable.

10.2 ACTIONS

Templates permit the definition of certain logic depending on the data value. Example 10.2 changes the output according to the user’s gender. The statement `{{if .Woman}}Mrs.{{- else}}Mr.{{- end}}` prints two different messages depending on the boolean field `Female`.

Example 10.2: Template `if/else`.

```
package main

import (
    “text/template”
    “os”
)

type User struct{
    Name string
     Female bool
}

const Msg = ‘
{{if .Female}}Mrs.{{- else}}Mr.{{- end}} {{.Name}},
Your package is ready.
Thanks,
‘
```

```
Mr. John,
Your package is ready.
Thanks,


Mrs. Mary,
Your package is ready.
Thanks,
```

```go

func main() {
    u1 := User{”John”, false}
    u2 := User{”Mary”, true}

    t := template.Must(template.New(”msg”).Parse(Msg))
    err := t.Execute(os.Stdout, u1)
    if err != nil {
        panic(err)
    }
    err = t.Execute(os.Stdout, u2)
    if err != nil {
        panic(err)
    }
}
```

For binary variable comparisons templates use a different syntax than the one used in Go. Check Table 10.1 for a list of equivalences.

Go	Templates
`<`	`lt`
`>`	`gt`
`<=`	`le`
`>=`	`ge`
`==`	`eq`
`!=`	`ne`

Table 10.1: Comparison operators in Go and templates.

Example 10.3 defines personalized messages depending on the score of every user.

Example 10.3: Template `if/else` binary variable comparisons.

```
package main

import (
    “text/template”
    “os”
)

type User struct{
    Name string
     Score uint32
}

const Msg = ‘
{{.Name}} your score is {{.Score}}
your level is:
{{if le .Score 50}}Amateur
{{else if le .Score 80}}Professional
{{else}}Expert
{{end}}
‘

func main() {
     u1 := User{”John”, 30}
     u2 := User{”Mary”, 80}

     t := template.Must(template.New(”msg”).Parse(Msg))
```

```
John your score is 30
your level is:
Amateur

Mary your score is 80
your level is:
Professional
```

```
    err := t.Execute(os.Stdout, u1)
    if err != nil {
        panic(err)
    }
    err = t.Execute(os.Stdout, u2)
    if err != nil {
        panic(err)
    }
}
```

Another common use case is the iteration through a collection of items. We can call a `range` iterator as we would do in a Go loop. The statement `{{range .}}{{print .}} {{end}}` iterates through a collection and prints every item. Example 10.4 uses this action to print the content of a string array containing the name of the musketeers.

Example 10.4: Template `range` iteration.

```
package main

import (
    “text/template”
    “os”
)

const msg = ‘
The musketeers are:
{{range .}}{{print .}} {{end}}
‘

func main() {
```

```
The musketeers are:
Athos Porthos Aramis D’Artagnan
```

```
    musketeers := []string{”Athos”, “Porthos”, “Aramis”,”D’Artagnan”}

    t := template.Must(template.New(”msg”).Parse(msg))
    err := t.Execute(os.Stdout, musketeers)
    if err != nil {
        panic(err)
    }
}
```

10.3 FUNCTIONS

Functions can be defined inside the template or in the global function map. The templates specification sets a variety of already available functions. Many of them are similar to functions already available in Go. However, we recommend you to take a look at the reference[23] for further details. Example 10.5 uses the function `{{slice . 3}}` which is equivalent to `x[3]` to get the item with index 3 from a slice.

Example 10.5: Template `slice` function.

```
package main

import (
    “text/template”
    “os”
)

const Msg = ‘
The fourth musketeer is:
{{slice . 3}}
‘
```

```
The fourth musketeer is:

[D’Artagnan]
```

```

func main() {
    musketeers := []string{”Athos”, “Porthos”, “Aramis”,”D’Artagnan”}

    t := template.Must(template.New(”msg”).Parse(Msg))

    err := t.Execute(os.Stdout, musketeers)
    if err != nil {
        panic(err)
    }
}
```

In other situations, we may need other functions not available at the current specification. In these situations, we can use the `FuncMap` type to define a map of available functions. Afterwards, we have to add this map of functions to our template to be callable. Example 10.6 declares a functions map where `“join”` maps the `strings.Join` function. Now we can call this new function from our template using `{{join . “,␣”}}`.

Example 10.6: Use of `FuncMap`.

```
package main

import (
   “strings”
   “text/template”
   “os”
)

const Msg = ‘
The musketeers are:
```

```
The musketeers are:
Athos, Porthos, Aramis,
D’Artagnan
```

```
{{join . “, “}}
‘

func main() {
    musketeers := []string{”Athos”, “Porthos”, “Aramis”,”D’Artagnan”}

    funcs := template.FuncMap{”join”: strings.Join}

    t, err := template.New(”msg”).Funcs(funcs).Parse(Msg)
    if err != nil {
        panic(err)
    }
    err = t.Execute(os.Stdout, musketeers)
    if err != nil {
        panic(err)
    }
}
```

It is possible to execute templates inside other templates using the `{{block “name”.}}{{end}}` and `{{define “name”}}{{end}}` actions. In Example 10.7, we define two template strings in `Header` and `Welcome`. The idea is that our `Welcome` template prints a collection of items after printing a header. The `{{define “hello”}}` statement looks for the template `“hello”` and executes it. Notice that we parse both templates independently. However, the second one uses the `helloMsg` instead of creating a new one. Finally, we have to execute both templates in the order the are expected to appear.

Example 10.7: Rendering templates inside other templates.

```
package main
```

```
Hello and welcome
```

```

import (
    “os”
    “text/template”
)

const Header = ‘
{{block “hello” .}}Hello and welcome{{end}}‘

const Welcome = ‘
{{define “hello“}}
{{range .}}{{print .}} {{end}}
{{end}}
‘

func main() {
    musketeers := []string{”Athos”, “Porthos”, “Aramis”,”D’Artagnan”}

    helloMsg, err := template.New(”start”).Parse(Header)
    if err != nil {
        panic(err)
    }

    welcomeMsg, err := template.Must(helloMsg.Clone()).Parse(Welcome)
    if err != nil {
        panic(err)
    }

```

```
Athos Porthos Aramis
D’Artagnan
```

```
    if err := helloMsg.Execute(os.Stdout, musketeers); err != nil {
        panic(err)
    }
    if err := welcomeMsg.Execute(os.Stdout, musketeers); err != nil {
        panic(err)
    }
}
```

10.4 HTML

Templates are particularly useful to serve HTML content. However, due to safety reasons and potential code injection attacks, Go provides a specific package for HTML templates in `html/template`. The different functions and methods from this package are fairly similar to those already explained for `text/template`.

In Example 10.8 we populate a web page with a collection of items. As you can see, we generate the output in the same way we did for text templates. Notice that the output follows the HTML specification automatically escaping characters when needed.

Example 10.8: HTML list with a template.

```
package main

import (
    “html/template”
    “os”
)

const Page = ‘
<html>
```

```
<head>
    <title>{{.Name}}’s Languages</title>
</head>
<body>
    <ul>
    {{range .Languages}}<li>{{print .}}</li>{{end}}
    </ul>
</body>
</html>
‘

type UserExperience struct {
    Name string
    Languages []string
}

func main() {
    languages := []string{”Go”,”C++”,”C#”}
    u := UserExperience{”John”, languages}

    t:= template.Must(template.New(”web”).Parse(Page))

    err := t.Execute(os.Stdout, u)
    if err != nil {
        panic(err)
    }
}
```

```
<html>
<head>
   <title>John’s Languages</title>
</head>
<body>
   <ul>
   <li>Go</li><li>C&#43;&#43;</li><li>C#</li>
   </ul>
</body>
</html>
```

10.5 SUMMARY

This Chapter explains how Go provides a language to define templates that can be filled on run-time. The examples from this Chapter demonstrate how to create templates and how to define template variations depending on incoming data.

CHAPTER 11 TESTING

Testing is one of the most important tasks to develop a successful project. Unfortunately, testing is usually postponed or not seriously taken into consideration by developers. Testing can be repetitive and sometimes even convoluted. Fortunately, Go defines tests in a simple manner that reduces tests adoption and help developers to focus more on the test and less on language constructions. This Chapter explores testing, benchmarking and profiling.

11.1 TESTS

Go provides an integrated solution for testing. Actually, any function with signature `TestXXX(t *testing.T)` is interpreted as a testing function.
Example 11.1 is a naive demonstration of how to check the correctness of a 2+2 operation. In this case, no error is going to occur. However, if any error is found, the `t` argument provides access to the `Error` function that sets this test to be failed.

Example 11.1: Single test.

```
package example_01

import “testing”

func TestMe(t *testing.T) {
    r := 2 + 2
    if r != 4 {
        t.Error(“expected 2 got”, r)
    }
}
```

Tests are thought to be used in conjunction with the `go test` tool. For the example above we can run the test as follow:

```
>>> go test
PASS
ok   github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/testing/example_010.341s
>>> go test -v
=== RUN   TestMe
– PASS: TestMe (0.00s)
PASS
ok   github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/testing/example_010.090s
```

The basic output from `go test` is the final status of the testing (`OK`) and the elapsed time. When using the verbose flag (`-v`) this information is printed for every test. Now, if we force an error we get:

```
>>> go test
– FAIL: TestMe (0.00s)
    example01_test.go:8: expected 2 got 3
FAIL
exit status 1
FAILgithub.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/testing/example_010.089s
```

The output shows that the final status is `FAIL` with the error message we added in case of error.

Tests are intended to improve the reliability of our code while helping us find potential errors. In Example 11.2 , we test a toy function that prints a Schwarzenegger’s movie quote. To test the completeness of our implementation, we call `MovieQuote` with all the movies we have. Unfortunately, we forgot to define a quote for the Predator movie[24].

Example 11.2: Function test.

```
package example_02

import “testing”

type Quote string
type Movie string

const (
    Crush Quote = “To crush your enemies…”
    T1000 Quote = “I’ll be back”
    Unknown Quote = “unknown”

    Conan Movie = “conan”
    Terminator2 Movie = “terminator2”
    Predator Movie = “predator”
)

func MovieQuote(movie Movie) Quote {
    switch movie {
    case Conan:
        return Crush
    case Terminator2:
        return T1000
    default:
        return Unknown
    }
}

```

```
=== RUN   TestMovieQuote
    example02_test.go:33: unknown  quote for movie predator
– FAIL: TestMovieQuote (0.00s)
FAIL
```

```
func TestMovieQuote(t *testing.T) {
    movies := []Movie{Conan, Predator, Terminator2}
    for _, m := range movies {
        if q := MovieQuote(m); q==Unknown{
            t.Error(“unknown  quote for movie”, m)
        }
    }
}
```

The `testing` package offers a good number of features to help test creation. In this sense, testing requirements may change depending on factors such as timing. Sometimes depending on the testing stage time restrictions may appear. Tests can be skipped using `t.Skip()`. This can be used with the `testing.Short()` function that returns `true` when the `-short` param is set in the `go test` command to avoid exhaustive testing. Example 11.3 shows how to skip a test when the duration of an iteration goes beyond a given timeout. If the `-short` flat is set, test skipping will be automatically done.

Example 11.3: Test skipping.

```
package example_03

import (
    “fmt”
    “math”
    “testing”
    “time”
)

func Sum(n int64, done chan bool) {
    var result int64 = 0
12     var i int64
13     for i = 0; i<n; i++ {
14         result = result + i
15     }
16     done <- true
17 }
18
19 func TestSum(t *testing.T) {
20     var i int64
21     done := make(chan bool)
22     for i = 1000; i<math.MaxInt64; i+=100000 {
23         go Sum(i, done)
24         timeout := time.NewTimer(time.Millisecond)
25         select {
26         case <- timeout.C:
27             t.Skip(fmt.Sprintf(“%d took longer than 1 millisecond”,i))
28         case <- done:
29
30         }
31     }
32 }
```

```
go test -v
=== RUN   TestSum
   example03_test.go:27: 4101000 took longer than 1 millisecond
– SKIP: TestSum (0.02s)
PASS
ok   github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/testing/example_030.256s
```

11.1.1 Subtests

Certain tests can be reutilized to test particular code properties. permit triggering tests from other tests using the `t.Run()` function.

```
func TestFoo(t *testing.T) {

    // …

    t.Run(“A=1”, func(t *testing.T) { … })

    t.Run(“A=2”, func(t *testing.T) { … })

    t.Run(“B=1”, func(t *testing.T) { … })

    // …

}
```

In the example above, `TestFoo` invokes three testing functions. Subsequently, these testing functions can invoke others. The first argument is a string that can be conveniently set to a `key=value` format. This identifies the test and permits us to isolate what tests to be run in the `go test` command.

We show how to work with subtests with a practical example. Because the code can be a bit large to be shown at once, we split it into two sections for better readability. In this example, we want to test a `User` type that can be saved in XML and JSON formats. Check Chapter 8 if you are not familiar with encodings.

Example 11.4: Subtests in practice (part I).

```
package example_03

import (
    “encoding/json”
    “encoding/xml”
    “errors”
    “fmt”
    “io/ioutil”
    “os”
     “testing”
 )

 type Encoding int

 const (
     XML Encoding = iota
     JSON
 )

 type User struct {
     UserId string ‘xml:“id” json:“userId”‘
     Email string ‘xml:“email” json:“email”‘
     Score int ‘xml:“score” json:“score”‘
 }

 var Users []User

 func (u *User) Equal(v User) bool{
     if u.UserId != v.UserId ||
         u.Email != v.Email ||
         u.Score != v.Score {
         return false
     }
     return true
 }
```

```go


func (u *User) encode(format Encoding) ([]byte, error) {
    var encoded []byte = nil
    var err error
    switch format {
    case XML:
        encoded, err = xml.Marshal(u)
    case JSON:
        encoded, err = json.Marshal(u)
    default:
        errors.New(“unknown encoding format”)
    }
    return encoded, err
}

func (u *User) fromEncoded(format Encoding, encoded []byte) error {
    recovered := User{}
    var err error
    switch format {
    case XML:
        err = xml.Unmarshal(encoded, &recovered)
    case JSON:
        err = json.Unmarshal(encoded, &recovered)
    default:
        err = errors.New(“unknown encoding format”)
    }

    if err == nil {
        *u = recovered
    }
    return err
}

func (u *User) write(encoded []byte, path string) error {
    err := ioutil.WriteFile(path, encoded, os.ModePerm)
    return err
}

func (u *User) read(path string) ([]byte, error) {
    encoded, err := ioutil.ReadFile(path)
    return encoded, err
}

func (u *User) ToEncodedFile(format Encoding, filePath string) error {
    encoded,err := u.encode(format)
    if err != nil {
       return err
    }
    err = u.write(encoded, filePath)
    return err
}

func (u *User) FromEncodedFile(format Encoding, filePath string) error {
    encoded, err := u.read(filePath)
    if err != nil {
        return err
    }
    err = u.fromEncoded(format, encoded)
    return err
}
```

Example 11.4 shows the code to be tested. The `User` type can be written to a file and recovered from a file with `ToEncodedFile` and `FromEncodedFile` methods respectively. The `Encoding` type is an enum to indicate the format we are using.

For the same type we have various features: we can work with XML and JSON formats and we can write and read both formats. We can use subtests to obtain finer granularity and better control. Example 11.5 continues Example 11.4 with all the corresponding tests. The entry point for testing is `TestMain` where we can set up code to be later used during the testing process. Afterwards, we invoke `m.Run()` to start the tests as usual.

Example 11.5: Subtests in practice (part II).

```
func testWriteXML(t *testing.T) {
    tmpDir := os.TempDir()
    for _, u := range Users {
        f := tmpDir+u.UserId+".xml"
        err := u.ToEncodedFile(XML, f)
        if err != nil {
            t.Error(err)
        }
    }
}

func testWriteJSON(t *testing.T) {
    tmpDir := os.TempDir()
    for _, u := range Users {
        f := tmpDir+u.UserId+”.json”
        err := u.ToEncodedFile(JSON, f)
        if err != nil {
            t.Error(err)
        }
    }
}


func testReadXML(t *testing.T) {
    tmpDir := os.TempDir()
    for _, u := range Users {
        f := tmpDir+”/”+u.UserId+”.xml”
        newUser := User{}
        err := newUser.FromEncodedFile(XML, f)
        if err != nil {
            t.Error(err)
        }
        if !newUser.Equal(u) {
            t.Error(fmt.Sprintf(“found %v, expected %v”,newUser, u))
        }
    }
}

func testReadJSON(t *testing.T) {
    tmpDir := os.TempDir()
    for _, u := range Users {
        f := tmpDir+”/”+u.UserId+”.json”
        newUser := User{}
        err := newUser.FromEncodedFile(JSON, f)
        if err != nil {
            t.Error(err)
        }
        if !newUser.Equal(u) {
            t.Error(fmt.Sprintf(“found %v, expected %v”,newUser, u))
        }
    }
}

func testXML(t *testing.T) {
    t.Run(“Action=Write”, testWriteXML)
    t.Run(“Action=Read”, testReadXML)

    tmpDir := os.TempDir()
    for _, u := range Users {
        f := tmpDir + ”/” + u.UserId + ”.xml”
        _ = os.Remove(f)
    }
}

func testJSON(t *testing.T) {
    t.Run(“Action=Write”, testWriteJSON)
    t.Run(“Action=Read”, testReadJSON)

    tmpDir := os.TempDir()
    for _, u := range Users {
        f := tmpDir + ”/” + u.UserId + ”.json”
        _ = os.Remove(f)
    }
}

func TestEncoding(t *testing.T) {
    t.Run(“Encoding=XML”, testXML)
    t.Run(“Encoding=JSON”, testJSON)
}

func TestMain(m *testing.M) {
    UserA := User{“UserA”,“usera@email.org”,42}
    UserB := User{“UserB”, “userb@email.org”, 333}
    Users = []User{UserA, UserB}

    os.Exit(m.Run())
}
```

Notice that only `TestEncoding` is a valid testing function because it is named with an uppercase letter. The other functions (`testJSON`, `testXML`, `testReadJSON`, etc.) are not test functions and will not be executed alone when running `go test`. This is done in this way to permit a testing route using subsets starting at `TestEncoding`. The main idea is to test encodings (XML and JSON) separately as well as their write and read operations.

```
>>> go test -v
=== RUN   TestEncoding
=== RUN   TestEncoding/Encoding=XML
=== RUN   TestEncoding/Encoding=XML/Action=Write
=== RUN   TestEncoding/Encoding=XML/Action=Read
=== RUN   TestEncoding/Encoding=JSON
=== RUN   TestEncoding/Encoding=JSON/Action=Write
=== RUN   TestEncoding/Encoding=JSON/Action=Read
– PASS: TestEncoding (0.00s)
   – PASS: TestEncoding/Encoding=XML (0.00s)
      – PASS: TestEncoding/Encoding=XML/Action=Write (0.00s)
      – PASS: TestEncoding/Encoding=XML/Action=Read (0.00s)
   – PASS: TestEncoding/Encoding=JSON (0.00s)
      – PASS: TestEncoding/Encoding=JSON/Action=Write (0.00s)
      – PASS: TestEncoding/Encoding=JSON/Action=Read (0.00s)
PASS
ok   github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/testing/example_030.087s
```

The output shows the routes containing the labels we defined in the subtests (e.g. `TestEncoding/Encoding=XML/Action=Write`). Now we can filter what tests to be run. For example, we can test JSON operations only by using the `-run` flag with a regular expression matching the labels we created for the subtests (`Encoding=JSON`).

```
go test -v -run /Encoding=JSON
=== RUN   TestEncoding
=== RUN   TestEncoding/Encoding=JSON
=== RUN   TestEncoding/Encoding=JSON/Action=Write
=== RUN   TestEncoding/Encoding=JSON/Action=Read
– PASS: TestEncoding (0.00s)
   – PASS: TestEncoding/Encoding=JSON (0.00s)
      – PASS: TestEncoding/Encoding=JSON/Action=Write (0.00s)
      – PASS: TestEncoding/Encoding=JSON/Action=Read (0.00s)
```

```
PASS

ok   github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/testing/example_030.087s
```

Or we can run every write operation with the regular expression `/./Action=Write` that matches any encoding.

```
go test -v -run /./Action=Write

=== RUN   TestEncoding

=== RUN   TestEncoding/Encoding=XML

=== RUN   TestEncoding/Encoding=XML/Action=Write

=== RUN   TestEncoding/Encoding=JSON

=== RUN   TestEncoding/Encoding=JSON/Action=Write

– PASS: TestEncoding (0.00s)

   – PASS: TestEncoding/Encoding=XML (0.00s)

      – PASS: TestEncoding/Encoding=XML/Action=Write (0.00s)

   – PASS: TestEncoding/Encoding=JSON (0.00s)

      – PASS: TestEncoding/Encoding=JSON/Action=Write (0.00s)

PASS

ok   github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/testing/example_030.083s
```

Observe that only testing the read operation will fail because this operation expects a file previously written by the writing tests. Modifying the code to permit running `go test -v -run /./Action=Read` remains as an exercise for the reader.

11.2 EXAMPLES

In the same way, tests are defined in Go with functions starting with `Test`, examples can be defined with functions starting with . An interesting feature of examples is that the standard output they generate can be checked during testing. This is an important feature to ensure code coherence. Additionally, examples are included in Go generated documentation as described in Chapter 12.2.

Examples follow the naming convention described below.

```
func Example(…) // for the package

func ExampleF(…) // F for functions

func ExampleT(…) // T for types

func ExampleT_M(…) // M method of type T

func ExampleT_M_suffix(…) // with a suffix if more than one
```

To check the output correctness, every example must define the expected output to be generated. This is indicated using the `Output` comment as shown below. Where the last line must match the expected output. In those cases the output order is not guaranteed, we can use `Unordered output:` instead of `Output:`.

```
func ExampleX() {

    …

    // Output:

    // Expected output

}
```

Example 11.6 defines the `User` type and its methods. Notice that the notation for the different examples follows the above guidelines. Examples are run like tests.

Example 11.6: Definition of examples.

```
package example_01

import (
    “fmt”
    “strings”
```

```
>>> go test -v
=== RUN   ExampleUser
– PASS: ExampleUser (0.00s)
```

```
)

type User struct {
    UserId string
    Friends []User
}

func (u *User) GetUserId() string {
    return strings.ToUpper(u.UserId)
}

func (u *User) CountFriends() int {
    return len(u.Friends)
}

func CommonFriend(a *User, b *User) *User {
    for _, af := range a.Friends {
        for _, bf := range b.Friends {
            if af.UserId == bf.UserId {
                return &af
            }
        }
    }
    return nil
}

func ExampleUser() {
    j := User{“John”, nil}
```

```
=== RUN   ExampleCommonFriend
– PASS: ExampleCommonFriend (0.00s)
=== RUN   ExampleGetUserId_User
– PASS: ExampleGetUserId_User (0.00s)
=== RUN   ExampleCountFriends_User
– PASS: ExampleCountFriends_User
(0.00s)
PASS
```

```
    m := User{“Mary”, []User{j}}
    fmt.Println(m)
    // Output:
    // {Mary [{John []}]}
}

func ExampleCommonFriend() {
    a := User{“a”, nil}
    b := User{“b”, []User{a}}
    c := User{“c”, []User{a,b}}

    fmt.Println(CommonFriend(&b,&c))
    // Output:
    // &{a []}
}

func ExampleUser_GetUserId() {
    u := User{“John”,nil}
    fmt.Println(u.GetUserId())
    // Output:
    // JOHN
}

func ExampleUser_CountFriends() {
    u := User{“John”, nil}
    fmt.Println(u.CountFriends())
    // Output:
    // 0
```

```
}
```

11.3 BENCHMARKING

Go provides benchmarking for those functions starting with the `Benchmark` prefix. Benchmarks are executed with the `go test` command when the `-bench` flag is present. They use the `testing.B` type instead of `testing.T` like normal tests.

A basic benchmarking is shown in Example 11.7 for a loop computing a sum of numbers. The value of `b.N` contains the number of repetitions of the operation in which performance is being measured. As can be confirmed from the benchmark output, the benchmark is run three times with different `b.N` values. This is done to ensure the reliability of the benchmark.

Benchmarks are highly configurable using the available `go test` flags[25]. The benchmark output indicates the benchmark with the number of available goroutines (`BenchmarkSum-16`), the number of executed iterations (3817), and the nanoseconds elapsed per iteration (265858).

Example 11.7: Function benchmarking.

```
package example_01

import (
    “fmt”
    “testing”
)

func Sum(n int64) int64 {
    var result int64 = 0
    var i int64
    for i = 0; i<n; i++ {
        result = result + i
```

```go
    }
    return result
}

func BenchmarkSum(b *testing.B) {
    fmt.Println(“b.N:”,b.N)
    for i:=0;i<b.N;i++ {
        Sum(1000000)
    }
}
```

```
>>> go test -v -bench .
goos: darwin
goarch: amd64
pkg: github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/benchmarking/example_01
BenchmarkSum
b.N: 1
b.N: 100
b.N: 3817
BenchmarkSum-16         3817    265858 ns/op
PASS
ok   github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/benchmarking/example_011.243s
```

By default, benchmark iterations are executed sequentially. In certain scenarios, particularly in those with shared resources, executing these iterations in parallel may be more valuable. The `RunParallel` method executes a function in parallel using the available testing goroutines. The number of available goroutines can be set with the `-cpu` flag in `go test`. The `Next()` function indicates if more iterations have to be executed.

```
func BenchmarkSumParallel(b *testing.B) {
    b.RunParallel(func(pb *testing.PB){
        for pb.Next() {
            // Do something
        }
    })
}
```

Example 11.8 executes the same iterative addition from the previous example using parallel benchmarking. We vary the number of available goroutines to demonstrate how we can increase the number of operations per time. See how the suffix added to the benchmark function name reflects the current number of goroutines.

Example 11.8: Function parallel benchmarking.

```
package example_02

import “testing”

func Sum(n int64) int64 {
    var result int64 = 0
    var i int64
    for i = 0; i<n; i++ {
        result = result + i
    }
    return result
}

func BenchmarkSumParallel(b *testing.B) {
    b.RunParallel(func(pb *testing.PB){
        for pb.Next() {
            Sum(1000000)
        }
    })
}
```

```
>>> go test -v -bench . -cpu 1,2,4,8,16,32
goos: darwin
goarch: amd64
pkg: github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/benchmarking/example_02
BenchmarkSumParallel
BenchmarkSumParallel             4324    249886 ns/op
BenchmarkSumParallel-2           9462    127147 ns/op
BenchmarkSumParallel-4          18202     66514 ns/op
BenchmarkSumParallel-8          31191     33927 ns/op
BenchmarkSumParallel-16        36373     32871 ns/op
BenchmarkSumParallel-32        36981     34199 ns/op
PASS
ok   github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/benchmarking/example_028.874s
```

11.4 COVERAGE

Test coverage aims at measuring what percentage of the code has been tested. Any Go test can be executed with the `-cover` flag to activate coverage metrics.

To run coverage tests we have to write tests and the code to be tested in different files. We measure the coverage of Example 11.9 with Example 11.10. We can expect poor coverage because we only explore one of the branches in the `switch` statement. We only get 22.2% of coverage.

Example 11.9: File to be tested.

```
package example_01

func Periods(year int) string {
    switch {
    case year < -3000:
        return “Copper Age”
    case year < -2000:
        return “Bronze Age”
    case year < -1000:
        return “Iron Age”
    case year < 0:
        return “Classic Age”
    case year < 476:
        return “Roman Age”
    case year < 1492:
        return “Middle Age”
    case year < 1800:
        return “Modern Age”
    default:
        return “unknown”
    }
}
```

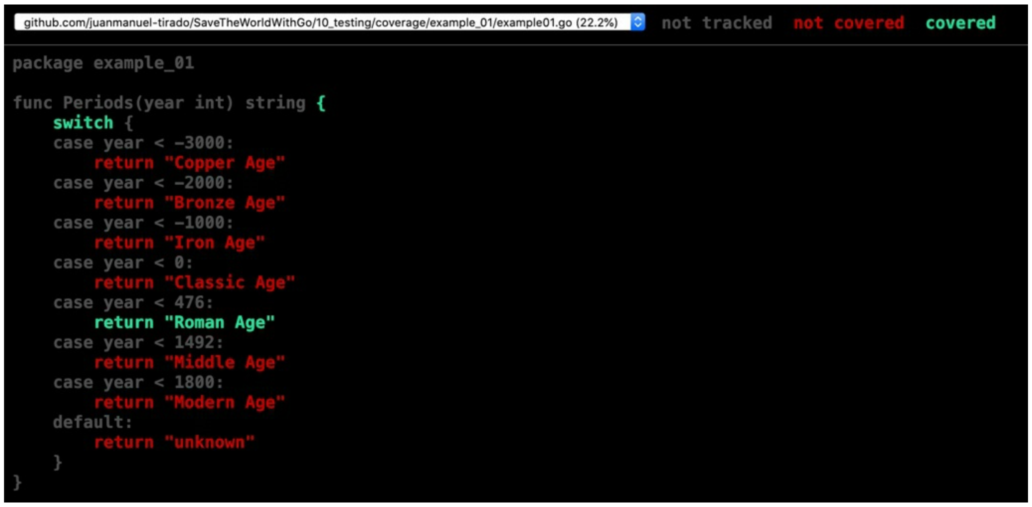
Example 11.10: Tests to show coverage.

```
package example_01

import “testing”
```

```

func TestOptions(t *testing.T) {
    Periods(333)
}
```

```
>>> go test -v -cover .
=== RUN   TestOptions
– PASS: TestOptions (0.00s)
PASS
coverage: 22.2% of statements
ok   github.com/juanmanuel-tirado/savetheworldwithgo/10_testing/coverage/example_01  0.541s  coverage: 22.2% of statements
```

Go provides tools to get deeper insights into coverage tests. The output from the coverage test can be exported to an intermediate file using the `-coverprofile=filepath` flag. The information dumped to the chosen file can be used to print additional information using `go tool cover`. For example, `go tool cover -func=filepath` prints the coverage of every function. A more visual analysis can be done with `go tool cover -html=filepath`. This option opens up a browser and shows the coverage for code regions as shown in Figure 11.1 where it is clear that only one branch was tested.


github.com/juanmanuel-tirado/SaveTheWorldWithGo/10_testing/coverage/example_01/example01.go (22.2%)
not tracked
not covered
covered

```
package example_01

func Periods(year int) string {
    switch {
    case year < -3000:
        return "Copper Age"
    case year < -2000:
        return "Bronze Age"
    case year < -1000:
        return "Iron Age"
    case year < 0:
        return "Classic Age"
    case year < 476:
        return "Roman Age"
    case year < 1492:
        return "Middle Age"
    case year < 1800:
        return "Modern Age"
    default:
        return "unknown"
    }
}
```

Figure 11.1: HTML detail for test coverage in Example 11.10 .

This coverage visualization cannot be especially useful because it does not detail how many times each statement was run. This can be changed with the `-covermode` flag to one of the three available modes: `set` boolean indicating if the statement was run or not, `count` counting how many times it was run, and `atomic` similar to `count` but safe for multithreaded executions.

11.5 PROFILING

Getting high performance and finding potential optimizations in your code is not an easy task. Profiling is a complex task that requires a deep understanding of code and the underlying language. Go provides the `runtime/pprof`[26] package with functions to define profiling entities. Fortunately, profiling functions are ready-to-use with benchmarks which facilitates the exploration of CPU and memory consumption in our code.

Profiling can be generated by launching benchmarks like described in this chapter. To force profiling, the `-cpuprofile filepath` and `-memprofile filepath` flags must be set for CPU and memory profiling respectively during `go test`. For both profiles, a file is created. These files can be interpreted by `pprof`[27] to generate and visualize useful reports. Thi tool is already integrated into the `go tool pprof` command.

To illustrate how to use the profiling, we take the benchmark from Example 11.11. The `BuildGraph` function returns a directed graph with edges connecting randomly selected vertices. The graph may not have sense, but it will help us understand how to approach a case of performance profiling.

Example 11.11: Profiling of a graph generation program.

```
package example_01

import (
    “math/rand”
    “testing”
    “time”
)

func BuildGraph(vertices int, edges int) [][]int {
     graph := make([][]int, vertices)
     for i:=0;i<len(graph);i++{
         graph[i] = make([]int,0,1)
     }
     for i:=0;i<edges;i++{
         from := rand.Intn(vertices)
         to := rand.Intn(vertices)
         graph[from]=append(graph[from],to)
     }

     return graph
 }

 func BenchmarkGraph(b *testing.B) {
     rand.Seed(time.Now().UnixNano())
     for i:=0;i<b.N;i++ {
         BuildGraph(100,20000)
     }
 }
```

First, we run the benchmarks dumping profiling information for memory and CPU using `go test`.

```
>>> go test -bench=. -benchmem -memprofile mem.out -cpuprofile cpu.out

goos: darwin

goarch: amd64

pkg: github.com/juanmanuel-tirado/SaveTheWorldWithGo/10_testing/profiling/example_01

BenchmarkGraph-16   1365   827536 ns/op   411520 B/op   901 allocs/op

PASS

ok   github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/profiling/example_011.759s
```

Now `mem.out` and `cpu.out` files can be used to generate the corresponding reports. First, we generate visual reports to have a superficial understanding of what is going on. In our case, we generate a PDF output for both, memory and CPU profiles.

```
>>> go tool pprof -pdf -output cpu.pdf cpu.out

Generating report in cpu.pdf

>>> go tool pprof -pdf -output mem.pdf mem.out

Generating report in mem.pdf
```

Now `cpu.pdf` and `mem.pdf` are visualizations of CPU and memory profiles. Check out the `go tool` help section for further available output formats [28]. From the visual report of the memory utilization shown in Figure 11.2 we can observe that `BuildGraph` is basically consuming all the available memory.

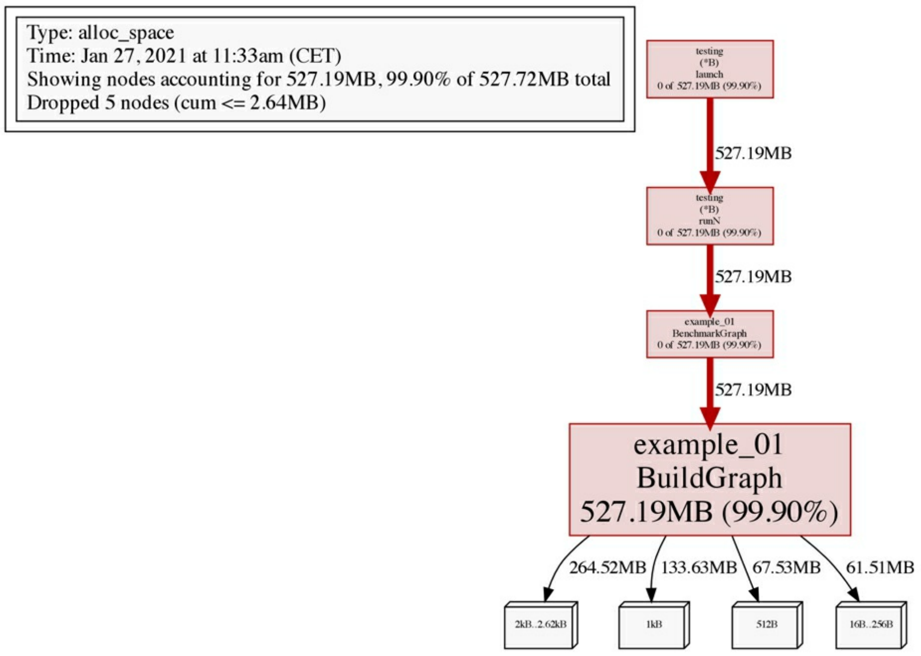

Type: alloc_space
Time: Jan 27, 2021 at 11:33am (CET)
Showing nodes accounting for 527.19MB, 99.90% of 527.72MB total
Dropped 5 nodes (cum <= 2.64MB)
testing (*B) launch 0 of 527.19MB (99.90%)
527.19MB
testing (*B) runN 0 of 527.19MB (99.90%)
527.19MB
example_01 BenchmarkGraph 0 of 527.19MB (99.90%)
527.19MB
example_01 BuildGraph 527.19MB (99.90%)
264.52MB
133.63MB
67.53MB
61.51MB
2kB..2.62kB
1kB
512B
16B..256B

Figure 11.2: Memory profile visualization of Example 11.11 .

The CPU visualization from the excerpt in Figure 11.3 looks more complex as it includes all the internal calls of the program. However, we can see that there is a bottleneck in the utilization of the random generation numbers. The largest the node size in the report, the longer time was spent there.

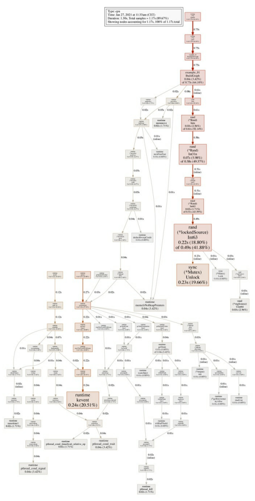


Figure 11.3: Excerpt of the CPU profile visualization from Example 11.11 .

We can go for a more detailed analysis using the online mode. This mode comes with plenty of options that will help us find what is going on line by line. Entering the `go tool pprof` we can explore the profiling output we have previously generated. The command `top` prints in decreasing order the functions where the program spent more time. The column `%flat` displays the percentage of time the program spent in the function. Column `%sum` is the total percentage spent after we leave that function. For example, `math/rand.(*Rand).Intn` only uses 5% of the execution time (`%flat`). However, 74.17% of the program time has already been spent when we reach that function.

```
>>> go tool pprof cpu.out
Type: cpu
Time: Jan 27, 2021 at 12:06pm (CET)
Duration: 1.30s, Total samples = 1.20s (92.08%)
Entering interactive mode (type “help” for commands, “o” for options)
(pprof) top
Showing nodes accounting for 1010ms, 84.17% of 1200ms total
Showing top 10 nodes out of 82
     flat  flat%   sum%        cum   cum%
    250ms 20.83% 20.83%      250ms 20.83%  sync.(*Mutex).Unlock (inline)
    220ms 18.33% 39.17%      530ms 44.17%  math/rand.(*lockedSource).Int63
    210ms 17.50% 56.67%      210ms 17.50%  runtime.kevent
    150ms 12.50% 69.17%      690ms 57.50%  math/rand.(*Rand).Int31n
     60ms  5.00% 74.17%      750ms 62.50%  math/rand.(*Rand).Intn
     40ms  3.33% 77.50%       60ms  5.00%  math/rand.(*rngSource).Int63 (inline)
     20ms  1.67% 79.17%      870ms 72.50%  github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/profiling/example_01.BuildGraph
     20ms  1.67% 80.83%       20ms  1.67%  math/rand.(*rngSource).Uint64
     20ms  1.67% 82.50%       20ms  1.67%  runtime.madvise
```

```
      20ms  1.67% 84.17%       20ms  1.67%  runtime.memclrNoHeapPointers
```

We can continue exploring our program with a more detailed view of the elapsed time per line of code. We list how `BuildGraph` consumes CPU with the command `list BuildGraph`. Now it becomes clear that the generation of random numbers is the main CPU bottleneck in our program. With this information, we can try to replace these functions for other functions with better performance, or think about a different approach to our solution.

```
(pprof) list BuildGraph

Total: 1.20s

ROUTINE ======================== github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/profiling/example_01.BuildGraph in
/github.com/juanmanuel-
tirado/SaveTheWorldWithGo/10_testing/profiling/example_01/example01_test.go

    20ms  870ms (flat, cum) 72.50% of Total

      .      .      5:    “testing”

      .      .      6:    “time”

      .      .      7:)

      .      .      8:

      .      .      9:func BuildGraph(vertices int, edges int) [][]int {

      .   20ms     10:    graph := make([][]int, vertices)

      .      .     11:    for i:=0;i<len(graph);i++{

      .   10ms     12:        graph[i] = make([]int,0,1)

      .      .     13:    }

      .      .     14:    for i:=0;i<edges;i++{

      .  390ms     15:        from := rand.Intn(vertices)

      .  370ms     16:        to := rand.Intn(vertices)

    20ms   80ms     17:        graph[from]=append(graph[from],to)

      .      .     18:    }

      .      .     19:

      .      .     20:    return graph
```

```
   .      .     21:}
     .      .     22:
```

11.6 SUMMARY

This Chapter presents the tools for testing available in Go. We explain how tests are carried out and organized with practical examples that may serve as a basis for any real project testing. We explore how the `testing` package also offers benchmarking tools that can take advantage of parallel execution. Finally, we show how code coverage and profiling can be done to achieve a better understanding of our code.

CHAPTER 12

MODULES AND DOCUMENTATION

Nowadays is very difficult to find programming projects to be designed as isolated pieces of code. Projects use code developed by other developers. Other projects are designed to be incorporated into other projects. This dependency between projects requires tools to facilitate the definition of what external code a project requires. And even more important, a systematic way to document our code so it can be shared with others. This Chapter explains how Go incorporates code into our projects using modules and how we can document our programs.

12.1 MODULES

Dealing with dependencies is always key to ensure code compatibility and reproducibility. As it was explained in Chapterr 2.1.1 and showed through this book, the most basic tool to make code accessible to our programs is using `go get`. However, this is a very basic solution that does not solve issues such as code versioning. Nowadays projects require several third-party projects to run. For some years various solutions were proposed to manage code dependencies in Go, until Go modules became the official solution.

Go modules facilitates the management of dependencies and makes it easier to share project requirements among developers. The idea is simple, to store the requirements of a project in a common format that can be used to acquire all the necessary code for a successful compilation. Go modules are governed by the `go.mod` file that contains information about what projects in what versions are required. Assume we start with the code from Example 12.1. This code uses a third-party logger to print a message. In the most basic approach, we would download the code using `go get`. However, with modules, we can simplify this process.

Example 12.1: Program using modules.

```
package main

import “github.com/rs/zerolog/log”

func main() {
    log.Info().Msg(“Save the world with Go!!!”)
}
```

In the folder containing our code we run `go mod init`. This creates a go.mod file which looks like:

```
module github.com/juanmanuel-tirado/SaveTheWorldWithGo/11_modules/modules/example_01


go 1.15
```

It contains the module name and the Go version we are running. The `go.mod` file is only generated in the root folder of the package. Now we can add a dependency. Dependencies are declared one by line using the `require package “version”` syntax. Fortunately, Go modules does this automatically for us. Every action done by the `go` command that requires our code to be executed (`build`, `run`, `test`, etc.) triggers the analysis of the required modules.

```
>>> go build main.go

go: finding module for package github.com/rs/zerolog/log

go: found github.com/rs/zerolog/log in github.com/rs/zerolog v1.20.0
```

Now if we check our `go.mod` file we can find a new line with the `zerolog` package and its current version. The *`//indirect`* comment indicates that this package has indirect dependencies. This means that additional packages were downloaded to satisfy the requirements of the package. Actually, the zerolog package has its own `go.mod` file with its own dependencies.

```
module github.com/juanmanuel-tirado/SaveTheWorldWithGo/11_modules/modules/example_01
```

```
go 1.15


require github.com/rs/zerolog v1.20.0 // indirect
```

The required packages are downloaded into the `GOPATH` folder as usual. However, we may need to store the packages in a vendor folder. This is not the recommended way, but if we still need to do it executing `go mod vendor` stores all the packages into the vendor folder.

⚠
When sharing our modules with others privately or in a public platform such as Github, be sure that you do not use a vendor folder. If so, you will ruin all the benefits of using go mod. Furthermore, you will replicate large pieces of code that are intended to be maintained by others.

⚠
Go modules is the latest standard for dependencies management in Go. You may find projects using other solutions such as `go dep`. Be careful when importing these packages to your code, and be sure they are compatible with your go modules.

12.2 DOCUMENTATION

Documenting code is one of those tasks developers always postpone, although we know it is a big mistake. A successful project must have good, up-to-date, and accessible documentation. Not only to ensure others understand the code but to make code maintainable. Go follows a minimalist approach to code documentation that can be summarized with the following rules:

- Comments before `package` declaration are considered to be a package comment.
- Every package *should* have a comment. For packages with multiple files, the package comment only needs to be present in one file.

- Every exported name *should* have a comment.
- Every commented item begins with the name of the item it describes.

Example 12.2 documents a piece of code following the mentioned conventions. It is important to notice that every comment starts with the name of the commented item. For large explanations, in particular, those containing code the `/**/` comment marks can be used.

Example 12.2: Program documentation.

```
// Package example_01 contains a documentation example.
package example_01

import “fmt”

// Msg represents a message.
type Msg struct{
    // Note is the note to be sent with the message.
    Note string
}

// Send a message to a target destination.
func (m *Msg) Send(target string) {
    fmt.Printf(“Send %s to %s\n”, m.Note, target)
}

// Receive a message from a certain origin.
func (m *Msg) Receive(origin string) {
    fmt.Printf(“Received %s from %s\n”, m.Note, origin)
}
```

Documented code can be processed with the `go doc` tool executing `go doc -all` in the project folder.

```
>>> go doc -all

 package example_01 // import “github.com/juanmanuel-
tirado/SaveTheWorldWithGo/11_modules/godoc/example_01”



 Package example_01 contains a documentation example.



 TYPES



 type Msg struct {

    // Note is the note to be sent with the message.

    Note string

 }

     Msg represents a message.



 func (m *Msg) Receive(origin string)

     Receive a message from a certain origin.



 func (m *Msg) Send(target string)

     Send a message to a target destination.
```

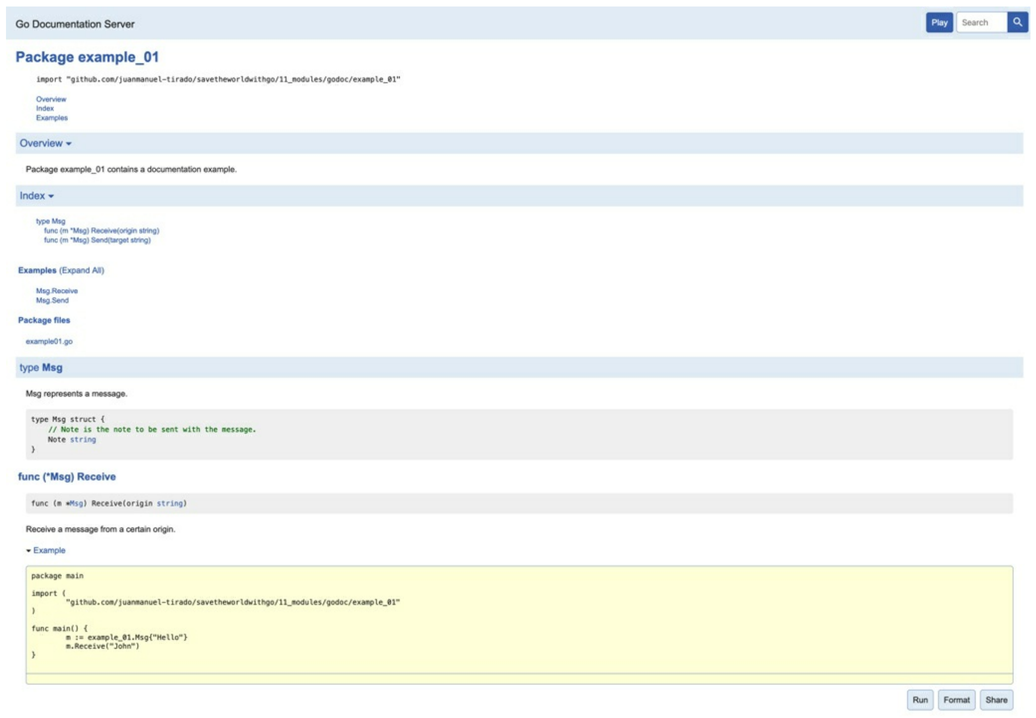
The `go doc` tool can actually display the documentation of any package available in the `GOPATH`. Apart from the flags and options [29] arguments are intended to follow the Go syntax. For example, `go doc fmt` prints the help for the `fmt` package, while `go doc json.decode` prints the documentation of the `Decode` method from the `json` package.

For a more interactive solution for documentation, Go provides the `godoc` server. Executing `godoc -http=:8080` serves incoming requests at port 8080 of your localhost. An interesting feature of `godoc` is the addition of runnable examples. The server interprets examples following the notation convention explained in Section 11.2. These examples must be allocated into a separated package to be interpreted. Normally, this is the same package with the `_test` suffix. In Example 12.3, we have written examples to be displayed with our documentation.

Example 12.3: Documented examples for Example 12.2 .

```
package example_01_test

import “github.com/juanmanuel-tirado/savetheworldwithgo/11_modules/godoc/example_01”

func ExampleMsg_Send() {
    m := example_01.Msg{“Hello”}
    m.Send(“John”)
    // Output:
    // Send Hello to John
}

func ExampleMsg_Receive() {
    m := example_01.Msg{“Hello”}
    m.Receive(“John”)
    // Output:
    // Received Hello from John
}
```

To make examples more accessible, it is possible to run the server in an interactive mode that permits examples to be executed in a runtime box. The command `godoc -http=:8080 -play` activates these boxes as shown in Figure 12.1.



Package example_01

```
import "github.com/juanmanuel-tirado/savetheworldwithgo/11_modules/godoc/example_01"
```

Package example_01 contains a documentation example.

Msg represents a message.

```
type Msg struct {
    // Note is the note to be sent with the message.
    Note string
}
```

func (*Msg) Receive

```
func (m *Msg) Receive(origin string)
```

Receive a message from a certain origin.

```
package main

import (
        "github.com/juanmanuel-tirado/savetheworldwithgo/11_modules/godoc/example_01"
)

func main() {
        m := example_01.Msg{"Hello"}
        m.Receive("John")
}
```

Figure 12.1: Excerpt of `godoc` Example 12.2 documentation.

12.3 SUMMARY

This Chapter shows how to manage dependencies in Go using Go modules to facilitate the adoption and exchange of projects. It also introduced the pragmatic approach to code documentation in Go and how this can be queried using the Go doc server.

Part II

Building systems

CHAPTER 13

PROTOCOL BUFFERS

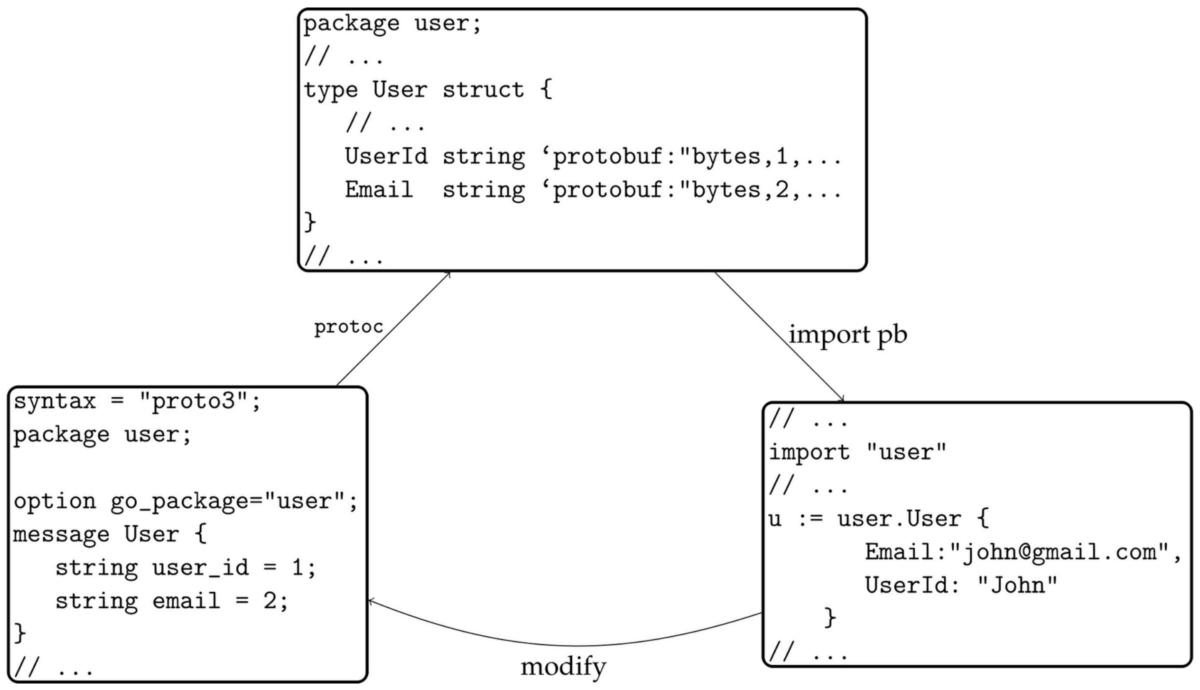
Nowadays systems are mainly distributed. The distribution comes with major advantages such as redundancy, resiliency, or scalability. However, the components of any system are not isolated and have to talk to each other. This communication is usually carried out with messages, understanding by message a piece of information exchanged between two speakers with some purpose.

Messages must be platform and language agnostic so they can be used by the largest number of solutions. Formats such as XML, JSON or YAML (see Chapter 8) are commonly used to exchange information because they are easy to parse and human friendly. However, they are not efficient solutions in terms of message size and serialization costs. To mitigate these problems, Google developed protocol buffers[30] as a language-neutral and platform-neutral serialization mechanism. Protocol buffers is available for various languages including Go.

This Chapter introduces protocol buffers, explains what they are, how they are built, and how they help us to define a common exchanging framework between applications and systems.

13.1 THE PROTO FILE

The definition of messages in protocol buffers (PB from now on) is independent of the programming language. The proto file manages the definition of exchangeable entities in PB. This file is interpreted by the `protoc` tool which generates the code required to marshal and unmarshal (see Chapter 8) these entities. A single proto file can be used to generate the corresponding code for every supported language. This is an abstraction mechanism that releases the developer from the complexities of defining low level data transfer mechanisms.



```
package user;
// ...
type User struct {
   // ...
   UserId string ‘protobuf:"bytes,1,...
   Email  string ‘protobuf:"bytes,2,...
}
// ...
```

```
syntax = "proto3";
package user;

option go_package="user";
message User {
   string user_id = 1;
   string email = 2;
}
// ...
```

```
// ...
import "user"
// ...
u := user.User {
       Email:"john@gmail.com",
       UserId: "John"
    }
// ...
```

Figure 13.1: Steps to define and use protocol buffer.

Figure 13.1 depicts the steps carried out to define and use entities defined with PB. This is represented as an iterative cycle where changes in the definition of PB entities require recompilations of the serialization mechanisms, but may not require changes in the final code using these entities. This detaches communication from the program logic.

The following examples show how to define a basic PB message entity and use it in a Go program using PB version 3. Before starting, be sure to install the PB environment following the corresponding instructions for your platform[31]. The basic tool is the `protoc` command that transforms the messages defined in the protos file into Go code.

Imagine we plan to build a system with a `User` entity. This entity will be exchanged between different applications that may not be written in the same programming language. Additionally, we expect the message definition to evolve over time. For these reasons, PB seems to be a reasonable solution. We start defining a `User` message like shown in Example 13.1.

Example 13.1: Proto file `user.proto` defining a `User` message.

```
syntax = “proto3”;
package user;

option go_package=“github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_01/user”;

message User {
    string user_id = 1;
    string email = 2;
}
```

PB permits defining messages with their fields and corresponding types. This definition is done in a file with the `.proto` suffix. The PB version is set with the `syntax` statement. PB entities are organized into packages that can be imported from other proto files. In our example, we work with package `user`. PB permits the definition of various options that may help the generation of code for certain languages. The `go_package` option indicates the path of the package where the generated Go code will be saved. This package must be accessible to be imported into our code.

Any message is composed of fields and every field has a type, a name, and a field tag. Field types resemble those used in Go `int32`, `uint32`, `string`, `bool` although there are some differences `float`, `double`, `sint64`. The equivalent type in every PB supported language is described in the official documentation[32]. The field tag indicates the position of the field after marshaling. In this case, `user_id` will be the first marshalled field followed by the `email`.

Once we have defined our message, we can create the corresponding Go implementation. To do so, we have to run the `protoc` command. For this example, `protoc –go_out=$GOPATH/src user.proto`. The `–go_out` parameter indicates where to write the generated Go files. Remember that these files must be accessible from your final code and must match the `go_package` statement. The execution output is the `user.pb.go` file. This file contains all the code that defines the type `User` and the functions to encode it following the PB protocol specification.

Finally, we can import the types defined in our `user.pb.go` file from any package. Example 13.2 creates a `User` and uses the PB code to marshal it and then, unmarshal it. Remember that once the type is marshalled, it can be sent through the network and unmarshalled by any other recipient independently of the programming language and platform.

Example 13.2: Using protos from Example 13.1 .

```
package main

import (
    “fmt”
    “github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_01/user”
    “google.golang.org/protobuf/proto”
)

func main() {
    u := user.User{Email:“john@gmail.com”, UserId: “John”}

    fmt.Println(“To encode:”, u.String())
    encoded, err := proto.Marshal(&u)
    if err != nil {
        panic(err)
    }

    v := user.User{}
    err = proto.Unmarshal(encoded, &v)
    if err != nil {
        panic(err)
    }
    fmt.Println(“Recovered:”, v.String())
}
```

```
To encode: user_id:”John” email:”john@gmail.com”

Recovered: user_id:”John” email:”john@gmail.com”
```

There are some considerations before you run this code. In Example 13.2 we import the package `google.golang.org/protobuf/proto` that contains the `Marshal` and `Unmarshal` functions. It is a third-party package and it is not available in the basic Go installation. The package can be installed using `go get` (see Section 2.1.1). However, we recommend to use `go mod` to make it more flexible. To revisit how `go mod` works visit Chapter 12. As a brief reminder, initialize the modules file with `go mod init`. Then you can build the code with `go build main.go` and the `go.mod` file will have all the required dependencies.

There is another proto package defined in `github.com/golang/protobuf`. This package is also valid and can be used. However, it has been superseded by `google.golang.org/protobuf/proto` and it is not recommended to be used.

13.2 COMPLEX MESSAGES

PB facilitates the definition of complex message structures with sophisticated types. We extend our `User` message example to demonstrate how to use other types such as enumerations and repeated fields.

Example 13.3 defines a message that represents a group of users. Every group has an id, a category, a certain score, and a list of users that belong to the group. Notice that the user category is an enumerated value. Enumerated values are defined with the reserved word `enum`. For every item of the enumeration, we need the name and the associated value. Do not confuse this value with the field tag. The `category` field in `Group` is defined as type `Category`. For the list of users, we indicate that a field can appear several times with the reserved word `repeated`. This list of users has no limit size and can be empty.

Example 13.3: Definition of complex messages.

```
syntax = “proto3”;
package user;

option go_package=“github.com/juanmanuel-
tirado/savetheworldwithgo/12_protocolbuffers/pb/example_02/user”;

message User {
    string user_id = 1;
    string email = 2;
}

enum Category {
    DEVELOPER = 0;
    OPERATOR = 1;
}

message Group {
    int32 id = 1;
    Category category = 2;
    float score = 3;
    repeated User users = 4;
}
```

The new `Group` type is used in Example 13.4. When defining enumerations, for every item PB creates a variable named with the concatenation of the enumeration and the name of the item. The enumeration of categories generates `Category_DEVELOPER` and `Category_OPERATOR` constants. The list of users is translated into a slice of `User` type. Observe that actually, this slice uses pointers to the `User` type.

Example 13.4: Utilization of messages from Example 13.3 .

```
package main

import (
    “github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_02/user”
    “google.golang.org/protobuf/proto”
    “fmt”
)

func main() {
    userA := user.User{UserId: “John”, Email: “john@gmail.com”}
    userB := user.User{UserId: “Mary”, Email:“mary@gmail.com”}

    g := user.Group{Id: 1,
        Score: 42.0,
        Category: user.Category_DEVELOPER,
        Users: []*user.User{&userA,&userB},
    }
    fmt.Println(“To encode:”, g.String())

    encoded, err := proto.Marshal(&g)
    if err != nil {
        panic(err)
    }
    recovered := user.Group{}
    err = proto.Unmarshal(encoded, &recovered)
    fmt.Println(“Recovered:”, recovered.String())
}
```

```
To encode: id:1 score:42 users:{user_id:”John” email:”john@gmail.com”} users:
{user_id:”Mary” email:”mary@gmail.com”}
Recovered: id:1 score:42 users:{user_id:”John” email:”john@gmail.com”} users:
{user_id:”Mary” email:”mary@gmail.com”}
```

13.3 IMPORTING OTHER PROTO DEFINITIONS

Protocol buffer definitions can be imported into other proto files. This facilitates the reuse of messages and enables the creation of complex solutions. Continuing with the previous example where we defined users and groups of users, we decided to separate both types into separate packages. Now there will be a user package and a group package. For this case, we have to define two separated proto files. However, the `Group` message uses the definition of a `User` message. Examples 13.5 and 13.6 show the definition of `User` and `Group` messages respectively.

Example 13.5: User proto definition.

```
syntax = “proto3”;
package user;

option go_package=“user”;

message User {
    string user_id = 1;
    string email = 2;
}
```

Example 13.6: Group proto importing Users proto.

```
syntax = “proto3”;
package group;

option go_package=“group”;

import “user.proto”;

enum Category {
    DEVELOPER = 0;
    OPERATOR = 1;
}

message Group {
    int32 id = 1;
    Category category = 2;
    float score = 3;
    repeated user.User users = 4;
}
```

The `import “user.proto”` statement makes accessible the messages defined at the `user.proto` file to `group.proto`. The `users` field in the `Group` message has to be taken from its corresponding package with `user.User`. Additionally, the target Go packages are different[33].

In order to replicate the logic from Example 13.3 now we have to import `group` and `user` after running `protoc`. Check how this is done in Example 13.6 and compare it with the previous version using only one package.

Example 13.7: Utilization of protos from Examples 13.5 and 13.6.

```
package main

import (
    “github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_03/group”
    “github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_03/user”
    “google.golang.org/protobuf/proto”
    “fmt”
)

func main() {
    userA := user.User{UserId: “John”, Email: “john@gmail.com”}
    userB := user.User{UserId: “Mary”, Email:“mary@gmail.com”}

    g := group.Group{Id: 1,
        Score: 42.0,
        Category: group.Category_DEVELOPER,
        Users: []*user.User{&userA,&userB},
    }
    fmt.Println(“To encode:”, g.String())

    encoded, err := proto.Marshal(&g)
    if err != nil {
        panic(err)
    }
    recovered := group.Group{}
    err = proto.Unmarshal(encoded, &recovered)
    fmt.Println(“Recovered:”, recovered.String())
}
```

13.4 NESTED TYPES

Certain messages may only make sense when found within other messages. PB permits the definition of nested types that are only accessible through other types. Our `Group` message can be rewritten using the `User` message as an embedded field as shown in Example 13.8 . Now the `User` type is defined in the context of `Group`. The `Winner` message represents a user who won in a given category. Observe, that the `user` field has to be referred to as `Group.User`.

Example 13.8: `Group` definition with nested `User` type.

```
syntax = “proto3”;
package group;

option go_package=“github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_04/group”;

enum Category {
    DEVELOPER = 0;
    OPERATOR = 1;
}

message Group {
    int32 id = 1;
    Category category = 2;
    float score = 3;
    message User {
        string user_id = 1;
        string email = 2;
    }
    repeated User users = 4;
}

message Winner {
    Group.User user = 1;
    Category category = 2;
}
```

PB renames these nested types in Go code by adding suffixes as shown in Example 13.9. The `User` message is renamed `Group_User`.

Example 13.9: Utilization of messages from Example 13.8 .

```
package main

import (
    “github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_04/group”
    “google.golang.org/protobuf/proto”
    “fmt”
)

func main() {
    userA := group.Group_User{UserId: “John”, Email: “john@gmail.com”}
    userB := group.Group_User{UserId: “Mary”, Email:“mary@gmail.com”}

    g := group.Group{Id: 1,
        Score: 42.0,
        Category: group.Category_DEVELOPER,
        Users: []*group.Group_User{&userA, &userB},
    }
    fmt.Println("To encode:", g.String())

    encoded, err := proto.Marshal(&g)
    if err != nil {
        panic(err)
    }
    recovered := group.Group{}
    err = proto.Unmarshal(encoded, &recovered)
    fmt.Println("Recovered:", recovered.String())
}
```

```
To encode: id:1 score:42 users:{user_id:"John" email:"john@gmail.com"} users:
{user_id:"Mary" email:"mary@gmail.com"}

Recovered: id:1 score:42 users:{user_id:"John" email:"john@gmail.com"} users:
{user_id:"Mary" email:"mary@gmail.com"}
```

13.5 TYPE ANY

Messages are allowed to have fields with no defined type. This may occur when at the time of defining a field type the content of this field is not clear yet. The type `Any` is a byte serialization of any size with a URL that works as a unique identifier for the type contained in the field.

The proto file in Example 13.10 uses `Any` to define the `info` field to allocate any available data. This type is not available by default, and it has to be imported from the `any.proto` definition.

Example 13.10: Utilization of type `Any`.

```
syntax = “proto3”;
package user;

option go_package=“github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_05/user”;

import “google/protobuf/any.proto”;

message User {
    string user_id = 1;
    string email = 2;
    repeated google.protobuf.Any info = 3;
}
```

When translated into Go (Example 13.11), the type `Any` can be initialized with any array of bytes and a string as the URL. Notice that the `anypb` package has to be imported to have the Go definition of the `Any` type.

Example 13.11: Utilization of messages from Example 13.10 .

```
package main

import (
    “fmt”
    “google.golang.org/protobuf/proto”
    “google.golang.org/protobuf/types/known/anypb”
    “github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_05/user”
)

func main() {
    info := anypb.Any{Value: []byte(‘John rules‘), TypeUrl: “urltype”}
    userA := user.User{UserId: “John”, Email: “john@gmail.com”, Info:
[]*anypb.Any{&info}}

    fmt.Println(“To encode:”, userA.String())

    encoded, err := proto.Marshal(&userA)
    if err != nil {
        panic(err)
    }
    recovered := user.User{}
    err = proto.Unmarshal(encoded, &recovered)
    fmt.Println(“Recovered:”, recovered.String())
}
```

```
To encode: user_id:”John” email:”john@gmail.com” info:{type_url:”urltype” value:”John
rules”}
Recovered: user_id:”John” email:”john@gmail.com” info:{type_url:”urltype” value:”John
rules”}
```

13.6 TYPE ONEOF

If a message with many fields can only have one field when is sent, there is no point in sending all the fields. The type `oneof` forces messages to only include one field from a given collection.

Example 13.12 extends the `User` message with a field indicating which type of user we have. A user can only be a developer or an operator. Additionally, developers have information about the language they use, while operators have information about the platform they administrate.

Example 13.12: Utilization of type `OneOf`.

```
syntax = “proto3”;
package user;

option go_package=“github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_06/user”;

message Developer {
    string language = 1;
}
message Operator {
    string platform = 1;
}

message User {
    string user_id = 1;
    string email = 2;
    oneof type {
        Developer developer = 3;
        Operator operator = 4;
    }
}
```

When using `oneof`, PB defines nested types in the `User` message. In our case, types `User_Developer` and `User_Operator` must contain a `Developer` or `Operator` type respectively. This forces the message to only contain one of those types as shown in Example 13.13.

Example 13.13: Utilization of messages from Example 13.12 .

```
package main

```

```
import (
    “fmt”
    “google.golang.org/protobuf/proto”
    “github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_06/user”
)

func main() {
    goDeveloper := user.Developer{Language: “go”}
    userA := user.User{UserId: “John”, Email: “john@gmail.com”,
        Type: &user.User_Developer{&goDeveloper}}
    aksOperator := user.Operator{Platform: “aks”}
    userB := user.User{UserId: “Mary”, Email: “mary@gmail.com”,
        Type: &user.User_Operator{&aksOperator}}

    encodedA, err := proto.Marshal(&userA)
    if err != nil {
        panic(err)
    }
    encodedB, err := proto.Marshal(&userB)
    if err != nil {
        panic(err)
    }

    recoveredA, recoveredB := user.User{}, user.User{}
    _ = proto.Unmarshal(encodedA, &recoveredA)
    _ = proto.Unmarshal(encodedB, &recoveredB)
    fmt.Println(“RecoveredA:”, recoveredA.String())
    fmt.Println(“RecoveredB:”, recoveredB.String())
}
```

```
RecoveredA: user_id:"John"  email:"john@gmail.com"  developer:{language:"go"}
RecoveredB: user_id:"Mary"  email:"mary@gmail.com"  operator:{platform:"aks"}
```

13.7 MAPS

Messages can contain maps with key/value pairs. Example 13.14 defines the `Teams` message containing a field with a map of string keys and a `UserList` message as value. Additionally, the `UserList` is a collection of users defined in the same `.proto` file.

Example 13.14: Utilization of maps.

```
syntax = "proto3";
package user;

option go_package="github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_07/user";

message User {
    string user_id = 1;
    string email = 2;
}

message UserList {
    repeated User users = 1;
}

message Teams {
    map<string, UserList> teams = 1;
}
```

A map field is treated in Go like a regular `map` type. See in Example 13.13 how a `map[string]*User.UserList` is enough to populate the corresponding `Teams` type.

Example 13.15: Utilization of messages from Example 13.14 .

```
package main

import (
    “fmt”
    “google.golang.org/protobuf/proto”
    “github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_07/user”
)

func main() {
    userA := user.User{UserId: “John”, Email: “john@gmail.com”}
    userB := user.User{UserId: “Mary”, Email: “mary@gmail.com”}

    teams := map[string]*user.UserList {
        “teamA”: &user.UserList{Users:[]*user.User{&userA,&userB}},
        “teamB”: nil,
    }

    teamsPB := user.Teams{Teams: teams}

    fmt.Println(“To encode:”, teamsPB.String())

    encoded, err := proto.Marshal(&teamsPB)
    if err != nil {
        panic(err)
    }
    recovered := user.Teams{}
    err = proto.Unmarshal(encoded, &recovered)
    if err != nil {
        panic(err)
    }
    fmt.Println("Recovered:", recovered.String())
}
```

```
To encode: teams:{key:"teamA" value:{users:{user_id:"John" email:"john@gmail.com"}
users:{user_id:"Mary" email:"mary@gmail.com"}}} teams:{key:"teamB" value:{}}

Recovered: teams:{key:"teamA" value:{users:{user_id:"John" email:"john@gmail.com"}
users:{user_id:"Mary" email:"mary@gmail.com"}}} teams:{key:"teamB" value:{}}
```

13.8 JSON

PB encodings are JSON compatible extending the number of systems that can use a `.proto` file. If you check any PB generated `*.pb.go` file you can see that the structs representing message types have JSON encoding tags (see Section 8.2), therefore they can be represented in a JSON format.

Example 13.16 uses the `encodings/json` package to marshal and unmarshal the message types defined in Example 13.14. The applicability of JSON encoding is straight forward and does not require additional code. For a low-level detail explanation of JSON mappings in PB check the official documentation[34].

Example 13.16: Encoding PB messages from Example 13.14 into JSON.

```
package main
```

```go

import (
    “encoding/json”
    “fmt”
    “github.com/juanmanuel-tirado/savetheworldwithgo/12_protocolbuffers/pb/example_08/user”
)

func main() {
     userA := user.User{UserId: “John”, Email: “john@gmail.com”}
     userB := user.User{UserId: “Mary”, Email: “mary@gmail.com”}

     teams := map[string]*user.UserList {
         “teamA”: &user.UserList{Users:[]*user.User{&userA,&userB}},
         “teamB”: nil,
     }

     teamsPB := user.Teams{Teams: teams}

     encoded, err := json.Marshal(&teamsPB)
     if err != nil {
         panic(err)
     }
     recovered := user.Teams{}
     err = json.Unmarshal(encoded, &recovered)
     if err != nil {
         panic(err)
     }
     fmt.Println(“Recovered:”, recovered.String())
}
```

```
Recovered: teams:{key:"teamA" value:{users:{user_id:"John" email:"john@gmail.com"}
users:{user_id:"Mary" email:"mary@gmail.com"}}} teams:{key:"teamB" value:{}}
```

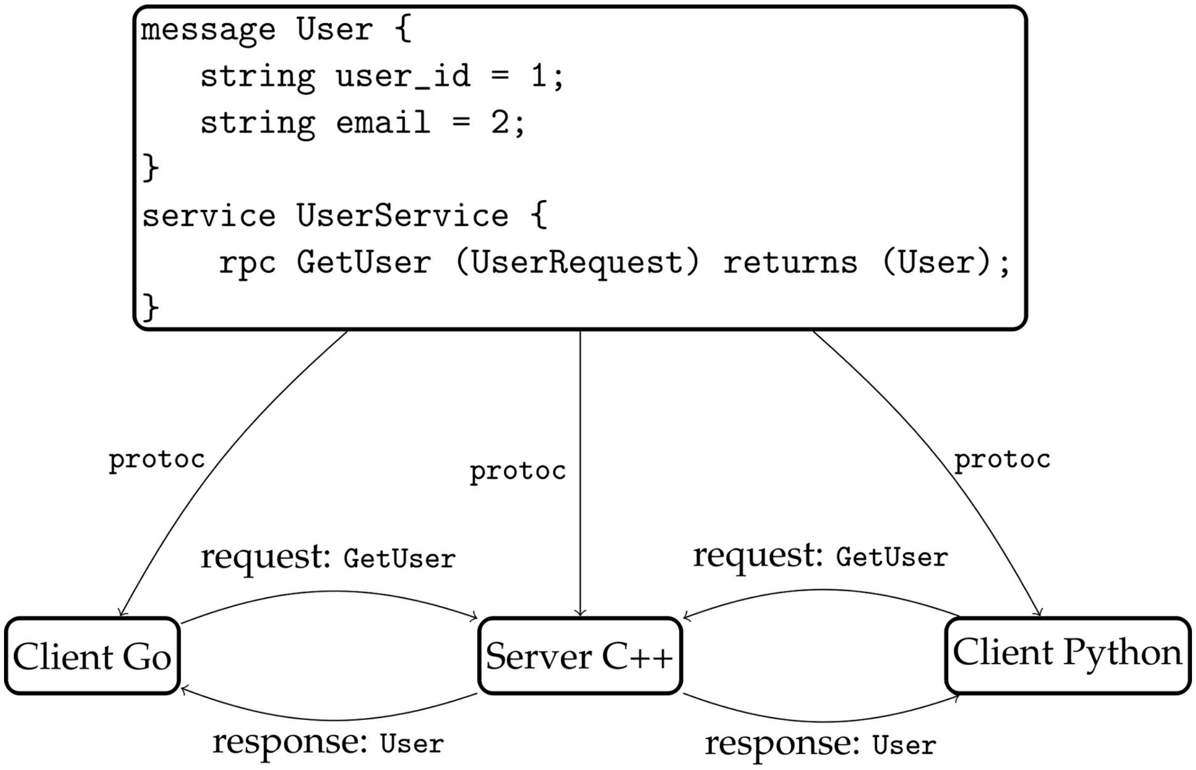
13.9 SUMMARY

This Chapter summarizes how to use protocol buffers to define Go serializable types that can be exchanged among different solutions independently of the platform and language. The examples from this Chapter cover the definition of protocol buffer messages, the syntax, and how to use these messages in Go. An understanding of these concepts is required before exploring Chapter 14.

CHAPTER 14

GRPC

gRPC[35] is a language and platform agnostic remote procedure call (RPC) solution. With gRPC, we can write a common definition of services and their signatures and then, create the corresponding clients and servers. This abstraction reduces development overhead and helps to maintain a healthy and evolving ecosystem of APIs available for several platforms. Figure 14.1 shows the components participating in a gRPC deployment. RPC works in conjunction with protocol buffers (see Chapter 13) to define messages and services. Defined messages can be consumed and returned by services and using `protoc` we can generate gRPC stubs in charge of ensuring the correctness of communications (data serialization, errors, calls signature) where we only have to write down the logic of these functions.



```
message User {
   string user_id = 1;
   string email = 2;
}
service UserService {
    rpc GetUser (UserRequest) returns (User);
}
```

Figure 14.1: Summary of elements in a gRPC deployment.

This Chapter introduces gRPC and how it can be used with Go explaining how to define services, and how to operate with clients and servers in the different communication solutions provided by gRPC.

14.1 DEFINITION OF SERVICES

A Remote Procedure Call (RPC) can be seen as an extension of the available functions we can invoke in a local machine. These functions are defined by an IDL (Interface Definition Language) that is used to generate stubs following the instructions from the IDL. In this case, we define messages and services in a `.proto` file as we did in Chapter 13 for protocol buffers.

In gRPC, a `service` is composed of one or several RPCs. We can think of services as a mechanism to group RPCs that operate with similar entities. For example, a user management service can have several operations: get a user, register a new user, remove a user, etc. Example 14.1 shows the `.proto` file for a service to manage users. The service only has one RPC called `GetUser` which receives a `UserRequest` and returns a `User`. Notice that we have only included one RPC call to the `UserService` for the sake of simplification. Services can have as many RPCs as required.

Example 14.1: Service definition using gRPC.

```
syntax = “proto3”;
package user;

option go_package=”github.com/juanmanuel-
tirado/savetheworldwithgo/13_grpc/example_01/user”;

message User {
    string user_id = 1;
    string email = 2;
}

message UserRequest {
    string user_id = 1;
}

service UserService {
    rpc GetUser (UserRequest) returns (User);
}
```

We already have defined our messages and services. Now we have to create the corresponding stubs to use the services and messages in our Go code. For this task, we need the `protoc` command-line tool with the gRPC plugin. Be sure, you have already installed the `protoc` tool (see Section 13.1). The gRPC plugin works with the `protoc` tool to generate the corresponding clients and servers using the specifications from the `.proto` file. You can get the gRPC plugin running[36]:

```
>>> export GO11MODULE=on  # Enable module mode
>>> go get google.golang.org/protobuf/cmd/protoc-gen-go \
         google.golang.org/grpc/cmd/protoc-gen-go-grpc
```

Make sure that `protoc` can find the plugins:

```
>>> export PATH="$PATH:$(go env GOPATH)/bin"
```

If the environment is correctly set, now we can execute `protoc` and generate the gRPC stubs.

```
>>> protoc -I=. —go_out=$GOPATH/src —go-grpc_out=$GOPATH/src *.proto
```

The `—go-grpc_out` argument indicates the path for the go stubs. Remember that the generated code must be accessible to your code and match the value set in the `go_package`. After a successful execution we have two files: `user.pb.go` and `user_grpc.pb.go`. The first one contains the definition of the messages as described in Chapter 13. The second one contains all the code generated to support servers and clients that derive from our gRPC definitions in the `.proto` file.

Example 14.2: Excerpt of a gRPC stub.

```go
type UserServiceClient interface {
   GetUser(ctx context.Context, in *UserRequest, opts …grpc.CallOption) (*User, error)
}

type userServiceClient struct {
   cc grpc.ClientConnInterface
}

func NewUserServiceClient(cc grpc.ClientConnInterface) UserServiceClient {
   return &userServiceClient{cc}
}

func (c *userServiceClient) GetUser(ctx context.Context, in *UserRequest, opts …grpc.CallOption) (*User, error) {
// …
}

type UserServiceServer interface {
   GetUser(context.Context, *UserRequest) (*User, error)
   mustEmbedUnimplementedUserServiceServer()
}
```

Example 14.2 contains an excerpt of the `user_grpc.pb.go`. We can see that `protoc` has automatically generated a set of Go types that represent the specified service. This code is now ready to be used.

14.2 CREATING A SERVER

Once the gRPC stub is generated, we can import the corresponding package into our code. This code provides us with the skeleton of the Go functions we have to implement to have the server defined in our `.proto` file. Notice that right now we have only defined the signature of the remote functions, not the logic itself. This has to be done on a server. Building a server has three steps: server implementation, registering the service, and listening to incoming requests.

Example 14.3: User service server.

```
package main

import (
    “context”
    “fmt”
    pb “github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/grpc/example_01/user”
    “google.golang.org/grpc”
    “net”
)

type UserServer struct{
     pb.UnimplementedUserServiceServer
}

func (u *UserServer) GetUser(ctx context.Context, req *pb.UserRequest) (*pb.User, error) {
     fmt.Println(”Server received:”, req.String())
     return &pb.User{UserId: “John”, Email: “john@gmail.com”}, nil
}

```

```
func main() {
    lis, err := net.Listen(”tcp”, “localhost:50051”)
    if err != nil {
        panic(err)
    }
    s := grpc.NewServer()
    pb.RegisterUserServiceServer(s, &UserServer{})

    if err := s.Serve(lis); err != nil {
        panic(err)
    }
}
```

1. Example 14.3 is a minimalist server implementation of `UserService`. The `UserServer` type must implement the `UserServiceServer` interface defined in the stub. You can find the interface definition in Example 14.2. Every RPC defined in the `.proto` file inside the `UserService` must have its corresponding method implemented by the server. Only if all the methods are implemented, the server type will be a valid server.

   RPC functions have the same signature:

   ```
   func RPCName(ctx context.Context, req *Request) (*Response, error)
   ```

   We have a context for the request (see Section 6.6) and an incoming request if any. The method returns a response if any, and an error. In our example, we simply print the incoming request and return a manually populated `User` type.

2. The complexity of a server varies depending on the task and the number of RPCs to be implemented. However, once the server is defined we have to expose it in such a way that requests can be correctly served by our code. The gRPC stub generates the function `RegisterUserServiceServer` that links a type implementing the `UserServiceServer` with any incoming

request to this service. This must be done with a gRPC server type (lines 25–26).
3. Finally, we can run the GRPC server in an endless loop waiting for incoming requests. The `s.Serve` method blocks the execution until the program is stopped or `panic` occurs.

The server code can be compared with the implementation of HTTP functions seen in Chapter 9. Notice that this code requires third-party components that may not be available on your platform. You can use `go mod` to ensure you have the required code (See Chapter 12).

14.3 CREATING CLIENTS

Creating a gRPC client requires fewer steps than running a server. The only thing we have to define is the address of the target server as shown in Example 14.4. The `UserServiceClient` type provides methods to call the RPCs defined in the `.proto`. A `UserServiceClient` instance can be obtained with the `NewUserServiceClient` and a connection to the server. We can create a connection with the `grpc.Dial` function. This function receives the server address and none or several `DialOption` values. The `DialOption` type setups the connection with the server. In the example, we use `WithInsecure` to indicate that there is no encryption and `WithBlock` to block the execution flow until the connection with the server is up. Further details about how to use `DialOption` are given in Section 14.6 to explain interceptors.

Example 14.4: User service client.

```
package main

import (
    “context”
    “fmt”
    pb “github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/grpc/example_01/user”
    “google.golang.org/grpc”
    “time”
)

 func main() {
     conn, err := grpc.Dial(”localhost:50051”, grpc.WithInsecure(), grpc.WithBlock())
     if err != nil {
         panic(err)
     }
     defer conn.Close()

     c := pb.NewUserServiceClient(conn)

     ctx, cancel := context.WithTimeout(context.Background(), time.Second)
     defer cancel()

     r, err := c.GetUser(ctx, &pb.UserRequest{UserId: “John”})
     if err != nil {
         panic(err)
     }
     fmt.Println(”Client received:”, r.String())
 }
```

Running the server and then the client code you should get the following outputs.

```
Client received: user_id:”John”  email:”john@gmail.com”
```

```
Server received: user_id:"John"
```

14.4 STREAMING

HTTP/2 permits full-duplex communication between client and server. This makes it possible to establish a streaming channel between both ends. The channel can be reused reducing the overhead and latency of involved networking operations in the channel negotiation process. gRPC leverages this HTTP/2 feature to offer procedures that receive or return data inside streams. Next, we explain how to define procedures with streaming for servers and clients in one direction or bidirectional flavours.

14.4.1 Server streaming

Consider the `.proto` file from Example 14.5 . The `Rnd` remote procedure receives a request to receive `n` random numbers in the between `from` and `to` values. These numbers are not returned in a single batch, instead they are returned using a stream. This is particularly useful in scenarios where `n` is very large.

Example 14.5: Definition of a server streaming procedure.

```
syntax = "proto3";
package numbers;

option go_package="github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/streaming/example_01/numbers";

message NumRequest {
    int64 from = 1;
    int64 to = 2;
    int64 n = 3;
}
```

```

message NumResponse {
    int64 i = 1;
    int64 remaining = 2;
}

service NumService {
    rpc Rnd (NumRequest) returns (stream NumResponse);
}
```

The data flow for this scenario is depicted in Figure 14.2. The client starts the communication with a request (`NumRequest` in this example). Afterwards, the server sends responses to the client using the same channel until the server closes the channel sending an `EOF`.

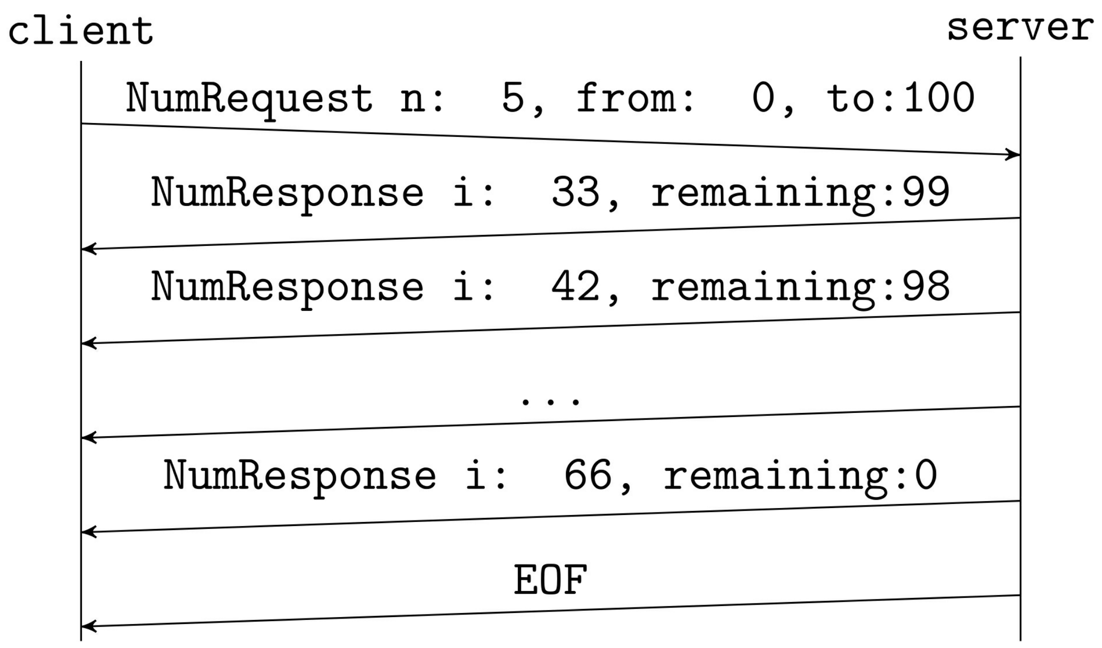

client
server
NumRequest n: 5, from: 0, to:100
NumResponse i: 33, remaining:99
NumResponse i: 42, remaining:98
...
NumResponse i: 66, remaining:0
EOF

Figure 14.2: Client-server communication in a server streaming scenario.

The server implementation in Example 14.6 generates `n` random numbers between the given range. Notice that the signature function to be implemented differs from unary RPC functions. The server implements the remote procedure `Rnd` which receives a request and a type that encapsulates a stream type. This stream type is the communication channel the client opened when invoking this procedure. Our server uses the method `stream.Send` to send new responses to the client. When we are finished, returning `nil` is enough to close the channel and send and `EOF` informing the client that the communication is finished.

Example 14.6: Implementation of streaming on the server side.

```
package main

import (
    “errors”
    “fmt”
    pb “github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/streaming/example_01/numbers”
    “google.golang.org/grpc”
    “math/rand”
    “net”
    “time”
)

type NumServer struct{
    pb.UnimplementedNumServiceServer
}

func (n *NumServer) Rnd(req *pb.NumRequest, stream pb.NumService_RndServer) error {
    fmt.Println(req.String())
    if req.N <= 0 {
        return errors.New(”N must be greater than zero”)
    }
    if req.To <= req.From {
        return errors.New(”to must be greater or equal than from”)
    }
    done := make(chan bool)
    go func() {
        for counter:=0;counter<int(req.N);counter++{
            i := rand.Intn(int(req.To) - int(req.From) +1) + int(req.To)
            resp := pb.NumResponse{I:int64(i), Remaining:req.N-int64(counter)}
            stream.Send(&resp)
            time.Sleep(time.Second)
        }
        done <- true
    }()
    <- done
    return nil
}

func main() {
    lis, err := net.Listen(”tcp”, “localhost:50051”)
    if err != nil {
        panic(err)
    }
    s := grpc.NewServer()

    pb.RegisterNumServiceServer(s, &NumServer{})

    if err := s.Serve(lis); err != nil {
        panic(err)
    }
}
```

On the other side, the client listens to new incoming information from the server after sending the initial request. When invoking the remote procedure from the client-side, it returns a streaming client with a `Recv` function. Example 14.7 shows how to consume data from this function until an `EOF` is found. An important aspect to be considered is the utilization of contexts. The context declared when invoking the function (`ctx` in line 21) lives during all the stream. This means that in the case of using a context with timelines if this context expires the stream will be closed. For the current example, the channel will be closed after 10 seconds.

Example 14.7: Client consuming server streaming responses.

```
1 package main
2
3 import (
4     “context”
5     “fmt”
6     pb “github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/streaming/example_01/numbers”
7     “google.golang.org/grpc”
8     “io”
9     “time”
10 )
11
12 func main() {
13     conn, err := grpc.Dial(”:50051”, grpc.WithInsecure(), grpc.WithBlock())
14     if err != nil {
15         panic(err)
```

```go
}
defer conn.Close()

c := pb.NewNumServiceClient(conn)

ctx, cancel := context.WithTimeout(context.Background(), time.Second*10)
defer cancel()

stream, err := c.Rnd(ctx, &pb.NumRequest{N:5, From:0, To: 100})
if err != nil {
    panic(err)
}

done := make(chan bool)
go func() {
    for {
        resp, err := stream.Recv()
        if err == io.EOF {
            done <- true
            return
        }
        if err != nil {
            panic(err)
        }
        fmt.Println(”Received:”, resp.String())
    }
}()
<- done
```

```
    fmt.Println("Client done")
}
```

The asynchronous nature of these operations is a perfect use-case for timeouts and goroutines. If you do not feel familiar with these concepts, please revisit Chapter 6.

```
Received: i:165  remaining:5
Received: i:182  remaining:4
Received: i:129  remaining:3
Received: i:187  remaining:2
Received: i:148  remaining:1
Client done
```

```
to:100 n:5
```

14.4.2 Client streaming

In a client streaming scenario, the client directly opens a data stream with the server. When the client decides to stop sending data, it can close the channel or close the channel and wait for a response from the server. What regards the server, it receives all the data from the client until an `EOF` is found. Afterwards, depending on the procedure it can send a response back or not.

Example 14.8 defines the `Sum` procedure that receives a stream of requests. This procedure will compute the sum of all the numbers sent through the stream and return the result to the client. To define a streaming input, we have to add the `stream` modifier to the incoming message.

Example 14.8: Definition of a client streaming procedure.

```
syntax = "proto3";
package numbers;
```

```

option go_package=”github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/streaming/example_02/numbers”;

message NumRequest {
    int64 x = 1;
}

message NumResponse {
    int64 total = 1;
}

service NumService {
    rpc Sum (stream NumRequest) returns (NumResponse);
}
```

The expected client-server communication is depicted in Figure 14.3. Observe that the client directly opens a stream sending requests and the server does not respond until the `EOF` is sent.

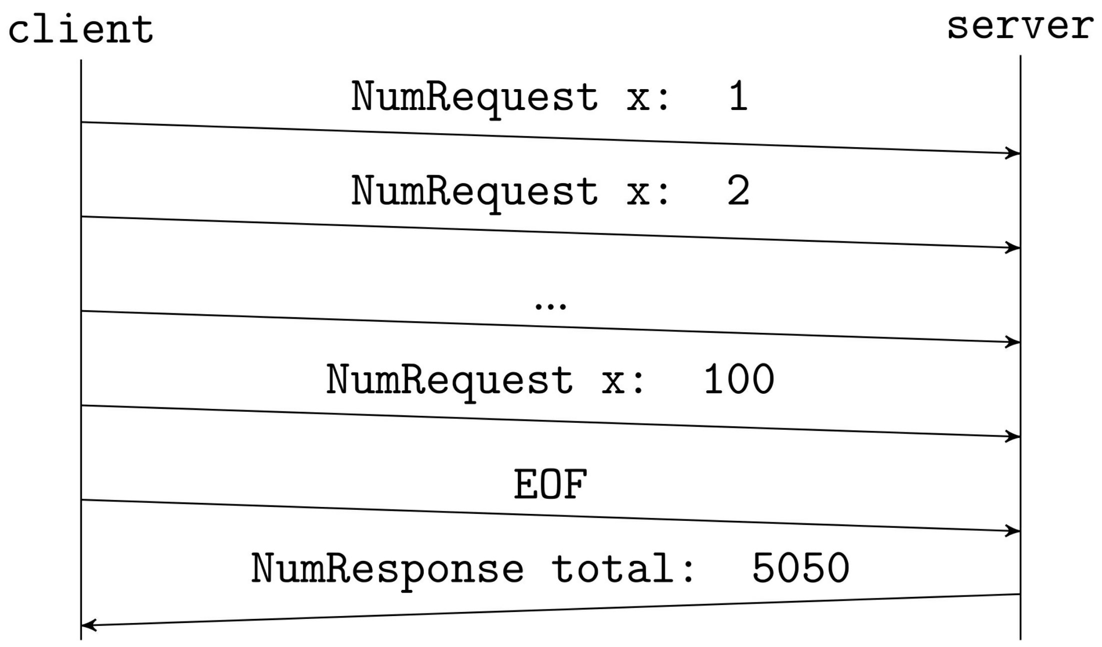


Figure 14.3: Client-server communication in a client streaming scenario.

On the client-side (Example 14.9) the `Sum` function does not receive any request, only a context. This function returns a streaming client that can be used to send requests to the server once the connection is established. If the stream has to be finished, then there are two options. We can close the stream if we do not expect any answer with `Close`, or we can close the stream and wait for a server response with `CloseAndRecv`. In both cases, the server is informed that the client has closed the channel. In our example, we close and wait for the server response containing the result with the sum of all the numbers it has received.

Example 14.9: Client sending data in streaming.

```
package main

import (
    “context”
    “fmt”
    pb “github.com/juanmanuel-
```

```
tirado/savetheworldwithgo/13_grpc/streaming/example_02/numbers”
    “google.golang.org/grpc”
    “time”
)

func main() {
    conn, err := grpc.Dial(”:50051”, grpc.WithInsecure(), grpc.WithBlock())
    if err != nil {
        panic(err)
    }
    defer conn.Close()

    c := pb.NewNumServiceClient(conn)

    ctx, cancel := context.WithTimeout(context.Background(), time.Second*10)
    defer cancel()

    stream, err := c.Sum(ctx)
    if err != nil {
        panic(err)
    }

    from, to := 1,100

    for i:=from;i<=to;i++ {
        err = stream.Send(&pb.NumRequest{X:int64(i)})
        if err!= nil {
            panic(err)
        }
    }
    fmt.Println("Waiting for response…")
    result, err := stream.CloseAndRecv()
    if err != nil {
        panic(err)
    }
    fmt.Printf("The sum from %d to %d is %d\n", from,to, result.Total)
}
```

The server implementation from Example 14.10 is mostly contained in an endless `for` loop. When the `Sum` method is invoked in the server, this can check the input stream until the `EOF` or an error occurs. If no error or `EOF` is found, the server accumulates the incoming values (line 28) otherwise, the server invokes `SendAndClose` sending the accumulated value to the client.

Example 14.10: Server processing client stream and responding.

```
1 package main
2
3 import (
4     "fmt"
5     pb "github.com/juanmanuel-
tirado/savetheworldwithgo/13_grpc/streaming/example_02/numbers"
6     "google.golang.org/grpc"
7     "io"
8     "net"
9 )
10
11 type NumServer struct{
12     pb.UnimplementedNumServiceServer
```

```go
}

func (n *NumServer) Sum(stream pb.NumService_SumServer) error {
    var total int64 = 0
    var counter int = 0
    for {
       next, err := stream.Recv()
       if err == io.EOF {
           fmt.Printf(”Received %d numbers sum: %d\n”,counter,total)
           stream.SendAndClose(&pb.NumResponse{Total: total})
           return nil
       }
       if err != nil {
           return err
       }
       total = total + next.X
       counter++
    }

    return nil
}

func main() {
    lis, err := net.Listen(”tcp”, “localhost:50051”)
    if err != nil {
        panic(err)
    }
    s := grpc.NewServer()

    pb.RegisterNumServiceServer(s, &NumServer{})

    if err := s.Serve(lis); err != nil {
        panic(err)
    }
}
```

```
Waiting for response…
The sum from 1 to 100 is 5050
```

```
Received 100 numbers sum: 5050
```

14.4.3 Bidirectional streaming

We have already mentioned that gRPC provides full-duplex communication using HTTP/2. This makes it possible to use bidirectional streaming between client and server. Bidirectional streaming may be particularly suitable for asynchronous scenarios or even to define your protocols on top of gRPC.

In Example 14.11, we define a chat service that sends back to the user her data consumption stats. In a chat service, client and server are expected to be completely asynchronous. The client can send text messages to the server at any moment. On the other side, the server periodically sends stats to the user with some periodicity that may vary. Observe that the remote procedure `SendTxt` sets the argument and response to be streamings.

Example 14.11: Definition of a bidirectional streaming procedure.

```
syntax = “proto3”;
package numbers;

option go_package=”github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/streaming/example_03/chat”;

message ChatRequest {
    int64 id = 1;
    int64 to = 2;
    string txt = 3;
}

message StatsResponse {
    int64 total_char = 1;
}

```

```
service ChatService {
    rpc SendTxt (stream ChatRequest) returns (stream StatsResponse);
}
```

Figure 14.4 depicts this communication scenario. The client starts the communication by establishing a new stream with the server. Both ends listen to the stream for any incoming message. Finally, both the client or server can close the connection. In our case, it is the client who finishes the communication.

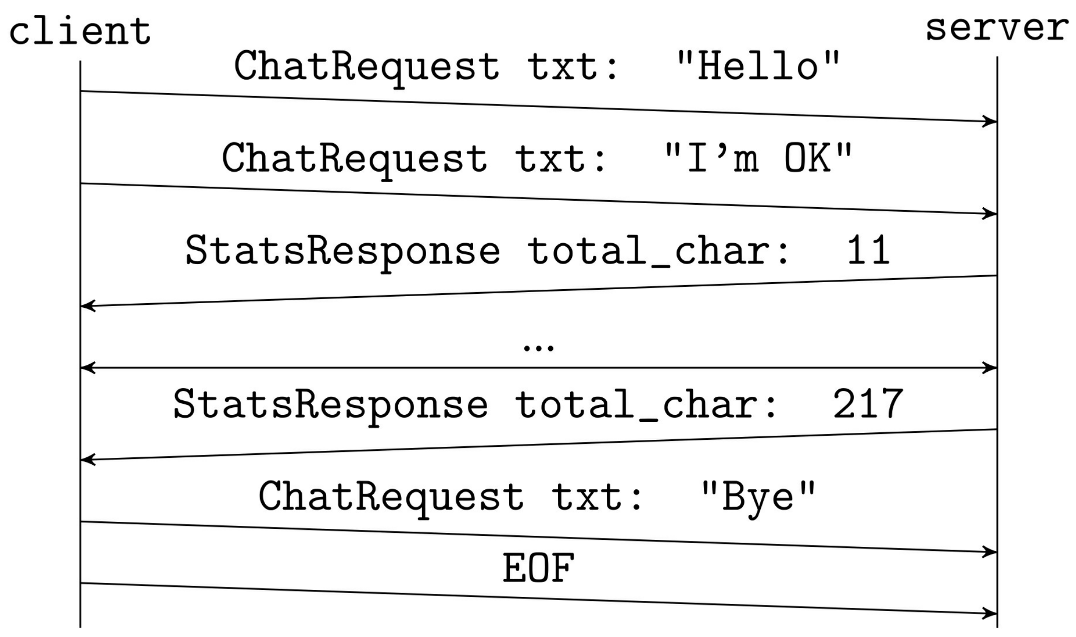

client
server
ChatRequest txt: "Hello"
ChatRequest txt: "I'm OK"
StatsResponse total_char: 11
...
StatsResponse total_char: 217
ChatRequest txt: "Bye"
EOF

Figure 14.4: Client-server communication in a bidirectional streaming scenario.

The client implementation from Example 14.12 sends a chat message until the server stats informs that the number of sent characters is greater than a limit. Because the two actions are asynchronous we use two goroutines: one to send chat messages and the other to monitor chat stats. The `ChatService_SendTxtClient` has methods for both sending and receiving data. The `Stats` function receives stats from the server and closes the stream (line 36).

Example 14.12: Client using bidirectional streaming.

```
package main

import (
    “context”
    “fmt”
    pb “github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/streaming/example_03/chat”
    “google.golang.org/grpc”
    “time”
)

func Chat(stream pb.ChatService_SendTxtClient, done chan bool) {
    t := time.NewTicker(time.Millisecond*500)
    for {
        select {
        case <- done:
            return
        case <- t.C:
            err := stream.Send(&pb.ChatRequest{Txt:”Hello”, Id:1, To:2})
            if err != nil {
                panic(err)
            }
        }
    }
}

func Stats(stream pb.ChatService_SendTxtClient, done chan bool) {
    for {
        stats, err := stream.Recv()
```

```go
        if err != nil {
            panic(err)
        }
        fmt.Println(stats.String())
        if stats.TotalChar > 35 {
            fmt.Println(”Beyond the limit!!!”)
            done <- true
            stream.CloseSend()
            return
        }
    }
}

func main() {
    conn, err := grpc.Dial(”:50051”, grpc.WithInsecure(), grpc.WithBlock())
    if err != nil {
        panic(err)
    }
    defer conn.Close()

    c := pb.NewChatServiceClient(conn)

    stream, err := c.SendTxt(context.Background())
    if err != nil {
        panic(err)
    }
    done := make(chan bool)
    go Stats(stream, done)
    go Chat(stream, done)

    <- done
}
```

The server implementation from Example 14.13 listens for incoming chat messages and count the number of received characters. Additionally, every two seconds it sends a message with the current characters counter value. Notice that sending stats is completely independent of the incoming messages and occurs at a regular pace. The server expects the client to close the connection by checking any incoming EOF.

Example 14.13: Server using bidirectional streaming.

```
package main

import (
   “fmt”
   pb “github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/streaming/example_03/chat”
   “google.golang.org/grpc”
   “io”
   “net”
   “time”
)

type ChatServer struct{
    pb.UnimplementedChatServiceServer
}

func (c *ChatServer) SendTxt(stream pb.ChatService_SendTxtServer) error {
```

```go
    var total int64 = 0
    go func(){
       for {
           t := time.NewTicker(time.Second*2)
           select {
           case <- t.C:
               stream.Send(&pb.StatsResponse{TotalChar: total})
           }
       }
    }()
    for {
       next, err := stream.Recv()
       if err == io.EOF {
           fmt.Println(”Client closed”)
           return nil
       }
       if err != nil {
           return err
       }
       fmt.Println(”->”, next.Txt)
       total = total + int64(len(next.Txt))
    }

    return nil
}

func main() {
    lis, err := net.Listen(”tcp”, “localhost:50051”)
    if err != nil {
        panic(err)
    }
    s := grpc.NewServer()

    pb.RegisterChatServiceServer(s, &ChatServer{})

    if err := s.Serve(lis); err != nil {
        panic(err)
    }
}
```

The output from tuning the server and client reveals how the client closes the connection when it exceeds its limit of characters and how the server realizes that.

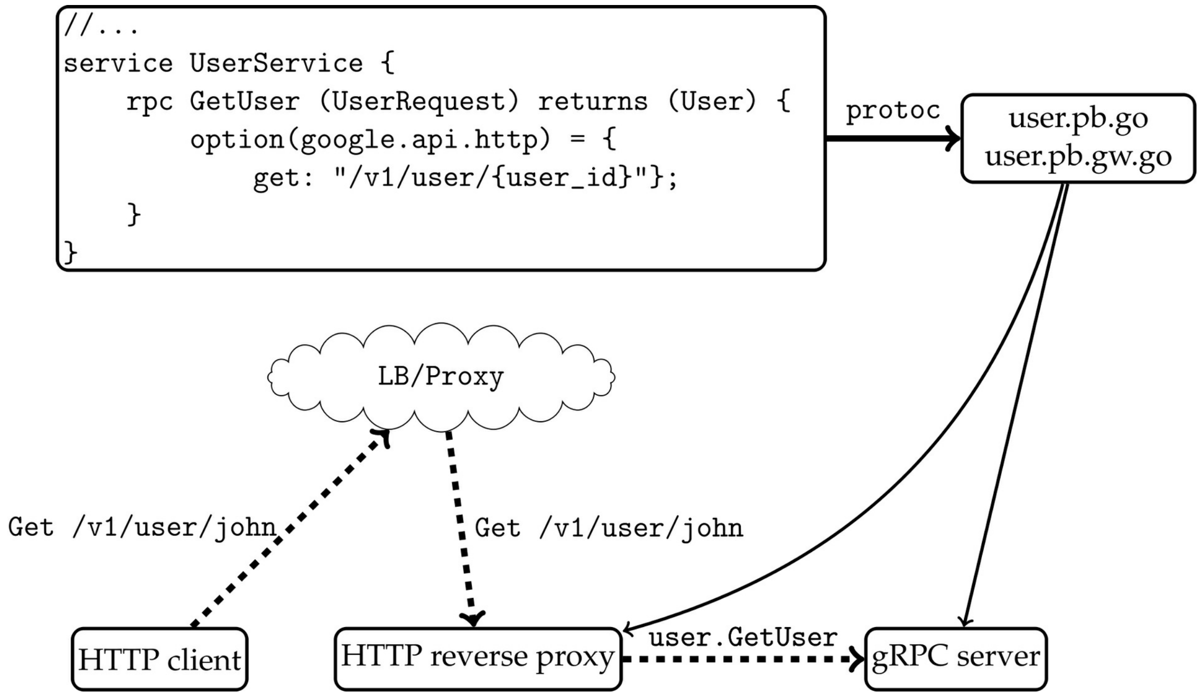
```
total_char:15
total_char:40
Beyond the limit!!!
```

```
-> Hello
-> Hello
-> Hello
-> Hello
-> Hello
-> Hello
-> Hello
-> Hello
Client closed
```

14.5 TRANSCODING

gRPC works on top of the HTTP/2 transfer protocol. This protocol enhances gRPC transfer performance and permits full-duplex communication among other features supported by HTTP2. Unfortunately, HTTP2 is not available in every scenario and sometimes can become a drawback. Many servers are behind load balancers and/or proxies that do not support HTTP2. In other scenarios, remote clients cannot work with gRPC due to other limitations simply because they cannot easily embrace this technology.

Transcoding permits gRPC to emulate REST APIs (HTTP + JSON). This can be done using annotations in the `.proto` file. Using the grpc-gateway[37] plugin, the `protoc` can generate reverse proxies transcoding HTTP requests to the corresponding gRPC handler. Figure 14.5 shows a schema of HTTP transcoding to gRPC where a HTTP client sends a `GET` request to a remote server. This request travels through the Internet and may traverse different load balancers or service providers. Finally, the request reaches the target HTTP server. The incoming request is transformed into a gRPC compatible request and sent to the corresponding gRPC server.



```
//...
service UserService {
    rpc GetUser (UserRequest) returns (User) {
        option(google.api.http) = {
            get: "/v1/user/{user_id}"};
    }
}
```

Figure 14.5: Summary of elements in a gRPC deployment.

This transformation is obtained adding the corresponding `google.api.http` notations to the `.proto` file. Observe in the figure that the output from `protoc` includes two files like in a regular gRPC scenario, but `user_grpc.pb.go` has been replaced by `usr.pb.gw.go`. This file contains the definition of the necessary transcoding operations. Transcoding is a powerful option to enable accessing remote procedures to a vast number of clients with minimal coding.

14.5.1 Generate stubs and gRPC-gateway

Transcoding is defined in the `.proto` file using annotations. Example 14.14 , defines a user service that creates and gets users. There are two important elements we have added for transcoding. First, we have imported the annotations.proto file that defines the `google.api.http` option.

Example 14.14: gRPC transcoding using notations.

```
syntax = “proto3”;
package user;

option go_package=”github.com/juanmanuel-tirado/savetheworldwithgo/13_grpc/transcoding/example_01/user”;

import “google/api/annotations.proto”;

message User {
    string user_id = 1;
    string email = 2;
}

message UserRequest {
    string user_id = 1;
}

service UserService {
    rpc Get (UserRequest) returns (User) {
        option(google.api.http) = {
            get: “/v1/user/{user_id}”
        };
    }
    rpc Create (User) returns (User) {
        option(google.api.http) = {
            post: “/v1/user”
            body: “*”
        };
    }
}
```

For every RPC function we want to expose through HTTP we must define the path and the required HTTP method. The remote procedure `Get` will be served at `/v1/user/ {user_id}`, where `{user_id}` corresponds to the field with the same name from the corresponding input message. In this case, this will fill the unique field from the `UserRequest` message. Similarly, the `Create` remote function is a HTTP `POST` method with a `User` message to be sent into the body of the HTTP request. By setting `body: “*”`, we indicate that a `User` message will be filled with all the available data from the body. If required, single fields can be specified, for example, `body: “email”` will only expect the body to have the email field.

Remember that the URL of a remote procedure should not be changed unilaterally once this function is consumed by others. There are some good practices to be followed like indicating the API version in the URL. In other cases, you can define additional bindings for the same procedure.

```
rpc Get (UserRequest) returns (User) {
    option(google.api.http) = {
        get: "/v1/user/{user_id}"
        additional_bindings {
            get: "/v2/user/{user_id}"
        }
    };
}
```

Before running `protoc`, we must install the grpc-gateway plugin[38].

```
>>> go install \
github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-grpc-gateway \
github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-openapiv2 \
google.golang.org/protobuf/cmd/protoc-gen-go \
google.golang.org/grpc/cmd/protoc-gen-go-grpc
```

Now `protoc` should be able to use the additional plugins by running:

```
>>> protoc -I . \
-I $GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/third_party/googleapis \
–go_out=plugins=grpc:$GOPATH/src \
–grpc-gateway_out=logtostderr=true:$GOPATH/src *.proto
```

This execution assumes you are in the same folder as the `.proto` file. Some arguments may vary depending on your environment.

14.5.2 Create an HTTP server

To demonstrate how transcoding works, we are going to implement the remote procedures we have defined in the previous `.protoc` file and serve them using gRPC as we have already explained, then we show how this can be done using an HTTP server. For clarity, we split the code into two parts. First, in Example 14.15 contains the code with the logic of the remote procedures defined in our `.proto` file. This code is similar to the examples we have already explained. The only method you may find weird is `ServeGrpc`, which simply encapsulates the gRPC server starting logic.

Example 14.15: gRPC gateway using HTTP (part I).

```
package main

import (
    “context”
    “fmt”
    “github.com/grpc-ecosystem/grpc-gateway/v2/runtime”
    pb “github.com/juanmanuel-
tirado/savetheworldwithgo/13_grpc/transcoding/example_01/user”
    “google.golang.org/grpc”
    “net”
     “net/http”
)

type UserServer struct{
     httpAddr string
     grpcAddr string
     pb.UnimplementedUserServiceServer
}

func (u *UserServer) Get(ctx context.Context, req *pb.UserRequest) (*pb.User,
error) {
     fmt.Println(”Server received:”, req.String())
     return &pb.User{UserId: “John”, Email: “john@gmail.com”}, nil
}

func (u *UserServer) Create(ctx context.Context, req *pb.User) (*pb.User, error) {
     fmt.Println(”Server received:”, req.String())
     return &pb.User{UserId: req.UserId, Email: req.Email}, nil
}

func (u *UserServer) ServeGrpc() {
     lis, err := net.Listen(”tcp”, u.grpcAddr)
     if err != nil {
         panic(err)
     }
     s := grpc.NewServer()
     pb.RegisterUserServiceServer(s, u)
     fmt.Println(”Server listening GRCP:”)

     if err := s.Serve(lis); err != nil {
         panic(err)
     }
}
```

When all the elements needed to run our gRPC server are ready, we can include our HTTP transcoding. Example 14.16 contains the code to serve the remote procedures defined in the previous example using a HTTP server. We have to register every service XXX from the `.proto` file using the corresponding `RegisterXXXHandlerFromEndPoint` function. This function binds the gRPC server address to a `Mux` server[39]. For simplicity, the `UserServer` type has two addresses one for gRPC and the other for HTTP. Remember that the

HTTP server is a gateway that forwards incoming requests to our gRPC server.

Example 14.16: gRPC gateway using HTTP (part II).

```
func (u *UserServer) ServeHttp() {
    mux := runtime.NewServeMux()
    opts := []grpc.DialOption{grpc.WithInsecure()}
    endpoint := u.grpcAddr

    err := pb.RegisterUserServiceHandlerFromEndpoint(context.Background(), mux, endpoint, opts)
    if err != nil {
        panic(err)
    }

    httpServer := &http.Server{
        Addr: u.httpAddr,
        Handler: mux,
    }

    fmt.Println(”Server listing HTTP:”)
    if err = httpServer.ListenAndServe(); err!=nil{
        panic(err)
    }
}

func main() {
    us := UserServer{httpAddr:”:8080”,grpcAddr:”:50051”}
    go us.ServeGrpc()
    us.ServeHttp()
}
```

Finally, in the `main` function we start our gRPC and HTTP servers. Now gRPC clients can invoke remote procedures in port 5051 and HTTP clients can use port 8080. Now we can send HTTP requests to the indicated URLs using an HTTP client like `curl`.

```
>>> curl  http://localhost:8080/v1/user/john
{”userId”:”John”,”email”:”john@gmail.com”}
>>> curl  -d ’{“user_id“:”john“,”email“:”john@gmail“}’ http://localhost:8080/v1/user
{”userId”:”john”,”email”:”john@gmail”}
```

As you can observe, the addition of the gRPC gateway has not impacted the logic of our code. We have added a small piece of code and now our server makes available the same API using gRPC and HTTP/JSON.

14.6 INTERCEPTORS

Interceptors is the name for middleware components in gRPC. Operations such as user authentication, message tracing, request validation, etc. can be done with interceptors. Interceptors are allocated between the client and the server data flow like in Figure 14.6. When an interceptor is introduced into the data flow, this is executed for every message and has access to all the communication elements. An interceptor can manipulate message metadata, for example adding authentication headers on the client-side, or annotating additional information about IP addresses, client time zone, etc.

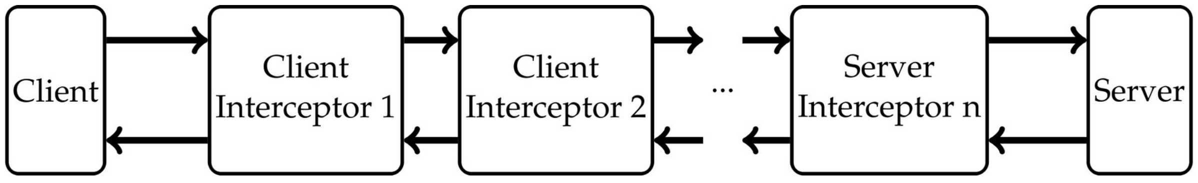


Figure 14.6: Data flow from client to server using interceptors.

Interceptors are thought to be used for common tasks and can be reutilized in several solutions. The gRPC-ecosystem has an available collection of interceptors to be used[40]. However, the exact interceptor required for your problem may not be available. Fortunately, gRPC facilitates the implementation of four types of interceptors depending if they are on the client or server-side and if they are designed for unary or streaming requests. In this section, we cover the server and client interceptors for unary requests. Streaming versions of these interceptors are very similar and can be easily implemented starting from unary request versions.

14.6.1 Server interceptors

gRPC servers are instantiated with the `grpc.NewServer` function that receives arguments of type `ServerOption`. We can build types that implement the `ServerOption` type to work as interceptors. The following lines sketch the elements required to build a server interceptor.

```
func MyServerInterceptor(…) (interface{}, error){

// …

func withMyServerInterceptor() grpc.ServerOption{

    return grpc.UnaryInterceptor(MyServerInterceptor)

}

// …

s := grpc.NewServer(withMyServerInterceptor())
```

To build a `ServerOption` we invoke the `grpc.UnaryInterceptor` function which expects a `UnaryServerInterceptor` type. This type is a function that must be implemented by our interceptor. The function signature is:

```
type UnaryServerInterceptor func(ctx context.Context, req interface{},

    info *UnaryServerInfo, handler UnaryHandler) (resp interface{}, err error)
```

where,

- `ctx` is the context.

- `req` contains the actual request.
- `info` contains server information and the URI of the invoked method.
- `handler` is the handler in charge of the request.

To illustrate how to implement a server interceptor, assume the initial implementation from Example 14.3 . We want to enhance this server to only permit authenticated users. Example 14.17 is an excerpt of the server implementation containing an interceptor that provides a naive authentication method.

Example 14.17: Server interceptor (excerpt).

```
func AuthServerInterceptor(
    ctx context.Context,
    req interface{},
    info *grpc.UnaryServerInfo,
    handler grpc.UnaryHandler) (interface{}, error) {
    md, found := metadata.FromIncomingContext(ctx)
    if !found {
        return nil, status.Errorf(codes.InvalidArgument, “metadata not found”)
    }
    password, found := md[”password”]
    if !found {
        return nil, status.Errorf(codes.Unauthenticated, “password not found”)
    }

    if password[0] != “go” {
        return nil, status.Errorf(codes.Unauthenticated, “password not valid”)
    }

    h, err := handler(ctx, req)
    return h, err
}

func withAuthServerInterceptor() grpc.ServerOption {
    return grpc.UnaryInterceptor(AuthServerInterceptor)
}

func main() {
    lis, err := net.Listen(”tcp”, “localhost:50051”)
    if err != nil {
        panic(err)
    }
    s := grpc.NewServer(withAuthServerInterceptor())
    pb.RegisterUserServiceServer(s, &UserServer{})

    if err := s.Serve(lis); err != nil {
        panic(err)
    }
}
```

The logic of our interceptor is contained into the `AuthServerInterceptor` function that implements the `UnaryServerInterceptor` type. The code looks into the gRPC request metadata for the key/value pair `“password”`. If it is found and is equal to `“go”`, we authenticate the user. In this case, we simply return the result of the handler with the given context and original request. We could go further by removing the password from the metadata and adding another field with the authenticated user id, role, etc.

Additional code prepares the interceptor to be converted into a `ServerOption` as explained above. Now every request will pass through our interceptor. However, the client must include the corresponding metadata otherwise, requests will not be authorized.

Example 14.18: Client extended metadata (excerpt).

```
12 func main() {
13     conn, err := grpc.Dial(”localhost:50051”, grpc.WithInsecure(), grpc.WithBlock())
14     if err != nil {
15         panic(err)
16     }
17     defer conn.Close()
18
19     c := pb.NewUserServiceClient(conn)
20
21     ctx, cancel := context.WithTimeout(context.Background(), time.Second)
22     ctx = metadata.AppendToOutgoingContext(ctx, “password”,”go”)
23     defer cancel()
24
25     r, err := c.GetUser(ctx, &pb.UserRequest{UserId: “John”})
26     if err != nil {
27         panic(err)
28     }
29     fmt.Println(”Client received:”, r.String())
30 }
```

Example 14.18 shows how the client puts additional metadata into the context before sending the request (line 22). For demonstration purposes and simplicity, the user was fixed. You can check that modifying the username returns an error from the server-side. Adding this metadata could be done using an interceptor on the client-side. Check the next section to understand how to write a client interceptor.

14.6.2 Client interceptors

Client interceptors work similarly to server interceptors. The three major elements to be filled are schematically shown below.

```
func MyClientInterceptor(…) error {

// …

func withMyClientInterceptor() grpc.DialOption{

    return grpc.WithUnaryInterceptor(MyClientInterceptor)

}

// …

conn, err := grpc.Dial(”:50051”, withMyClientInterceptor())
```

Our interceptor must implement the function signature defined by the `UnaryClient Interceptor` type.

```
type UnaryClientInterceptor func(ctx context.Context, method string, req,

     reply interface{}, cc *ClientConn, invoker UnaryInvoker, opts …CallOption) error
```

where,

- `ctx` is the context.
- `method` is the URI of the invoked procedure.
- `req` is the request.
- `reply` is the server response.
- `cc` is the connection.
- `invoker` the function to be called to continue the gRPC call.

Now we plan to extend Example 14.4 by adding to every request metadata information about the environment used by the client. Example 14.19 defines the function `ClientLoggerInterceptor` that adds metadata with the operating system and timezone of the client. Observe that before returning, the interceptor calls the `invoker` function to continue the data workflow.

Example 14.19: Client interceptor with logging metadata (excerpt).

```
func ClientLoggerInterceptor(
   ctx context.Context,
   method string,
   req interface{},
   reply interface{},
   cc *grpc.ClientConn,
   invoker grpc.UnaryInvoker,
   opts …grpc.CallOption) error {

   os := runtime.GOOS
   zone, _ := time.Now().Zone()

   ctx = metadata.AppendToOutgoingContext(ctx, “os”, os)
   ctx = metadata.AppendToOutgoingContext(ctx, “zone”, zone)

   err := invoker(ctx, method, req, reply, cc, opts…)
   return err
}

func withUnaryClientLoggerInterceptor() grpc.DialOption {
   return grpc.WithUnaryInterceptor(ClientLoggerInterceptor)
}

func main() {
   conn, err := grpc.Dial(”localhost:50051”, grpc.WithInsecure(),
      grpc.WithBlock(), withUnaryClientLoggerInterceptor())
   if err != nil {
      panic(err)
   }
   defer conn.Close()

   c := pb.NewUserServiceClient(conn)

   ctx, cancel := context.WithTimeout(context.Background(), time.Second)
   defer cancel()

   r, err := c.GetUser(ctx, &pb.UserRequest{UserId: “John”})
   if err != nil {
      panic(err)
   }
   fmt.Println(”Client received:”, r.String())
}
```

For completeness, we have created a server interceptor that captures this metadata for logging purposes in Example 14.20 . The steps to prepare this interceptor are similar to those described in the previous section.

Example 14.20: Server interceptor consuming logging metadata (excerpt).

```
func RequestLoggerInterceptor(ctx context.Context,
    req interface{},
    info *grpc.UnaryServerInfo,
    handler grpc.UnaryHandler) (interface{}, error){
        md, found := metadata.FromIncomingContext(ctx)
        if found {
            os, _ := md[”os”]
            zone, _ := md[”zone”]
            fmt.Printf(”Request from %s using %s\n”, zone, os)
        }

        h, err := handler(ctx, req)
        return h, err
}
```

The output obtained from running the server and its client reveals how the client sent transparently the time zone and operating system to the server.

```
Client received: user_id:”John”  email:”john@gmail.com”
```

```
Request from [CET] using [darwin]
Server received: user_id:”John”
```

14.7 SUMMARY

gRPC is a modern, powerful, and versatile communication protocol. This Chapter has shown how to harness all the powerful features of gRPC. In conjunction with protocol buffers, gRPC is a must in every data system designed to be scalable, portable, and maintainable. We have explained how to implement servers with their corresponding clients, to use streaming connections, and how to provide HTTP interfaces for our gRPC procedures with minimal code using transcoding. Finally, for advanced designs, we have shown how interceptors can be used to enhance the logic of our data communication channels.

CHAPTER 15

LOGGING WITH ZEROLOG

The execution of a program generates a substantial amount of information. Part of this information is dedicated to ensuring the correctness of the program or recording what happened. Logging is therefore a basic component of any program. For any software developer logging must be one of the first elements to introduce in any project. Projects such as Zerolog, take advantage of Go features to offer a modern and complete logging solution. This Chapter exhaustively explores how to log in Go with special attention to the `Zerolog` project.

15.1 THE LOG PACKAGE

Go provides a basic logging solution in the `log` package[41]. This package defines three types of messages `Fatal`, `Panic`, and `Println`. The first two messages are equivalent to executing `Print` followed by `os.Exit` and `panic` functions respectively. Messages can be customized using prefixes and flags as shown in Example 15.1.

Example 15.1: `log` package messaging.

```
package main

import “log”

func main() {
    // Panic or fatal messages stop the execution flow
    // log.Fatal(“This is a fatal message”)
    // log.Panic(“This is a panic message”)
    log.Println(“This is a log message”)
    log.SetPrefix(“prefix -> ”)
    log.Println(“This is a log message”)
    log.SetFlags(log.Lshortfile)
}
```

```
2021/02/08 19:30:34 This is a log message
prefix -> 2021/02/08 19:30:34 This is a log message
```

More customized loggers can be defined using the `log.New` function which receives a writer argument, a prefix, and a flag. This facilitates logs using file writers as shown in Example 15.2.

Example 15.2: `log` package messaging.

```
package main

import (
    “io/ioutil”
    “log”
    “os”
)

func main() {
    tmpFile, err := ioutil.TempFile(os.TempDir(),“logger.out”)
    if err != nil {
        log.Panic(err)
    }
    logger := log.New(tmpFile, “prefix -> ”, log.Ldate)
    logger.Println(“This is a log message”)
}
```

15.2 ZEROLOG BASICS

The previous Section explored how to use the basic logging features provided within the `log` package. The limitations of this package in terms of customization, efficiency, and adaptability made third-party developers write their own logging solutions. A remarkable solution is Zerolog[42] which provides JSON oriented logging using zero-allocation.

The simplest Zerolog message defines a logging level and a message string. The generated output is a JSON object with the level, the time, and the message. There are seven log levels as shown in Example 15.3 from panic (5) to trace (-1).

Example 15.3: Logging messages in zerolog.

```
package main

import(
    “github.com/rs/zerolog/log”
)

func main() {
    // Panic or fatal messages stop the execution flow
    // log.Panic().Msg(“This is a panic message”)
    // log.Fatal().Msg(“This is a fatal message”)
    log.Error().Msg(“This is an error message”)
    log.Warn().Msg(“This is a warning message”)
    log.Info().Msg(“This is an information message”)
    log.Debug().Msg(“This is a debug message”)
    log.Trace().Msg(“This is a trace message”)
}
```

```
{"level":"error","time":"2021-02-08T19:00:21+01:00","message":"This is an error
message"}
{"level":"warn","time":"2021-02-08T19:00:21+01:00","message":"This is a warning
message"}
{"level":"info","time":"2021-02-08T19:00:21+01:00","message":"This is an information
message"}
{"level":"debug","time":"2021-02-08T19:00:21+01:00","message":"This is a debug
message"}
{"level":"trace","time":"2021-02-08T19:00:21+01:00","message":"This is a trace
message"}
```

Log levels help to identify the severity of messages and can be set on the fly using the global level variable. Example 15.4 changes the global level from `Debug` to `Info`. Notice that after setting the info level, debug messages are discarded.

Example 15.4: Set global level zerolog.

```go
package main

import (
    "github.com/rs/zerolog/log"
    "github.com/rs/zerolog"
)

func main() {
    zerolog.SetGlobalLevel(zerolog.DebugLevel)

     log.Debug().Msg("Debug message is displayed")
     log.Info().Msg("Info Message is displayed")

     zerolog.SetGlobalLevel(zerolog.InfoLevel)
    log.Debug().Msg("Debug message is no longer displayed")
    log.Info().Msg("Info message is displayed")
}
```

```
{"level":"debug","time":"2021-02-08T11:12:56+01:00","message":"Debug message is displayed"}
{"level":"info","time":"2021-02-08T11:12:56+01:00","message":"Info Message is displayed"}
{"level":"info","time":"2021-02-08T11:12:56+01:00","message":"Info message is displayed"}
```

Zerolog provides messages within a context. This context has zero or more variables accompanying the logging message. Variables are typed and can be added on the fly to any message like shown in Example 15.5. A complete list of available types can be found at the official documentation [43].

Example 15.5: Set message context.

```
package main

import(
   "github.com/rs/zerolog/log"
)

func main() {
   log.Info().Str("mystr","this is a string").Msg("")
   log.Info().Int("myint",1234).Msg("")
   log.Info().Int("myint",1234).Str("str","some string").Msg("And a regular message")
}
```

```
{"level":"info","mystr":"this is a string","time":"2021-02-08T11:19:55+01:00"}
```

```
{"level":"info","myint":1234,"time":"2021-02-08T11:19:55+01:00"}

{"level":"info","myint":1234,"str":"some string","time":"2021-02-
08T11:19:55+01:00","message":"And a regular message"}
```

Zerolog is specially oriented to leverage JSON encoding[44]. In this sense, using JSON tags helps to display structs consistently. Example 15.6 shows how two structs are displayed in the context of a log message when they have JSON tags or not. Both types are sent to the context like interfaces. Observe that the tagged type follows the field names given in the JSON tags. Additionally, the `RawJSON` context type permits printing JSON encoded objects.

Example 15.6: JSON tagging and encoding in message logs.

```go
package main

import (
    "encoding/json"
    "github.com/rs/zerolog/log"
)

type AStruct struct {
    FieldA string
    FieldB int
    fieldC bool
}

type AJSONStruct struct {
    FieldA string   `json:"fieldA,omitempty"`
    FieldB int      `json:"fieldB,omitempty"`
    fieldC bool
}
```

```

func main() {
    a := AStruct{“a string”, 42, false}
    b := AJSONStruct{“a string”, 42, false}

    log.Info().Interface(“a”,a).Msg(“AStruct”)
    log.Info().Interface(“b”,b).Msg(“AJSONStruct”)

    encoded, _ := json.Marshal(b)
    log.Info().RawJSON(“encoded”,encoded).Msg(“Encoded JSON”)
}
```

```
{”level”:”info”,”a”:{”FieldA”:”a string”,”FieldB”:42},”time”:”2021-02-08T19:20:59+01:00”,”message”:”AStruct”}
{”level”:”info”,”b”:{”fieldA”:”a string”,”fieldB”:42},”time”:”2021-02-08T19:20:59+01:00”,”message”:”AJSONStruct”}
{”level”:”info”,”encoded”:{”fieldA”:”a string”,”fieldB”:42},”time”:”2021-02-08T19:20:59+01:00”,”message”:”Encoded JSON”}
```

Errors are very important in logs. Understanding where errors occur in a program runtime is a crucial task for any developer. The `error` type is available as a context type like shown in Example 15.7.

Example 15.7: Single error logging.

```
package main

import (
   “errors”
   “github.com/rs/zerolog/log”
)
```

```

func main() {
  err := errors.New("there is an error")

   log.Error().Err(err).Msg("this is the way to log errors")
}
```

```
{"level":"error","error":"there is an error","time":"2021-02-08T19:38:35+01:00","message":"this is the way to log errors"}
```

Print the point in the code where an error occurred may not always be enough. Understanding where an error comes from obtaining the full stack trace is more useful. The stack trace can be obtained as shown in Example 15.8. Setting the `ErrorStackMarshaler` to the implementation offered at `pkgerrors` permits the context `Stack()` to get the complete stack. The log output contains the list of invoked functions until the error was found.

Example 15.8: Stack trace logging.

```
package main

import (
    "github.com/pkg/errors"
    "github.com/rs/zerolog/log"
    "github.com/rs/zerolog"
    "github.com/rs/zerolog/pkgerrors"
)

func failA() error {
    return failB()
}
```

```

func failB() error {
    return failC()
}

func failC() error {
    return errors.New(“C failed”)
}

func main() {
    zerolog.ErrorStackMarshaler = pkgerrors.MarshalStack

    err := failA()
    log.Error().Stack().Err(err).Msg(””)
}
```

```
{”level”:”error”,”stack”:[{”func”:”failC”,”line”:”19”,”source”:”main.go”},
{”func”:”failB”,”line”:”15”,”source”:”main.go”},
{”func”:”failA”,”line”:”11”,”source”:”main.go”},
{”func”:”main”,”line”:”25”,”source”:”main.go”},
{”func”:”main”,”line”:”204”,”source”:”proc.go”},
{”func”:”goexit”,”line”:”1374”,”source”:”asm_amd64.s”}],”error”:”C
failed”,”time”:”2021-02-08T19:42:22+01:00”}
```

15.3 ZEROLOG SETTINGS

Zerolog has an extensive set of configuration options. These options can modify the message output format, where is the output sent to, define subloggers, just to mention a few. In this Section, we explore some settings that a developer may find useful when defining how and what to log in to a project.

15.3.1 Write logs to a file

Logs in the standard output are not always the best solution, especially in unsupervised scenarios. Writing logs to a file not also permits to have a non-volatile version, it additionally facilitates forensics and can even be used to feed analytical systems. A file logger can be created using the `New` function which receives an `io.Writer`. Example 15.9 creates a logger which dumps all the messages into a temporary file.

Example 15.9: Stack trace logging.

```
package main

import (
    “io/ioutil”
    “os”
    “fmt”
    “github.com/rs/zerolog/log”
    “github.com/rs/zerolog”
)

func main() {
    tempFile, err := ioutil.TempFile(os.TempDir(),“deleteme”)
    if err != nil {
        // Can we log an error before we have our logger? :)
        log.Error().Err(err).
            Msg(“there was an error creating a temporary file four our log”)
    }
    defer tempFile.Close()
    fileLogger := zerolog.New(tempFile).With().Logger()
    fileLogger.Info().Msg(“This is an entry from my log”)
    fmt.Printf("The log file is allocated at %s\n", tempFile.Name())
}
```

```
The log file is allocated at
/var/folders/6h/xffhh45j077157cb5mbk48zh0000gp/T/deleteme930052425
```

15.3.2 Output customization

We have already mentioned that Zerolog is designed to use JSON. However, JSON may not be the most user-friendly format, especially if we print logs to a console. The `Console Writer` gets rid of colons, brackets, and quotation marks from the JSON syntax and permits us to easily define our output format. Example 15.10 customizes a `ConsoleWriter` to define new formats for the level, message, field name and field value. Every item can be redefined using a function that receives an interface and returns a string with the new value. See how in this example, the field name is surrounded by square brackets or how the log level is always fixed to a six characters string.

Example 15.10: Customized output with `ConsoleWriter`.

```
package main

import (
    "os"
    "strings"
    "time"
    "fmt"
    "github.com/rs/zerolog"
)

func main() {
    output := zerolog.ConsoleWriter{Out: os.Stdout, TimeFormat: time.RFC3339}
    output.FormatLevel = func(i interface{}) string {
        return strings.ToUpper(fmt.Sprintf("| %-6s|", i))
    }
    output.FormatMessage = func(i interface{}) string {
        return fmt.Sprintf(">>>%s<<<", i)
    }
    output.FormatFieldName = func(i interface{}) string {
        return fmt.Sprintf("[%s]:", i)
    }
    output.FormatFieldValue = func(i interface{}) string {
        return strings.ToUpper(fmt.Sprintf("[%s]", i))
    }

    log := zerolog.New(output).With().Timestamp().Logger()

    log.Info().Str("foo", "bar").Msg("Save the world with Go!!!")
}
```

```
2021-02-08T19:16:09+01:00 | INFO  | >>>Save the world with Go!!!<<< [foo]:[BAR]
```

15.3.3 Multi-logger

It is common to separate logs depending on the log level, or the message final output (file, standard output). The `MultiLevelWriter` aggregates several loggers making it possible for a single log message to be written to different destinations with different formats. Example 15.11 defines a multi-logger that simultaneously sends every message to a file, the standard output, and a `ConsoleWriter`. The output shows the outcome from the two loggers dumping messages to the standard output and the file with the logs. Additional customizations such as setting log levels to have specialized loggers can be done. This is a common approach to separate error messages into specific log files.

Example 15.11: Simultaneously logging to several outputs with `MultiLevelWriter`.

```
package main

import (
    “io/ioutil”
    “os”
    “fmt”
    “github.com/rs/zerolog”
    “github.com/rs/zerolog/log”
)

func main() {

    tempFile, err := ioutil.TempFile(os.TempDir(),“deleteme”)
    if err != nil {
        log.Error().Err(err).
            Msg(“there was an error creating a temporary file four our log”)
    }
    defer tempFile.Close()
    fmt.Printf(“The log file is allocated at %s\n”, tempFile.Name())

    fileWriter := zerolog.New(tempFile).With().Logger()
    consoleWriter := zerolog.ConsoleWriter{Out: os.Stdout}

    multi := zerolog.MultiLevelWriter(consoleWriter, os.Stdout, fileWriter)

    logger := zerolog.New(multi).With().Timestamp().Logger()

    logger.Info().Msg(“Save the world with Go!!!”)
}
```

```
The log file is allocated at
/var/folders/6h/xffhh45j077157cb5mbk48zh0000gp/T/deleteme581703284

12:32PM INF Save the world with Go!!!

{”level”:”info”,”time”:”2021-02-08T12:32:18+01:00”,”message”:”Save the world with
Go!!!”}
```

15.3.4 Sub-logger

By definition, Zerolog loggers are extensible. This makes it possible to create new loggers enhancing the existing ones. These sub-loggers inherit the existing configuration and can extend the context. In Example 15.12 we create a new logger from `mainLogger` with additional context indicating the current component. Check how the output from the sub-logger maintains the same configuration from the main logger with the additional context without additional info.

Example 15.12: Extensible logging using sub-loggers.

```
package main

import (
    “os”
    “github.com/rs/zerolog”
)

```

```
func main() {
   mainLogger := zerolog.New(os.Stdout).With().Logger()
    mainLogger.Info().Msg(“This is the main logger”)

    subLogger := mainLogger.With().Str(“component”,“componentA”).Logger()
    subLogger.Info().Msg(“This is the sublogger”)
}
```

```
{”level”:”info”,”message”:”This is the main logger”}
{”level”:”info”,”component”:”componentA”,”message”:”This is the sublogger”}
```

15.4 ZEROLOG ADVANCED SETTINGS

This Section, extends the current overview of Zerolog features and customization by explaining examples of additional solutions for specific scenarios.

15.4.1 Hooks

Hooks are executed every time a logger is invoked. The `Hook` interface defines the `Run` method that gives access to the arguments of a log message. The Example 15.13 uses a `Hook` to add the component name to the context of every debug message. Additionally, a second hook adds a random number to the context. Both hooks are added to the same logger modifying the final behaviour of the logger. Observe that only, the debug message contains the component string in the context.

Example 15.13: Extensible logging using sub-loggers.

```
package main

import (
    “github.com/rs/zerolog”
    “github.com/rs/zerolog/log”
    “math/rand”
)

type ComponentHook struct {
     component string
 }

 func (h ComponentHook) Run(e *zerolog.Event, level zerolog.Level, msg string) {
     if level == zerolog.DebugLevel {
         e.Str(“component”, h.component)
     }
 }

 type RandomHook struct{}

 func (r RandomHook) Run(e *zerolog.Event, level zerolog.Level, msg string) {
     e.Int(“random”,rand.Int())
 }

 func main() {
     logger := log.Hook(ComponentHook{“moduleA”})
     logger = logger.Hook(RandomHook{})
     logger.Info().Msg(“Info message”)
     logger.Debug().Msg(“Debug message”)
 }
```

```
{"level":"info","time":"2021-02-08T13:16:37+01:00","random":5577006791947779410,"message":"Info message"}

{"level":"debug","time":"2021-02-08T13:16:37+01:00","component":"moduleA","random":8674665223082153551,"message":"Debug message"}
```

15.4.2 Sampling

Logging messages can be particularly disturbing when executed inside loops. It becomes even worse for a large number of iterations. Zerolog provides sampling loggers that can be configured to adequate the number of generated log messages. The `log.Sample` function returns a Logger based on a `sampler` type. The `Sampler` interfaces only contain a `Sample` method that returns true when a message has to be sampled.

The `BasicSampler` sends a message once out of N times. This is particularly useful inside loops, like in Example 15.14 , where a message is printed every 200 iterations.

Example 15.14: Logger using basic sampler.

```go
package main

import (
    "github.com/rs/zerolog"
    "github.com/rs/zerolog/log"
)

func main() {
    logger := log.Sample(&zerolog.BasicSampler{N:200})
     for i:=0;i<1000;i++{
         logger.Info().Int("i",i).Msg("")
    }
}
```

```
{”level”:”info”,”i”:0,”time”:”2021-02-09T19:24:03+01:00”}
{”level”:”info”,”i”:200,”time”:”2021-02-09T19:24:03+01:00”}
{”level”:”info”,”i”:400,”time”:”2021-02-09T19:24:03+01:00”}
{”level”:”info”,”i”:600,”time”:”2021-02-09T19:24:03+01:00”}
{”level”:”info”,”i”:800,”time”:”2021-02-09T19:24:03+01:00”}
```

The `BurstSampler` permits more sophisticated policies indicating how many messages are allowed per period. Additionally, a `NextSampler` can be used to indicate what sampler has to be invoked when the burst limit is reached. In Example 15.15 , the sampler defines a burst of two messages every five seconds and then one sample every 90000000 iterations. After the first two entries, the burst limit is reached. Looking at the timestamp, we see that no more consecutive messages will be printed after five seconds.

Example 15.15: Logger using burst sampler.

```
package main

import (
    “github.com/rs/zerolog”
    “github.com/rs/zerolog/log”
    “time”
)

func main() {
    logger := log.Sample(&zerolog.BurstSampler{
        Burst: 2,
        Period: time.Second*5,
        NextSampler: &zerolog.BasicSampler{N: 90000000},
    })

    for i:=0;i<99999999;i++{
        logger.Info().Int(“i”,i).Msg(“”)
    }
}
```

```
{”level”:”info”,”i”:0,”time”:”2021-02-09T19:32:08+01:00”}
{”level”:”info”,”i”:1,”time”:”2021-02-09T19:32:08+01:00”}
{”level”:”info”,”i”:2,”time”:”2021-02-09T19:32:08+01:00”}
{”level”:”info”,”i”:54825538,”time”:”2021-02-09T19:32:13+01:00”}
{”level”:”info”,”i”:54825539,”time”:”2021-02-09T19:32:13+01:00”}
{”level”:”info”,”i”:90000004,”time”:”2021-02-09T19:32:17+01:00”}
```

15.4.3 Integration with HTTP Handlers

A common scenario for logging messages is HTTP handlers. Zerolog provides additional tools to integrate with `http.Handler` types[45]. The `hlog.NewHandler` returns a function that receives an `http.Handler` and returns another handler. This makes possible the concatenation of loggers and their integration with other handlers. Some logging functions are already available[46] to extend the message context adding information extracted from the requests.

In Example 15.16, we create a simple HTTP Server that returns a predefined message for every request. We extend our logger `log` with additional context extracted from HTTP requests. We add the remote IP address, the agent handler, and the request id with `RemoteAddrHandler`, `UserAgentHandler`, and `RequestIDHandler` respectively. The concatenation of the loggers with our handler creates a middleware of loggers that is available for every request. Because `hlog` works as a contextual logger, we have to invoke `hlog.FromRequest` to get a logger with the contextual information (line 17). In the final output the message from the `hlog` logger contains additional context from the incoming request without any additional intervention.

Example 15.16: Integration of contextual HTTP loggers.

```
1 package main
2
3 import (
4     “net/http”
5     “github.com/rs/zerolog”
6     “github.com/rs/zerolog/hlog”
7     “os”
8 )
9
10 var log zerolog.Logger = zerolog.New(os.Stdout).With().
11     Str(“app”,“example_04”).Logger()
12
13 type MyHandler struct {}
14
15 func(c MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
16     log.Info().Msg(“This is not a request contextual logger”)
17     hlog.FromRequest(r).Info().Msg(“”)
18     w.Write([]byte(“Perfect!!!”))
19     return
20 }
21
22 func main() {
23     mine := MyHandler{}
24     a := hlog.NewHandler(log)
```

```
    b := hlog.RemoteAddrHandler(“ip”)
    c := hlog.UserAgentHandler(“user_agent”)
    d := hlog.RequestIDHandler(“req_id”, “Request-Id”)

    panic(http.ListenAndServe(“:8090”, a(b(c(d(mine))))))
}
```

```
{”level”:”info”,”app”:”example_04”,”message”:”This is not a request contextual
logger”}

{”level”:”info”,”app”:”example_04”,”ip”:”::1”,”user_agent”:”curl/7.64.1”,”req_id”:”c0h5
```

Logging HTTP requests may imply several pieces of contextual information. The way we have concatenated our loggers in Example 15.16 is valid. However, it may not be handy when the number of loggers is large or may vary frequently. In Example 15.17 the `Wrapper` type and its method `GetWrapper` extend any handler with a collection of HTTP loggers. This method invokes recursively all the layers and finally applies the received `http.Handler`. Finally, the method returns an `http.Handler` that invokes all the layers. This method simplifies the code required to start the server and makes more flexible the definition of context loggers to use. A similar solution for HTTP middleware is shown in Example 9.10 using a sequential loop instead of recursion.

Example 15.17: Integration of several contextual HTTP loggers.

```
package main

import (
   “net/http”
   “github.com/rs/zerolog”
   “github.com/rs/zerolog/hlog”
   “os”
)

var log zerolog.Logger = zerolog.New(os.Stdout).With().
    Str(“app”,“example_05”).Logger()

type MyHandler struct {}

func(c MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    hlog.FromRequest(r).Info().Msg(””)
    w.Write([]byte(“Perfect!!!”))
    return
}

type Wrapper struct {
    layers []func(http.Handler) http.Handler
}

func NewWrapper() *Wrapper {
    layers := []func(http.Handler) http.Handler {
        hlog.NewHandler(log),
        hlog.RemoteAddrHandler(“ip”),
        hlog.UserAgentHandler(“user_agent”),
        hlog.RequestIDHandler(“req_id”, “Request-Id”),
        hlog.MethodHandler(“method”),
        hlog.RequestHandler(“url”),
    }
    return &Wrapper{layers}
}
```

```

func(w *Wrapper) GetWrapper(h http.Handler,i int) http.Handler {
    if i >= len(w.layers) {
        return h
    }
    return w.layers[i](w.GetWrapper(h,i+1))
}

func main() {
    mine := MyHandler{}
    wrapper := NewWrapper()
    h := wrapper.GetWrapper(mine,0)

    panic(http.ListenAndServe(":8090", h))
}
```

```
{"level":"info","app":"example_05","ip":"::1","user_agent":"curl/7.64.1","req_id":"c0h5
/"}
```

15.5 SUMMARY

Program logs are one of those tasks developers postpone. However, good logs always help to have better problems resolution and a more comprehensive and maintainable project. This Chapter demonstrates how the Zerolog library can help to define powerful and modular logs for any Go project.

CHAPTER 16

COMMAND LINE INTERFACE

A software developer is used to interact with command-line interfaces (CLI) daily. Good systems not only solve problems but also are easy to use. Nowadays solutions may offer hundreds of commands with several parameters each. Defining a good command-line interface that enables users to easily execute any operation is challenging. Fortunately, the Go ecosystem offers powerful solutions that reduce the effort required to define powerful and solid CLIs. In particular, this Chapter shows how the Cobra library helps developers to tackle this task.

16.1 THE BASICS

Cobra[47] is a library that provides all the necessary items to develop CLIs in Go. This library is used by major projects such as Kubernetes, Github or Istio and has some interesting features such as intelligent suggestions, automatic help generation, documentation, shell autocompletion, it is POSIX compliant, etc.

Before explaining the basics make sure you have Cobra available in your environment. The examples from this Chapter require the library to be installed. As usual, you can use Go modules (see Chapter 12) or run the following command to download the code in your `GOPATH`.

```
go get github.com/spf13/cobra
```

Cobra CLIs are built around the `Command` type which fields define the name of the actions to be executed, their arguments, and help information. Example 16.1 contains a minimal example of a Cobra CLI for a program. This naive program is intended to be executed like `./hello` and print a message using the standard output.

Example 16.1: Basic Cobra CLI

```
package main

import (
    “fmt”
    “github.com/spf13/cobra”
    “os”
)

var RootCmd = &cobra.Command{
     Use: “hello”,
     Short: “short message”,
     Long: “Long message”,
     Version: “v0.1.0”,
     Example: “this is an example”,
     Run: func(cmd *cobra.Command, args []string) {
         fmt.Println(”Save the world with Go!!!”)
     },
}

func main() {
     if err := RootCmd.Execute(); err != nil {
         fmt.Fprintln(os.Stderr, err)
         os.Exit(1)
     }
}
```

Observe that the `main` function only executes the root command. The execution flow is now delegated to the root command. `RootCmd` is a Cobra command with some fields that are used to populate the program help displayed below. This minimal example already generates this help automatically.

```
>>> ./hello —help

Long message


Usage:

  hello [flags]


Examples:

this is an example


Flags:

  -h, —help      help for hello

   -v, —version   version for hello
```

We defined a long message that is displayed when the command help is invoked, an example text, and the program version that is displayed with `./hello -v`.

By convention, Cobra expects commands to be defined one per file in the `cmd` folder as shown below.

```
appName
├─ cmd
│  ├─ RootCommand.go
│  ├─ command1.go
│  ├─ command2.go
│  └─ command3.go
└─ main.go
```

The `main.go` file is minimal and only executes the root command. To facilitate the understanding of examples at a glance, these are defined in a single file if not mentioned otherwise. However, they follow the same concepts and ideas so they are easily portable to this organization.

16.2 ARGUMENTS AND FLAGS

Commands may require arguments to be executed. These arguments are passed to the `Run` field of every `Command`. In Example 16.2, the arguments received by the root command are printed.

Example 16.2: Command receving arguments

```
package main

import (
    “fmt”
    “github.com/spf13/cobra”
    “os”
    “strings”
)

 var RootCmd = &cobra.Command{
     Use: “main”,
     Long: “Long message”,
     Run: func(cmd *cobra.Command, args []string) {
         fmt.Printf(”%s\n”,strings.Join(args,”,”))
     },
 }

 func main() {
     if err := RootCmd.Execute(); err != nil {
         fmt.Fprintln(os.Stderr, err)
         os.Exit(1)
     }
 }
```

```
>>> ./main These are arguments
These,are,arguments
>>> ./main 1 2 three 4 five
1,2,three,4,five
```

Observe that these arguments are not typed and there is no control over the number of arguments the program expects. Using flags, Cobra parses incoming arguments that can be later used in the commands. The Example 16.3 expects a `–msg` flag with a message to be displayed with the program.

Example 16.3: Command with single string flag.

```
package main
```

```

import (
    “fmt”
    “github.com/spf13/cobra”
    “os”
)

var RootCmd = &cobra.Command{
    Use: “main”,
    Long: “Long message”,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Printf(”[[–%s–]]\n”, *Msg)
    },
}

var Msg *string

func init() {
    Msg = RootCmd.Flags().String(”msg”, “Save the world with Go!!!”,
        “Message to show”)
}

func main() {
    if err := RootCmd.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

Flags have to be extracted in an `init` function to make it accessible before `main` is invoked. Cobra parses flags only when it detects the command is expecting them. In the example, we define a flag named `msg` of type `string` with a default value, and a usage message that is displayed in the command help. The returned value is stored in the `Msg` variable which value contains the incoming argument. This flag can now be used by the command in the `Run` function.

```
>>> ./main —msg Hello

[[—Hello—]]

>>> ./main

[[—Save the world with Go!!!—]]

>>> ./main —message Hello

Error: unknown flag: —message

Usage:

  main [flags]


Flags:

  -h, —help         help for main

      —msg string   Message to show (default “Save the world with Go!!!”)


unknown flag: —message

exit status 1
```

The output shows how we can set the flag to print a custom message or use the default value when no flag is passed. However, if the flag is unknown Cobra returns an error message and the command help.

A command can have several flags with different types like in Example 16.4. This code extends the previous example with the number of times our message will be printed.

Example 16.4: Command with several typed flags.

```
package main

import (
    “fmt”
    “github.com/spf13/cobra”
    “os”
)

var RootCmd = &cobra.Command{
    Use: “main”,
    Long: “Long message”,
    Run: func(cmd *cobra.Command, args []string) {
        for i:=0;i<*Rep;i++ {
            fmt.Printf(”[[–%s–]]\n”, *Msg)
        }
    },
}

var Msg *string
var Rep *int

func init() {
    Msg = RootCmd.Flags().String(”msg”, “Save the world with Go!!!”,
        “Message to show”)
    Rep = RootCmd.Flags().Int(”rep”,1, “Number of times to show the message”)
}

func main() {
    if err := RootCmd.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

```
>>> ./main
[[—Save the world with Go!!!—]]
>>> ./main —msg Hello —rep 3
[[—Hello—]]
[[—Hello—]]
[[—Hello—]]
```

Flags can be marked to be required forcing the user to indicate its value. Example 16.5 extends the previous one to force users to indicate the number of repetitions. Observe that the `msg` flag can be missing but the missing `rep` flag returns a required flag error.

Example 16.5: Command with required flags.

```
package main

import (
    “fmt”
    “github.com/spf13/cobra”
    “os”
)

var RootCmd = &cobra.Command{
    Use: “main”,
    Long: “Long message”,
    Run: func(cmd *cobra.Command, args []string) {
        for i:=0;i<*Rep;i++ {
            fmt.Printf(”[[—%s—]]\n”, *Msg)
        }
    },
}

var Msg *string
var Rep *int

func init() {
    Msg = RootCmd.Flags().String(”msg”, “Save the world with Go!!!”,
        “Message to show”)
    Rep = RootCmd.Flags().Int(”rep”,1, “Number of times to show the message”)

    RootCmd.MarkFlagRequired(”rep”)
}

func main() {
    if err := RootCmd.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

```
>>> ./main
Error: required flag(s) “rep” not set
Usage:
  main [flags]
```

```
Flags:
  -h, —help         help for main
      —msg string   Message to show (default "Save the world with Go!!!")
      —rep int      Number of times to show the message (default 1)

required flag(s) "rep" not set
exit status 1
>>> ./main —rep 2
[[—Save the world with Go!!!—]]
[[—Save the world with Go!!!—]]
```

Previous examples fill the variables to be used by commands with direct assignations. A less verbose approach uses `xxxVar` functions with `xxx` the flag type. These functions do not return any value but expect a pointer argument to the variable to be filled with the flag value. Example 16.6 defines a `Config` type to contain all the configuration parameters passed by flags. The fields of the `cnfg` variable can be directly filled with `StringgVar` and `IntVar`.

Example 16.6: Flag parsing using pointer variables.

```go
package main

import (
    "fmt"
    "github.com/spf13/cobra"
    "os"
)

var RootCmd = &cobra.Command{
    Use: "main",
    Short: "short message",
    Run: func(cmd *cobra.Command, args []string) {
        for i:=0;i<cnfg.Rep;i++ {
            fmt.Printf(”[[—%s—]]\n”, cnfg.Msg)
        }
    },
}

type Config struct {
    Msg string
    Rep int
}
var cnfg Config = Config{}

func init() {
    RootCmd.Flags().StringVar(&cnfg.Msg, “msg”,”Save the world with Go!!!”,”Message to show”)
    RootCmd.Flags().IntVar(&cnfg.Rep,”rep”,1, “Number of times to show the message”)
    RootCmd.MarkFlagRequired(”rep”)
}

func main() {
    if err := RootCmd.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

16.3 COMMANDS

A CLI may contain one or several commands, and these commands can have subcommands. Generally, the root command is not intended to run any operation. Example 16.7 prints a `hello` or `bye` message. Syntactically `say hello` is an operation that makes sense. However, the root command `say` makes no sense and simply prints the help.

Example 16.7: CLI with several commands.

```
package main

import (
    “fmt”
    “github.com/spf13/cobra”
    “os”
)

var RootCmd = &cobra.Command{
    Use: “say”,
    Long: “Root command”,
}

var HelloCmd = &cobra.Command{
    Use: “hello”,
    Short: “Say hello”,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println(”Hello!!!”)
    },
}

var ByeCmd = &cobra.Command{
    Use: “bye”,
    Short: “Say goodbye”,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println(”Bye!!!”)
    },
}

func init() {
    RootCmd.AddCommand(HelloCmd, ByeCmd)
}

func main() {
    if err := RootCmd.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

Running the root command as shown below prints the help message. Now this message includes the `bye` and `hello` commands. Commands have their specific help which shows the command definition and any subcommand if proceeds.

```
>>> ./say
Root command

Usage:
  say [command]

Available Commands:
  bye         Say goodbye
  hello       Say hello
  help        Help about any command

Flags:
  -h, —help   help for say

Use “say [command] —help” for more information about a command.
>>> ./say hello
Hello!!!
>>> ./say bye
Bye!!!
>>> ./say bye —help
Say goodbye

Usage:
  say bye [flags]

Flags:
   -h, —help   help for bye
```

16.3.1 Persistent and local flags

Flags can be shared among commands or be local. Cobra differentiates between the `PersistentFlags` and `Flags` types. The first type assumes the flag to be propagated to the command’s children while `Flags` are locally available only. Example 16.8 extends the previous example with the name of the person we are talking to and a custom command that sets the message to be said. In this configuration, the custom command requires an additional flag that should not be available for other commands. This is done in the code by setting the `person` flag as persistent and defining the local flag `msg` for the `CustomCmd` command.

Example 16.8: Commands using persistent and local flags.

```
package main

import (
    “fmt”
    “github.com/spf13/cobra”
    “os”
)

var RootCmd = &cobra.Command{
    Use: “say”,
    Long: “Root command”,
}

var HelloCmd = &cobra.Command{
    Use: “hello”,
    Short: “Say hello”,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Printf(”Hello %s!!!\n”,person)
    },
}

var ByeCmd = &cobra.Command{
    Use: “bye”,
    Short: “Say goodbye”,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Printf(”Bye %s!!!\n”,person)
    },
}

var CustomCmd = &cobra.Command{
    Use: “custom”,
    Short: “Custom greetings”,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Printf(”Say %s to %s\n”,msg,person)
    },
}

var msg string
var person string

func init() {
    RootCmd.AddCommand(HelloCmd, ByeCmd, CustomCmd)

    RootCmd.PersistentFlags().StringVar(&person, “person”, “Mr X”, “Receiver”)
    CustomCmd.Flags().StringVar(&msg,”msg”,”what’s up”,”Custom message”)
}

func main() {
    if err := RootCmd.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

The custom command help indicates the person flag to be global. This means that the value is inherited and already available for the command.

Anyway, we can modify it because it is defined by the root command which is the parent of the custom command.

```
>>> ./say bye –person John

Bye John!!!

>>> ./say custom –help

Custom greetings


Usage:

 say custom [flags]


Flags:

 -h, –help         help for custom

    –msg string   Custom message (default “what’s up”)


Global Flags:

    –person string   Receiver (default “Mr X”)

>>> ./say custom –person John

Say what’s up to John
```

16.3.2 Hooks

The `Command` type offers functions that can be executed before and after the `Run` function. `PreRun` and `PostRun` are executed before and after respectively. `PersistentPreRun` and `PersistentPostRun` are inherited by children commands.

Example 16.9: Commands using hook functions.

```
package main

import (
    “fmt”
    “github.com/spf13/cobra”
    “os”
)

var RootCmd = &cobra.Command{
     Use: “say”,
     Long: “Root command”,
     PersistentPreRun: func(cmd *cobra.Command, args []string) {
         fmt.Printf(”Hello %s!!!\n”, person)
     },
     Run: func(cmd *cobra.Command, args []string) {},
     PostRun: func(cmd *cobra.Command, args []string) {
         fmt.Printf(”Bye %s!!!\n”, person)
     },
 }

 var SomethingCmd = &cobra.Command{
     Use: “something”,
     Short: “Say something”,
     Run: func(cmd *cobra.Command, args []string) {
         fmt.Printf(”%s\n”,msg)
     },
     PostRun: func(cmd *cobra.Command, args []string) {
         fmt.Printf(”That’s all I have to say %s\n”, person)
     },
 }

 var person string
 var msg string

 func init() {
     RootCmd.AddCommand(SomethingCmd)
     RootCmd.Flags().StringVar(&person, “person”, “Mr X”, “Receiver”)
     SomethingCmd.Flags().StringVar(&msg, “msg”, “”, “Message to say”)
     SomethingCmd.MarkFlagRequired(”msg”)
 }

 func main() {
     if err := RootCmd.Execute(); err != nil {
         fmt.Fprintln(os.Stderr, err)
         os.Exit(1)
     }
 }
```

Example 16.9 combines hooks with commands to include additional messages. Observe that simply executing the root command prints two messages corresponding to the pre and post-run functions. Running the subcommand `something` prints the `PersistentRun` from the root command but not the `PostRun` which is replaced by the `PostRun` from the command `something`.

```
>>> ./say
Hello Mr X!!!
Bye Mr X!!!
>>> ./say something —msg “How are you?”
Hello Mr X!!!
How are you?
That’s all I have to say Mr X
```

16.4 ADVANCED FEATURES

Cobra is a powerful and customizable library with a vast number of options. Your CLIs may require additional features beyond the definition of commands and their parameters. This Section explores some advanced features.

16.4.1 Custom help and usage

Cobra automatically generates help and usage messages for our CLI. However it is possible to define our own solutions with `SetHelpCommand`, `SetHelpFunc`, and `SetHelpTemplate` methods from the `Command` type. For the usage message, there are similar methods. Example 16.10 defines the `action` command with the flag `now`. We replace the default help and usage with functions `helper` and `usager`. By setting these functions in the root command we change the whole CLI behaviour. However, this could be changed only for specific commands.

Example 16.10: Command custom help.

```
package main

import (
    “errors”
    “fmt”
    “github.com/spf13/cobra”
    “os”
)

var RootCmd = &cobra.Command{
     Use: “main”,
     Short: “short message”,
}

var ActionCmd = &cobra.Command{
    Use: “action”,
    Args: cobra.MinimumNArgs(2),
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println(”Do something with “,args)
    },
}

func helper (cmd *cobra.Command, args []string) {
    fmt.Printf(”You entered command %s\n”, cmd.Name())
    fmt.Println(”And that is all the help we have right now :)”)
}

func usager (cmd *cobra.Command) error {
    fmt.Printf(”You entered command %s\n”, cmd.Name())
    fmt.Println(”And you do not know how it works :)”)
    return errors.New(”Something went wrong :(”)
}

func main() {
    RootCmd.AddCommand(ActionCmd)
    RootCmd.SetHelpFunc(helper)
    RootCmd.SetUsageFunc(usager)

    ActionCmd.Flags().Bool(”now”,false,”Do it now”)

    if err := RootCmd.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

Observe that the help is overwritten with the `helper` function returning a different message using the name of the requested command. If we misspelt the flag `now`, Cobra launches the usage function shown below.

```
>>> ./main  help action
You entered command action
And that is all the help we have right now :)
>>> ./main action —naw
Error: unknown flag: —naw
You entered command action
And you do not know how it works :)
Error: Something went wrong :(
exit status 1
```

16.4.2 Documented CLIs

The help entities generated by Cobra are a sort of CLI documentation. However, it cannot be considered as documentation because it requires an interactive exploration. Cobra can automatically generate the documentation for a CLI using command descriptions and examples in Man pages, Markdown, Rest, or Yaml formats. The `cobra/doc` package contains functions that can explore the commands tree and generate the corresponding documentation. Example 16.11 generates the documentation for the CLI in every available format.

Example 16.11: Self-documented CLI.

```
package main

import (
    “fmt”
    “github.com/spf13/cobra”
    “github.com/spf13/cobra/doc”
    “os”
)

var RootCmd = &cobra.Command{
    Use: “test”,
    Short: “Documented test”,
    Long: “How to document a command”,
    Example: “./main test”,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println(”Save the world with Go!!!”)
    },
}

func main() {

    RootCmd.Flags().Bool(”flag”,true,”Some flag”)

    header := &doc.GenManHeader{
        Title: “Test”,
        Manual: “MyManual”,
        Section: “1”,
    }
    err := doc.GenManTree(RootCmd, header, “.”)
    if err != nil {
        panic(err)
    }
    err = doc.GenMarkdownTree(RootCmd, “.”)
    if err != nil {
        panic(err)
    }
    err = doc.GenReSTTree(RootCmd, “.”)
    if err != nil {
        panic(err)
    }
    err = doc.GenYamlTree(RootCmd, “.”)
    if err != nil {
        panic(err)
    }
    if err := RootCmd.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

If for example we get the content from the `test.yaml` file we can see the populated fields below.

```
>>> cat test.yaml
name: test
synopsis: Documented test
description: How to document a command
```

```
usage: test [flags]

options:

- name: flag

  default_value: “true”

  usage: Some flag

- name: help

  shorthand: h

  default_value: “false”

  usage: help for test

example: ./main test
```

Additional customization options can be done to the generated output. Check the package documentation for more details [48].

16.4.3 Cobra generator

The Cobra library comes with a program for commands generation. If following Cobra conventions for commands, this generator can be helpful to fill commands with an initial template and add new ones. The following command lines initialize a project with a command named test.

```
 >>> $GOPATH/bin/cobra init —pkg-name github.com/juanmanuel-
tirado/savetheworldwithgo/15_cli/cobra/advanced/example_03

 >>> $GOPATH/bin/cobra add test
```

The Cobra generator creates the folders tree below containing a license, the main file, and two templates to define the root command and the test command we added.

```
example_03
├─ cmd
│  ├─ root.go
│  └─ test.go
├─ LICENSE
└─ main.go
```

You can run the cobra command help to check additional parameters to customize the templates for new commands, the license, or the author information.

16.4.4 Shell completion

Shell completion help users to interactively find the command name, arguments and flags of a CLI. Cobra generates shell completion scripts for Bash, Zsh, Fish and PowerShell. The `Command` type offers the methods `GetXXXCompletions` to generate the corresponding completion script for the shell `XXX`. Normally, a `completion` command is added to the CLI to permit users to generate the corresponding completion script for their shells. When the script is loaded into the shell, pressing the key tab twice displays the valid commands and the help. Example 16.12 shows a possible implementation of a completion command using the root command [49].

Example 16.12: Shell completion command.

```
package cmd

import (
    “os”
    “github.com/spf13/cobra”
)

var CompletionCmd = &cobra.Command{
   Use:   “completion [bash|zsh|fish|powershell]”,
    Short: “Generate completion script”,
    Long: “Load it into your shell for completions”,
    DisableFlagsInUseLine: true,
    ValidArgs: []string{”bash”, “zsh”, “fish”, “powershell”},
    Args: cobra.ExactValidArgs(1),
    Run: func(cmd *cobra.Command, args []string) {
       switch args[0] {
       case “bash”:
          cmd.Root().GenBashCompletion(os.Stdout)
       case “zsh”:
          cmd.Root().GenZshCompletion(os.Stdout)
       case “fish”:
          cmd.Root().GenFishCompletion(os.Stdout, true)
       case “powershell”:
          cmd.Root().GenPowerShellCompletionWithDesc(os.Stdout)
       }
    },
}
```

Assuming this command is already integrated into our CLI we can generate and load the shell completion script for Bash as follows.

```
>>> ./say completion bash > /tmp/completion

>>> source /tmp/completion
```

For the CLI from Example 16.7 these are the shell completions displayed for the root command. Notice that `[tab]` represents the tab key pressed.

```
>> ./say [tab][tab]

bye          — Say goodbye

completion   — Generate completion script

hello        — Say hello

help         — Help about any command
```

Command arguments can be displayed for additional help. A list of valid arguments can be provided with the `ValidArgs` field of `Command`. Our completion command has already filled this field showing the following list of valid arguments.

```
>> ./say completion [tab][tab]

bash         fish         powershell  zsh
```

In some scenarios, the arguments of command can only be determined at run time. For example, assume we have an application that queries the information of a certain user in a database using her identifier. The user id is only a valid argument if it exists in the database. For these scenarios, the list of valid arguments can be defined using a function in the field `ValidArgsFunction` like in Example 16.13. This Example emulates the availability of different users with a random selector in the `UserGet` function. The `ShellCompDirective` is a binary flag used to modify the shell behaviour. Check the documentation for more information about this flag and what it does.

Example 16.13: Dynamic definition of arguments for a command.

```
package main

import (
    “fmt”
    “github.com/juanmanuel-tirado/savetheworldwithgo/15_cli/cobra/advanced/example_05/cmd”
    “github.com/spf13/cobra”
    “os”
    “math/rand”
    “time”
)

var RootCmd = &cobra.Command{
    Use: “db”,
    Long: “Root command”,
}

var GetCmd = &cobra.Command{
    Use: “get”,
    Short: “Get user data”,
    Args: cobra.ExactValidArgs(1),
    DisableFlagsInUseLine: false,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Printf(”Get user %s!!!\n”,args[0])
    },
    ValidArgsFunction: UserGet,
}

func UserGet (cmd *cobra.Command, args []string, toComplete string) ([]string, cobra.ShellCompDirective) {
    rand.Seed(time.Now().UnixNano())
    if rand.Int() % 2 == 0 {
        return []string{”John”, “Mary”}, cobra.ShellCompDirectiveNoFileComp
    }
    return []string{”Ernest”, “Rick”, “Mary”}, cobra.ShellCompDirectiveNoFileComp
}
```

```
func init() {
   RootCmd.AddCommand(GetCmd, cmd.CompletionCmd)
}

func main() {
   if err := RootCmd.Execute(); err != nil {
      fmt.Fprintln(os.Stderr, err)
      os.Exit(1)
   }
}
```

After generating and loading the shell completion script, the completion dynamically suggests user ids with the `UserGet` function as shown below.

```
>>> ./db get [tab][tab]
John  Mary
>>> ./db [tab][tab]
completion  — Generate completion script
get         — Get user data
help        — Help about any command
>>> ./db get [tab][tab]
Ernest  Mary    Rick
```

16.5 SUMMARY

In this Chapter, we explore how to easily define command-line interfaces using the Cobra library. This library enables developers to define complex and adaptable solutions to a wide variety of use cases. It is really difficult to show examples for every potential use case or need you may find in your developments. However, the shown examples cover the basics and some advanced features that can push your CLI to the next level. Do not be afraid of checking the Cobra repository or how other large projects such as Kubernetes or Github implement their CLIs with this library.

CHAPTER 17

RELATIONAL DATABASES

Databases are basic components for any data-driven systems and the Structured Query Language (SQL) is the most common and accepted language for relational databases. Widely adopted solutions such as MySQL, PostgreSQL, Apache Hive, etc. use SQL to create, retrieve and manage relational data. Go offers an elegant and easy-to-use interface for SQL databases with a large variety of database drivers from different projects. This Chapter assumes the reader to have basic notions of relational databases (tables, entities, relationships, transactions, etc.). The purpose of this Chapter is not to cover all the topics of relational databases because that would require an entire book. This Chapter aims at offering the reader the foundations to understand how relational databases can be used in any Go project. With this purpose in mind, this Chapter explores how the Go standard library defines an interface for SQL databases and introduces GORM an object-relational mapping solution.

17.1 SQL IN GO

The package `database/sql`[50] defines a common interface for SQL databases. However, it does not implement any particular database driver. Specific drivers are expected to be imported from third-party projects maintaining the same interface for all of them. This permits to reuse of the same code independently of the underlying database. There can be certain differences depending on the employed database, although a large portion of the code can be reused.

17.1.1 Drivers and database connections

The `database/sql` package is built around the `DB` type[51]. This type handles a pool of database connections. A `DB` instance can be obtained opening a database connection with the `sql.Open` or `sql.OpenDB`functions.

Example 17.1 shows how to connect with a SQLite database[52]. First, we need to import a valid SQL driver. There is a large number of available drivers that can be checked in the documentation[53]. In our examples, we use a driver for SQLite [54] for simplicity. However, other drivers can be easily applied with minor changes. Observe that we only import the side effects of the package (line 8). By doing this, we register the driver and make it available for the standard library[55]. The `Open` function receives the name of the driver and the data source. For the current example, this is `“sqlite3”` and the file path of the database. For most use-cases, the data source is the database hostname with the username and database name. Check the documentation of the driver you employ to be sure you use the correct data source definition.

Example 17.1: Database connection.

```
1 package main
2
3 import (
4     “context”
5     “database/sql”
6     “fmt”
7     “time”
8     _ “github.com/mattn/go-sqlite3”
9 )
10
11 func main() {
12
13     db, err := sql.Open(“sqlite3”,”/tmp/example.db”)
14
15     if err != nil {
16         fmt.Println(err)
17         panic(err)
```

Database responds

```
    }
    defer db.Close()
    db.SetConnMaxLifetime(time.Minute * 3)
    db.SetMaxOpenConns(10)
    db.SetMaxIdleConns(10)

    ctx := context.Background()
    if err := db.PingContext(ctx); err == nil {
        fmt.Println("Database responds")
    } else {
        fmt.Println("Database does not respond")
        panic(err)
    }
}
```

The `DB` instance can be configured with some parameters such as the maximum connection lifetime, the maximum number of open connections or the maximum number of idle connections. If the `Open` function succeeds, this does not establish any connection with the database. To ensure the connection availability, `Ping` or `PingContext` methods can be invoked. These methods establish a connection if required and can be used to determine the current status of the target database. Finally, `DB` instances should be released using the `close` method. In this case, we use `defer` for a more proper design.

17.1.2 Modifying data statements

The Go standard library differentiates between queries that return rows of data and queries that modify the database. When modifying the database we do not expect any row to be returned as a result. Methods `Exec` and `ExecContext` execute queries to create, update, or insert new rows into a table. They return a `Result` type which contains the id of the latest inserted row when applies, and the number of rows affected by update, insert, or delete operations. Not every database may provide these values. Check the documentation before proceeding to operate with these values.

Example 17.2 assumes an existing `DB` connection and uses it to create a table and insert an entry. The `Exec` and `ExecContext` methods receive a query in a string and none or several arguments. These arguments are used to fill the query string. Observe that the main difference between the `create_table` and the `insert_rows` functions is that the second one uses the `Result` value returned from the query. If we get the same value when creating the table, this will be set to zero because it does not affect any existing row.

Example 17.2: Modifying data statements (excerpt).

```

const (
    USERS_TABLE=‘CREATE TABLE users(
    name varchar(250) PRIMARY KEY,
    email varchar(250)
)‘
    USERS_INSERT=“INSERT INTO users (name, email) VALUES(?,?)”
)

func create_table(db *sql.DB) {
    ctx := context.Background()
    _, err := db.ExecContext(ctx, USERS_TABLE)
    if err != nil {
        panic(err)
    }

}

```

```
Database responds

Row ID: 1, Rows: 1
```

```
func insert_rows(db *sql.DB) {
    ctx := context.Background()
    result, err := db.ExecContext(ctx, USERS_INSERT,
    “John”, “john@gmail.com”)
    if err != nil {
        panic(err)
    }
    lastUserId, err := result.LastInsertId()
    if err != nil {
        panic(err)
    }
    numRows, err := result.RowsAffected()
    if err != nil {
        panic(err)
    }
    fmt.Printf(“Row ID: %d, Rows: %d\n”, lastUserId, numRows)
}
```

The query syntax may vary depending on the target database. For example, the insert query used in the example has to be written differently for MySQL, PostgreSQL, and Oracle.

- MySQL: `INSERT INTO users (name, email)VALUES(?,?)`
- PostgreSQL: `INSERT INTO users (name, email)VALUES($1,$2)`
- Oracle: `INSERT INTO users (name, email)VALUES(:val1,:val2)`

17.1.3 Fetching data statements

When executing a query that returns one or several rows, typically a `SELECT`, the `QueryRow` and `Query` and their contextual variants `QueryRowContext` and `QueryContext` are used. These methods return a `*Rows` that permits to iterate through the returned rows and extract their values.

Example 17.3 selects all the rows from the users' table. To retrieve all the results from the query, we iterate until no more rows are available checking `rows.Next`. This sets the pointer to the next available row where the `Scan` method can extract the corresponding values. In this example, we have created a `User` to be filled with the returned values. Notice that the current query is equivalent to `SELECT name, email from users` so we can populate the `Name` and `Email` fields in that order otherwise, the order must be the same.

Example 17.3: Querying data statements (excerpt).

```go
type User struct {
    Name string
    Email string
}

func get_rows(db *sql.DB) {
    ctx := context.Background()
    rows, err := db.QueryContext(ctx, “SELECT * from users”)
    if err != nil {
        panic(err)
    }
    defer rows.Close()

    for rows.Next() {
        u := User{}
        rows.Scan(&u.Name,&u.Email)
        fmt.Printf(“%v\n”,u)
    }
}
```

```
Database responds
{John john@gmail.com}
{Mary mary@gmail.com}
```

For single-row queries the `Scan` method described above applies.

17.1.4 Transactions

A transaction consists of a set of operations that are executed in a context and all of them must be successful otherwise, a rollback resets the database to the state previous to the start of the transaction. For Go, a transaction allocates a database connection until the transaction is finished. Transactions offer similar methods for the data modifier and data retriever scenarios we have already seen. However, these methods are provided by a `Tx` type that represents the transaction.

Example 17.4 inserts one row into the users' table inside a transaction. Observe how the `ExecContext` method occurs inside the transaction. The `rollback` is deferred and will only be executed in case of an error occurs. The `Commit` method triggers the execution of all the queries contained within the transaction.

Example 17.4: Transaction modifying database (excerpt).

```
func runTransaction(db *sql.DB) {
    tx, err := db.BeginTx(context.Background(), nil)
    defer tx.Rollback()

    ctx := context.Background()
    _, err = tx.ExecContext(ctx, “INSERT INTO users(name, email) VALUES(?,?)”,
        “Peter”, “peter@email.com”)
    if err != nil {
        panic(err)
    }
    err = tx.Commit()
    if err != nil {
        panic(err)
    }
}
```

17.1.5 Prepared statements

Every time a query is sent to a database server this is parsed, analyzed and processed accordingly. This is an expensive operation, especially for queries that are expected to be run several times. Prepared statements permit servers to process a query in such a way that subsequent executions of the same query do not have to be parsed again.

A prepared statement `Stmt` is associated with a `DB` or `Tx` instance. Example 17.5 shows a statement to retrieve a single row matching a user name. The `Prepare` method instantiates the statement that is ready to be used if no errors were found. Observe that statements are associated with a connection and they have to be released using the `Close` method.

Example 17.5: Prepared statement (excerpt).

```
type User struct {
    Name string
    Email string
}

func runStatement(db *sql.DB) {
    stmt, err := db.Prepare(“select * from users where name = ?”)
    if err != nil {
        panic(err)
    }
    defer stmt.Close()
    result := stmt.QueryRow(“Peter”)
    u := User{}

```

Database responds

{Peter peter@email.com}

```
    err =  result.Scan(&u.Name, &u.Email)
    switch {
    case err == sql.ErrNoRows:
        panic(err)
    case err != nil:
        panic(err)
    default:
        fmt.Printf(“%v\n”,u)
    }
}
```

17.2 GORM

GORM[56] is an ORM[57] library for Go. Like any other ORM, GORM offers an abstraction layer that translates operations done with structs to SQL. It supports MySQL, PostgreSQL, SQLite, SQL Server, and Clickhouse. For simplicity, the examples assume SQLite to be available in your environment.

17.2.1 Basics

GORM provides its own database drivers to be used. In our case, we use the SQLite driver available at package `gorm.io/driver/sqlite`. For information about other supported databases check the documentation[58]. First, we have to initialize a session to instantiate a pool of connections represented by type `DB`. In Example 17.6 we use the SQLite driver to initialize a database at path `/tmp/test.db`.

Example 17.6: Basic GORM program.

```
package main

import (
```

```
Recovered {1 John
john@gmail.com}
```

After update {1 John newemail}

```
    “fmt”
    “gorm.io/gorm”
    “gorm.io/driver/sqlite”
)

type User struct {
    ID uint
    Name string
    Email string
}

func main() {
    db, err := gorm.Open(sqlite.Open(“/tmp/test.db”), &gorm.Config{})
    if err != nil {
        panic(“failed to connect database”)
    }

    err = db.AutoMigrate(&User{})
    if err != nil {
        panic(err)
    }

    u := User{Name: “John”, Email: “john@gmail.com”}
    db.Create(&u)

    var recovered User
    db.First(&recovered,“name=?”,“John”)
    fmt.Println(“Recovered”, recovered)

    db.Model(&recovered).Update(“Email”,“newemail”)
    db.First(&recovered,1)
    fmt.Println(“After update”, recovered)

    db.Delete(&recovered, 1)
}
```

Next, we have to inform GORM about our data schema. In this case, we define the `User` type with a unique ID, the user name, and her email. With `db.AutoMigrate` GORM translates `User` into a valid representation for our target database. Once the migration is correctly done, create, read, update, and delete operations (CRUD) can be performed by GORM with none or minimal SQL code. With the current SQLite configuration the migration will create the corresponding tables. In our case a table named users with the following parameters:

field	type
id	integer
name	text
email	text

Notice that the name of the destination table has been automatically generated and managed by GROM. In the Example, we use the `u` variable to populate a new entry in the database. Next, we recover the entry from the database. Observe that when recovering the entry (`db.Read`) we use different approaches. The first uses SQL notation indicating the field and the value to be queried. In the second version, we simply indicate the primary key number which corresponds to the `ID` field of type `User`. By default GORM sets any ID field to be the primary key if not stated otherwise.

17.2.2 Modelling entities

In GORM a model is a struct with basic Go types. Some conventions are applied to field names. By default, GORM assumes any field `ID` to be the primary key. Additionally, `CreatedAt` and `UpdatedAt` fields are used to track manipulation times. The struct below is an example with the `ID` as the primary key and the time tracking fields.

```go
type User struct {
   ID uint
   Name string
   Email string
   CreatedAt time.Time
   UpdatedAt time.Time
}
```

Actually, GORM defines the struct gorm.Model as follows:

```go
type Model struct {
   ID        uint           ‘gorm:“primaryKey”‘
   CreatedAt time.Time
   UpdatedAt time.Time
   DeletedAt gorm.DeletedAt ‘gorm:“index”‘
}
```

This struct can be embedded to automatically extend row metadata. Structs embedding the `gorm.Model` can automatically have the `ID` as the primary key and the corresponding time tracking fields.

In the context of database tables, embedding means that all the information from a struct has to be contained in a column value. GORM permits embedding any struct that implements the `Scanner` and `Valuer` interfaces from the `database/sql` package[59]. The example below defines the `DevOps` struct to be a user with an associated operator. In this case, the embedded tag indicates the `User` field to be embedded into `Operator`.

```
type User struct {
   Name string
   Email string
}

type Operator struct {
   ID uint
   User User ‘gorm:“embedded”‘
   Platform string ‘gorm:“not null”‘
}
```

Observe that we have additionally used the `“not null”` tag to indicate that the field `Platform` cannot be empty in the database. The complete list of field tags can be found in the documentation[60]. Example 17.7 shows the code that permits struct `User` to be embedded. To serialize the struct we use JSON encoding (see Section 8.2) to marshal the struct value in the `Value` method. In the `Scan` method we use the JSON unmarshal to populate the struct. Notice that the `Scan` method has the `*User` type as receiver to ensure that we modify the original value. Additional tag constraints on the `Dedication` field are set to demonstrate how they can be used. When this field is set to values lower than five the create query fails.

Example 17.7: Embedded structs in GORM.

```
Created {1 {John john@gmail.com} k8s 10}
Recovered {1 {John john@gmail.com} k8s 10}
```

```
package main

import (
    “database/sql/driver”
    “encoding/json”
    “fmt”
    “gorm.io/driver/sqlite”
    “gorm.io/gorm”
)

 type User struct {
     Name string
     Email string
 }

 func(u *User) Scan(src interface{}) error {
     input := src.([]byte)
     json.Unmarshal(input,u)
     return nil
 }
 func(u User) Value()(driver.Value, error) {
     enc, err := json.Marshal(u)
     return enc,err
 }

 type Operator struct {
     ID uint
     User User ‘gorm:“embedded,embeddedPrefix:user_”‘
     Platform string ‘gorm:“not null”‘
     Dedication uint ‘gorm:“check:dedication>5”‘
 }

 func main() {
     db, err := gorm.Open(sqlite.Open(“/tmp/example02.db”), &gorm.Config{})
     if err != nil {
         panic(“failed to connect database”)
     }

     err = db.AutoMigrate(&Operator{})
     if err != nil {
         panic(err)
     }

     op := Operator{
         User: User{
             Name:  “John”,
             Email: “john@gmail.com”,
         },
         Platform: “k8s”,Dedication:10,
     }
     db.Create(&op)
     fmt.Println(“Created”, op)
     var recovered Operator
     db.First(&recovered,1)
     fmt.Println(“Recovered”, recovered)
 }
```

17.2.3 Relationships

GORM maps relationships between SQL entities using struct fields and some additional field tags. The schema migration generates the corresponding foreign keys and additional tables required to support these relationships. We explore four potential relationships.

Belongs-to

This relationship sets an entity to belong to another entity. This is a one-to-one relationship. For example, a user belongs to a group. In GORM we can represent this relationship indicating the foreign key and the entity. In the Example below a `User` belongs to one `Group`. This is indicated with the `GroupID` field which is the foreign key pointing to the corresponding entity group in the database, and the field `Group`.

Example 17.8: Belongs to relationship.

```go
type User struct {
    ID uint
    Name string
    Email string
    GroupID uint
    Group Group
}

type Group struct {
    ID uint
    Name string
}
```

GORM analyzes field names to find relationships and set foreign key names and target entities. If the default behaviour has to be changed, this can be done using field tags.

Example 17.9: Belongs to relationship with custom foreign key.

```go
type User struct {
    ID uint
    Name string
    Email string
    Ref uint
    Group Group ‘gorm:“foreignKey:Ref”‘
}

type Group struct {
    ID uint
    Name string
}
```

The example above changes the foreign key referencing the group to be the field `Ref` instead of the default `GroupID`.

Has-one

A has-one relationship indicates that an entity has a reference to another entity. This is similar to a belongs-to relationship in the sense that both are one-to-one relationships. However, there is a semantic difference depending on the side the relationship is observed. In the Example below, we have a user who has a laptop. In this case, the foreign key is in the laptop entity pointing to the user.

Example 17.10: Has one relationship.

```
type User struct {
    ID uint
    Name string
    Email string
    Laptop Laptop
}
type Laptop struct {
    ID uint
    SerialNumber string
    UserID uint
}
```

Has-many

A has-many relationship permits an entity to be referred to by several instances as their owner. In the Example below, we declare that a user can have several laptops. Notice that to follow GORM conventions, we named a field `Laptops` to contain an array of `Laptop` type. This is translated into a table named laptops with a `user_id` column as a foreign key pointing at the corresponding user.

Example 17.11: One to many relationship.

```
type User struct {
    ID uint
    Name string
    Email string
    Laptops []Laptop
}
type Laptop struct {
    ID uint
    SerialNumber string
     UserID uint
}
```

Many-to-many

This is the most complex kind of relationship where there is no restriction in the number of entities that can be referenced. In this Example, we have that a user can speak many languages and a language can be spoken by many users. This many-to-many relationship has to be represented using a join table. The field tag `many2many` is used to define the name of the join table to be used.

Additionally, we have that laptops are shared by users. This means that a user can use many laptops and a laptop can be used by many users. In this case, `Laptops` and `Users` fields are arrays that have to be found in a many2many table. Observe, that the array items are pointers. This differs from the user speaking many languages because the language is meaningful on its own and does not belong to any user.

Example 17.12: Many to many relationship.

```
type User struct {
    ID uint
    Name string
    Email string
    Languages []Language ‘gorm:“many2many:user_languages”‘
    Laptops []*Laptop ‘gorm:“many2many:user_laptops”‘
}
type Language struct {
    ID uint
    Name string
}
type Laptop struct {
    ID uint
    SerialNumber string
    Users []*User ‘gorm:“many2many:user_laptops”‘
}
```

17.3 MANIPULATE DATA

The GORM `DB` type is the main entry to the database and the data models. It controls all the SQL statements to be generated to create, read, update, or delete instances.

Create

The `db.Create` methods receives any object with a migrated schema and writes a new record with the information it contains. Fields corresponding to autoincrement values such as the `ID` from Example 17.13 are set. The `Create` method returns a `*DB` that contains any error found during the operation and the number of rows affected by the operation. In the case of creating several records, the `CreateInBatches` method optimizes the process by generating SQL statements with a given number of records (5 in the Example).

Example 17.13: GORM creation of records.

```
package main

import (
    “fmt”
    “gorm.io/driver/sqlite”
    “gorm.io/gorm”
)

type User struct {
     ID uint
     Name string
}

func main() {
     db, err := gorm.Open(sqlite.Open(“/tmp/example01.db”), &gorm.Config{})
     if err != nil {
         panic(“failed to connect database”)
     }

```

```
User ID: 1, rows:
1
```

```
    err = db.AutoMigrate(&User{})
    if err != nil {
        panic(err)
    }

    u := User{Name: “John”}
    res := db.Create(&u)
    fmt.Printf(“User ID: %d, rows: %d\n”,u.ID,res.RowsAffected)

    users := []User{{Name:“Peter”},{Name:“Mary”}}
    for _,i := range users {
        db.Create(&i)
    }
    db.CreateInBatches(users,5)
}
```

Query

GORM has a fully functional interface to perform SQL queries. Exploring all the available combinations is far away from the purpose of this Chapter. A complete explanation of every supported query is available at the official documentation[61]. Examples 17.14 and 17.15 are fragments of the same code showing a collection of common SQL queries.

The first Example 17.14 populates the database with some users and shows how individual users can be found. `First`, `Take`, and `Last` return one single record depending on the primary key. `First` and `Last` return the first and last entries while `Take` has no associated order. Notice that between every query we set a new `u` value to reuse the variable. Otherwise, GORM interprets the fields in `u` to be the arguments of the query. If the first query returns the value where the name is John, invoking `db.Last(u)` will return the same record because it is the latest id with the indicated name.

Example 17.14: GORM queries (excerpt).

```
package main

import (
    “fmt”
    “gorm.io/driver/sqlite”
    “gorm.io/gorm”
)

type User struct {
    ID uint
    Name string
}

func main() {
    db, err := gorm.Open(sqlite.Open(“/tmp/example02.db”), &gorm.Config{})
    if err != nil {
        panic(“failed to connect database”)
    }

    err = db.AutoMigrate(&User{})
    if err != nil {
        panic(err)
    }
    users := []User{{Name: “John”},{Name: “Mary”},{Name: “Peter”},{Name:“Jeremy”}}
    db.CreateInBatches(users,4)

    var u User
```

```
28      db.First(&u)
29      fmt.Println(“First”,u)
30      u=User{}
31      db.Take(&u)
32      fmt.Println(“Take”,u)
33      u=User{}
34      db.Last(&u)
35      fmt.Println(“Last”,u)
36      u=User{}
37      db.First(&u,2)
38      fmt.Println(“First ID=2”,u)
```

```
First {1 John}
Take {1 John}
Last {4 Jeremy}
First ID=2 {2 Mary}
```

Example 17.15 shows how to retrieve several records and use additional clauses. The `db.Find` method returns all the records of a table. Additional filters can be set like using primary keys (line 42). Conditional queries using `Where` plus `First` or `Find` permit to be more specific about the queried record and the number of returned entries. Observe that `Where` uses the syntax of a where SQL statement without setting the arguments. The `?` mark in the where clause is replaced by the arguments. Finally, mention that `Where` accepts types from the schema as search arguments (line 52).

Example 17.15: GORM queries (continuation).

```
39      var retrievedUsers []User
40      db.Find(&retrievedUsers)
41      fmt.Println(“Find”,retrievedUsers)
```

```
    db.Find(&retrievedUsers,[]int{2,4})
    fmt.Println("Find ID=2,ID=4",retrievedUsers)
    u=User{}
    db.Where("name = ?","Jeremy").First(&u)
    fmt.Println("Where name=Jeremy",u)
    db.Where("name LIKE ?","%J%").Find(&retrievedUsers)
    fmt.Println("Where name=%J%",retrievedUsers)
    db.Where("name LIKE ?","%J%").Or("name LIKE ?","%y").Find(&retrievedUsers)
    fmt.Println("Name with J or y",retrievedUsers)
    u=User{}
    db.Where(&User{Name: "Mary"}).First(&u)
    fmt.Println("User with name Mary",u)
    db.Order("name asc").Find(&retrievedUsers)
    fmt.Println("All users ordered by name",retrievedUsers)
}
```

```
Find [{1 John} {2 Mary} {3 Peter} {4 Jeremy}]
Find ID=2,ID=4 [{2 Mary} {4 Jeremy}]
Where name=Jeremy {4 Jeremy}
Where name=%J% [{1 John} {4 Jeremy}]
Name with J or y [{1 John} {2 Mary} {4 Jeremy}]
User with name Mary {2 Mary}
All users ordered by name [{4 Jeremy} {1 John} {2 Mary} {3 Peter}]
```

Eager loading

GORM has eager loading for relations. This feature permits programmatically set queries to be executed before others. This can be easily demonstrated with an example. In Example 17.16, we have a one-to-many relationship where a user can have several laptops. If we retrieve a user from the database using `db.First`, we can observe that the returned value has no laptops. These entries are already in the database. However, they are in a different table and that requires an additional query. Adding the eager preloading with `db.Preload(“Laptops”)` indicates to GORM that there are laptops to be queried to populate the resulting record. The output does contain the `Laptops` field populated.

Example 17.16: GORM preload.

```
package main

import (
    “fmt”
    “gorm.io/driver/sqlite”
    “gorm.io/gorm”
)

type User struct {
    ID uint
    Name string
    Email string
    Laptops []Laptop
}

type Laptop struct {
    ID uint
    SerialNumber string
    UserID uint
}

```

```
func main() {
    // SQLite does not support foreign key constraints
    db, err := gorm.Open(sqlite.Open("/tmp/example03.db"),
        &gorm.Config{DisableForeignKeyConstraintWhenMigrating: true,})

    if err != nil {
        panic("failed to connect database")
    }

    err = db.AutoMigrate(&User{},&Laptop{})
    if err != nil {
        panic(err)
    }

    laptops := []Laptop{{SerialNumber: "sn0000001"},{SerialNumber: "sn0000002"}}
    u := User{
        Name:    "John",
        Email:   "john@gmail.com",
        Laptops: laptops,
    }
    db.Create(&u)
    fmt.Println("Created", u)
    var recovered User
    db.First(&recovered)
    fmt.Println("Recovered without preload",recovered)
    recovered = User{}
    db.Preload("Laptops").First(&recovered)
    fmt.Println("Recovered with preload", recovered)
}
```

```
Created {2 John john@gmail.com [{3 sn0000001 2} {4 sn0000002 2}]}

Recovered without preload {1 John john@gmail.com []}

Recovered with preload {1 John john@gmail.com [{1 sn0000001 1} {2 sn0000002 1}]}
```

There is something important to highlight in the manner the eager loading works. The number of items to be preloaded must be correctly indicated. The call `Preload("Laptops")` expects several entries. If only one entry is going to be loaded this should be `Preload("Laptop")`. Notice the utilization of the plural. Example 17.17 defines a belongs-to relationship, where a user belongs to a group. Notice that in this case, we only expect to have one group for the user. This is mentioned in the `db.Preload("Group")` statement.

Example 17.17: GORM preload with single record.

```
package main

import (
    "fmt"
    "gorm.io/driver/sqlite"
    "gorm.io/gorm"
)

type User struct {
     ID uint
     Name string
     GroupID uint
     Group Group
}
```

```
Recovered {1 John 1 {1
TheCoolOnes}}
```

```

type Group struct {
    ID uint
    Name string
}

func main() {
    db, err := gorm.Open(sqlite.Open("/tmp/example04.db"), &gorm.Config{})
    if err != nil {
        panic("failed to connect database")
    }

    err = db.AutoMigrate(&Group{},&User{})
    if err != nil {
        panic(err)
    }

    g := Group{Name: "TheCoolOnes"}
    u := User{Name: "John", Group: g}
    db.Create(&u)

    var recovered User
    db.Preload("Group").First(&recovered,1)
    fmt.Println("Recovered", recovered)
}
```

17.3.1 Transactions

By default, GORM executes every writes operation inside a transaction clause. This can be disabled as indicated in the documentation by setting the `SkipDefaultTransaction` flag[62]. This brings relevant performance improvement. Similarly to the `database/sql` from the standard library, GORM can define transactional environments that rollback if something goes wrong.

Example 17.18 uses a transaction to update the email of an existing user. Observe that the operations running in the transactional context are inside a function. This function receives a connection `tx` that can execute the operations inside the transaction. The transaction will fail on purpose when trying to recreate an already existing user. The rollback operation restores the previous state. We can check this by retrieving the user after the transaction.

Example 17.18: GORM transaction.

```
package main

import (
    “fmt”
    “gorm.io/driver/sqlite”
    “gorm.io/gorm”
)

type User struct {
    ID uint
    Name string
    Email string
}
func main() {
    db, err := gorm.Open(sqlite.Open(“/tmp/example01.db”), &gorm.Config{})

    if err != nil {
        panic(“failed to connect database”)
    }

    err = db.AutoMigrate(&User{})
    if err != nil {
        panic(err)
    }

    u := User{Name: “John”, Email: “john@gmail.com”}
    db.Create(&u)

    db.Transaction(func(tx *gorm.DB) error {
        if err := tx.Model(&u).Update(“Email”,“newemail”).Error; err != nil {
            return err
        }
        var inside User
        tx.First(&inside)
        fmt.Println(“Retrieved inside transaction”, inside)
        if err := tx.Create(&u).Error; err != nil {
            return err
        }
        return nil
    })
    var retrieved User
    db.First(&retrieved)
    fmt.Println(“Retrieved”, retrieved)
}
```

```
Retrieved inside transaction {1 John newemail}

…github.com/juanmanuel-
tirado/savetheworldwithgo/16_sql/gorm/transactions/example_01/main.go:33 UNIQUE
constraint failed: users.id

[0.040ms] [rows:0] INSERT INTO ‘users‘ (‘name‘,‘email‘,‘id‘) VALUES
(”John”,”newemail”,1)

Retrieved {1 John john@gmail.com}
```

Transactions can be controlled manually using `Begin`, `Commit`, and `RollBack` similarly to how it is done in the standard library. Example 17.19 is a similar situation to the one in the previous Example. However, we add a savepoint to avoid losing the changes done to the user record. The savepoint is set after the record update and we rollback to that savepoint when the create operation fails. Finally, we execute the commit to finish the transaction.

Example 17.19: GORM manual transaction.

```go
package main

import (
    “fmt”
    “gorm.io/driver/sqlite”
    “gorm.io/gorm”
)

type User struct {
    ID uint
    Name string
    Email string
}

```

```
func RunTransaction(u *User, db *gorm.DB) error{
    tx := db.Begin()
    if tx.Error != nil {
        return tx.Error
    }
    if err := tx.Model(u).Update(“Email”,“newemail”).Error; err != nil {
        return err
    }
    tx.SavePoint(“savepoint”)
    if err := tx.Create(u).Error; err != nil{
        tx.RollbackTo(“savepoint”)
    }
    return tx.Commit().Error
}

func main() {
    db, err := gorm.Open(sqlite.Open(“/tmp/example02.db”), &gorm.Config{})

    if err != nil {
        panic(“failed to connect database”)
    }

    err = db.AutoMigrate(&User{})
    if err != nil {
        panic(err)
    }

    u := User{Name: “John”, Email: “john@gmail.com”}
    db.Create(&u)

    err = RunTransaction(&u, db)
    if err != nil {
        fmt.Println(err)
    }
    var retrieved User
    db.First(&retrieved)
    fmt.Println(“Retrieved”, retrieved)
}
```

```
…/github.com/juanmanuel-
tirado/savetheworldwithgo/16_sql/gorm/transactions/example_02/main.go:24 UNIQUE
constraint failed: users.id

[0.040ms] [rows:0] INSERT INTO ‘users‘ (‘name‘,‘email‘,‘id‘) VALUES
(”John”,”newemail”,1)

Retrieved {1 John newemail}
```

17.4 SUMMARY

This Chapter brings the reader the tools to understand how SQL databases can be used in any Go project. Managing databases is a topic itself that would require a complete book to cover all the details and potential situations the developer may find. This is why this Chapter simply offers a detailed explanation of tools provided by the Go standard library for SQL and additionally introduces a powerful object-relational mapping solution such as GORM. The examples and explanations given in this Chapter bring the reader familiar with SQL concepts a strong background to easily embrace SQL solutions in her Go projects.

CHAPTER 18

NOSQL DATABASES

The adoption of big data solutions accelerated the exploration of new database models. This exploration exposed the limitations of classic relational databases and proposed new models beyond the Standard Query Language and the entity-relationship model. This Chapter is focused on the Cassandra NoSQL database and the available Go drivers.

18.1 CASSANDRA AND GOCQL

The Apache Cassandra database[63] is a NoSQL distributed database solution. It has been adopted in scenarios demanding high performance and availability, and it has demonstrated linear scalability while ensuring fault tolerance. In case the reader is not familiar with Cassandra, let these lines partially illustrate what are the main differences between this database and classic relational databases such as MySQL or PostgreSQL.

- **Keyspaces** contain tables and define several properties such as replication policies.
- **Tables** like in SQL databases, store collections of entries. However, conceptually a table only defines the schema to be stored. A table is instantiated across several nodes with several partitions.
- **Partition** defines the mandatory part of the primary key all rows must have.
- **Rows** similarly to SQL databases are a collection of columns identified by a unique primary key and optional clustering keys.
- **Column** is a typed datum that belongs to a row and is defined by the schema of a table.

Cassandra uses a language similar to SQL named CQL. The syntax of CQL is fairly similar to SQL but keeping the particularities of Cassandra. The complete definition of the CQL language is available at the official Cassandra documentation page[64].

Like major databases, Cassandra has available drivers in Go. In this Section, we explore the utilization of the GoCQL[65] driver. This driver provides a straight forward solution to connect to Cassandra databases and execute CQL statements for officially supported Cassandra versions[66]. The examples shown in this Section assume an out of the box Cassandra instance listening at `localhost:9042`. Most examples assume the `example` keyspace to be already available.

The GoCQL project is very dynamic and the examples may not be fully compatible with your current version of Cassandra by the time you execute them. Hopefully, the reader may find enough information in these examples to overcome possible execution errors.

18.1.1 Database connection

Like any other database driver, GoCQL works with sessions that provide the user with the connections required to send queries and manipulate data. A `Session` can be generated by the `ClusterConfig` type through the `CreateSession` method. The `ClusterConfig` type represents the configuration to be used to instantiate new connections with a cluster.

Example 18.1 shows how to instantiate a new session for a local cluster. The `NewCluster` function returns a `ClusterConfig` type that can be customized. Check the documentation to see all the possible configurable items [67]. In the Example, we set the keyspace to be used, the consistency level, and a one minute connection timeout. Several hosts can be passed by argument and the rest of the nodes will be automatically discovered. Note that the creation of the session can fail and it has to be closed to release resources.

Example 18.1: GoCQL database connection.

```
package main

import (
   “context”
    “github.com/gocql/gocql”
    “time”
)

const CREATE_TABLE=‘CREATE TABLE example.user (
 id int PRIMARY KEY,
 name text,
 email text
 )
 ‘

 func main() {
     cluster := gocql.NewCluster(“127.0.0.1:9042”)
     cluster.Keyspace = “example”
     cluster.Consistency = gocql.Quorum
     cluster.Timeout = time.Minute
     session, err := cluster.CreateSession()
     if err != nil {
         panic(err)
     }
     defer session.Close()

     ctx := context.Background()
     err = session.Query(CREATE_TABLE).WithContext(ctx).Exec()
     if err != nil {
         panic(err)
     }
```

```
32 }
```

For the sake of demonstration, we create a table to check that everything is OK. If we use the command line CQL shell (`cqlsh`), we will see the new table.

```
cqlsh> DESCRIBE example.user ;

CREATE TABLE example.user (
   id int PRIMARY KEY,
   email text,
   name text
) WITH bloom_filter_fp_chance = 0.01
   AND caching = ’KEYS_ONLY’
   AND comment = ’’
   AND compaction = {’class’:
’org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy’}
   AND compression = {’sstable_compression’:
’org.apache.cassandra.io.compress.LZ4Compressor’}
   AND dclocal_read_repair_chance = 0.1
   AND default_time_to_live = 0
   AND gc_grace_seconds = 864000
   AND index_interval = 128
   AND memtable_flush_period_in_ms = 0
   AND populate_io_cache_on_flush = false
   AND read_repair_chance = 0.0
   AND replicate_on_write = true
    AND speculative_retry = ’99.0PERCENTILE’;
```

18.1.2 Modelling

Data modelling in Cassandra differs from traditional relational databases.

Using modelling concepts from traditional databases will probably bring bad results. The rules for data modelling in Cassandra[6] clearly state that the goals to achieve when modelling data are 1) evenly data spread around clusters and 2) minimize the number of partitions. This refers to the fact that writes in Cassandra are cheap and disk space is not usually a problem. The main goals are more related to the current Cassandra architecture and how it can help to achieve incredibly efficient and scalable queries. Data modelling must be done keeping in mind how the data is going to be processed.

This Section is intended to show examples of how data relationships can be modelled using Cassandra. The solutions given here are a matter of discussion and only pretend to introduce the questions to be made during modelling and present the particularities of Cassandra.

Has-one

Imagine a users database where every user has a laptop. In a relational database, we could have two differentiated entities, user and laptop and link the laptop with its owner. However, in Cassandra this is not the case. The best solution depends on the final utilization of these entities. Assume we have the laptops and users tables defined as follows:

Example 18.2: Users and laptops entities.

```
CREATE TABLE users (
    user_id int PRIMARY KEY,
    name text,
    email text);
CREATE TABLE laptops (
    sn int PRIMARY KEY,
    model text,
    memory int);
```

One solution is to create an additional table that stores the relationship between a user and its laptop. The `user_laptop` table shown below sets the `user_id` as the primary key, then a user can only appear once.

Example 18.3: A user has a laptop relationship using an extra table.

```
CREATE TABLE user_laptop (
    user_id int PRIMARY KEY,
    sn int);
```

The solution above is suitable if we want to periodically get the list of users with their laptops. However, it permits a laptop to be linked with several users. Furthermore, it assumes that the laptop entity is important enough to exist on its own. However, if a laptop entry is always associated with a user we can define a laptop type and associate it with the user.

Example 18.4: A user has a laptop relationship using a laptop type.

```
CREATE TYPE laptop (
    sn int,
    model text,
    memory int);
CREATE TABLE users (
    user_id int PRIMARY KEY,
    name text,
    email text,
    laptop frozen<laptop>);
```

Now laptops are a type associated with users. The contained information is the same but it will be an attribute of a user. We use the `frozen` modifier to indicate that the individual fields of a laptop cannot be modified independently actually, the laptop field has to be reinserted as a whole. However, this solution does not guarantee a laptop to be owned by a single user. If we must guarantee this situation from a database perspective we can maintain the laptop table and include a user id.

Example 18.5: Guarantee that every laptop has a unique user.

```
CREATE TABLE users (
    user_id int PRIMARY KEY,
    name text,
    email text);
CREATE TABLE laptops (
    sn int PRIMARY KEY,
    model text,
    memory int,
    user_id int);
```

Has-many

Extending our previous example with users and laptops, let’s assume that now every user can have several laptops. Using the `laptop` type we can set every user to have a set of laptops.

Example 18.6: A user has many laptops using collections.

```
CREATE TABLE users (
    user_id int PRIMARY KEY,
    name text,
    email text,
    laptops set<frozen<laptop>>);
```

Using the set collection guarantees that the same laptop cannot appear twice in the same users and querying the laptops of every user would be straight forward. However, the same laptop can appear in two different users. To guarantee that there is only one user per laptop we can maintain the `laptop` table with the `user_id`.

Example 18.7: Guarantee that every laptop has a unique user.

```
CREATE TABLE laptops (
    sn int PRIMARY KEY,
    model text,
    memory int,
    user_id int);
```

Note that the primary key is used to partition the table across the cluster. If we query all the laptops of a user this query could be extended across the cluster to just return a few records. However, this would be a good solution to find the user who owns a laptop.

Many-to-many

Now let’s assume that laptops can be shared by many users and we have two potential queries: getting the laptops of a user and getting the users of a laptop. We can create two tables to facilitate both queries.

Example 18.8: Users and laptops in a many to many relationship.

```
CREATE TABLE user_laptops (
    user_id int,
    sn int,
    PRIMARY KEY (user_id,sn));
CREATE TABLE laptops_user (
    sn int,
    user_id int,
    PRIMARY KEY (sn,user_id));
```

Using a compound primary key `(user_id,sn)` partitions by the user and then inside the partition, the rows are ordered using the laptop serial number. Querying the laptops of a user with `SELECT * from user_laptops where user_id=?` is intended to be particularly efficient by hitting a single partition.

18.1.3 Manipulate data

Data manipulation operations create, update, insert, and delete are executed using the `Exec` method form the `Query` type. The `Query` type can be customized through different methods[68] that help to define the current behaviour of the query. Example 18.9 create the users' table and inserts a single row. Note that for simplicity, we can define the queries as constant strings to be reused. The `CREATE TABLE` query has no arguments, while the `INSERT` query waits for three arguments that are passed when creating the query. For additional control, we can execute the queries inside a context to, for example, limit the waiting time.

Example 18.9: GoCQL data manipulation queries.

```
package main

import (
    “context”
    “github.com/gocql/gocql”
)

const CREATE_TABLE=‘CREATE TABLE users (
id int PRIMARY KEY,
name text,
email text)‘

const INSERT_QUERY=‘INSERT INTO users
(id,name,email) VALUES(?,?,?)‘

func main() {
    cluster := gocql.NewCluster(“127.0.0.1:9042”)
    cluster.Keyspace = “example”
    session, err := cluster.CreateSession()
    if err != nil {
        panic(err)
    }
    defer session.Close()

    ctx := context.Background()
    err = session.Query(CREATE_TABLE).WithContext(ctx).Exec()
    if err != nil {
        panic(err)
    }

    err = session.Query(INSERT_QUERY,1, “John”,
“john@gmail.com”).WithContext(ctx).Exec()
    if err != nil {
        panic(err)
    }
}
```

CQL can execute insert, update, and delete queries in batches. Batches ensure atomicity and can improve performance. Example 18.10 uses a batch to insert a new user and modify its email. A `Batch` type is obtained from the current session with a given context using the `NewBatch` method. This batch has a collection of `BatchEntry` elements representing queries. The collection of queries is executed as a whole invoking the `ExecuteBatch` method from the current session.

Example 18.10: GoCQL data manipulation using batches.

```
package main

```

```
import (
    “context”
    “github.com/gocql/gocql”
)

const CREATE_TABLE=‘CREATE TABLE users (
id int PRIMARY KEY,
name text,
email text)‘

const INSERT_QUERY=‘INSERT INTO users
(id,name,email) VALUES(?,?,?)‘

const UPDATE_QUERY=‘UPDATE users SET email=? WHERE id=?‘

func main() {
    cluster := gocql.NewCluster(“127.0.0.1:9042”)
    cluster.Keyspace = “example”
    session, err := cluster.CreateSession()
    if err != nil {
        panic(err)
    }
    defer session.Close()

    ctx := context.Background()
    err = session.Query(CREATE_TABLE).WithContext(ctx).Exec()
    if err != nil {
        panic(err)
    }

    b := session.NewBatch(gocql.UnloggedBatch).WithContext(ctx)
    b.Entries = append(b.Entries,
        gocql.BatchEntry {
            Stmt: INSERT_QUERY,
            Args: []interface{}{1, “John”, “john@gmail.com”},
        },
        gocql.BatchEntry {
            Stmt: UPDATE_QUERY,
            Args: []interface{}{“otheremail@email.com”,1},
        })
    err = session.ExecuteBatch(b)
    if err != nil {
        panic(err)
    }
}
```

Batch queries are particularly suitable when the target of the query is a single partition. If more than one partitions are involved in the batch, this will impact the performance. One possible reason to use batches when multiple partitions are involved could be the need to ensure the modification of two related tables. For example, changing a user email and its corresponding entry in a table using this email. Find more about good practices for batches in the CQL documentation[69].

18.1.4 Queries

GoCQL differentiates between queries that return a single result or multiple results. For queries returning a single result, we can use the `Scan` method passing by argument the destination variables to be populated with the returned results. These variables must have a type compatible with the columns returned. Example 18.11 shows a query for a single entry returning the name of the user.

Example 18.11: GoCQL single result query.

```
package main

import (
    “context”
    “fmt”
    “github.com/gocql/gocql”
)

const (
    QUERY        =“SELECT name FROM users WHERE id=1”
    CREATE_TABLE =‘CREATE TABLE users (
id int PRIMARY KEY,
name text,
email text
)‘
     INSERT_QUERY =‘INSERT INTO users
(id,name,email) VALUES(?,?,?)‘
)

func main() {
     cluster := gocql.NewCluster(“127.0.0.1:9042”)
     cluster.Keyspace = “example”
     session, err := cluster.CreateSession()
```

```
Retrieved name
John
```

```
    if err != nil {
        panic(err)
    }
    defer session.Close()

    ctx := context.Background()
    err = session.Query(CREATE_TABLE).WithContext(ctx).Exec()
    if err != nil {
        panic(err)
    }

    err = session.Query(INSERT_QUERY,1, “John”,
“john@gmail.com”).WithContext(ctx).Exec()
        if err != nil {
        panic(err)
    }
    name := ””
    err = session.Query(QUERY).WithContext(ctx).Scan(&name)
    if err != nil {
        panic(err)
    }
    fmt.Println(“Retrieved name”, name)
}
```

When a query is expected to return multiple rows like a `SELECT *` query, the `Iter` method from `Query` permits to iterate through the results using pagination. Pagination is controlled internally although it can be customized. Example 18.12 queries all the entries from the `users` table. For better navigation across the returned results, we use a `Scanner` type that can be easily iterated until no more results are returned. Additional examples of how `Scanner` works can be found at Section 7.3.

Example 18.12: GoCQL multiple results query.

```
package main

import (
    “context”
    “fmt”
    “github.com/gocql/gocql”
)

const (
     QUERY        =“SELECT * FROM users”
     CREATE_TABLE =‘CREATE TABLE users (
id int PRIMARY KEY,
name text,
email text
)‘
     INSERT_QUERY =‘INSERT INTO users
(id,name,email) VALUES(?,?,?)‘
)

func main() {
     cluster := gocql.NewCluster(“127.0.0.1:9042”)
     cluster.Keyspace = “example”
     session, err := cluster.CreateSession()
     if err != nil {
         panic(err)
```

```
Found: 1 john@gmail.com John
Found: 2 mary@gmail.com Mary
```

```
    }
    defer session.Close()

    ctx := context.Background()
    err = session.Query(CREATE_TABLE).WithContext(ctx).Exec()
    if err != nil {
        panic(err)
    }

    err = session.Query(INSERT_QUERY,2, “Mary”, 
“mary@gmail.com”).WithContext(ctx).Exec()
    if err != nil {
        panic(err)
    }
    err = session.Query(INSERT_QUERY,1, “John”, 
“john@gmail.com”).WithContext(ctx).Exec()
    if err != nil {
        panic(err)
    }

    scanner := session.Query(QUERY).WithContext(ctx).Iter().Scanner()
    for scanner.Next() {
        var id int
        var  name, email string
        err = scanner.Scan(&id,&name,&email)
        if err != nil {
            panic(err)
        }
        fmt.Println(“Found:”,id,name,email)
    }
}
```

As a final note, observe that we have inserted the records in the reverse order they are retrieved. This occurs because Cassandra sorts the records using the primary key.

18.1.5 User Defined Types

User-defined types (UDTs) can be expressed using structs and tag fields. Example 18.13 defines a laptop UDT to be incorporated to the users table. The `Laptop` struct defines the field of the UDT we have defined. GoCQL uses field tags like `cql:“sn”` to indicate that struct field correspond to variable `sn` in the UDT. The `Scan` method can populate this struct without additional guidance as shown in the code.

Example 18.13: GoCQL UDT struct definition and query.

```
package main

import (
    “context”
    “fmt”
    “github.com/gocql/gocql”
    “time”
)

const LAPTOP_TYPE = ‘CREATE TYPE example.Laptop (
sn int,
model text,
memory int)‘
```

```
Retrieved {100 Lenovo
10}
```

```

const USERS_TABLE =‘CREATE TABLE example.users (
user_id int PRIMARY KEY,
name text,
email text,
Laptop frozen<Laptop>)‘

const INSERT = ‘INSERT INTO example.users (
user_id, name, email, Laptop) VALUES (?,?,?,?)‘

type Laptop struct {
    Sn int ‘cql:“sn”‘
    Model string ‘cql:“model”‘
    Memory int ‘cql:“memory”‘
}

func main() {
    cluster := gocql.NewCluster(“127.0.0.1:9042”)
    cluster.Keyspace = “example”
    cluster.Consistency = gocql.Quorum
    cluster.Timeout = time.Minute
    session, err := cluster.CreateSession()
    if err != nil {
        panic(err)
    }
    defer session.Close()

    ctx := context.Background()
    err = session.Query(LAPTOP_TYPE).WithContext(ctx).Exec()
    if err != nil {
        panic(err)
    }
    err = session.Query(USERS_TABLE).WithContext(ctx).Exec()
    if err != nil {
        panic(err)
    }

    err = session.Query(INSERT,1,“John”,“john@gmail.com”,&Laptop{100,“Lenovo”,10}).Exec()
    if err != nil {
        panic(err)
    }

    var retrieved Laptop
    err = session.Query(“select laptop from users where user_id=1”).Scan(&retrieved)
    fmt.Println(“Retrieved”, retrieved)
}
```

18.2 SUMMARY

This Chapter overviews the utilization of NoSQL solutions in Go. Independently of the complexity and architectonic designs of the underlying technology, Go offers a good solution to manage these databases. In particular, the GoCQL library permits any Go program to manipulate a Cassandra database expanding the horizon of tools that can be used in any data-driven solution.

CHAPTER 19 KAFKA

It is common for a distributed system to have a large number of services designed to consume information in real-time. As the number of services grows, ensuring the scalability and availability of data becomes a challenge. Traditionally, the publish/subscribe pattern has been used to delegate the consumption and publication of new data to a third service in charge of ensuring this data is consumed by the corresponding subscribers as soon as possible. Apache Kafka is a widely adopted publish/subscribe event streaming distributed platform that ensures the scalability and availability of messages for large scale systems. This Chapter explores how to use Go to produce and consume data from Kafka using external clients and clients designed by ourselves.

19.1 THE BASICS

Apache Kafka[70] is designed as a distributed platform that can span multiple data centres and cloud regions. Kafka brokers are accessed by clients to publish new data or consume it. Kafka producers and consumers work around the concept of events. An event is a record or message containing a key, a value, a timestamp, and optionally some metadata. These events are organized into topics. Topics are a logical organization of events. Every event is stored into a topic, in this sense, topics work like folders. Events are stored in the system until they expire. This expiration occurs when they reach the limit of a predefined retention policy.

Topics are partitioned across brokers. By replicating events that belong to the same topic into several partitions, Kafka ensures fault tolerance and improves availability and throughput. Figure 19.1 shows a topic split into four partitions. Every square inside the partitions is an event in which colour corresponds to a different key. Depending on the replication policy, events can be replicated across different partitions. Normally there are three replicas of every event. Events with the same key are stored in the same partition. Kafka guarantees that events are consumed in the same order they were written.

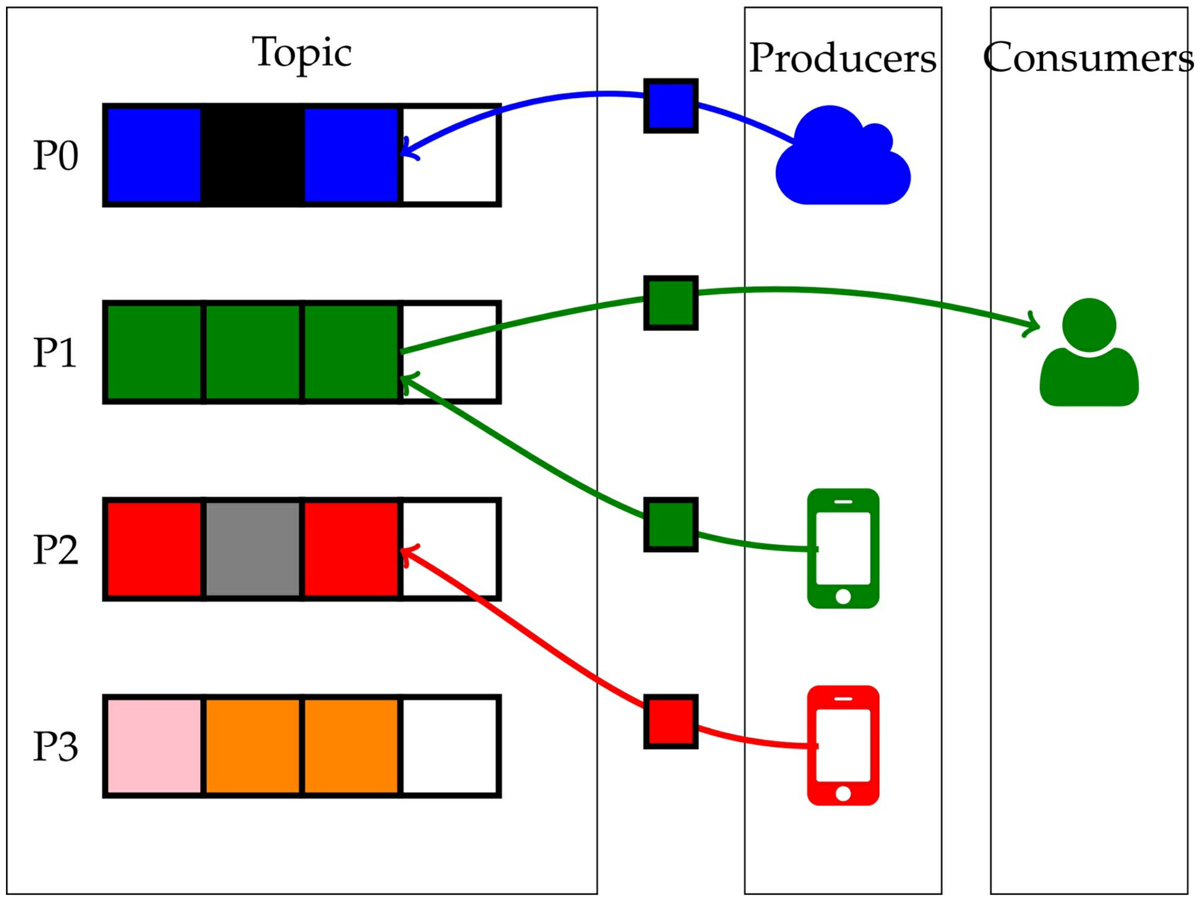


Figure 19.1: Kafka topics and partitions.

Events inside topics are indexed by a consecutive number. When a consumer is subscribed to a topic, she receives an offset with the index of the next event to be consumed. When producers consume events, they increment this offset to get new events if they are available. Different consumers can have different offsets like in Figure 19.2 . To ensure that consumers get all the available messages in their order of arrival, consumers can commit the index of the latest message they have consumed. This informs the broker about the current status of the consumer. Kafka by default executes this commit operation automatically although consumers can specify when to commit with the correct configuration.

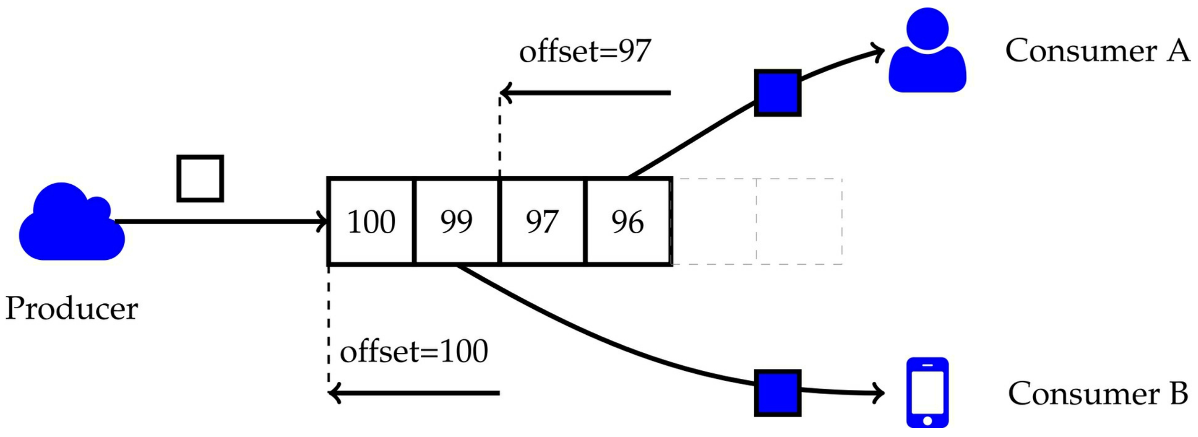


Figure 19.2: Kafka offsets in a topic partition.

From the Kafka drivers available in Go, we explore the official driver from confluent[71] and the pure Go implementation provided by Segmentio[72].

The examples described in this Chapter assume a Kafka broker is available and listening at `localhost:9092`. For the examples from Section 19.4 the Kafka API REST must be available at `localhost:8082`. You can deploy a complete virtualized Kafka environment using Docker following the official documentation[73]. A reduced Docker compose solution is provided in our GitHub repository.

19.2 USING THE CONFLUENT CLIENT

The `confluent-kafka-go`[74] is a wrapper around the the `librdkafka` C library. This wrapper defines a Kafka client with a producer and a consumer types. Examples 19.1 and 19.2 show standard implementations for a producer and a consumer respectively.

19.2.1 Synchronous producer

The `Producer` type is instantiated using a `ConfigMap` which contains key-value pairs with all the required configuration for the connection[75]. The `Producer` holds a connection that has to be released with `Close()`. For this Example, we send events to the topic named `helloTopic`. Events are defined by the `Message` type described below.

```
type Message struct {
   TopicPartition TopicPartition
   Value          []byte
   Key            []byte
   Timestamp      time.Time
   TimestampType  TimestampType
   Opaque         interface{}
   Headers        []Header
}
```

The most important field is `Value` which contains the message content in a raw byte slice. `Key` can be used for additional control inside the topic as mentioned before. `Opaque` can be used to pass arbitrary data to the corresponding event handler.

The producer shown in Example 19.1 passes a `Message` to the `Produce` method indicating the partition and the message value. Note that we let the target partition be set by Kafka using the `PartitionAny` policy. `Produce` receives a `chan Event` to handle the production of events. Messages are sent to the `librdkafka` which has its own queue of messages. To avoid an overflow, we can wait until the message is sent using the `deliveryChan`. This is not the most efficient approach as the production of messages is stopped until a message is processed. The `Flush` method waits for a given time in milliseconds until all the messages have been sent. In the Example it is commented because in our case we are waiting for every message submission.

Example 19.1: Confluent Kafka synchronous producer.

```
package main

```

```
Sent helloTopic[0]@30
```

```
import (
    “fmt”
    “github.com/confluentinc/confluent-kafka-go/kafka”
)

func main() {
    cfg := kafka.ConfigMap{”bootstrap.servers”: “localhost:9092”}
    p, err := kafka.NewProducer(&cfg)
    if err != nil {
        panic(err)
    }
    defer p.Close()

    deliveryChan := make(chan kafka.Event,10)
    defer close(deliveryChan)
    topic := “helloTopic”
    msgs := []string{”Save”, “the”, “world”, “with”, “Go!!!”}
    for _, word := range msgs {
        err = p.Produce(&kafka.Message{
            TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.PartitionAny},
            Value: []byte(word),
        }, deliveryChan)
        if err != nil {
            panic(err)
        }
        e := <-deliveryChan
        m := e.(*kafka.Message)
        fmt.Printf(”Sent %v\n”,m)
```

```
Sent helloTopic[0]@31
Sent helloTopic[0]@32
Sent helloTopic[0]@33
Sent helloTopic[0]@34
```

```
    }
    // p.Flush(1000)
}
```

19.2.2 Synchronous consumer

Like the `Producer` type, the `Consumer` is instantiated with a `ConfigMap` with the corresponding configuration parameters. Consumers must subscribe to one or several topics matching a given regular expression. The `Poll` blocks the execution for the given timeout and returns an `Event` or `nil`. Like shown in Example 19.2 , using a `switch` statement we can control the execution flow and process messages, errors or other situations.

Example 19.2: Confluent Kafka synchronous consumer.

```
package main

import (
    “fmt”
    “github.com/confluentinc/confluent-kafka-go/kafka”
)

func main() {
    c, err := kafka.NewConsumer(&kafka.ConfigMap{
        “bootstrap.servers”: “localhost:9092”,
        “group.id”:          “helloGroup”,
        “auto.offset.reset”: “earliest”,
    })
    if err != nil {
        panic(err)
```

```
Message on helloTopic[0]@30: Save
Message on helloTopic[0]@31: the
Message on helloTopic[0]@32: world
Message on helloTopic[0]@33: with
Message on helloTopic[0]@34: Go!!!
```

```
    }
    defer c.Close()

    c.Subscribe(”helloTopic”, nil)
    for {
        ev := c.Poll(1000)
        switch e := ev.(type) {
        case *kafka.Message:
            c.Commit()
            fmt.Printf(”Msg on %s: %s\n”, e.TopicPartition, string(e.Value))
        case kafka.PartitionEOF:
            fmt.Printf(”%v\n”,e)
        case kafka.Error:
            fmt.Printf(”Error: %v\n”, e)
            break
        default:
            fmt.Printf(”Ignored: %v\n”,e)
        }
    }
}
```

The Example above receives the messages produced by the previous example. Remember that in case the `enable.auto.commit` configuration parameter is set to false, consumers must commit to ensuring the correctness of the received messages. This can be done using the `Commit` method as shown in the Example.

19.2.3 Asynchronous producer

Our previous Example of a synchronous producer is not a good idea if we want to get a better throughput for our solution. Actually, Kafka is prepared to receive batches of events to be processed. Normally, producers should send batches of events and wait until they are processed. We can use the delivery channel from the `Produce` method to writing a non-blocking handler to observe if everything is going correctly.

Example 19.3 rewrites the previous producer with a goroutine that handles the production process without blocking the execution. In this case, to ensure that we wait until all the messages are sent, the `Flush` method blocks the execution until everything is flushed out.

Example 19.3: Confluent Kafka asynchronous producer.

```
package main

import (
    “fmt”
    “github.com/confluentinc/confluent-kafka-go/kafka”
)

func Handler(c chan kafka.Event) {
    for {
        e := <- c
        if e == nil {
            return
        }
        m := e.(*kafka.Message)
        if m.TopicPartition.Error != nil {
            fmt.Printf(”Partition error %s\n”,m.TopicPartition.Error)
        } else {
            fmt.Printf(”Sent %v: %s\n”,m,string(m.Value))
        }
    }
}

func main() {
    cfg := kafka.ConfigMap{”bootstrap.servers”: “localhost:9092”}
    p, err := kafka.NewProducer(&cfg)
    if err != nil {
        panic(err)
    }
    defer p.Close()

    delivery_chan := make(chan kafka.Event,10)
    defer close(delivery_chan)
    go Handler(delivery_chan)

    topic := “helloTopic”
    msgs := []string{”Save”, “the”, “world”, “with”, “Go!!!”}
    for _, word := range msgs {
        err = p.Produce(&kafka.Message{
            TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.PartitionAny},
            Value: []byte(word),
        },delivery_chan)
    }
    p.Flush(10000)
}
```

```
Sent helloTopic[0]@95: Save
Sent helloTopic[0]@96: the
Sent helloTopic[0]@97: world
Sent helloTopic[0]@98: with
Sent helloTopic[0]@99: Go!!!
```

19.2.4 Asynchronous consumer

Committing every message is not a good practice that produces a lot of overhead. It makes more sense to commit after a batch of messages is received. In Example 19.4, a commit is done every two messages in a separated goroutine. The batch size is probably very small and it only makes sense for demonstration purposes. The `Committer` function can detect if the commit fails so we can handle the situation.

Example 19.4: Confluent Kafka asynchronous consumer.

```
package main

import (
    “fmt”
    “github.com/confluentinc/confluent-kafka-go/kafka”
)

const COMMIT_N = 2

func Committer(c *kafka.Consumer) {
    offsets, err := c.Commit()
    if err != nil {
        fmt.Printf(”Error: %s\n”, err)
        return
    }
```

```
Msg on helloTopic[0]@95: Save
Msg on helloTopic[0]@96: the
Msg on helloTopic[0]@97: world
Msg on helloTopic[0]@98: with
Msg on helloTopic[0]@99: Go!!!
Offset: “helloTopic[0]@100”
Offset: “helloTopic[0]@100”
```

```
    fmt.Printf(”Offset: %#v\n”,offsets[0].String())
}

func main() {

    c, err := kafka.NewConsumer(&kafka.ConfigMap{
        “bootstrap.servers”: “localhost:9092”,
        “group.id”:          “helloGroup”,
        “auto.offset.reset”: “earliest”,
    })
    if err != nil {
        panic(err)
    }
    defer c.Close()

    c.Subscribe(”helloTopic”, nil)
    counter := 0
    for {
        ev := c.Poll(1000)
        switch e := ev.(type) {
        case *kafka.Message:
            counter += 1
            if counter % COMMIT_N == 0 {
                go Committer(c)
            }
            fmt.Printf(”Msg on %s: %s\n”, e.TopicPartition, string(e.Value))
        case kafka.PartitionEOF:
            fmt.Printf(”%v\n”,e)
        case kafka.Error:
            fmt.Printf(”Error: %v\n”, e)
            break
        default:
            fmt.Printf(”Ignored: %v\n”,e)
        }
    }
}
```

In the previous example, messages are processed independently of the result of the commit operation. We can continue receiving messages even when the commit fails. We can postpone the message process until we know that the commit was successful.

```
…
case *kafka.Message:
        offset, err := c.Commit()
        if err != nil {
            // process message…
        }
…
```

As mentioned before, this would generate a lot of overhead. A possible solution is to store incoming messages and process them after the commit in a batch. This could even be done in a goroutine.

```
…
case *kafka.Message:
        append(messages,msg)
        counter += 1
        if counter % COMMIT_N == 0 {
            offset, err := c.Commit()

            if err != nil {

                go ProcessMsgs(messages)

            }

        }

…
```

19.3 USING THE SEGMENTIO CLIENT

One of the drawbacks of the Confluent Kafka client is the fact that it is already a wrapper of a C library. This can be an issue in certain platforms or for those projects where only Go is going to be available. The Segmentio[76] Kafka client is fully written in Go and offers a complete implementation of consumers and producers with additional features such as compression or secure connections.

19.3.1 The Connection type

The Segmentio client uses a straight forward implementation based on a `Connection` instance. A `Connection` can be instantiated indicating the Kafka host, the topic, and the partition. This connection offers all the methods required by consumers and producers.

Example 19.5 shows the implementation of a producer. The connection uses the method `SetWriteDeadline` to define a deadline for the writing operation. The most basic method to produce messages is `conn.Write` which receives a byte array with the message. However, for demonstration purposes, we use the `conn.WriteMessages` which receives `Message` instances with more valuable information.

```
type Message struct {

   Topic string

   Partition int

   Offset    int64

   Key       []byte

   Value     []byte

   Headers   []Header

   Time time.Time

}
```

An important aspect to be considered is that the `Partition` field must not be set when writing messages, it is a read-only field. In the Example, we iterate through all the messages to be sent. However, we could simply pass a collection of messages to be sent. The write operation returns the number of written bytes and error if any.

Example 19.5: Segmentio Kafka producer.

```
package main

import (
    “context”
    “fmt”
    “github.com/segmentio/kafka-go”
    “time”
)

func main() {
    topic := “helloTopic”
    partition := 0
    conn, err := kafka.DialLeader(context.Background(),
        “tcp”, “localhost:9092”,topic,partition)
    if err != nil {
        panic(err)
    }

    defer conn.Close()

    conn.SetWriteDeadline(time.Now().Add(3*time.Second))
    msgs := []string{”Save”, “the”, “world”, “with”, “Go!!!”}
```

```
Sent 4 bytes: Save
Sent 3 bytes: the
Sent 5 bytes: world
Sent 4 bytes: with
Sent 5 bytes:
Go!!!
```

```
    for _, m := range msgs {
        l, err := conn.WriteMessages(kafka.Message{Value: []byte(m)})
        if err != nil {
            panic(err)
        }
        fmt.Printf(”Sent %d bytes: %s\n”, l,m)
    }
}
```

The structure of a consumer is similar to the producer. A `Connection` instance can be used to read one or several messages in batches. Example 19.6 defines a batch of messages with a minimum size of 10 KB and a maximum size of 10 MB. Note that we use a byte array with me minimum size we defined and we populate it until we find an error. Finally, the batches have to be closed.

Example 19.6: Segmentio Kafka consumer.

```
package main

import (
    “context”
    “fmt”
    “github.com/segmentio/kafka-go”
    “time”
)

func main() {
    topic := “helloTopic”
    partition := 0
    conn, err := kafka.DialLeader(context.Background(),
```

```
Received 4: Save
Received 3: the
Received 5: world
Received 4: with
Received 5:
Go!!!
```

```
        "tcp", "localhost:9092",topic,partition)
    if err != nil {
        panic(err)
    }

    defer conn.Close()

    conn.SetReadDeadline(time.Now().Add(time.Second))
    batch := conn.ReadBatch(10e3, 10e6)
    defer batch.Close()
    for {
        b := make([]byte, 10e3)
        l, err := batch.Read(b)
        if err != nil {
            break
        }
        fmt.Printf("Received %d: %s\n",l,string(b))
    }
}
```

19.3.2 Writer and Reader types

The Segmentio client has high level abstraction types named `Writer` and `Reader`. These types are designed to simplify producer and consumer implementations.

Example 19.7 uses a `Writer` instance to send messages to Kafka. The code is very similar to the one shown in Example 19.5. However, note that all the operations are performed using the `Writer` instead of a `Connection` instance.

Example 19.7: Segmentio Kafka high level API producer.

```
Sent message: Save

Sent message: the

Sent message: world

Sent message: with

Sent message: Go!!!

Producer sent: 131
bytes
```

```
package main

import (
    “context”
    “fmt”
    “github.com/segmentio/kafka-go”
    “time”
)

func main() {
    topic := “helloTopic”
    partition := 0
    conn, err := kafka.DialLeader(context.Background(),
        “tcp”, “localhost:9092”,topic,partition)
    if err != nil {
        panic(err)
    }

    defer conn.Close()

    conn.SetWriteDeadline(time.Now().Add(3*time.Second))
    msgs := []string{”Save”, “the”, “world”, “with”, “Go!!!”}
    for _, m := range msgs {
        l, err := conn.WriteMessages(kafka.Message{Value: []byte(m)})
        if err != nil {
            panic(err)
        }
        fmt.Printf("Sent %d bytes: %s\n", l,m)
    }
}
```

The `WriterMessages` method is designed to receive a collection of messages. The example is not properly using the method. To improve performance, all the available messages should be passed to this method to improve throughput. The `Stats` method returns an instance with the statistics of this writer. The Example prints the total number of bytes written although there is an interesting set of available stats.

A consumer can be implemented using the `Reader` type as shown in Example 19.8 . A `Reader` is configured using a `ReaderConfig` indicating the brokers, partition, topic or group to be subscribed to. The `FetchMessage` method blocks until new events are available or the context expires.

Example 19.8: Segmentio Kafka high level API consumer.

```
package main

import (
    "context"
    "fmt"
    "github.com/segmentio/kafka-go"
)

func main () {
    r := kafka.NewReader(kafka.ReaderConfig{
        Brokers:    []string{"localhost:9092"},
        Partition:  0,
        Topic:      "helloTopic",
```

```
Topic helloTopic msg: Save
Topic helloTopic msg: the
Topic helloTopic msg: world
Topic helloTopic msg: with
Topic helloTopic msg: Go!!!
```

```
        GroupID: “testGroup”,
        MinBytes:   10e3,
        MaxBytes:   10e6,
    })

    defer r.Close()

    for {
        m, err := r.FetchMessage(context.Background())
        if err != nil {
            break
        }
        if err := r.CommitMessages(context.Background(), m); err != nil {
            panic(err)
        }
        fmt.Printf(”Topic %s msg: %s\n”, m.Topic, m.Value)
    }
}
```

In this Example, we commit every consumed message. This produces an unnecessary overhead that can be mitigated by committing several messages at the same time. The `CommitMessages` method has a variadic argument for the number of messages to be committed.

Remember that manual committing is only required when the auto-commit configuration parameter in the Kafka brokers is disabled.

19.4 USING THE KAFKA REST API

Kafka defines its own protocol over TCP [3]. However, raw TCP connections are not always a good solution in certain environments. For example, browser-running solutions or platforms where the Kafka native client is not available. Fortunately, Kafka has a REST-proxy [77] that exposes Kafka features through an HTTP API REST. This makes it possible to work with Kafka brokers from standard HTTP clients. In this Section, we explore how to produce and consume messages using the standard `net/http` Go package.

The examples from this Section assume the Kafka restful API to be running with a default configuration at `localhost:8082`. If you are not familiar with HTTP clients in Go, review Chapter 9.

19.4.1 Producer

The REST Proxy writes messages with the post method at `/topics/<name>/partitions/<number>`, where `<name>` and `<number>` are the topic name and the corresponding partition respectively. The message is expected to be attached to the body of the request. This method admits data in JSON, Avro, binary, and Protobuf formats. For example, the following JSON object would represent two messages in the request body.

Example 19.9: Kafka REST Proxy JSON encoded messages

```
{
    “records”:
    [
        {”value”:{”name”:”John”,”email”:”john@gmail.com”}},
        {”value”:{”name”:”Mary”,”email”:”mary@email.com”}}
    ]
}
```

Note that we are only indicating the value of every message, but other values such as the key of the timestamp can be set. The encoding format has to be indicated in the `Content-Type` header of the request. For JSON messages the corresponding header value is `application/vnd.kafka.json.v2+json`.

With these elements we can use the `net/http` package to produce Kafka messages like in Example 19.10. To simplify the generation of JSON objects, we use the JSON notation (see Section 8.2). The `BuildBody` function is a helper to build the JSON object requested by Kafka with the content of our `User` instances. Note that we set the corresponding `Content-Type` header by passing the `CONTENT_TYPE` constant to the `Post` function.

Example 19.10: Kafka producer using the REST API.

```
package main

import (
     “bufio”
     “bytes”
     “encoding/json”
     “fmt”
     “net/http”
     “strings”
)

const (
    URL = “http://localhost:8082/topics/%s/partitions/%d”
    CONTENT_TYPE = “application/vnd.kafka.json.v2+json”
)

type User struct{
     Name string ‘json:”name”‘
     Email string ‘json:”email”‘
}

```

```
func BuildBody (users []User) string {
    values := make([]string,len(users))
    for i, u := range users {
        encoded, err := json.Marshal(&u)
        if err != nil {
            panic(err)
        }
        values[i] = fmt.Sprintf(”{"value":%s}”, encoded)
    }
    result := strings.Join(values,”,”)
    return fmt.Sprintf(”{"records": [%s]}”, result)
}

func main() {
    users := []User{{”John”, “john@gmail.com”},{”Mary”,”mary@email.com”}}
    body := BuildBody(users)
    fmt.Println(body)
    bufferBody := bytes.NewBuffer([]byte(body))

    resp, err := http.Post(fmt.Sprintf(URL,”helloTopic”,0),CONTENT_TYPE, bufferBody)
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()

    fmt.Println(resp.Status)
    bodyAnswer := bufio.NewScanner(resp.Body)
    for bodyAnswer.Scan() {
        fmt.Println(bodyAnswer.Text())
    }
}
```

```
{”records”: [{”value”:{”name”:”John”,”email”:”john@gmail.com”}},{”value”:
{”name”:”Mary”,”email”:”mary@email.com”}}]}

200 OK

{”offsets”:[{”partition”:0,”offset”:165,”error_code”:null,”error”:null},
{”partition”:0,”offset”:166,”error_code”:null,”error”:null}],”key_schema_id”:null,”valu
```

The body of the server response contains information about the partitions and offsets of every written message.

19.4.2 Consumer

Consuming events from the API REST requires more steps than simply producing events. Actually, to get a message a consumer must: 1) create a new consumer instance, 2) subscribe to a topic and group, 3) consume records, and finally 4) delete the instance if no more messages are going to be consumed. The following Examples are fragments of the same piece of code we have split into chunks according to the mentioned steps.

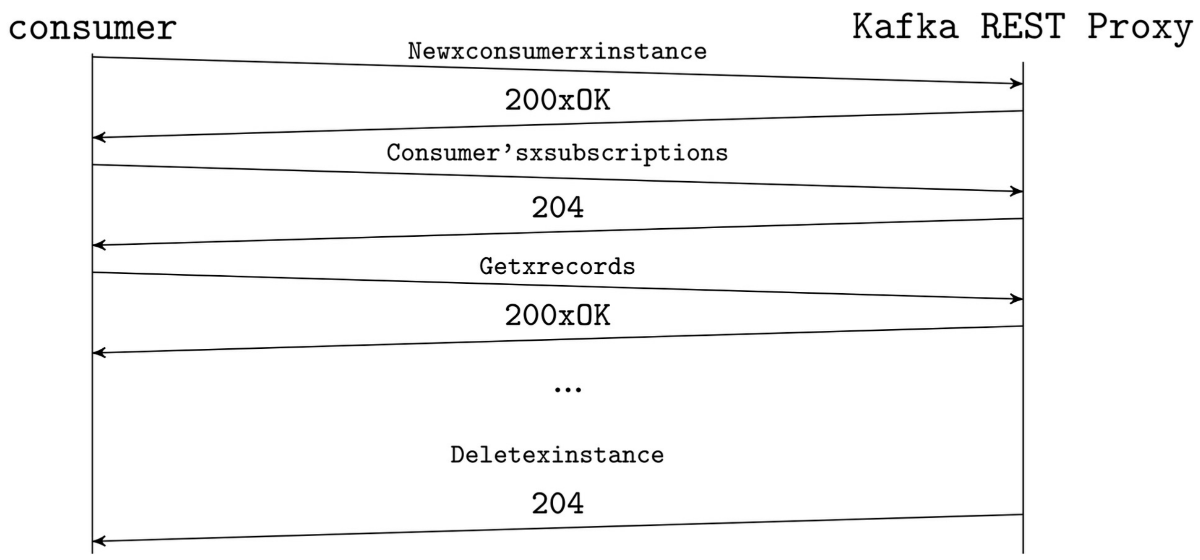


Figure 19.3: Consumer and Kafka REST Proxy communication diagram.

In Example 19.11 we define the URLs of the methods we need and the `DoHelper` function to help with POST methods. To create a new consumer, we call the POST method `/consumers/testGroup` where `testGroup` in the name of the consumers' group. The body request contains the name of the consumer (`testConsumer`) and the format messages must be used by messages (`json`). The response body contains the consumer id and the base URI to be used by this consumer.

Example 19.11: Kafka consumer using the REST API (excerpt I, new consumer).

```
package main

import (
     “bufio”
     “bytes”
     “time”
     “fmt”
     “net/http”
)

const (
      HOST              = “http://localhost:8082”
      CONSUMER          = “testConsumer”
      GROUP             = “testGroup”
      NEW_CONSUMER      = “%s/consumers/%s”
      SUBSCRIBE_CONSUMER = “%s/consumers/%s/instances/%s/subscription”
      FETCH_CONSUMER = “%s/consumers/%s/instances/%s/records”
      DELETE_CONSUMER = “%s/consumers/%s/instances/%s”
```

```go
    CONTENT_TYPE   = “application/vnd.kafka.json.v2+json”
)

func DoHelper(client *http.Client, url string, body []byte ) error {
    bufferBody := bytes.NewBuffer(body)
    req, err := http.NewRequest(http.MethodPost,url, bufferBody)
    if err != nil {
        return err
    }
    fmt.Printf(”—>Call %s\n”,req.URL)
    fmt.Printf(”—>Body %s\n”,string(body))
    resp, err := client.Do(req)
    if err != nil {
        return err
    }
    defer resp.Body.Close()
    bodyResp := bufio.NewScanner(resp.Body)
    fmt.Printf(”<—Response %s\n”, resp.Status)
    for bodyResp.Scan() {
        fmt.Printf(”<—Body %s\n”,bodyResp.Text())
    }
    return nil
}

func main() {
    client := http.Client{}
    // New consumer
    url := fmt.Sprintf(NEW_CONSUMER,HOST,GROUP)
```

```
body := fmt.Sprintf('{“name“:”%s“, “format“: “json“}',CONSUMER)
err := DoHelper(&client, url, []byte(body))
if err != nil {
    panic(err)
}
time.Sleep(time.Second)
```

Next, we subscribe the consumer to the `helloTopic` in Example 19.12 . The target POST method matches the base URI received in the response from the creation of the consumer instance extended with the suffix `subscription`. The body contains the list of topics this consumer wants to subscribe to. The response returns a 204 code indicating no body in the response.

Example 19.12: Kafka consumer using the REST API (excerpt II, subscription).

```
// Subscribe to topic
url = fmt.Sprintf(SUBSCRIBE_CONSUMER,HOST,GROUP,CONSUMER)
body = '{“topics“:[“helloTopic“]}'
err = DoHelper(&client, url, []byte(body))
if err != nil {
    panic(err)
}
time.Sleep(time.Second)
```

Now the consumer is ready to receive records from the topics it has been subscribed to (Exampe 19.13). A GET post to the base URI with the `records` suffix will return any available record. Note that the `Accept` header must be set with the corresponding content type we want the incoming messages to be encoded, in this case, the JSON format. The response body contains the available messages. Note that if in this case, no messages are available at the time of sending the request, the returned response will be empty. Additional query parameters are `timeout` to specify the maximum time the server will spend fetching records and `max_bytes` with the maximum size of the returned records.

Example 19.13: Kafka consumer using the REST API (excerpt III, acquisition).

```
    // Get records
    req, err := http.NewRequest(http.MethodGet, fmt.Sprintf(FETCH_CONSUMER,HOST,GROUP,CONSUMER), nil)
    if err != nil {
        panic(err)
    }
    req.Header.Add(”Accept”,CONTENT_TYPE)
    fmt.Printf(”–>Call %s\n”,req.URL)
    respRecords, err := client.Do(req)
    if err != nil {
        panic(err)
    }
    defer respRecords.Body.Close()
    fmt.Printf(”<–Response %s\n”, respRecords.Status)
    recordsBodyResp := bufio.NewScanner(respRecords.Body)
    for recordsBodyResp.Scan() {
        fmt.Printf(”<–Body %s\n”,recordsBodyResp.Text())
    }
```

Finally, we delete the consumer instance to release resources like shown in Example 19.14. The body response is empty with a 204 status.

Example 19.14: Kafka consumer using the REST API (excerpt IV, delete).

```
    // Delete consumer instance
    deleteReq, err :=
http.NewRequest(http.MethodDelete,fmt.Sprintf(DELETE_CONSUMER,HOST,GROUP,CONSUMER),nil)
    if err != nil {
        panic(err)
    }
    fmt.Printf("—>Call %s\n",deleteReq.URL)
    resp, err := client.Do(deleteReq)
    if err != nil {
        panic(err)
    }
    fmt.Printf("<—Response %s\n",resp.Status)
}
```

The program shows the requests and responses between the customer and the server during the whole process.

```
—>Call http://localhost:8082/consumers/testGroup
—>Body {"name":"testConsumer", "format": "json"}
<—Response 200 OK
<—Body {"instance_id":"testConsumer","base_uri":"http://rest-
proxy:8082/consumers/testGroup/instances/testConsumer"}
—>Call http://localhost:8082/consumers/testGroup/instances/testConsumer/subscription
—>Body {"topics":["helloTopic"]}
<—Response 204 No Content
—>Call http://localhost:8082/consumers/testGroup/instances/testConsumer/records
<—Response 200 OK
<—Body [{"topic":"helloTopic","key":null,"value":
{"name":"John","email":"john@gmail.com"},"partition":0,"offset":179},
{"topic":"helloTopic","key":null,"value":
{"name":"Mary","email":"mary@email.com"},"partition":0,"offset":180}]
—>Call http://localhost:8082/consumers/testGroup/instances/testConsumer
<—Response 204 No Content
```

Note that in this case, we are not modifying the consumer offset. Calling the POST method at `/consumers/testGroup/instances/testConsumer/positions` with the next offset indicated in the body prepares the consumer for the next batch of messages. For our example where the latest record had offset 180, we could send the following body to set the next offset to 181.

```
{
  “offsets”: [
    {
      “topic”: “helloTopic”,
      “partition”: 0,
      “offset”: 181
    }
  ]
}
```

19.5 SUMMARY

This Chapter explores how to consume and produce data with the Apache Kafka message streaming platform. We analyze different Kafka clients provided by Confluent and Segmentio, and show how they access the same solution from different perspectives. Finally, we demonstrate how using the standard Go library we can easily create our clients without external dependencies using the Kafka API Rest.

Table of Contents

BIBLIOGRAPHY

[1] Apache Cassandra. Apache Cassandra documentation. https://cassandra.apache.org/doc/latest/, 2021.

[2] Apache Kafka. Apache Kafka official documentation. https://kafka.apache.org/documentation/, 2021.

[3] Apache Kafka. Kafka protocol guide. https://kafka.apache.org/protocol, 2021.

[4] ECMA. The JSON Data Interchange Syntax. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, 2021.

[5] golang.org. Go official documentation. https://golang.org/doc/, 2021.

[6] Tyler Hobbs. Basic Rules of Cassandra Data Modeling. https://www.datastax.com/blog/basic-rules-cassandra-data-modeling, 2015.

[7] IETF. Common Format and MIME Type for Comma-Separated Values (CSV) Files. http://https://tools.ietf.org/html/rfc4180, 2021.

[8] IETF. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://tools.ietf.org/html/rfc7231, 2021.

[9] Sobel Jonathan M. and Friedman Daniel P. An Introduction to Reflection-Oriented Programming. https://web.archive.org/web/20100204091328/http://www.cs.indi 1996.

[10] Rob Pike. The Laws of Reflection. https://blog.golang.org/laws-of-reflection, 2021.

[11] W3C. Extensible Markup Language (XML). https://www.w3.org/XML/, 2021.

[12] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall PTR, USA, 1978.

[13] yaml.org. YAML Ain't Markup Language (YAML™) Version 1.2. https://yaml.org/spec/1.2/spec.html, 2021.

[14] yaml.org. Yet Another Markup Language (YAML) 1.0. https://yaml.org/spec/history/2001-12-10.html, 2021.

NOTES

1 https://golang.org/doc/install

2 https://gobyexample.com/command-line-arguments

3 https://golang.org/cmd/go/\#hdr-GOPATH_environment_variable

4 Actually, `rune` could be an alias for `uint32` instead of `int32`.

5 We can check the type of a variable using `reflect.TypeOf`. Visit Chapter 5 for more details.

6 https://golang.org/ref/spec\#Slice_types

7 https://golang.org/pkg/reflect/

8 https://blog.golang.org/generics-proposal

9 https://golang.org/pkg/context

10 https://golang.org/pkg/io

11 https://golang.org/pkg/io/ioutil/

12 https://golang.org/pkg/os/\#File

13 https://golang.org/pkg/bufio/

14 https://golang.org/pkg/bufio/\#Scanner

15 https://golang.org/pkg/encoding/xml/\#Marshal

16 Check the package reference for other examples.

17 https://github.com/go-yaml/yaml

18 https://golang.org/pkg/net/http/\#Client

19 The `curl` command is displayed in the output frame `curl -H “Header1: Value1” :8090/info`

20 https://golang.org/pkg/net/http/\#Cookie

21 https://golang.org/pkg/net/http/cookiejar/

22 https://golang.org/pkg/text/template/

23 https://golang.org/pkg/text/template/\#hdr-Functions

24 A candidate quote could be `Choppa Quote = “Get to the choppa!!!”.`

25 Check `go`help testflag for more information.

26 https://golang.org/pkg/runtime/pprof/

27 https://github.com/google/pprof

28 `go tool pprof –help`

29 `go help doc` for more details.

30 https://developers.google.com/protocol-buffers

31 https://developers.google.com/protocol-buffers/docs/downloads

32 https://developers.google.com/protocol-buffers/docs/proto3\#scalar

33 In the example package paths have been trimmed for clarity purposes.

34 https://developers.google.com/protocol-buffers/docs/proto3\#json

35 https://grpc.io

36 Check the documentation for more details
https://grpc.io/docs/languages/go/quickstart/\#prerequisites

37 https://github.com/grpc-ecosystem/grpc-gateway

38 For additional details check: https://github.com/grpc-ecosystem/grpc-gateway.

39 https://golang.org/pkg/net/http/\#ServeMux

40 https://github.com/grpc-ecosystem/go-grpc-middleware

41 https://golang.org/pkg/log

42 https://github.com/rs/zerolog

43 https://github.com/rs/zerolog\#standard-types

44 If you are not familiar with JSON encodings check Chapter 8.2

45 A detailed explanation of how HTTP Handlers work can be found in Chapter 9.

46 https://github.com/rs/zerolog/blob/master/hlog/hlog.go

47 https://github.com/spf13/cobra

48 https://github.com/spf13/cobra/blob/master/doc/README.md

49 Check the documentation for additional details
https://github.com/spf13/cobra/blob/master/shell_completions.md.

50 https://golang.org/pkg/database/sql

51 https://golang.org/pkg/database/sql/\#DB

52 https://www.sqlite.org/

53 https://github.com/golang/go/wiki/SQLDrivers

54 https://github.com/mattn/go-sqlite3

55 You can find a more detailed explanation of `init`functions in Section 2.1.1 .

56 `https://gorm.io`

57 Object-relational Mapping

58 `https://gorm.io/docs/connecting_to_the_database.html\#Unsupported-Databases`

59 `https://pkg.go.dev/database/sql/\#Scanner` and `https://pkg.go.dev/database/sql/driver\#Valuer`for more details.

60 `https://gorm.io/docs/models.html\#Fields-Tags`

61 `https://gorm.io/docs/query.html`

62 `https://gorm.io/docs/transactions.html\#Disable-Default-Transaction`

63 `https://cassandra.apache.org/`

64 `https://cassandra.apache.org/doc/latest/cql/index.html`

65 `https://github.com/gocql/gocql`

66 Versions 2.1.x, 2.2.x, and 3.x.x when these lines were written.

67 `https://pkg.go.dev/github.com/gocql/gocql\#ClusterConfig`

68 `https://pkg.go.dev/github.com/gocql/gocql\#Query`

69 `https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useBatchGoodExample.html`

70 `https://kafka.apache.org`

71 `https://github.com/confluentinc/confluent-kafka-go`

72 `https://github.com/segmentio/kafka-go`

73 `https://docs.confluent.io/platform/current/quickstart/ce-docker-quickstart.html\#ce-docker-quickstart`

74 https://github.com/confluentinc/confluent-kafka-go

75 https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md

76 https://github.com/segmentio/kafka-go

77 https://docs.confluent.io/3.0.0/kafka-rest/docs/index.html