**Go**

**Chapter -2**

**Go programs are read top to bottom, left to right.**

**package main**

This is known as a “package declaration”. Every Go program must start with a package declaration. Packages are Go's way of organizing and reusing code.

There are two types of Go programs:

* executables and
* libraries.

Executable applications are the kinds of programs that we can run directly from the terminal. (in Windows they end with .exe)

Libraries are collections of code that we package together so that we can use them in other programs.

**Go mostly doesn't care about whitespace**

import "fmt"

The import keyword is how we include code from other packages to use with our program. The fmt package (shorthand for format) implements formatting for input and output.

Notice that fmt above is surrounded by double quotes. The use of double quotes like this is known as a “string literal” which is a type of “expression”.

**The line that starts with // is known as a comment.**

**To comment multiple lines: /\* \*/**

Functions are the building blocks of a Go program. They have inputs, outputs and a series of steps called statements which are executed in order. All functions start with the keyword **func** followed by the name of the function (main in this case), a list of zero or more “parameters” surrounded by parentheses, an optional return type and a “body” which is surrounded by curly braces.

This function has no parameters, doesn't return anything and has only one statement.

The name **main** is special because it's the function that gets called when you execute the program.

**fmt.Println("Hello World")**

This statement is made of three components. First we access another function inside of the fmt package called Println (that's the fmt.Println piece, Println means Print Line). Then we create a new string that contains Hello World and invoke (also known as call or execute) that function with the string as the first and only argument.

**\_, err := fmt.Scanf("%d", &total)**

**if err != nil {**

**fmt.Println(err)**

**}**

**Scanf returns a value and an error. That’s why we need to do this one.**

We used “**\_**” for **unused variables.**

**Chapter- 3**

Go is a statically typed programming language. This means that variables always have a specific type and that type cannot change.

Go's integer types are: uint8, uint16, uint32, uint64, int8, int16, int32 and int64.

uint means “unsigned integer”

**there two alias types:**

byte which is the same as uint8 and

rune which is the same as int32

* Floating point numbers are inexact. Occasionally it is not possible to represent a number. For example computing 1.01 - 0.99 results in 0.020000000000000018 – A number extremely close to what we would expect, but not exactly the same.

Go has two floating point types:

float32 and

float64

(also often referred to as single precision and double precision respectively)

as well as two additional types for representing complex numbers (numbers with imaginary parts): complex64 and complex128.

Generally we should stick with float64 when working with floating point numbers.

**Chapter- 4**

**A variable is a storage location, with a specific type and an associated name**

**var x string = "Hello World"**

**Var variable\_Name Data\_Type**

**name := "Max" -> Directly assign without var**

**Constants are basically variables whose values cannot be changed later.**

**const x string = "Hello World"**

**Defining Multiple Variables:**

**var (**

**a = 5**

**b = 10**

**c = 15**

**)**

**Chapter-5**

**Loop:**

**i := 1**

**for i <= 10 {**

**fmt.Println(i)**

**i = i + 1**

**}**

**Or**

**for j := 1; j <= 10; j++ {**

**fmt.Println(j)**

**}**

**Go only has one loop that can be used in a variety of different ways.**

**Conditional Statements:**

**IF Else:**

**if j%2 == 0 && j%3 == 0 {**

**fmt.Println(j)**

**}**

**if j%2 == 0 {**

**fmt.Println("Even")**

**} else if j%2==1 {**

**fmt.Println("Odd")**

**} else {**

**}**

**Switch Case:**

**switch j {**

**case 0: fmt.Println("Zero")**

**case 1: fmt.Println("One")**

**default: fmt.Println("fffffff")**

**}**

**Chapter- 6**

**Array:**

**var x [4]float64**

**len(x) => 4**

**for i := 0; i < 4; i++ {**

**total += x[i]**

**}**

**Or**

**for \_, value := range x {**

**total += value**

**}**

**Underscore is important-> A single \_ (underscore) is used to tell the compiler that we don't need this. (In this case we don't need the iterator variable)**

**Assign elements to an array:**

**var x [4]float64**

**x[0] = 1**

**x[1] = 2**

**x[2] = 3**

**x[3] = 4**

**Or**

**x := [5]float64{ 98, 93, 77, 82, 83 }**

**Or**

**x := [5]float64{**

**98,**

**93,**

**77,**

**82,**

**83,**

**}**

**Or**

**x := [4]float64{**

**98,**

**93,**

**77,**

**82,**

**// 83,**

**}**

**We can also make comments inside the bracket.**

**float64(len(x) Type casting**

## **Slices**

A slice is a segment of an array.

Like arrays, slices are indexable and have a length.

Unlike arrays this **length is allowed to change**

**Type length**

**x := make([]float64, 5)**

**Capacity**

**x := make([]float64, 5, 10)**

**arr := [5]float64{1,2,3,4,5}**

**x := arr[0:5] take value index 0 to 5**

**X := arr[:] take value index 0 to len(arr)**

**X := arr[:4] take value index 0 to (4-1)= 3**

**slice1 := []int{1,2,3}**

**slice2 := append(slice1,4,5) append 4 & 5 after slice1**

**slice1 := []int{1,2,3}**

**slice2 := make([]int, 2)**

**copy(slice2, slice1) copy slice1 in Slice2**

**Map**

A map is an unordered collection of key-value pairs.

Also known as an associative array, a hash table or a dictionary, maps are used to look up a value by its associated key.

**x := make(map[string]int)**

**x["key"] = 10**

**x["key1"] = 100**

**fmt.Println(x)**  map[key:10 key1:100]

Map\_Name, Value

**delete(x, "key")**

**fmt.Println(x)** map[key1:100]

name, ok := elements["h"] name ok

fmt.Println(name, ok) if “h” is in the map value, true.

Else empty string, false

**Map Initialization:**

**elements := map[string]string{**

**"H": "Hydrogen",**

**"He": "Helium",**

**}**

**func main() {**

**elements := map[string]map[string]string{**

**"H": map[string]string{**

**"name":"Hydrogen",**

**"state":"gas",**

**},**

**"He": map[string]string{**

**"name":"Helium",**

**"state":"gas",**

**},**

**"Li": map[string]string{**

**"name":"Lithium",**

**"state":"solid",**

**}**

**if el, ok := elements["Li"]; ok {**

**fmt.Println(el["name"], el["state"])**

**}**

**a map of strings to maps of strings to strings.**

**The outer map is used as a lookup table based on the element's symbol,**

**while the inner maps are used to store general information about the elements**

**Chapter- 7**

**func function\_name(Parameter-list) (Return\_type) {**

**// function body.....**

**}**

**func average(xs []float64) float64 {**

**total := 0.0**

**for \_, v := range xs {**

**total += v**

**}**

**return total / float64(len(xs))**

**}**

**func main() {**

**s := []float64{1, 2, 3, 4}**

**fmt.Println(average(s))**

**}**

**We can also set the return type name**

**func f2() (r int) {**

**r = 1**

**return**

**}**

## **Returning Multiple Values**

**Go is also capable of returning multiple values from a function:**

**func f() (int, int) {**

**return 5, 6**

**}**

**func main() {**

**x, y := f()**

**}**

## **Variadic Functions**

**There is a special form available for the last parameter in a Go function:**

**func add(args ...int) int {**

**total := 0**

**for \_, v := range args {**

**total += v**

**}**

**return total**

**}**

**func main() {**

**fmt.Println(add(1,2,3))**

**}**

## **Closure**

It is possible to create functions inside of functions:

func main() {

add := func(x, y int) int {

return x + y

}

fmt.Println(add(1,1))

}

**add is a local variable that has the type func(int, int) int (a function that takes two ints and returns an int).**

**func main() {**

**x := 0**

**increment := func() int {**

**x++**

**return x**

**}**

**fmt.Println(increment())**

**fmt.Println(increment())**

**}**

**increment adds 1 to the variable x which is defined in the main function's scope. This x variable can be accessed and modified by the increment function. This is why the first time we call increment we see 1 displayed, but the second time we call it we see 2 displayed.**

**func makeEvenGenerator() func() uint {**

**i := uint(0)**

**return func() (ret uint) {**

**ret = i**

**i += 2**

**return**

**}**

**}**

**func main() {**

**nextEven := makeEvenGenerator() //assign the return value**

**fmt.Println(nextEven()) // 0**

**fmt.Println(nextEven()) // 2**

**fmt.Println(nextEven()) // 4**

**}**

**makeEvenGenerator returns a function which generates even numbers. Each time it's called it adds 2 to the local i variable which – unlike normal local variables – persists between calls**

**Recursion:**

**Finally a function is able to call itself. Here is one way to compute the factorial of a number:**

**func factorial(x uint) uint {**

**if x == 0 {**

**return 1**

**}**

**return x \* factorial(x-1)**

**}**

**Defer**

**Go has a special statement called defer which schedules a function call to be run after the function completes.**

**func first() {**

**fmt.Println("1st")**

**}**

**func second() {**

**fmt.Println("2nd")**

**}**

**func main() {**

**defer second() // It means execute first(), then second()**

**first()**

**}**

**Same as:**

**first()**

**second()**

**when we open a file we need to make sure to close it later. With defer:**

**f, \_ := os.Open(filename)**

**defer f.Close()**

**This has 3 advantages:**

**(1) it keeps our Close call near our Open call so it's easier to understand,**

**(2) if our function had multiple return statements (perhaps one in an if and one in an else) Close will happen before both of them and**

**(3) deferred functions are run even if a run-time panic occurs.**

**panic function to cause a run time error.**

**We can handle a run-time panic with the built-in recover function.**

**recover stops the panic and returns the value that was passed to the call to panic**

**func main() {**

**panic("PANIC")**

**str := recover()**

**fmt.Println(str)**

**}**

**But the call to recover will never happen in this case because the call to panic immediately stops execution of the function.**

**func main() {**

**defer func() {**

**str := recover()**

**fmt.Println(str)**

**}()**

**panic("PANIC")**

**}**

**This code will work.**

## **Multiple defer Statements in Go**

When we use multiple defer in a program, the order of execution of the defer statements will be LIFO (Last In First Out).

**func main() {**

**defer fmt.Println("One")**

**defer fmt.Println("Two")**

**defer fmt.Println("Three")**

**}**

**Output->**

**Three**

**Two**

**One**

func handlePanic() {

// detect if panic occurs or not

a := recover()

if a != nil {

fmt.Println("RECOVER", a)

}

}

func division(num1, num2 int) {

// execute the handlePanic even after panic occurs

defer handlePanic()

// if num2 is 0, program is terminated due to panic

if num2 == 0 {

panic("Cannot divide a number by zero")

} else {

result := num1 / num2

fmt.Println("Result: ", result)

}

}

func main() {

division(4, 2)

division(8, 0)

division(2, 8)

}

Result: 2

RECOVER Cannot divide a number by zero

Result: 0

# **Pointers**

func zero(xPtr \*int) {

\*xPtr = 0

}

func main() {

x := 5

zero(&x)

fmt.Println(x) // x is 0

}

Pointers reference a location in memory where a value is stored rather than the value itself.

(They point to something else) By using a pointer (\*int) the zero function is able to modify the original variable.

## **New**

Another way to get a pointer is to use the built-in new function.

func one(xPtr \*int) {

\*xPtr = 1

}

func main() {

xPtr := new(int)

one(xPtr)

fmt.Println(\*xPtr) // x is 1

}

new takes a type as an argument, allocates enough memory to fit a value of that type and returns a pointer to it.

Go is not like this, it's a garbage collected programming language which means memory is cleaned up automatically when nothing refers to it anymore.

**New with Slice:**

**func one(xp \*[]int) {**

**\*xp = append(\*xp, 1)**

**\*xp = append(\*xp, 2)**

**\*xp = append(\*xp, 3)**

**}**

**func main() {**

**xp := new([]int)**

**one(xp)**

**fmt.Println(\*xp)**

**}**

## **Structs**

An easy way to make this program better is to use a struct.

A struct is a type which contains named fields.

The type keyword introduces a new type. It's followed by the name of the type (Circle), the keyword struct to indicate that we are defining a struct type and a list of fields inside of curly braces. Each field has a name and a type. Like with functions we can collapse fields that have the same type:

**type Circle struct {**

**x, y, r float64**

**}**

### **Initialization**

**var c Circle**

Like with other data types, this will create a local Circle variable that is by default set to zero.

For a struct zero means each of the fields is set to their corresponding zero value

(0 for ints, 0.0 for floats, "" for strings, nil for pointers, …)

**c := new(Circle)**

This allocates memory for all the fields, sets each of them to their zero value and returns a pointer. (\*Circle) More often we want to give each of the fields a value.

**c := Circle{x: 0, y: 0, r: 5}**

**c := Circle{0, 0, 5}**

**Slice of Structs:**

**type Circle struct {**

**x, y, r float64**

**}**

**func (c Circle) area() float64 {**

**return math.*Pi* \* c.r \* c.r**

**}**

**func main() {**

**//c := Circle{0, 0, 5**

**var c []Circle = []Circle{}**

**c = append(c, Circle{1.0, 2.0, 3.0})**

**fmt.Println(c[0].area())**

**//fmt.Println(c1.area())**

**}**

**Array of Struct:**

**type Circle struct {**

**x, y, r float64**

**}**

**func (c Circle) area() float64 {**

**return math.*Pi* \* c.r \* c.r**

**}**

**func main() {**

**var c [4]Circle = [4]Circle{}**

**c[0] = Circle{0, 0, 5}**

**c[1] = Circle{0, 0, 5}**

**fmt.Println(c[0].area())**

**}**

**Embedded Types:**

**type Person struct {**

**Name string**

**}**

**func (p \*Person) Talk() {**

**fmt.Println("Hi, my name is", p.Name)**

**}**

**type Android struct {**

**Person // Embedded Struct**

**Model string**

**}**

**a := new(Android)**

**a.Talk()**

**we can call any Person methods directly on the Android:**

**type Android struct {**

**Person Person**

**Model string**

**}**

**a := new(Android)**

**a.Person.Talk()**

**In this code a.Talk() will not work. Because there is no embedded struct.**

## **Interfaces**

You may have noticed that we were able to name the Rectangle's area method the same thing as the Circle's area method. This was no accident. In both real life and in programming, relationships like these are commonplace. Go has a way of making these accidental similarities explicit through a type known as an Interface.

**type Shape interface {**

**area() float64**

**}**

Like a struct an interface is created using the type keyword, followed by a name and the keyword interface. But instead of defining fields, we define a “method set”. A method set is a list of methods that a type must have in order to “implement” the interface.

**func totalArea(shapes ...Shape) float64 {**

**var area float64**

**for \_, s := range shapes {**

**area += s.area()**

**}**

**return area**

**}**

**fmt.Println(totalArea(&c, &r))**

**MUlti-Shape Interface**

**type Shape interface {**

**area() float64**

**}**

**type MultiShape struct {**

**shapes []Shape**

**}**

**func (m \*MultiShape) area() float64 {**

**var area float64**

**for \_, s := range m.shapes {**

**area += s.area()**

**}**

**return area**

**}**

**type Rectangle struct {**

**x1, y1, x2, y2 float64**

**}**

**func distance(x1, y1, x2, y2 float64) float64 {**

**return math.Sqrt((x2-x1)\*(x2-x1) + (y2-y1)\*(y2-y1))**

**}**

**func (r \*Rectangle) area() float64 {**

**l := distance(r.x1, r.y1, r.x1, r.y2)**

**w := distance(r.x1, r.y1, r.x2, r.y1)**

**return l \* w**

**}**

**type Circle struct {**

**x, y, r float64**

**}**

**func (c \*Circle) area() float64 {**

**return math.*Pi* \* c.r \* c.r**

**}**

**func main() {**

**r := &Rectangle{0, 0, 4, 3}**

**c := &Circle{0, 0, 5}**

**multiSp := &MultiShape{make([]Shape, 2)}**

**multiSp.shapes[0] = r**

**multiSp.shapes[1] = c**

**fmt.Println(multiSp.area())**

**}**

**Chapter- 10**

## **Goroutines**

**A goroutine is a function that is capable of running concurrently with other functions.**

**To create a goroutine we use the keyword go followed by a function invocation:**

**func f(n int) {**

**for i := 0; i < 10; i++ {**

**fmt.Println(n, ":", i)**

**}**

**}**

**func main() {**

**go f(0)**

**var input string**

**fmt.Scanln(&input)**

**}**

**Here, is just Two goroutines: main and go f(0)**

**func f(n int) {**

**for i := 0; i < 10; i++ {**

**fmt.Println(n, ":", i)**

**amt := time.Duration(rand.Intn(250))**

**time.Sleep(time.Millisecond \* amt)**

**}**

**}**

**func main() {**

**for i := 0; i < 10; i++ {**

**go f(i)**

**}**

**var input string**

**fmt.Scanln(&input)**

**}**

## **Channels**

**Channels provide a way for two goroutines to communicate with one another and synchronize their execution.**

**func pinger(c chan string) {**

**for i := 0; ; i++ {**

**c <- "ping" // c<- Send**

**}**

**}**

**func printer(c chan string) {**

**for { // Infinity Loop**

**msg := <- c // <-c Receive**

**fmt.Println(msg)**

**time.Sleep(time.Second \* 1)**

**}**

**}**

**func main() {**

**var c chan string = make(chan string) // To create channel keyword “chan”, then type “string”.**

**go pinger(c)**

**go printer(c)**

**var input string**

**fmt.Scanln(&input)**

**}**

### **Channel Direction**

We can specify a direction on a channel type thus restricting it to either sending or receiving.

**func pinger(c chan<- string)**

Channel c send only

**func printer(c <-chan string)**

Channel c receive only

A bi-directional channel can be passed to a function that takes send-only or receive-only channels.

### 

### **Select**

Go has a special statement called select which works like a switch but for channels:

**func main() {**

**c1 := make(chan string)**

**c2 := make(chan string)**

**go func() {**

**for {**

**c1 <- "from 1"**

**time.Sleep(time.Second \* 2)**

**}**

**}()**

**go func() {**

**for {**

**c2 <- "from 2"**

**time.Sleep(time.Second \* 3)**

**}**

**}()**

**go func() {**

**for {**

**select {**

**case msg1 := <- c1:**

**fmt.Println("Message 1", msg1)**

**case msg2 := <- c2:**

**fmt.Println("Message 2", msg2)**

**case <- time.After(time.Second):**

**fmt.Println("timeout")**

**Default: // If nothing ready, default case happens immediately**

**fmt.Println("nothing ready")**

**}**

**}**

**}()**

**var input string**

**fmt.Scanln(&input)**

**}**

This program prints “from 1” every 2 seconds and “from 2” every 3 seconds. **select** picks the first channel that is ready and receives from it (or sends to it).

If more than one of the channels are ready then it randomly picks which one to receive from.

If none of the channels are ready, the statement blocks until one becomes available.

### **Buffered Channels**

It's also possible to pass a second parameter to the make function when creating a channel:

**c := make(chan int, 1)**

This creates a buffered channel with a **capacity of 1.**

Normally channels are synchronous; both sides of the channel will wait until the other side is ready.

A buffered channel is asynchronous; sending or receiving a message will not wait unless the channel is already full. If it is already full, then it will wait.

Chapter- 11

# **Packages**

another mechanism for **code reuse:** packages

**In go directory: Create src directory-> Create golang-book directory->**

**Create main.go + Create package\_name Directory -> math.go**

import m "golang-book/chapter11/math"

m is the alias.

In Go if something starts with a capital letter that means other packages (and programs) are able to see it.

If we had named the function **average** instead of **Average** our main program would not have been able to see it.

**Testing:**

Go includes a special program that makes writing tests easier, so let's create some tests for the package we made in the last chapter.

**package math**

**import "testing"**

**func TestAverage(t \*testing.T) {**

**var v float64**

**v = Average([]float64{1,2})**

**if v != 1.5 {**

**t.Error("Expected 1.5, got ", v)**

**}**

**}**

**Command to Run: go test**

**Core Packages:**

## **Strings**

* **strings.Contains("test", "es")**

Is “es” string is a sub-string of “test -> true/ false

## **Learn Go Nuances**

**Switch**

**Not just integers**

## **Missing expression**

you don’t need to switch on anything at all. A switch with no value means “switch true”, making it a cleaner version of an if-else chain

**switch {**

**case '0' <= c && c <= '9':**

**return c - '0'**

**case 'a' <= c && c <= 'f':**

**return c - 'a' + 10**

**case 'A' <= c && c <= 'F':**

**return c - 'A' + 10**

**}**

## **Break**

v := 3

switch {

case v >= 1:

fmt.Println("dfs")

fallthrough

case v >= 2:

fmt.Println("dfs")

if v >= 2 {

break

}

fallthrough

case v >= 3:

fmt.Println("dfs")

}

## **Fall through**

**v := 42**

**switch v {**

**case 100: fmt.Println(100)**

**fallthrough**

**case 42:**

**fmt.Println(42)**

**fallthrough**

**case 1:**

**fmt.Println(1)**

**fallthrough**

**default:**

**fmt.Println("default")**

**}**

**// Output:**

// 42

// 1

// default

**The ‘fallthrough’ must be the last thing in the case**

**Note: fallthrough does not work in type switch.**

## **Multiple cases**

**If you want to use multiple values in the same case, use a comma-separated list.**

**a := 3**

**switch a {**

**case 0, 1, 2, 3:**

**fmt.Println("Yes")**

**default:**

**fmt.Println("No")**

**}**

## **Type switch**

**func typeName(v interface{}) string {**

**switch v.(type) {**

**case int:**

**return "int"**

**case string:**

**return "string"**

**default:**

**return "unknown"**

**}**

**}**

**x := typeName(2)**

**fmt.Println(x)**

**Output -> int**

**func do(v interface{}) string {**

**switch u := v.(type) {**

**case int:**

**return strconv.Itoa(u \* 2) // u has type int**

**case string:**

**mid := len(u) / 2 // split - u has type string**

**return u[mid:] + u[:mid] // join**

**}**

**return "unknown"**

**}**

**func main() {**

**x := do(21)**

**fmt.Println(x)**

## **Noop case**

**Sometimes it useful to have cases that require no action. This can look confusing, because it can appear that both the noop case and the subsequent case have the same action, but isn’t so.**

**func pluralEnding(n int) string {**

**ending := ""**

**switch n {**

**case 1:**

**default:**

**ending = "s"**

**}**

**return ending**

**}**

**func main() {**

**fmt.Println("foo%s\n", pluralEnding(1))**

**fmt.Println("bar%s\n", pluralEnding(2))**

**}**

# **Iterating Over Slices In Go**

**Go is an imperative programming language**

**type Dog struct {**

**Name string**

**Age int**

**}**

**func main() {**

**jackie := Dog{**

**Name: "Jackie",**

**Age: 19,**

**}**

**fmt.Printf("Jackie Addr: %p\n", &jackie)**

**sammy := Dog{**

**Name: "Sammy",**

**Age: 10,**

**}**

**fmt.Printf("Sammy Addr: %p\n", &sammy)**

**dogs := []Dog{jackie, sammy}**

**fmt.Println("")**

**for \_, dog := range dogs {**

**fmt.Printf("Name: %s Age: %d\n", dog.Name, dog.Age)**

**fmt.Printf("Addr: %p\n", &dog)**

**fmt.Println("")**

**}**

**}**

**Output:**

**Jackie Addr: 0x2101bc000**

**Sammy Addr: 0x2101bc040**

**Name: Jackie Age: 19**

**Addr: 0x2101bc060**

**Name: Sammy Age: 10**

**Addr: 0x2101bc060**

**So why is the address of the dog value different inside the range loop and why does the same address appear twice? This all has to do with the fact that everything is pass by value in Go. In this code example we actually create 2 extra copies of each Dog value in memory.**

**Now we can see why the address of the dog variable inside range loop is always the same. We are displaying the address of the dog variable, which happens to be a local variable of type Dog and contains a copy of the Dog value for each index of the slice. With each iteration of the slice, the location of the dog variable is the same. The value of the dog variable is changing.**

**type Dog struct {**

**Name string**

**Age int**

**}**

**func main() {**

**jackie := &Dog{**

**Name: "Jackie",**

**Age: 19,**

**}**

**fmt.Printf("Jackie Addr: %p\n", jackie)**

**sammy := &Dog{**

**Name: "Sammy",**

**Age: 10,**

**}**

**fmt.Printf("Sammy Addr: %p\n\n", sammy)**

**dogs := []\*Dog{jackie, sammy}**

**for \_, dog := range dogs {**

**fmt.Printf("Name: %s Age: %d\n", dog.Name, dog.Age)**

**fmt.Printf("Addr: %p\n\n", dog)**

**}**

**}**

**Output:**

**Jackie Addr: 0x2101bb000**

**Sammy Addr: 0x2101bb040**

**Name: Jackie Age: 19**

**Addr: 0x2101bb000**

**Name: Sammy Age: 10**

**Addr: 0x2101bb040**

**This time we create a slice of pointers to Dog values. When we iterate over this slice, the value of the dog variable is the address of each Dog value we stored in the slice. Instead of creating two extra copies of each Dog value, we are using the same initial Dog value we created with the composite literal.**

**Empty Slice:**

**data := []string{}**

**fmt.Println(data)**

**Output: [] <- Empty, not zero.**

**Empty Struct:**

**var es struct{}**

**var es1 struct{}**

**fmt.Printf("%p ", &es)**

**fmt.Println(es)**

**fmt.Printf("%p ", &es1)**

**fmt.Println(es1)**

**Output:**

**0x54e3e0 {}**

**0x54e3e0 {} // Assign in same address & Empty.**

# **Common Mistakes**

# **Using reference to loop iterator variable**

**func main() {**

**var out []\*int**

**for i := 0; i < 3; i++ {**

**out = append(out, &i)**

**}**

**fmt.Println("Values:", \*out[0], \*out[1], \*out[2])**

**fmt.Println("Addresses:", out[0], out[1], out[2])**

**}**

**It will output unexpected results:**

**Values: 3 3 3**

**Addresses: 0x40e020 0x40e020 0x40e020**

**Explanation: in each iteration we append the address of i to the out slice, but since it is the same variable, we append the same address which eventually contains the last value that was assigned to i.**

**One of the solutions is to copy the loop variable into a new variable:**

**func main() {**

**var out []\*int**

**for i := 0; i < 3; i++ {**

**j := i**

**out = append(out, &j)**

**}**

**for i := 0; i < 3; i++ {**

**fmt.Printf("%d %p\n", \*out[i], out[i])**

**}**

**}**

**he loop variable can be an array and the reference can be a slice:**

**func main() {**

**var out [][]int**

**for \_, i := range [][1]int{{1}, {2}, {3}} {**

**out = append(out, i[:])**

**}**

**fmt.Println("Values:", out)**

**}**

**Output:**

**Values: [[3] [3] [3]]**

**Can be solved by taking another variable:**

**var out [][]int**

**fmt.Println(out)**

**for \_, i := range [][1]int{{1}, {2}, {3}} {**

**j := i**

**out = append(out, j[:])**

**}**

**fmt.Println(out)**

**var values []int**

**values = append(values, 1, 2)**

**for \_, val := range values {**

**go func() {**

**fmt.Println(val)**

**}()**

**}**

**var a int**

**fmt.Scanf("%d", &a)**

**Output: 2 2**

**Solution:**

**var values []int**

**values = append(values, 1, 2)**

**for \_, val := range values {**

**go func(val int) {**

**fmt.Println(val)**

**}(val)**

**}**

**var a int**

**fmt.Scanf("%d", &a)**

**By adding val as a parameter to the closure, val is evaluated at each iteration and placed on the stack for the goroutine,**

**so each slice element is available to the goroutine when it is eventually executed.**

**In Structure Method:**

**func (x \*v) MyMethod() {**

**fmt.Println(x)**

**}**

**type v struct {**

**x, y, r int**

**}**

**func main() {**

**var values []v = []v{}**

**values = append(values, v{1, 2, 3})**

**values = append(values, v{4, 5, 6})**

**for \_, val := range values {**

**go val.MyMethod()**

**}**

**var a int**

**fmt.Scanf("%d", &a)**

**Same issue. Printing the last elements.**

**Solve: Assigning a newval.**

**func (x \*v) MyMethod() {**

**fmt.Println(x)**

**}**

**type v struct {**

**x, y, r int**

**}**

**func main() {**

**var values []v = []v{}**

**values = append(values, v{1, 2, 3})**

**values = append(values, v{4, 5, 6})**

**for \_, val := range values {**

**newval := val**

**go newval.MyMethod()**

**}**

**var a int**

**fmt.Scanf("%d", &a)**

**}**

# **SliceTricks**

**Append**

**var a []int**

**var b []int**

**b = append(b, 1, 2, 3, 4, 5, 6, 7)**

**Copy**

**c := make([]int, len(b))**

**copy(c, b)**

**c = append([]int(nil), b...)**

**fmt.Println(c)**

**c = append(b[:0:0], b...)**

**fmt.Println(c)**

**Cut**

**a = append(b[:3], b[6:]...)**

**fmt.Println(a)**

**B[:3] -> 0 to 3**

**B[6:]... -> 7 to last index of b**

**Delete**

**c = append(c[:3], c[4:]...)**

**fmt.Println(c)**

**c = c[:3+copy(c[3:], c[3+1:])]**

**fmt.Println(c)**

**Delete without preserving order**

**c[3] = c[len(c)-1] // swap the ith index with the last index**

**c = c[:len(c)-1] // make the slice size len()-1**

**fmt.Println(c)**

**If the type of the element is a *pointer* or a struct with pointer fields, which need to be garbage collected, the above implementations of Cut and Delete have a potential *memory leak* problem: some elements with values are still referenced by slice a’s underlying array, just not “visible” in the slice. Because the “deleted” value is referenced in the underlying array, the deleted value is still “reachable” during GC, even though the value cannot be referenced by your code. If the underlying array is long-lived, this represents a leak. The following code can fix this problem:**

**Cut:**

**var c []int**

**c = append(c, 1, 2, 3, 4, 5, 6, 7)**

**copy(c[3:], c[6:])**

**fmt.Println(c) // printing rest of the index which are cut**

**n := len(c) - 6 + 3**

**for k := len(c); k < n; k++ {**

**c[k] = 0**

**}**

**c = c[:len(c)-6+3]**

**fmt.Println(c) // not printing the cutting index**

**Delete:**

**copy(a[i:], a[i+1:])**

**a[len(a)-1] = nil // or the zero value of T**

**a = a[:len(a)-1]**

**Delete without preserving order**

**a[i] = a[len(a)-1]**

**a[len(a)-1] = nil**

**a = a[:len(a)-1]**

#### **Expand**

**Insert n elements at position i:**

**a = append(a[:i], append(make([]T, n), a[i:]...)...)**

#### **Extend**

**Append n elements:**

**a = append(a, make([]T, n)...)**

**Insert**

**var a []int**

**a = append(a, 1, 2, 3, 4, 5, 6, 7)**

**fmt.Println(a)**

**i := 2**

**a = append(a[:i], append([]int{8, 9}, a[i:]...)...)**

**fmt.Println(a) // [1 2 8 9 3 4 5 6 7]**

**The second append creates a new slice with its own underlying storage and copies elements in a[i:] to that slice, and these elements are then copied back to slice a (by the first append).**

**The creation of the new slice (and thus memory garbage) and the second copy can be avoided by using an alternative way:**

**INsert:**

**i := 2 // Initially: [1 2 3 4 5 6 7]**

**a = append(a, 0)**

**copy(a[i+1:], a[i:])**

**a[i] = 9**

**fmt.Println(a) // [1 2 9 3 4 5 6 7]**

**Push (in Last)**

**b = append(b, 5)**

**Push (in Front)**

**b = append([]int{5}, b...)**

**Pop ( Last Element )**

**var x int**

**x, b = b[len(b)-1], b[:len(b)-1]**

**fmt.Println(b)**

**fmt.Println(x)**

**Pop (First Element)**

**var x int**

**x, b = b[0], b[1:]**

# **Arrays, slices (and strings):**

**Arrays have their place—they are a good representation of a transformation matrix for instance—but their most common purpose in Go is to hold storage for a slice.**

**Slices are where the action is, but to use them well one must understand exactly what they are and what they do.**

**A slice is a data structure describing a contiguous section of an array stored separately from the slice variable itself. *A slice is not an array*. A slice *describes* a piece of an array.**

**but for now think of a slice as a little data structure with two elements: a length and a pointer to an element of an array. You can think of it as being built like this behind the scenes:**

**type sliceHeader struct {**

**Length int**

**ZerothElement \*byte**

**}**

**slice := sliceHeader{**

**Length: 50,**

**ZerothElement: &buffer[100],**

**}**

**we can also slice a slice, like this:**

**slice2 := slice[5:10]**

**Just as before, this operation creates a new slice, in this case with elements 5 through 9 (inclusive) of the original slice, which means elements 105 through 109 of the original array.**

## **Passing slices to functions**

**func AddOneToEachElement(slice []byte) {**

**for i := range slice {**

**slice[i]++**

**}**

**}**

**func main() {**

**x := []byte{1, 2, 3}**

**AddOneToEachElement(x)**

**fmt.Println(x)**

**}**

**Output: 2 3 4 // As slice the header includes a pointer to elements of an array, change the both x & slice.**

## **Pointers to slices: Method receivers**

**Another way to have a function modify the slice header is to pass a pointer to it. Here’s a variant of our previous example that does this:**

**func PtrSubtractOneFromLength(slicePtr \*[]byte) {**

**slice := \*slicePtr**

**\*slicePtr = slice[0 : len(slice)-1]**

**}**

**func main() {**

**fmt.Println("Before: len(slice) =", len(slice))**

**PtrSubtractOneFromLength(&slice)**

**fmt.Println("After: len(slice) =", len(slice))**

**}**

**the third component of the slice header: its *capacity*. Besides the array pointer and length, the slice header also stores its capacity:**

**type sliceHeader struct {**

**Length int**

**Capacity int**

**ZerothElement \*byte**

**}**

**var iBuffer [10]int**

**slice := iBuffer[0:0]**

**fmt.Println(len(slice))**

**fmt.Println(cap(slice))**

**As buffer size is 10, the slice capacity becomes also 10.**

**The Capacity field is equal to the length of the underlying array, minus the index in the array of the first element of the slice**

**gophers := make([]int, 10)**

**In this declaration, capacity is default 10. Same as len**

**String:**

**Strings are actually very simple: they are just read-only slices of bytes with a bit of extra syntactic support from the language.**

**Because they are read-only, there is no need for a capacity (you can’t grow them), but otherwise for most purposes you can treat them just like read-only slices of bytes.**

**10 Things you don’t know**

### **Anonymous structs**

**emp1 := struct {**

**name string**

**age int**

**}{**

**name: "evan",**

**age: 40,**

**}**

**fmt.Println(reflect.TypeOf(emp1))**

**Anonymous structs on the other hand lets you create struct-based variables without needing to define the struct.**

**Nested Struct:**

**// Creating structure**

**type Student struct {**

**name string**

**branch string**

**year int**

**}**

**// Creating nested structure**

**type Teacher struct {**

**name string**

**subject string**

**exp int**

**details Student**

**}**

**result := Teacher{**

**name: "Suman",**

**subject: "Java",**

**exp: 5,**

**details: Student{"Bongo", "CSE", 2},**

**}**

**Godoc**

**extracts and generates documentation for Go programs.**

**It runs as a web server and presents the documentation as a web page.**

**By default, godoc looks at the packages it finds via $GOROOT and $GOPATH (if set).**

**This behavior can be altered by providing an alternative $GOROOT with the -goroot flag.**

**CLI - Cobra**

**Cobra is a library for creating powerful modern CLI applications.**

**Cobra is a library providing a simple interface to create powerful modern CLI interfaces similar to git & go tools.**

**Cobra is used in many Go projects such as** [**Kubernetes**](https://kubernetes.io/)**,** [**Hugo**](https://gohugo.io/)**, and** [**GitHub CLI**](https://github.com/cli/cli) **to name a few.** [**This list**](https://github.com/spf13/cobra/blob/main/site/content/projects_using_cobra.md) **contains a more extensive list of projects using Cobra.**

**POSIX stands for "Portable Operating System Interface for Unix." It's a family of standards aimed at ensuring compatibility between operating systems, specifically those that are Unix-like.**

**POSIX standards define the API (Application Programming Interface), shell interface, and utility interfaces for such systems.**

**Automatically generating man pages (manual pages) for applications can be a helpful way to provide comprehensive documentation and usage instructions.**

**Man pages are the standard documentation format on Unix and Unix-like systems, providing users with information about commands, utilities, and applications.**

**Command is the central point of the application. Each interaction that the application supports will be contained in a Command. A command can have children commands and optionally run an action.**

**Command**

**Cobra is built on a structure of commands, arguments & flags.**

**Commands represent actions, Args are things and Flags are modifiers for those actions.**

**APPNAME VERB NOUN --ADJECTIVE**

**Or**

**APPNAME COMMAND ARG --FLAG**

**In the following example, 'server' is a command, and 'port' is a flag:**

**hugo server --port=1313**

**A flag is a way to modify the behavior of a command.**

**Cobra supports fully POSIX-compliant flags as well as the Go** [**flag package**](https://golang.org/pkg/flag/)**.**

**A Cobra command can define flags that persist through to children commands and flags that are only available to that command.**

**cobra-cli is a command line program to generate cobra applications and command files. It will bootstrap your application scaffolding to rapidly develop a Cobra-based application. It is the easiest way to incorporate Cobra into your application.**

**The go.mod file is the root of dependency management in GoLang. All the modules which are needed or to be used in the project are maintained in go.mod file.**

**For all the packages we are going to import/use in our project, it will create an entry of those modules in go.mod.**

**Having a go mod file saves the efforts of running the go get command for each dependent module to run the project successfully.**

**go mod init — creates a new module, initializing the go.mod file that describes the module. At the start, it will only add the module path and go version in go mod file.**

**After running any package building command like go build, go test for the first time, it will install all the packages with specific versions i.e which are the latest at that moment.**

**It will also create a go.sum file which maintains the checksum so when you run the project again it will not install all packages again. But use the cache which is stored inside $GOPATH/pkg/mod directory (module cache directory).**

**go.sum is a generated file you don’t have to edit or modify this file.**

**In go.mod**

**“require” will include all dependency modules and the related version we are going to use in our project**

**“replace” points to the local version of a dependency in Go rather than the git-web. It will create a local copy of a vendor with versions available so no need to install every time when we want to refer the vendor.**

**“//indirect” implies that we are not using these dependencies inside our project but there is some module which imports these.  
all the transitive dependencies are indirect, these include dependencies which our project needs to work properly.**

### **Persistent Flags**

**A flag can be 'persistent', meaning that this flag will be available to the command it's assigned to as well as every command under that command. For global flags, assign a flag as a persistent flag on the root.**

### **Local Flags**

**A flag can also be assigned locally, which will only apply to that specific command.**

## **PreRun and PostRun Hooks**

**It is possible to run functions before or after the main Run function of your command. The PersistentPreRun and PreRun functions will be executed before Run. PersistentPostRun and PostRun will be executed after Run.**

**A plugin is a** [**software**](https://www.computerhope.com/jargon/s/software.htm) **add-on that is installed on a program, enhancing its capabilities.**

**For example, if you wanted to watch a video on a website, you may need a plugin to do so. If the plugin is not installed, your browser will not understand how to play the video.**

**// Flags and Arguments**

**cobra-cli add create**

**go run main.go create -s raisul islam**

**Flag Arguments**

**package cmd**

**import (**

**"fmt"**

**"github.com/spf13/cobra"**

**)**

**var Source bool**

**// createCmd represents the create command**

**var createCmd = &cobra.Command{**

**Use: "create",**

**Short: "A brief description of your command",**

**Long: `A longer description that spans multiple lines and likely contains examples**

**and usage of using your command. For example:**

**Cobra is a CLI library for Go that empowers applications.**

**This application is a tool to generate the needed files**

**to quickly create a Cobra application.`,**

**Args: cobra.MatchAll(cobra.ExactArgs(2), cobra.OnlyValidArgs),**

**Run: func(cmd \*cobra.Command, args []string) {**

**fmt.Println("create called")**

**fmt.Println(args)**

**if Source == true {**

**fmt.Println("It is True")**

**} else {**

**fmt.Println("It is False")**

**}**

**},**

**}**

**func init() {**

**rootCmd.AddCommand(createCmd)**

**createCmd.Flags().BoolVarP(&Source, "source", "s", false, "Source Dir")**

**// Here you will define your flags and configuration settings.**

**// Cobra supports Persistent Flags which will work for this command**

**// and all subcommands, e.g.:**

**// createCmd.PersistentFlags().String("foo", "", "A help for foo")**

**// Cobra supports local flags which will only run when this command**

**// is called directly, e.g.:**

**// createCmd.Flags().BoolP("toggle", "t", false, "Help message for toggle")**

**}**

**Go Module**

### **Modules**

**A *module* is a collection of related Go packages that are versioned together as a single unit.**

**Modules record precise dependency requirements and create reproducible builds.**

**Most often, a version control repository contains exactly one module defined in the repository root. (**[**Multiple modules are supported in a single repository**](https://go.dev/wiki/Modules#faqs--multi-module-repositories)**, but typically that would result in more work on an on-going basis than a single module per repository).**

**Summarizing the relationship between repositories, modules, and packages:**

* **A repository contains one or more Go modules.**
* **Each module contains one or more Go packages.**
* **Each package consists of one or more Go source files in a single directory.**

**How to create go.mod (Commands)**

**go mod init**

**go mod tidy**

**go mod vendor**

## **Tidying it up**

**Going back to the previous version that uses only testmod 2.0.0, if we check the contents of go.mod now, we’ll notice something:**

**module mod**

**require github.com/robteix/testmod v1.0.1**

**require github.com/robteix/testmod/v2 v2.0.0**

By default, Go does not remove a dependency from go.mod unless you ask it to. If you have dependencies that you no longer use and want to clean up, you can use the new tidy command:

**$ go mod tidy**

Now we’re left with only the dependencies that are really being used.

## **Vendoring**

**Go modules ignores the vendor/ directory by default. The idea is to *eventually* do away with vendoring[^0]. But if we still want to add vendored dependencies to our version control, we can still do it:**

**$ go mod vendor**

### **Error handling & Reporting**

**A buffer is a continuous region of memory. It's a term that's used a lot so it can be used kind of imprecisely, but a context I use it frequently is "allocate a buffer of 100 bytes" which requests the operating system/platform to provide you with a continuous region of 100 bytes.**

### **Type assertions**

**For an expression x of** [**interface type**](https://go.dev/ref/spec#Interface_types)**, but not a** [**type parameter**](https://go.dev/ref/spec#Type_parameter_declarations)**, and a type T, the primary expression**

**x.(T)**

**asserts that x is not nil and that the value stored in x is of type T. The notation x.(T) is called a *type assertion*.**

**More precisely, if T is not an interface type, x.(T) asserts that the dynamic type of x is** [**identical**](https://go.dev/ref/spec#Type_identity) **to the type T. In this case, T must** [**implement**](https://go.dev/ref/spec#Method_sets) **the (interface) type of x; otherwise the type assertion is invalid since it is not possible for x to store a value of type T. If T is an interface type, x.(T) asserts that the dynamic type of x** [**implements**](https://go.dev/ref/spec#Implementing_an_interface) **the interface T.**

**Interfaces in Go**

**a \*Dog value can utilize the Speak method defined on Dog, but as we saw earlier, a Cat value cannot access the Speak method defined on \*Cat.**

type Dog struct {

}

func (d Dog) Speak() string {

return "Woof!"

}

type Cat struct {

}

func (c \*Cat) Speak() string {

return "Meow!"

}

animals := []Animal{ &Dog{}, &Cat{}, Llama{}, JavaProgrammer{}}

for \_, animal := range animals {

fmt.Println(animal.Speak())

}

### **What is reflection?**

Reflection is the ability of a program to inspect its variables and values at run time and find their type.

To store a string as output in a variable -> Sprintf

i := fmt.Sprintf("insert into order values(%d, %d)", o.ordId, o.customerId)

return i

### **reflect package**

The [reflect](https://golang.org/pkg/reflect/) package implements run-time reflection in Go. The reflect package helps to identify the underlying concrete type and the value of a [*interface{}*](https://golangbot.com/interfaces-part-1/#emptyinterface) variable.

Type represents the actual type of the interface{}, in this case **main.Order** and Kind represents the specific kind of the type. In this case, it’s a **struct**.

t := reflect.TypeOf(q)

v := reflect.ValueOf(q)

k := t.Kind()

The [NumField()](https://golang.org/pkg/reflect/#Value.NumField) method returns the number of fields in a struct and

the [Field(i int)](https://golang.org/pkg/reflect/#Value.Field) method returns the reflect.Value of the ith field.

k:= reflect.ValueOf(q).Kind()

Or

t := reflect.TypeOf(q)

k := t.Kind()

Name of the q

t := reflect.TypeOf(q).Name()

**Mutex:**

One way we could prevent the data race is to ensure that if one goroutine is using the myBalance variable, then all other goroutines are prevented (or *mutually excluded*) from using it at the same time.

We can do this by creating a [sync.Mutex](https://pkg.go.dev/sync/#Mutex) and setting a *lock* around particular lines of code with it.

While one goroutine holds the lock, all other goroutines are prevented from executing any lines of code protected by the same mutex, and are forced to wait until the lock is yielded before they can proceed.

we've created a new mutex and assigned it to the variable mu.

We then use mu.Lock() to create a lock immediately before both racy parts of the code,

and mu.Unlock() to yield the lock immediately after.

var wg sync.WaitGroup // WaitGroup

wg.Add(1) // Wait for how many goRoutines

go b.Add(70) // goRoutines

wg.Done() // Decrement the Counter

wg.Wait() // Wait until goRoutines finished

Lock the mutex before accessing counters;

unlock it at the end of the function using a [defer](https://gobyexample.com/defer) statement.

### **What is concurrency?**

Concurrency is the capability to deal with lots of things at once. It’s best explained with an example.

Let’s consider a person jogging. During his morning jog, let’s say his shoelaces become untied. Now the person stops running, ties his shoelaces and then starts running again. This is a classic example of concurrency. The person is capable of handling both running and tying shoelaces, that is the person is able to deal with lots of things at once :)

When this browser is run in a single-core processor, the processor will context switch between the two components of the browser. It might be downloading a file for some time and then it might switch to render the html of a user requested web page. This is known as concurrency. Concurrent processes start at different points of time and their execution cycles overlap. In this case, the downloading and the rendering start at different points in time and their executions overlap.

Let’s say the same browser is running on a multi-core processor. In this case, the file downloading component and the HTML rendering component might run simultaneously in different cores. This is known as parallelism.

Concurrency is handled in Go using [Goroutines](https://golangbot.com/goroutines/) and channels.

### **What are Goroutines?**

Goroutines are [functions](https://golangbot.com/functions/) or [methods](https://golangbot.com/methods/) that run concurrently with other functions or methods. Goroutines can be thought of as lightweight threads. The cost of creating a Goroutine is tiny when compared to a thread. Hence it’s common for Go applications to have thousands of Goroutines running concurrently.

### **What are channels**

Channels can be thought of as pipes using which Goroutines communicate. Similar to how water flows from one end to another in a pipe, data can be sent from one end and received from the other end using channels.

a := make(chan int)

data := <- a // read from channel a

2a <- data // write to channel a

Sends and receives to a channel are blocked by default.

What does this mean?

When data is sent to a channel, the control is blocked in the send statement until some other Goroutine reads from that channel.

Similarly, when data is read from a channel, the read is blocked until some Goroutine writes data to that channel.

func hello(done chan int) {

fmt.Println("Hello world goroutine")

done <- 5

}

func main() {

done := make(chan int)

fmt.Println(done) // Default no value. Just Printing the Address

go hello(done)

<-done

fmt.Println("main function")

}

The line of code <-done receives data from the done channel but does not use or store that data in any variable. This is perfectly legal.

Now we have our main Goroutine blocked waiting for data on the done channel. The hello Goroutine receives this channel as a parameter, prints Hello world goroutine and then writes to the done channel. When this write is complete, the main Goroutine receives the data from the done channel, it is unblocked and then the text *main function* is printed.

In Golang, there are two types of channels:

1. Buffered Channel
2. Unbuffered Channel

## **Buffered Channel and Its Properties**

To conduct asynchronous communication, buffered channels are required. Before receiving any data, it could store one or more of them. We often don't require the goroutines for this type of channel to process send and receive operations simultaneously. Along with a few other conditions, received will only be blocked if there are no values in the channel to receive, and send will only be blocked if there is no buffer available to place the value being sent.

Syntax:

ch := make(chan type, capacity)

Example:

buffered := make(chan int, 10) *// Buffered channel of integer type*

*//Create a buffered channel*

*// with a capacity of 2.*

ss := make(chan string, 2)

ss <- "Scaler"

ss <- "Golang channels"

fmt.Println(<-ss)

fmt.Println(<-ss)

## **Unbuffered Channel and Its Properties**

In Go, an unbuffered channel is a channel that can be used for both sending and receiving but doesn't have a buffer to store data. This means that when a goroutine sends data to an unbuffered channel, the data is not stored in a buffer, but is instead immediately passed to the goroutine that is trying to receive data from the channel. Similarly, when a goroutine receives data from an unbuffered channel, it blocks until data is available to be received.

Some properties of unbuffered Channels in golang:

* No Buffer: As unbuffered channels don't have a buffer to store data capacity. This means that when a goroutine sends data to an unbuffered channel, the data is not stored in a buffer, but is instead immediately passed to the goroutine that is trying to receive data from the channel.
* Synchronous: Unbuffered channels are synchronous, which means that a goroutine sending data to one will block until another goroutine is prepared to accept it. A goroutine that accepts data from an unbuffered channel will similarly block until new data becomes available.

Unbuffered := make(chan int) // Unbuffered channel of integer type

### **Deadlock**

One important factor to consider while using channels is deadlock. If a Goroutine is sending data on a channel, then it is expected that some other Goroutine should be receiving the data. If this does not happen, then the program will panic at runtime with Deadlock.

Similarly, if a Goroutine is waiting to receive data from a channel, then some other Goroutine is expected to write data on that channel, else the program will panic.

### **Unidirectional channels**

All the channels we discussed so far are bidirectional channels, that is data can be both sent and received on them. It is also possible to create unidirectional channels, that is channels that only send or receive data.

Senders have the ability to **close** the channel to notify receivers that no more data will be sent on the channel.

Receivers can use an additional variable while receiving data from the channel to check whether the channel has been closed.

close(chnl)

v, ok := <-ch

# Channel Axioms

Most new Go programmers quickly grasp the idea of a channel as a queue of values and are comfortable with the notion that channel operations may block when full or empty.

This post explores four of the less common properties of channels:

* **A send to a nil channel blocks forever**
* **A receive from a nil channel blocks forever**
* **A send to a closed channel panics**
* **A receive from a closed channel returns the zero value immediately**
* **A send to a nil channel blocks forever**

**var c chan string**

**c <- "let's get started" // deadlock**

* **A receive from a nil channel blocks forever**

**var c chan string**

**fmt.Println(<-c) // deadlock**

**Why this happened?**

* The size of a channel’s buffer is not part of its *type* declaration, so it must be part of the channel’s value.
* If the channel is not initalised then its buffer size will be zero.
* If the size of the channel’s buffer is zero, then the channel is *unbuffered*.
* If the channel is unbuffered, then a send will block until another goroutine is ready to receive.
* If the channel is nil then the sender and receiver have no reference to each other; they are both blocked waiting on independent channels and will never unblock.

## **Select**

The select statement lets a goroutine wait on multiple communication operations.

A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

## 

## 

## **A nil channel always blocks**

The second property I want to talk about is polar opposite of the closed channel property. A nil channel; a channel value that has not been initalised, or has been set to nil will always block.

## **A closed channel never blocks**

The first property I want to talk about is a closed channel. Once a channel has been closed, you cannot send a value on this channel, but you can still receive from the channel.

Given a nil channel c:

* <-c receiving from c blocks forever
* c <- v sending into c blocks forever
* close(c) closing c panics

# 

# **The Behavior Of Channels**

I learned over time that it’s best to forget about how channels are structured and focus on how they behave.

So now when it comes to channels, I think about one thing: **signaling**.

A channel allows one goroutine to signal another goroutine about a particular event. Signaling is at the core of everything you should be doing with channels. Thinking of channels as a signaling mechanism will allow you to write better code with well defined and more precise behavior.

To understand how signaling works, we must understand its three attributes:

* Guarantee Of Delivery
* State
* With or Without Data

### **Guarantee Of Delivery**

The Guarantee Of Delivery is based on one question: “Do I need a guarantee that the signal sent by a particular goroutine has been received?”

Unbuffered -> Guaranteed Delivery

Buffered -> No Guaranteed Delivery

The behavior of a channel is directly influenced by its current State. The state of a channel can be **nil**, **open** or **closed**.

When a channel is in a **nil** state, any send or receive attempted on the channel will block.

When a channel is in an **open** state, signals can be sent and received. When a channel is placed into a **closed** state, signals can no longer be sent but it’s still possible to receive signals.

The three channel options are **Unbuffered**, **Buffered >1** or **Buffered =1**.

* **Guarantee**
  + An **Unbuffered** channel gives you a **Guarantee** that a signal being sent has been received.
    - Because the **Receive** of the signal **Happens Before** the **Send** of the signal completes.
* **No Guarantee**
  + A **Buffered** channel of size **>1** gives you **No Guarantee** that a signal being sent has been received.
    - Because the **Send** of the signal **Happens Before** the **Receive** of the signal completes.
* **Delayed Guarantee**
  + A **Buffered** channel of size **=1** gives you a **Delayed Guarantee**. It can guarantee that the previous signal that was sent has been received.
    - Because the **Receive** of the **First Signal, Happens Before** the **Send** of the **Second Signal** completes.

## **What is a pipeline?**

There’s no formal definition of a pipeline in Go; it’s just one of many kinds of concurrent programs. Informally, a pipeline is a series of *stages* connected by channels, where each stage is a group of goroutines running the same function. In each stage, the goroutines

* receive values from *upstream* via *inbound* channels
* perform some function on that data, usually producing new values
* send values *downstream* via *outbound* channels

**HTTP**

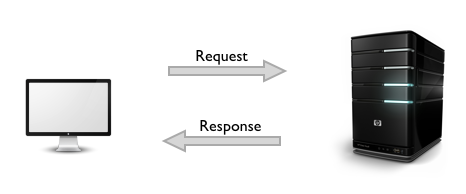
HTTP -> Normal Plain Text

HTTPs -> Encrypted Text

HTTP stands for Hypertext Transfer Protocol.

It's a stateless, application-layer protocol for communicating between distributed systems, and it's the foundation of the modern web.

When you open a website in a browser, you see text, images, and embedded content. All this content is loaded from servers elsewhere on the web. It is the role of the browser—also called a client—to raise a request for this content. The request is sent to a server, which in return sends a response back to the browser. Both the request and the response are sent as human-readable text.



Every request raised by the browser is independent. The HTTP protocol is stateless. That means that each individual request needs to carry all the information needed to fulfill it. In an HTTP request, this information is passed through headers.

The requests are sent, and responses are received over the TCP/IP layer.

The default port for HTTP communication is port 80, but this can be configured differently for different applications.

**HTTP/2.0:** This version of HTTP allows the client to send multiple requests simultaneously.

This technique is known as **multiplexing.**

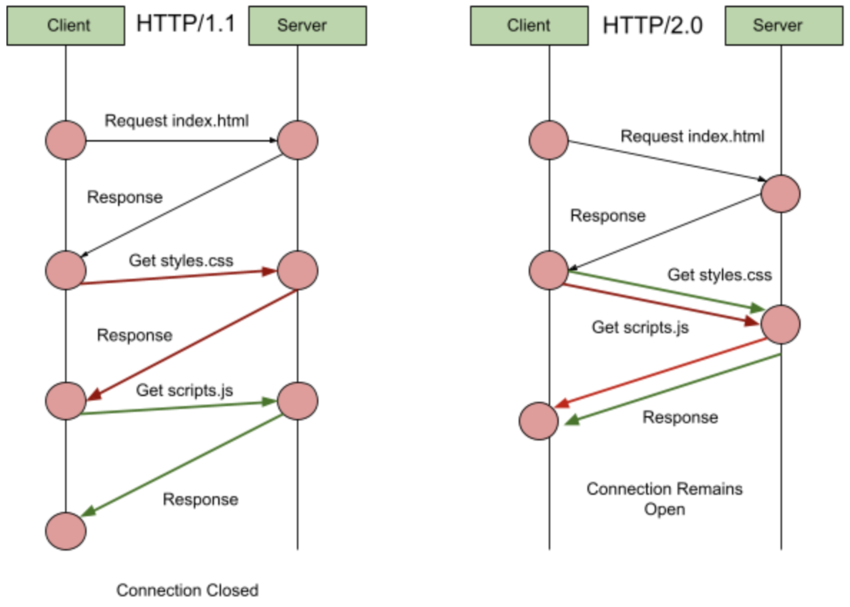
It cuts down on the time required to load a page.

**HTTP/1.1:** This version allows only a single outstanding request with every TCP/IP connection.

This mechanism is also known as **baseline.**

**HTTP/0.9:** was the **original version of the protocol.**

Currently, this version is completely deprecated.



HTTP/2.0 comes with **smart packet management strategies** and **header compression mechanisms to cut down on latency**.

## **HTTP URL:**

## Requests are sent to servers specified with Uniform Resource Locators (URLs).

## [**https://www.domain.com:1234/path/to/resource?a=b&x=y**](https://www.domain.com:1234/path/to/resource?a=b&x=y)

* **https** specifies the protocol. It can be http or https, which makes the communication secure.
* **www.domain.com** is the **host.**
* **1234** is the **port**. In many cases, the browser hides the port. The default is 80.
* **path/to/resource** is the **resource path**. It helps the server identify a specific resource and generate the right response.
* **?a=b&x=y** are **query string parameters**. Query string parameters are used by the server to spot the right resource.

The **URL identifies the specific host** with which the client wants to communicate. It does **not perform any action.** This is where the request comes.

## **HTTP Requests**

At the **heart** of web communications is the request message. A request is made up of the following parts:

* **request line**: this says what is being requested. It consists of **a verb, a path, and the HTTP version**. The HTTP verb says what action is requested of the host, e.g. to GET a resource or POST form data.
* **headers**: **additional information about the message**, the requester, and the communication format.
* **body** (optional): the content of the request. **For a simple request for a static resource like a web page, this will be empty.** For a form submission, this will contain the information from the form. The body is separated from the headers by a blank line.

Here's a typical HTTP request:

GET /articles/http-basics HTTP/1.1

Host: www.articles.com

Connection: keep-alive

Cache-Control: no-cache

Pragma: no-cache

Accept: text/html, application/xhtml+xml, application/xml;q=0.9, \*/\*;q=0.8

The first line has the verb, resource path, and HTTP version.

In this case, we are trying to GET the resource at **/articles/http-basics**.

The rest of the request lines are headers—this request has no body.

### **HTTP Request Verbs**

There are **four universally applicable HTTP verbs** in a request:

* **GET**: **fetch a resource** from the server. For a GET request, the URL should carry all the required pieces of information for the server to spot the right resource. It does not have a message body.
* **POST**: **create a new resource**. The request has an optional payload which helps the server create a new resource.
* **PUT**: **update an existing resource.** The request should have an optional payload to help the server update an existing resource.
* **DELETE**:delete an existing resource.

Interestingly, PUT and DELETE are sometimes considered as specialized versions of the POST verb.

There are some less-used verbs too. A few to consider are:

* **HEAD** is similar to GET, but without the message body. It's used to retrieve the server headers for a particular resource, generally to check if the resource has changed, via timestamps.
* **TRACE** is used to **retrieve the hops that a request takes during a round trip from the server.** Each intermediate proxy or gateway would inject its IP or DNS name into the Via header field. This can be used for diagnostic purposes.
* **OPTIONS** is used to retrieve server capabilities. On the client side, it can be used to modify the request based on what the server supports.

**In Trace->**

The hop count refers to the **number of network devices** through which data passes from source to destination.

## **What is DNS?**

The Domain Name System (DNS) is the phonebook of the Internet.

Humans access information online through [domain names](https://www.cloudflare.com/learning/dns/glossary/what-is-a-domain-name/), like nytimes.com or espn.com. Web browsers interact through [Internet Protocol (IP)](https://www.cloudflare.com/learning/network-layer/internet-protocol/) addresses. DNS translates domain names to [IP addresses](https://www.cloudflare.com/learning/dns/glossary/what-is-my-ip-address/) so browsers can load Internet resources.

Each device connected to the Internet has a unique IP address which other machines use to find the device.

DNS servers eliminate the need for humans to memorize IP addresses such as 192.168.1.1 (in IPv4), or more complex newer alphanumeric IP addresses such as 2400:cb00:2048:1::c629:d7a2 (in IPv6).

**CDN:**

A **content delivery network** is a distributed group of servers that caches content near end users.

A content delivery network, or content distribution network, is a geographically distributed network of proxy servers and their data centers. The goal is to provide high availability and performance by distributing the service spatially relative to end use

**Keep Alive**

A keepalive is a message sent by one device to another to check that the link between the two is operating, or to prevent the link from being broken.

### **HTTP Headers**

HTTP headers give the server information about the sender, the way the client wants to interact, and the message. Each header is a name-value pair. The HTTP protocol specifies all the valid HTTP headers the client and server can use.

* **Cache-Control**: a directive that controls how caching happens in CDNs, proxies, or browsers. It became effective from HTTP/1.1.
* **Connection**: used to decide if the network connection needs to be closed or open once a request is completed. Possible values are keep-alive or closed.
* **Pragma** is an interesting and heavily implementation-specific header. It is provided only for backwards compatibility with HTTP/1.0, which does not support Cache-Control.
* **Trailer**:tells the server it can append metadata to the message body, for example an integrity check or digital signature.
* **Transfer-encoding**:defines the encoding of the payload transferred from the server. Often, this is known as the hop-by-hopheader because the encoding is applied between nodes, and not between the server and client.
* **Via** is used in the header to track messages and the capabilities of the client or server.
* **Upgrade** is available only in HTTP/1.1 and above. If the client or server is allowed to shift from one protocol to another, this header has to be set. For example: Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11.

Here are some of the headers that are specific to the request:

* **Accept-** prefixed headers indicate the acceptable media-types, encoding, languages, and character sets on the client.
* **From**, **Host**, **Referer**, and **User-Agent** have details about the client that initiated the request.
* **Authorization**: used by the client to provide credentials which can be further used by the server to authenticate the request. This is useful for accessing password-protected resources.
* **If-** prefixed headers are used to make a request conditional—the server returns the resource only if the condition matches. Otherwise, it returns a 304 Not Modified.
* **Referrer**: contains either the partial or absolute address of the requesting page.

## **HTTP Responses**

The response is similar in structure to a request message, except for the status line and headers.

* **status line**: includes a status code that indicates whether the request succeeded (status code 200) or why the request failed. It also includes the HTTP version and a very brief description of the status.
* **headers**: additional information about the response—for example, the content type or information about the server.
* **body** (optional): the content of the response. For example, this might be the HTML content of a requested web page or the binary data of an image.

A successful response from the server will have a status line similar to HTTP/1.1 200 OK

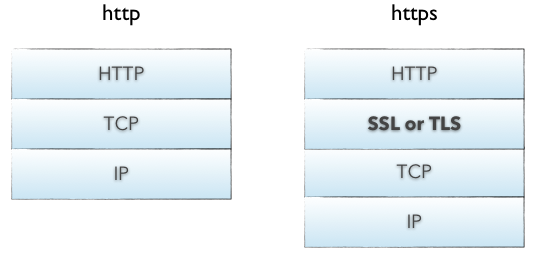
**A web debugging proxy** is a tool that sits between a client (such as a web browser) and a web server, allowing developers and testers to intercept, inspect, and modify the traffic between them. It acts as an intermediary, capturing and analyzing HTTP or HTTPS requests and responses exchanged during web communication

## **HTTP Connections**

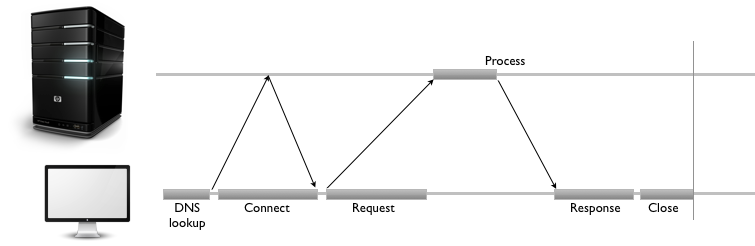
A connection must be established between the client and server before they can communicate with each other, and HTTP uses the reliable TCP transport protocol to make this connection.

By default, web traffic uses TCP port 80. A TCP stream is broken into IP packets, and it ensures that those packets always arrive in the correct order without fail. HTTP is an application layer protocol over TCP, which is over IP.

**HTTPS is a secure version of HTTP**, inserting an additional layer between HTTP and TCP called TLS or SSL (**Transport Layer Security** or **Secure Sockets Layer**, respectively). HTTPS communicates over port 443 by default, and we will look at HTTPS later in this article.



An HTTP connection is identified by <source-IP, source-port> and <destination-IP, destination-port>. On a client, an HTTP application is identified by a <IP, port> tuple. Establishing a connection between two endpoints is a multi-step process and involves the following:



* resolve IP address from host name via DNS
* establish a connection with the server
* send a request
* wait for a response
* close connection

**The server is responsible for always responding with the correct headers and responses.**

**A persistent connection,**

also known as a Hypertext Transfer Protocol ([HTTP](https://www.techtarget.com/whatis/definition/HTTP-Hypertext-Transfer-Protocol)) persistent connection, refers to a network communication channel that remains open for further HTTP requests and responses instead of closing after a single exchange. Persistent connections are also called HTTP keep-alive and HTTP connection reuse.

The likelihood of raising requests in the near future is called **site locality**.

To reduce connection-establishment delays, HTTP/1.1 introduced **persistent connections**, long-lived connections that stay open until the client closes them. Persistent connections are the default in HTTP/1.1, and making a single transaction connection requires the client to set the Connection: close request header. This tells the server to close the connection after sending the response.

To achieve this, HTTP/1.1 keeps TCP connections open, even after a transaction is complete. The existing connection will be reused for future references. This is known as a **persistent connection**.

**The persistent connection will remain open until the server or client decides to close the connection.**

The benefits of using persistent connections are:

* avoid delays due to slow setup
* avoid slow start congestion during the adaptation phase
* ensure faster data transfer

The rules and restrictions of a persistent connection are:

* Once the close header is sent, the connection cannot be reused.
* If the client does not intend to send another request, the close request header should be sent.
* All messages should be self-defined and correct to maintain a persistent connection.
* If the client connects using the HTTP/1.0 protocol, the persistent connection cannot be used.

### **Parallel Connections**

In addition to persistent connections, browsers/clients also employ a technique, called **parallel connections**, to minimize network delays.

The age-old concept of parallel connections involves creating a pool of connections (generally capped at six connections). If there are six assets that the client needs to download from a website, the client makes six parallel connections to download those assets, resulting in a faster turnaround. This is a huge improvement over serial connections where the client only downloads an asset after completing the download for a previous asset.

## **HTTP Authentication**

It is often mandatory to know who connects to a server for tracking an app's or site's usage and the general interaction patterns of users. The premise of identification is to tailor the response to provide a personalized experience; naturally, the server must know who a user is in order to provide that functionality.

There are a few different ways a server can collect this information, and most websites use a hybrid of these approaches:

* **Request headers**: From, Referer, and User-Agent—we saw these headers in [Part 1 of this tutorial](https://code.tutsplus.com/http-the-protocol-every-web-developer-must-know-part-1--net-31177t).
* **Client-IP:** the IP address of the client.
* **Fat URLs:** storing the state of the current user by modifying the URL and redirecting to a different URL on each click; each click essentially accumulates state.
* **Cookies:** the most popular and non-intrusive approach.

The best way to identify a user is to require them to sign up and log in, but implementing this feature requires some effort by the developer, as well as the user.

Techniques like [OAuth](http://oauth.net/) simplify this type of feature, but it still requires user consent in order to work properly. Authentication plays a large role here, and it is probably the only way to identify and verify the user.

HTTP does support a rudimentary form of authentication called **Basic Authentication**, as well as the more secure **Digest Authentication**.

HTTP **cookies** are small blocks of data created by a web server while a user is browsing a website and placed on the user's computer or other device by the user's web browser.

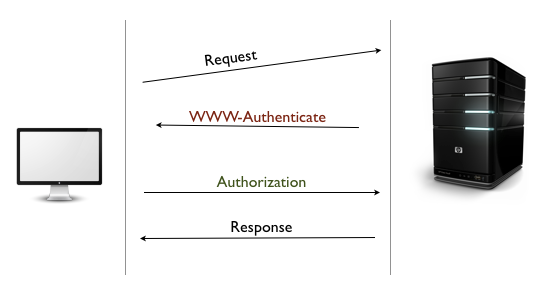
Cookies are placed on the device used to access a website, and more than one cookie may be placed on a user's device during a session.

Cookies are small pieces of text sent to your browser by a website you visit.

They help that website remember information about your visit, which can both make it easier to visit the site again and make the site more useful to you.

### **Basic Authentication**

**In Basic Authentication, the server initially denies the client's request with a WWW-Authenticate response header and a 401 Unauthorized status code. On seeing this header, the browser displays a login dialog, prompting for a username and password. This information is sent in a base-64 encoded format in the Authentication request header. The server can now validate the request and allow access if the credentials are valid. Some servers might also send an Authentication-Info header containing additional authentication details.**

****

* **If the client is not authorised, the server responds with a 401 Unauthorized response status.**
* **If the client wishes to authorise a request, the request should have the Authorization header.**
* **If the client sends valid credentials, without adequate privileges, a 403 Forbidden response is sent. Now, the client does not make a new attempt.**
* **Sometimes, the server fetches information from cookies to authenticate users. This is a common and non-intrusive strategy.**

#### **Tackling the 401 Unauthorised Response**

**The 401 error occurs when a client request was not successfully completed. The request failed because important authentication credentials were not present in the request.**

#### **Authorisation Header**

**Another commonly used method for sending client identity information to the server is through the Authorisation header.**

#### **Authentication Using Cookies**

**Cookies allow the server to attach arbitrary information for outgoing responses via the Set-Cookie response header. A cookie is set with one or more *name=value* pairs separated by a semicolon (;), as in Set-Cookie: session-id=12345ABC; username=nettuts.**

**A server can also restrict the cookies to a specific domain and path, and it can make them persistent with an expires value. Cookies are automatically sent by the browser for each request made to a server, and the browser ensures that only the domain- and path-specific cookies are sent in the request. The request header Cookie: name=value [; name2=value2] is used to send these cookies to the server.**

### **Digest Authentication**

**The strategy behind digest authentication is very similar to basic authentication. Similar handshake techniques around WWW-Authenticate and Authorization Headers are used.**

**From Client: Digest Authentication does not transfer a password to the server. Instead, the client takes the password and the username. Then, it applies an MD5 Hashing Algorithm to build a hash using the username and password. This hash is sent to the server.**

**At Server: The algorithm used to build the hash is used by the server to decode the password and username. The server searches for this combination in the database. If a valid record is found, access is granted. Else, a 401 or 403 error response is returned.**

**Benefit: One of the most benefits of the Digest Authentication algorithm is the ability to avoid replay attacks. How? The server links every request to a one-time number.**

## **HTTP Caching**

**HTTP caching is crucial since it cuts down on the bandwidth and cost of performing operations. Consequently, it improves a person's online experience. While using the cache functionality, however, it is important to acknowledge that resources don't remain the same forever.**

**A resource should be cached only until it is valid.**

**Caches are used at several places in the network infrastructure, from the browser to the origin server.**

**Depending on where it is located, a cache can be categorized as public or private:**

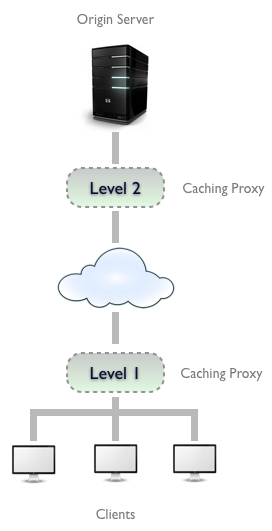
**Types of Caching**

There are two types of caches:

1. Public Cache: stores the server **response for multiple users**. This calls for customised infrastructure to allow a user to access the popular resource several times.
2. Private Cache: **limited to a single user.** The resource would be stored in the user's browser. As the user navigates, the resource will be loaded without multiple trips to the server. Caching makes content available, even when the user is offline. Commonly cached data includes usernames, passwords, URLs, browsing history, and web content.

Public caches are deployed as caching proxies between the server and client. These are much larger because they serve multiple users. A common practice is to keep multiple caching proxies between the client and the origin-server. This helps to serve frequently accessed content, while still allowing a trip to the server for infrequently needed content.

**Keeping the content fresh and up-to-date is one of the primary responsibilities of the cache.**

****

**where it is located, a cache can be categorized as public or private:**

#### 

#### 

#### **Document Expiration**

**HTTP allows an origin-server to attach an expiration date to each document using the Cache-Control and Expires *response* headers. This helps the client and other cache servers know how long a document is valid and fresh. The cache can serve the copy as long as the *age* of the document is within the expiration date. Once a document expires, the cache must check with the server for a newer copy and update its local copy accordingly.**

#### **Server Revalidation**

**Once a cached document expires, the cache must revalidate with the server to check if the document has changed. This is called *server revalidation* and serves as a querying mechanism for the *staleness* of a document. Just because a cached copy has expired doesn't mean that the server actually has newer content. Revalidation is just a means of ensuring that the cache stays fresh. Because of the expiration time (as specified in a previous server response), the cache doesn't have to check with the server for every single request, thus saving bandwidth and time and reducing the network traffic.**

**The combination of document expiration and server revalidation is a very effective mechanism, and it allows distributed systems to maintain copies with an expiration date.**

## **Secure HTTP**

**Our last area of focus is HTTP security with HTTPS. It is extremely important to make sure the web application is secure. Even if the code contains small issues, plenty of private data can get leaked. This is why securing HTTP is more than a luxury.**

### **HTTPS**

**The HTTPS protocol provides a secure connection on the web. The easiest way to know if you are using HTTPS is to check the browser's address bar. HTTPS's secure component involves inserting a layer of encryption/decryption between HTTP and TCP. This is the Secure Sockets Layer (SSL) or the improved Transport Layer Security (TLS).**

**Web Server with Golang**

**The name mux stands for "HTTP request multiplexer". Like the standard http. ServeMux, mux. Router matches incoming requests against a list of registered routes and calls a handler for the route that matches the URL or other conditions.**