Developed by google

**Features**

Fast compilation

It is simple, safe, conscious

Support for environment adopting pattern

Lightweight processing

Production of statically linked native binaries without external dependencies

**Features excluded intentionally**

Support for type inherited

Support for method or operator overloading

Support for circular dependencies among packages

Pointer arithmetics

Assertions

Support for generic programming

**How program written**

With extension .go

Can use vi or vin editor

**The go compiler**

Install GO in your relevant PC like linux, windows, mac os

Golang

**Packages and modules**

Packages are gos way of organizing

Programs are written in as one or more packages

Packages are inported from the go package registery

packages should be focused and perform single thing

- argument passing

- Drawing graphics

- Handling http request

**Using packages**

import “name”

for ex

import (

“name”

“namespace/packagename”

)

Can import everything using dot (.)

No need to reference package name in code

Import can be renamed

import (

. “name” // Can importeverything using dot

pk “namespace/packagename” // can rename package name with pk

)

**Modules**

Modules are the collection of packages

Created by using the go.mod file in the root directory of your project

Can be managed by go cli

Contain information about your project

Dependancies, go versions, package info

All go program have go.mod file

**Example module**

module example.com/practice

go 1.17

require

(

github.com/alexflint/go-arg v1.4.2

github.com/fatih/color v1.13.0

)

Hello world program

import "fmt"

func main() {

fmt.Println("Hello Beautifull world")

}

**String**

String are slice of byte.

So string are slices

Go will provide various libraries to manipulate string

unicode

regexp

strings

Creating string

var grreting = “Hello World!”

Check the length of string

fmt.Println(len(greeting))

Concating string

package main

import (

"fmt"

"strings"

)

func main() {

greeting := []string{"Hello", "World"}

fmt.Println(strings.Join(greeting, ""))

fmt.Printf("%+q\n", greeting)

fmt.Printf("%x\n", greeting)

}

HelloWorld

["Hello" "World"]

[48656c6c6f 576f726c64]

**Go slices**

Go slices are the abstraction over the go array.

Array will allows you to define several data items of same kind

But does not provide increase the size dynamically or to get sub array of its own

Slices overcome this limitation

var numbers []int /\* a slice of unspecified size \*/

/\* numbers == []int{0,0,0,0,0}\*/

numbers = make([]int,5,5) /\* a slice of length 5 and capacity 5\*/

**Defining the slice**

Declare the array without specifying the size will be slice

Alternatively can create the make function too

package main

import (

"fmt"

)

func main() {

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

var number1 = make([]int, 3, 5) //3 is length and 5 is the capacity here

fmt.Println(number)

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

fmt.Println(number1)

fmt.Println(len(number1), cap(number1))

var number2 []int //3 is length and 5 is the capacity here

if number2 == nil {

fmt.Printf("SLice is nil \n")

fmt.Println(number2)

}

}

[0 1 2 3 4]

5 5

[0 0 0]

3 5

SLice is nil

[]

**Subslice**

Subslice allows to create new lice from current slice use upper bound and lower bound limits as per below

**[lower-bound:upper-bound]**

package main

import "fmt"

func main() {

numbers := []int{0, 1, 2, 3, 4, 5, 6, 7, 8}

fmt.Println(numbers)

fmt.Println(numbers[2:3])

fmt.Println(numbers[:3])

fmt.Println(numbers[4:])

numbers1 := make([]int, 0, 5)

fmt.Println(numbers1)

number2 := numbers[1:5]

fmt.Println(number2)

number3 := numbers[3:]

fmt.Println(number3)

}

[0 1 2 3 4 5 6 7 8]

[2]

[0 1 2]

[4 5 6 7 8]

[]

[1 2 3 4]

[3 4 5 6 7 8]

**Slice append and copy**

package main

import "fmt"

func main() {

var number []int

fmt.Printf("Len = %d , Cap = %d , slice = %v \n", len(number), cap(number), number)

number = append(number, 0)

number = append(number, 1)

number = append(number, 2)

number = append(number, 3, 4, 5)

fmt.Printf("Len = %d , Cap = %d , slice = %v \n", len(number), cap(number), number)

numbers1 := make([]int, len(number), (cap(number))\*2)

copy(numbers1, number)

fmt.Printf("Len = %d, Cap = %d , slice = %v \n", len(numbers1), cap(numbers1), numbers1)

}

Len = 0 , Cap = 0 , slice = []

Len = 6 , Cap = 8 , slice = [0 1 2 3 4 5]

Len = 6, Cap = 16 , slice = [0 1 2 3 4 5]

**Function in go**

package main

import (

"fmt"

)

func main() {

var s string

fmt.Println("Hello Beautifull world")

s = passthestring()

fmt.Println("String from function is", s)

}

func passthestring() string {

return "Goodbye"

}

Hello Beautifull world

String from function is Goodbye

package main

import (

"fmt"

)

func main() {

//var s string

//var t string

fmt.Println("Hello Beautifull world")

s, t := passthestring()

fmt.Println("String from function is", s, t)

}

func passthestring() (string, string) {

return "Goodbye", "World"

}

Hello Beautifull world

String from function is Goodbye World

Function as a pointer

package main

import (

"fmt"

)

func main() {

var a string

passString(&a)

fmt.Println("Value come from function is", a)

}

func passString(b \*string) {

\*b = "name"

}

Value come from function is name

package main

import (

"fmt"

)

func main() {

var x int = 5748

var p \*int

p = &x

fmt.Println(x)

fmt.Println(&x)

fmt.Println(p)

}

5748

0xc0000aa058

0xc0000aa058

Function array

package main

import "fmt"

func main() {

a := []int{0, 5, 3, 3}

var j [4]\*int

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

j[i] = &a[i]

}

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

fmt.Println(i, \*j[i])

}

}

0 0

1 5

2 3

3 3

**Pointer on pointer in go**

**A pointer to a pointer is a form of chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.**

****

**package main**

**import "fmt"**

**func main() {**

**var a int = 10**

**var ptr \*int**

**var pptr \*\*int**

**ptr = &a**

**pptr = &ptr**

**fmt.Println(a)**

**fmt.Println(\*ptr)**

**fmt.Println(\*\*pptr)**

**}**

**10**

**10**

**10**

**Passing pointer to function**

**package main**

**import "fmt"**

**func main() {**

**var x int = 100**

**var y int = 200**

**fmt.Println(x, y)**

**swap(&x, &y)**

**fmt.Println(x, y)**

**}**

**func swap(a \*int, b \*int) int {**

**var tmp int**

**tmp = \*b**

**\*b = \*a**

**\*a = tmp**

**return 0**

**}**

**100 200**

**200 100**

**Structor**

package main

import (

"fmt"

"time"

)

type book struct {

author string

name string

revision int

yearrelease time.Time

}

func main() {

bookdetail := book{

author: "yagnik",

name: "Golang",

revision: 3,

}

fmt.Println(bookdetail.author)

fmt.Println(bookdetail.name)

fmt.Println(bookdetail.revision, bookdetail.yearrelease)

}

yagnik

Golang

3 0001-01-01 00:00:00 +0000 UTC

**Structor as a function**

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the same way as you did in the above example −

package main

import "fmt"

type book struct {

author string

publisher string

revision int

}

func main() {

var book1 book

var book2 book

book1.author = "yagnik"

book1.publisher = "milman"

book1.revision = 1

book2.author = "mahamad"

book2.publisher = "billigine"

book2.revision = 2

print\_func(book1)

print\_func(book2)

}

func print\_func(book\_detail book) {

fmt.Printf("%s \n", book\_detail.author)

fmt.Printf("%s \n", book\_detail.publisher)

fmt.Printf("%d \n", book\_detail.revision)

}

yagnik

milman

1

mahamad

billigine

2

**Structor as a pointer function**

**You can define pointers to structures in the same way as you define pointer to any other variable as follows −**

var struct\_pointer \*Books

struct\_pointer = &Book1;

struct\_pointer.title;

package main

import "fmt"

type book struct {

author string

publisher string

revision int

}

func main() {

var book1 book

var book2 book

book1.author = "yagnik"

book1.publisher = "milman"

book1.revision = 1

book2.author = "mahamad"

book2.publisher = "billigine"

book2.revision = 2

print\_func(&book1)

print\_func(&book2)

}

func print\_func(book\_detail \*book) {

fmt.Printf("%s \n", book\_detail.author)

fmt.Printf("%s \n", book\_detail.publisher)

fmt.Printf("%d \n", book\_detail.revision)

}

yagnik

milman

1

mahamad

billigine

2

**Datatypes**

It is the way that program can interpret the binary numbers

For ex numbers, letters,

Go uses type interferance to determine what type of data it is working with

**Signed integer**

int8 -128 to 127

int16 -32768 to 32767

int //int and int32 both are 32 bit by default

int32

int64

**Unsigned integers**

uint8 0 to 255

uint16 0 to 65535

uint //uint and uint32 both are same

uint32

uint64

byte 0 to 255

uintptr 0 to ptr size

**Other datatypes**

float32

float64

complex64

complex128

bool true or false

**Hello world in go**

Package main /\* package declaration \*/

Import “fmt” /\* preprocessor 8/

Func main()

{

fmt.Println(“Hello world”)

}

**Go program structure**

It contains following parts

* Package declaration
* Import packages
* Functions
* Variables
* Statements and expressions
* Comments

Go will runs with packages

Each package has its path and name associated with it

**Token in go**

Token is either keyword, an identifier, constants, string literature, or a symbol

For ex below statement consists of six tokens

fmt.Println(“Hello World!”)

For example individual tokens are

Fmt

Println

(

“Hello World”

)

**Line separator**

fmt.Println(“Hello, WOrld”)

fmt.Println(“I am in go programming world!”)

**Comments**

/\* my first program in go \*/

**Identifier**

Identifier = letter {letter | unicode\_digit}.

Go does not allow the punctuation character such as @, $, %

Go is the case sensitive programming language

Thus Manpower and manpower are 2 different identifiers

Here are some of the acceptable identifiers

mahesh kumar abc move\_name a\_123

myname50 \_temp j a3b9 retvl

**Keywords in go**

break

default

func

interface

select

case

defer

Go

map

Struct

chan

else

Goto

package

Switch

const

fallthrough

if

range

Type

continue

for

import

return

Var

**Whitespaces in Go**

It will used in go to describe blanks, tabs, new line characters and comments etc.

Line containing only white spaces possibly with a comments is known as blank line

var age int;

fruit = apples + oranges; //Get the total fruits

No white spaces is necessary between fruit and = or between = and apples

It is free to include if you wish for readability purposes

**GO Datatypes**

**Boolean** Consists of 2 predefined constants a true b false

**Derived** Arithmetics types, integer types or floating point types

**string** Sequence of byte It is immutable types Not possible to change the type of the string

**numeric** pointer, array, structor, union, function, slice, map, channel

**Go type conversion**

Type conversion is the way to convert one data type to another datatype.

If need to store the long value into simple integer then can type cast long to int

type\_name(expression)

package main

import "fmt"

func main() {

var value1 int = 17

var value2 int = 5

var output float32

output = float32(value1) / float32(value2)

fmt.Printf("Value of output is %f", output)

}

Value of output is 3.400000

**Go Array**

Go supports data structor called array

Which store fixed sequential bytes of same type of element

Declaration of array

var var\_name [size] type.

var var\_name [size] type{value1, value2, value3}

var name[3] string

var balance = [5]float32{1.1, 2.3, 5.4, 17.5, 5.2}

var balance = []float32{1.1, 2.3, 5.4, 17.5, 5.2}

var balance[4] = 17.5

package main

import "fmt"

func main() {

var n [11]int

var i, j int

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

n[i] = i + 100

}

for j = 0; j < 10; j++ {

fmt.Println(j, n[j])

}

}

0 100

1 101

2 102

3 103

4 104

5 105

6 106

7 107

8 108

9 109

package main

import (

"fmt"

)

func main() {

array := []string{"my", "name", "is", "yagnik"}

/\*

array = []string

array[0] = "my"

array[1] = "name"

array[2] = "is"

array[3] = "yagnik"

fmt.Println("Elements of Array:")

fmt.Println("Element 1: ", array[2])

\*/

// printing simple array

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

fmt.Printf(array[i])

}

}

mynameisyagnik

package main

import "fmt"

func main() {

// 5 row 2 column

a := [5][2]int{{0, 0}, {1, 5}, {9, 5}, {6, 2}, {7, 2}}

for i := 0; i < 5; i++ {

for j := 0; j < 2; j++ {

fmt.Printf("a[%d][%d] = %d \n", i, j, a[i][j])

}

}

}

a[0][0] = 0

a[0][1] = 0

a[1][0] = 1

a[1][1] = 5

a[2][0] = 9

a[2][1] = 5

a[3][0] = 6

a[3][1] = 2

a[4][0] = 7

a[4][1] = 2

Go will allows multi dimensional array

var var\_name[size1] [size2] [size3] …. [sizen] variable\_type

2D array

var arrayName [ x ][ y ] variable\_type

initialization of 2D array

a = [3] [4] int{

{0,1,2,3},

{4,5,3,6},

{8,4,3,7}

}

**Go pointers**

Go tasks easily perform by the pointer

Some cases such as call by reference will not perform without pointer

Every variable has a memory location

Memory location has address and can be accesses by & which is the address of mlocation

A **Pointers are the variable whose value is the address of another memory location**

var var\_name \*var-type

package main

import "fmt"

func main() {

var x int = 20

var y \*int

fmt.Println(y) // THis is nil pointer where we hav not allocated the adress just we initialized

y = &x

fmt.Println(&x)

fmt.Println(y)

fmt.Println(\*y)

}

<nil>

0xc000014088

0xc000014088

20

**Passing array to function**

**void myFunction(param [10]int)**

**{**

**.**

**.**

**.**

**}**

**void myFunction(param []int)**

**{**

**.**

**.**

**.**

**}**

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following two ways and all two declaration methods produce similar results because each tells the compiler that an integer array is going to be received. Similar way you can pass multi-dimensional array as formal parameters.

package main

import "fmt"

func main() {

a := []int{0, 100, 52, 30}

x := average(a, 4)

fmt.Println(x)

}

func average(y []int, size int) float64 {

var b float64

var sum int

for i := 0; i < size; i++ {

sum += y[i]

}

b = (float64)(sum / size)

return b

}

45

**Literature**

**Integer literals**

It can be decimal, octal, hexadecimal constant.

0x or 0X for hexa decimal

0 for octal

Nothing for decimal

212 // legal decimal

0213 // octal

0x4b // hexadecimal

30l // long

30ul // unsigned long

215u // legal unsigned integer

0xFeeL // legal

078 // illlegal octal digit

032UU // illegal octal digit

**Floating-point literature**

It is the part of floating point, fractional point and exponent part

3.14159 // legal

31459E-5L // legal

510E // illegal

210F // illegal

.e55 // illegal

**String literature in go**

“Hello, Dear”

“ Hello, \

dear”

“hello,”

**Const literature**

const var type = value;

const LENGTH = 10

const WIDTH = 5

**Go scope rules**

**Local variable**

**Globle variable**

**Firmal parameters**

**Local variable**

Inside th function is called as local variable

import "fmt"

/\* global variable declaration \*/

var g int

func main() {

/\* local variable declaration \*/

var a, b int

/\* actual initialization \*/

a = 10

b = 20

g = a + b

fmt.Printf("value of a = %d, b = %d and g = %d\n", a, b, g)

}

value of a = 10, b = 20 and g = 30

**Globle variable**

But local variable inside the main has higher preference hence output will be 10 instead of 20

package main

import "fmt"

/\* global variable declaration \*/

var g int = 20

func main() {

/\* local variable declaration \*/

var g int = 10

fmt.Printf ("value of g = %d\n", g)

}

value of g = 10

**Formal parameters**

Formal parameters says always stick to value in main variable if we used the same value in any other function.

Let us say variable a is declared in global and local both

And same called by the function

Then function will take priority from local only. See the below program

Formal parameters are treated as local variables with-in that function and they take preference over the global variables. For example −

package main

import "fmt"

/\* global variable declaration \*/

var a int = 20;

func main() {

/\* local variable declaration in main function \*/

var a int = 10 **// This value is always a preference**

var b int = 20

var c int = 0

fmt.Printf("value of a in main() = %d\n", a);

c = sum( a, b);

fmt.Printf("value of c in main() = %d\n", c);

}

/\* function to add two integers \*/

func sum(a, b int) int {

fmt.Printf("value of a in sum() = %d\n", a);

fmt.Printf("value of b in sum() = %d\n", b);

return a + b;

}

value of a in main() = 10

value of a in sum() = 10

value of b in sum() = 20

value of c in main() = 30

For loop as while loop

package main

import "fmt"

func main() {

var i int32

i = 0

for i < 5 {

fmt.Println("This loop runs five time")

i++

}

}

This loop runs five time

This loop runs five time

This loop runs five time

This loop runs five time

This loop runs five time

For loop as a do while loop

//There is no do while loop in the go

// There are few ways with the help of for loop we can define do loop

package main

import "fmt"

func main() {

var i int = 0

for {

fmt.Println("This loop will run 5 times", i)

i++

if i >= 5 {

break

}

}

}

PS C:\Go\_WorkSpace\forasdowhile> go run forasdowhile.go

This loop will run 5 times 0

This loop will run 5 times 1

This loop will run 5 times 2

This loop will run 5 times 3

This loop will run 5 times 4

Break statement in go

package main

import "fmt"

func main() {

var i int = 10

for {

fmt.Println(i)

i++

if i > 15 {

break

}

}

}

PS C:\Go\_WorkSpace\breakloop> go run break.go

10

11

12

13

14

15

/\*package main

import "fmt"

func main() {

var i int = 10

for i < 20 {

i++

fmt.Println(i)

//continue

}

}\*/

package main

import "fmt"

func main() {

/\* local variable definition \*/

var a int = 10

/\* do loop execution \*/

for a < 20 {

if a == 15 {

/\* skip the iteration \*/

a = a + 1

continue

}

fmt.Printf("value of a: %d\n", a)

a++

}

}

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

Go to statement

package main

import "fmt"

func main() {

learnGoTo()

}

func learnGoTo() {

fmt.Println("a")

goto FINISH

fmt.Println("b")

FINISH:

fmt.Println("c")

}

PS C:\Go\_WorkSpace\goto> go run goto.go

a

c

**Go range**

Range keyword is used to iterate over items of an array, slice, channel or map.

With array and slice it will return the index of the item as integer.

With maps it will return the key of the next key-pair.

Range either return the once value or two.

| **Range expression** | **1st Value** | **2nd Value(Optional)** |
| --- | --- | --- |
| Array or slice a [n]E | index i int | a[i] E |
| String s string type | index i int | rune int |
| map m map[K]V | key k K | value m[k] V |
| channel c chan E | element e E | none |

package main

import "fmt"

func main() {

// creating a slice

numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

for i := range numbers {

fmt.Println(numbers[i])

}

mystateCitymap := map[string]string{"gujarat": "ahmedabad", "kernataka": "banglore", "maharastra": "mumbai"}

//Print map using keys

for state := range mystateCitymap {

fmt.Println("capital city of", state, "is", mystateCitymap[state])

}

//Print map using the key value

for state, city := range mystateCitymap {

fmt.Println("capital city of", state, "is", city)

}

}

1

2

3

4

5

6

7

8

9

10

capital city of gujarat is ahmedabad

capital city of kernataka is banglore

capital city of maharastra is mumbai

capital city of gujarat is ahmedabad

capital city of kernataka is banglore

capital city of maharastra is mumbai

Function returns the maximum value

package main

import "fmt"

func main() {

var a int = 10

var b int = 20

c := max(a, b)

fmt.Println(c)

}

func max(num1 int, num2 int) int {

var result int

if num1 > num2 {

result = num1

} else {

result = num2

}

return result

}

20

Swap the value with function and passing 2 values to function and get 2 values from function

package main

import "fmt"

func main() {

a, b := swap("casey", "jacob")

fmt.Println(a, b)

}

func swap(value1, value2 string) (string, string) {

return value2, value1

}

jacob casey

Call by value function

package main

import (

"fmt"

)

func main() {

var i int = 10

var j int = 20

fmt.Println("before swap", i)

fmt.Println("before swap", j)

swap(i, j)

fmt.Println("after swap", i)

fmt.Println("after swap", j)

// output will not change after and before swap

}

func swap(value1, value2 int) int {

var temp int

temp = value1

value1 = value2

value2 = temp

return temp

}

before swap 10

before swap 20

after swap 10

after swap 20

Call by reference function

package main

import (

"fmt"

)

func main() {

var i int = 10

var j int = 20

fmt.Println("before swap", i)

fmt.Println("before swap", j)

swap(&i, &j)

fmt.Println("after swap", i)

fmt.Println("after swap", j)

// output will not change after and before swap

}

func swap(value1 \*int, value2 \*int) int {

var temp int

temp = \*value1

\*value1 = \*value2

\*value2 = temp

return temp

}

before swap 10

before swap 20

after swap 20

after swap 10

**Go - functions as values**

Go programming language provides the flexibility to create functions on the fly and use them as values.

package main

import (

"fmt"

"math"

)

func main() {

getSquareRoot := func(x float64) float64 {

return math.Sqrt(x)

}

fmt.Println(getSquareRoot(9))

}

3

**Go function closure**

Go support anonymous function which can acts as a function closure.

package main

import "fmt"

func return\_increment() func() int {

i := 0

return func() int {

i += 1

return i

}

}

func main() {

a := return\_increment()

fmt.Println(a())

fmt.Println(a())

fmt.Println(a())

b := return\_increment()

fmt.Println(b())

fmt.Println(b())

}

1

2

3

1

2

Method in Go

Go programming language supports special types of functions called methods. In method declaration syntax, a "receiver" is present to represent the container of the function. This receiver can be used to call a function using "." operator. For example −

package main

import (

"fmt"

"math"

)

/\* define a circle \*/

type Circle struct {

x, y, radius float64

}

/\* define a method for circle \*/

func (circle Circle) area() float64 {

return math.Pi \* circle.radius \* circle.radius

}

func main() {

circle := Circle{x: 0, y: 0, radius: 5}

fmt.Printf("Circle area: %f", circle.area())

}

Circle area: 78.539816

**Go error handling**

Go will prvides pretty simple error handling framework.

With inbuilt error handling type of following declaration

Inthe function we will use return 2 times.

Now will check if there is error then will give that error, if no error then will give the actual return value.

Both time in return value we will give 2 parameters

in error return will return with 0 + error y default

in actual value will return actual value + nil (Because in main we check if it is nil then there is error else not)

package main

import (

"errors"

"fmt"

"math"

)

func Sqrt(value float64) (float64, error) {

if value < 0 {

return 0, errors.New("Math: negative number passed to Sqrt")

}

return math.Sqrt(value), nil

}

func main() {

result, err := Sqrt(-1)

if err != nil {

fmt.Println(err)

} else {

fmt.Println(result)

}

result, err = Sqrt(9)

if err != nil {

fmt.Println(err)

} else {

fmt.Println(result)

}

}

Math: negative number passed to Sqrt

3

**Go recursion**

Recursion is the process of repeating the item in selfsimilar way

func recursion(){

recursion()

}

func main(){

recursion()

}

Use of factorial in go with recursion

package main

import "fmt"

func factorial(i int) int {

if i <= 1 {

return 1

}

return i \* factorial(i-1)

}

func main() {

x := factorial(4)

fmt.Println(x)

}

24

package main

import "fmt"

func fibonacci(i int) int {

if i == 0 {

return 0

}

if i == 1 {

return 1

}

return fibonacci(i-1) + fibonacci(i-2)

}

func main() {

var i int

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

fmt.Printf("%d \n", fibonacci(i))

}

}

0

1

1

2

3

5

8

13

21

34

**Map**

**Delete function in map**

**package main**

**import "fmt"**

**func main() {**

**stateofCityMap := map[string]string{"Gujarat": "Ahmedabad", "Maharastra": "Mumbai"}**

**for state := range stateofCityMap {**

**fmt.Println("state of", state, "is", stateofCityMap[state])**

**}**

**delete(stateofCityMap, "Gujarat")**

**fmt.Println("After delete")**

**for state := range stateofCityMap {**

**fmt.Println("state of", state, "is", stateofCityMap[state])**

**}**

**}**

state of Gujarat is Ahmedabad

state of Maharastra is Mumbai

After delete

state of Maharastra is Mumbai

**Go Interfaces**

Go provides another datatype called as interface

It represents a set of method signatures

The struct data type implements these interface to have a method definition for the method of the interface

Syntex

type interface\_type name{

method name1[return type]

method name2[return type]

method name3[return type]

}

type struct\_name struct{

variables

}

// implement interface methods

func (struct\_name\_variable struct\_name) method\_name1() [return type]{

// Method implementation

}

func (struct\_name\_variable struct\_name) method\_name() [return type] {

// Method implementation  
}

package main

import (

"fmt"

"math"

)

/\* define an interface \*/

type Shape interface {

area() float64

}

/\* define a circle \*/

type Circle struct {

x, y, radius float64

}

/\* define a rectangle \*/

type Rectangle struct {

width, height float64

}

/\* define a method for circle (implementation of Shape.area())\*/

func (circle Circle) area() float64 {

return math.Pi \* circle.radius \* circle.radius

}

/\* define a method for rectangle (implementation of Shape.area())\*/

func (rect Rectangle) area() float64 {

return rect.width \* rect.height

}

/\* define a method for shape \*/

func getArea(shape Shape) float64 {

return shape.area()

}

func main() {

circle := Circle{x: 0, y: 0, radius: 5}

rectangle := Rectangle{width: 10, height: 5}

fmt.Printf("Circle area: %f\n", getArea(circle))

fmt.Printf("Rectangle area: %f\n", getArea(rectangle))

}

Circle area: 78.539816

Rectangle area: 50.000000

**Go unit testing**

Test your code during the development will expose the bugs

Gos built in function will makes easier to test as you go

It uses the go testing commands and go testing packages

//math.go

// This is a simple module file which will be our code

package math

// Add is our function that sums two integers

func Add(x, y int) (res int) {

return x + y

}

//math\_test.go

//math\_test.go this is testing file and need to import testing module for this

package math

import "testing"

func TestAdd(t \*testing.T) {

got := Add(4, 6)

want := 10

// want := 11 then it will gives error

if got != want {

t.Errorf("got %q, wanted %q", got, want)

}

}

PS C:\Go\_WorkSpace\testingowithmath> go test

PASS

ok \_/C\_/Go\_WorkSpace/testingowithmath 1.855s

**Go logging**

What to log

* Spot bugs in application
* Discover performance problems
* Do the postmortem analysys of outage and security incidents

Some time you needed to log

* Time stamp
* Log level such as debug, error or info
* Contextual data to understand what happen to make it possible to easily reproduce data.

What not to log

* Names
* IP Adress
* Credit card numbers

As per GDPR and HIPPA logging data

**Introducing the log package**

package main

import “log”

func main(){

log.Println(“Hello World”)

}

2019/12/09 17:21:53 Hello World

**Logging to a file**

The below file will create the log file with the name of text.

package main

import (

"log"

"os"

)

func main() {

file, err := os.OpenFile("logs.txt", os.O\_APPEND|os.O\_CREATE|os.O\_WRONLY, 0666)

if err != nil {

log.Fatal(err)

}

log.SetOutput(file)

log.Println("Hello World")

}



**Go Database operations**

Importing a database driver & module

Driver for the sql is sql.Open()

There are drivers for the sqlite3 and postgres to

Before executing the below program there are few things need to install

1. SQL server

mysql-installer-community-8.0.29.0

<https://cdn.mysql.com//Downloads/MySQLInstaller/mysql-installer-community-8.0.29.0.msi>

1. Microsoft SQL server management studio

SSMS-Setup-ENU

<https://download.microsoft.com/download/c/7/c/c7ca93fc-3770-4e4a-8a13-1868cb309166/SSMS-Setup-ENU.exe>

package main

import (

"database/sql"

"fmt"

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

)

func main() {

db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)/employeedb")

if err != nil {

fmt.Println(err)

} else {

fmt.Println("Connection established")

}

defer db.Close()

}

Connecion established

**Go directives**

* **Retract directive**

It means draw back or with draw

Let us assume we publish our module using version control mechanism

In one module suppose did a mistake and released to production with number v0.1.0

After that realise a mistake and publish a new version with v0.2.0

We cant modify the cose in v0.1.0

And there is no way to tell the people that use v0.2.0

This problem will solved by the retract module

Can upgrade module

Can downgrade modules

* **Go module directives**

Applicable in and after version 1.13 of go

It is the new way of adding libraries called go modules

Go module solves the gopath problems

package main

import (

"fmt"

"mymodule/mypackage"

)

func main() {

fmt.Println("Hello, Modules!")

mypackage.PrintHello()

}

package mypackage

import "fmt"

func PrintHello() {

fmt.Println("Hello, Modules! This is mypackage speaking!")

}

Directory: C:\Go\_WorkSpace\projects\mymodules

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 10-05-2022 02:18 PM mypackage

-a---- 10-05-2022 02:14 PM 25 go.mod

-a---- 10-05-2022 02:18 PM 142 main.go

Taken reference from <https://www.digitalocean.com/community/tutorials/how-to-use-go-modules>

**Adding a remote module as a dependencies**

go get github.com/spf13/cobra@07445ea

module mymodule

go 1.16

require (

github.com/inconshreveable/mousetrap v1.0.0 // indirect

github.com/spf13/cobra v1.1.2-0.20210209210842-07445ea179fc // indirect

github.com/spf13/pflag v1.0.5 // indirect

)

go get github.com/spf13/cobra@v1.1.1

module mymodule

go 1.16

require (

github.com/inconshreveable/mousetrap v1.0.0 // indirect

github.com/spf13/cobra v1.1.1 // indirect

github.com/spf13/pflag v1.0.5 // indirect

)

go get github.com/spf13/cobra@latest

module mymodule

go 1.16

require (

github.com/inconshreveable/mousetrap v1.0.0 // indirect

github.com/spf13/cobra v1.2.1 // indirect

github.com/spf13/pflag v1.0.5 // indirect

)

* **Replace directory**

Replace directory will replace the content of the specific version of the midule from other wheres.

If the version present on the left side of the arrow only that specific version is replaced

Replace directory only applied on the main modules go.mod file, ignored by others

If there is multiple main than it will appy to all

right habd side begin with ./ or ../ then it is local path for replacement

Example:

replace golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5

replace (

golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5

golang.org/x/net => example.com/fork/net v1.4.5 //Module with no local path

golang.org/x/net v1.2.3 => ./fork/net

golang.org/x/net => ./fork/net // Local path

)

* **Exclude directory**

Exclude directory prevents the module version loaded by go command

Before Go 1.16 exclude version was reference by the required directory

Exclude directory only applied in main go.mod it will be ignored by others

ExcludeDirective = "exclude" ( ExcludeSpec | "(" newline { ExcludeSpec } ")" newline ) .

ExcludeSpec = ModulePath Version newline .

Example:

exclude golang.org/x/net v1.2.3

exclude (

golang.org/x/crypto v1.4.5

golang.org/x/text v1.6.7

)

* **Require directory**

Required directory declares the minimum required version of the given module dependencies.

Go command loads the go.mod file for required version & incorporate requirement from the file

Go command will automatically adds // indirect comments

Which indicates that no package from the required modules is directly imported by any package in main module

RequireDirective = "require" ( RequireSpec | "(" newline { RequireSpec } ")" newline ) .

RequireSpec = ModulePath Version newline .

Example:

require golang.org/x/net v1.2.3

require (

golang.org/x/crypto v1.4.5 // indirect

golang.org/x/text v1.6.7

)

* **Go directives**

Go directive indicates that a module was written assuming the semantic of a given version of go.

The version is like 1.9, 1.14 etc

The go directive originally intended to support backward incompatibility changes to the go language.

There have no been incompatible language changes since modules was introduced, go directory still affects new language supports

The go.mod file after 1.17 includes an explicite require directive for each module that provides any package transively import by package or test in main module.

As of the Go 1.17 release, if the go directive is missing, go 1.16 is assumed.

GoDirective = "go" GoVersion newline .

GoVersion = string | ident . /\* valid release version; see above \*/

Example:

go 1.14

**GIN Framework**

This will introduced the basic of writing the RESTFULL web services API with go and gin web framework.

GIN will simplifies many coding tasks associated with building with web applications, including web services.

Here we will write the GIN to route request, retrieve request, details and marshal json for responses.

Here we also build RESTful API server with 2 end points.

Example project will be a repository of data about vintage jazz record.

* **Design API end point**

API provides an access to a store selling vintage recording the end points.

Hence need to provide endpoints through which a client can get and add album for users.

/album

* GET Get a list of album, return as JSON
* POST Add a new album from request data sent as json

/albums/:id

* GET Get an albums by ID, returning album data as json.

**Tools**

* **Race detection**

Race conditions are most insidious and elusive programming errors. It often long after the code deployed to mass production

Go 1.1 includes the race detection a new tool for finding the race conditions in go code.

It is currently support linux and windows

The race detector is based on the c/C++ thread sanitizer runtime library. Which will used to detect many errorsin googles internal codes.

when -race command-line will be set the compiler instruments all memory access and record, and see how the memory accessed.

Race enables binaries will use 10 times CPU and memory

Commands

go test -race mypkg // Test the package

go run -race mysrc.go //Compile and run the program

go build -race mycmd //build the command

go install -race mypkg //install the package

package main

import "fmt"

func main() {

done := make(chan bool)

m := make(map[string]string)

m["name"] = "world"

go func() {

m["name"] = "data race"

done <- true

}()

fmt.Println("Hello,", m["name"])

<-done

}

go run -race racy.go

func main() {

11 start := time.Now()

12 var t \*time.Timer

13 t = time.AfterFunc(randomDuration(), func() {

14 fmt.Println(time.Now().Sub(start))

15 t.Reset(randomDuration())

16 })

17 time.Sleep(5 \* time.Second)

18 }

19

20 func randomDuration() time.Duration {

21 return time.Duration(rand.Int63n(1e9))

22 }

23

panic: runtime error: invalid memory address or nil pointer dereference

[signal 0xb code=0x1 addr=0x8 pc=0x41e38a]

goroutine 4 [running]:

time.stopTimer(0x8, 0x12fe6b35d9472d96)

src/pkg/runtime/ztime\_linux\_amd64.c:35 +0x25

time.(\*Timer).Reset(0x0, 0x4e5904f, 0x1)

src/pkg/time/sleep.go:81 +0x42

main.func·001()

race.go:14 +0xe3

created by time.goFunc

src/pkg/time/sleep.go:122 +0x48

The race detector shows the problem: an unsynchronized read and write of the variable t from different goroutines.

To fix the race condition we change the code to read and write the variable t only from the main goroutine:

10 func main() {

11 start := time.Now()

12 reset := make(chan bool)

13 var t \*time.Timer

14 t = time.AfterFunc(randomDuration(), func() {

15 fmt.Println(time.Now().Sub(start))

16 reset <- true

17 })

18 for time.Since(start) < 5\*time.Second {

19 <-reset

20 t.Reset(randomDuration())

21 }

22 }

Here the main goroutine is wholly responsible for setting and resetting the Timer t and a new reset channel communicates the need to reset the timer in a thread-safe way.

**Go performance tool**

Go has a lot of performance tool available for the CPU utilization and time usage.

The common tool is

<https://gitlab.com/steveazz-blog/go-performance-tools-cheat-sheet>

One of the most convinient method to see is use benchmark tool built in go

go test -bench=. -test.benchmem ./rand/

goos: darwin

goarch: amd64

pkg: gitlab.com/steveazz/blog/go-performance-tools-cheat-sheet/rand

cpu: Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz

BenchmarkHitCount100-8 3020 367016 ns/op 269861 B/op 3600 allocs/op

BenchmarkHitCount1000-8 326 3737517 ns/op 2696308 B/op 36005 allocs/op

BenchmarkHitCount100000-8 3 370797178 ns/op 269406189 B/op 3600563 allocs/op

BenchmarkHitCount1000000-8 1 3857843580 ns/op 2697160640 B/op 36006111 allocs/op

PASS

ok gitlab.com/steveazz/blog/go-performance-tools-cheat-sheet/rand 8.828s

Note: -test.benchmem is an optional flag to show memory allocations

**Comparing Benchmarks**

Go created [perf](https://github.com/golang/perf) which provides [benchstat](https://github.com/golang/perf/tree/master/cmd/benchstat) so that you can compare to benchmark outputs together and it will give you the delta between them.

For example, let’s compare the [main](https://gitlab.com/steveazz-blog/go-performance-tools-cheat-sheet/-/tree/main) and [best](https://gitlab.com/steveazz-blog/go-performance-tools-cheat-sheet/-/tree/best) branches.

# Run benchmarks on `main`

git checkout main

go test -bench=. -test.benchmem -count=5 ./rand/ > old.txt

# Run benchmarks on `best

git checkout best

go test -bench=. -test.benchmem -count=5 ./rand/ > new.txt

# Compare the two benchmark results

benchstat old.txt new.txt

name old time/op new time/op delta

HitCount100-8 366µs ± 0% 103µs ± 0% -71.89% (p=0.008 n=5+5)

HitCount1000-8 3.66ms ± 0% 1.06ms ± 5% -71.13% (p=0.008 n=5+5)

HitCount100000-8 367ms ± 0% 104ms ± 1% -71.70% (p=0.008 n=5+5)

HitCount1000000-8 3.66s ± 0% 1.03s ± 1% -71.84% (p=0.016 n=4+5)

name old alloc/op new alloc/op delta

HitCount100-8 270kB ± 0% 53kB ± 0% -80.36% (p=0.008 n=5+5)

HitCount1000-8 2.70MB ± 0% 0.53MB ± 0% -80.39% (p=0.008 n=5+5)

HitCount100000-8 270MB ± 0% 53MB ± 0% -80.38% (p=0.008 n=5+5)

HitCount1000000-8 2.70GB ± 0% 0.53GB ± 0% -80.39% (p=0.016 n=4+5)

name old allocs/op new allocs/op delta

HitCount100-8 3.60k ± 0% 1.50k ± 0% -58.33% (p=0.008 n=5+5)

HitCount1000-8 36.0k ± 0% 15.0k ± 0% -58.34% (p=0.008 n=5+5)

HitCount100000-8 3.60M ± 0% 1.50M ± 0% -58.34% (p=0.008 n=5+5)

HitCount1000000-8 36.0M ± 0% 15.0M ± 0% -58.34% (p=0.008 n=5+5)

Notice that we pass the -count flag to run the benchmarks multiple times so it can get the mean of the runs.

**Benchmarks**

You can generate profiles using benchmarks that we have in the demo project.

CPU:

go test -bench=. -cpuprofile cpu.prof ./rand/

Memory:

go test -bench=. -memprofile mem.prof ./rand/

**Go Static code analysis**

Static code analysis is the greatest tool to find the issues related to the security, performance, coverage, coding style, and some time even logic running without the running your application.

When invoked with the -analysis flag, godoc performs static analysis on the Go packages it indexes and displays the results in the source and package views. This document provides a brief tour of these features.

## **Type analysis features**

godoc -analysis=type performs static checking similar to that done by a compiler: it detects ill-formed programs, resolves each identifier to the entity it denotes, computes the type of each expression and the method set of each type, and determines which types are assignable to each interface type. **Type analysis** is relatively quick, requiring about 10 seconds for the >200 packages of the standard library, for example.

### **Compiler errors**

If any source file contains a compilation error, the source view will highlight the errant location in red. Hovering over it displays the error message.



### **Identifier resolution**

In the source view, every referring identifier is annotated with information about the language entity it refers to: a package, constant, variable, type, function or statement label. Hovering over the identifier reveals the entity's kind and type (e.g. var x int or func f func(int) string).





Clicking the link takes you to the entity's definition.



### **Type information: size/alignment, method set, interfaces**

Clicking on the identifier that defines a named type causes a panel to appear, displaying information about the named type, including its size and alignment in bytes, its [method set](https://go.dev/ref/spec#Method_sets), and its *implements* relation: the set of types T that are assignable to or from this type U where at least one of T or U is an interface. This example shows information about net/rpc.methodType.



The method set includes not only the declared methods of the type, but also any methods "promoted" from anonymous fields of structs, such as sync.Mutex in this example. In addition, the receiver type is displayed as \*T or T depending on whether it requires the address or just a copy of the receiver value.

The method set and *implements* relation are also available via the package view.



**Go API DOCS**

Why API doc is needed?

To aware the functionality of the API to new developers.

Which tool we needed?

We will use swag tool.

Installation of swag tool

Go to the main directory of the project where your rest api based project is there.

Go to the terminal and fire below commands

go get -u github.com/swaggo/swag/cmd/swag

go get -u github.com/swaggo/http-swagger

go get -u github.com/alecthomas/template

This 3 commands will do the necessory installation of swag, http and templates

## **Routes**

I have my app’s endpoints as follows:

user := r.Group("/user")

{

user.GET("/", controller.GetUsers)

user.POST("/", controller.CreateUser)

user.GET("/:id", controller.GetUserByID)

}

I will be documenting these endpoints in this artcile.

## **Models**

I have a User model as:

type User struct {

ID BinaryUUID `json:"id"`

Name string `json:"name"`

Email string `json:"email"`

Phone string `json:"phone"`

Address string `json:"address"`

UN sql.NullString `json:"user\_num" swaggertype:"string"`

}

Above, I have ID and UN fields of **customized data types**. Swag supports customized data type. In case of field ID, the marshallings and unmarshallings are written in binary\_uuid.go file [check the example repo in GitHub]. Since, the data type sql.NullString is imported from “database/sql” , the corresponding field i.e. UN requires swaggertype tag so that Swag can support these kinds of data type.

## **Handlers**

I have three handlers for three endpoints as follows:

// GetUsers ... Get all users

func GetUsers(c \*gin.Context) {

var user []model.User

err := model.GetAllUsers(&user)

if err != nil {

c.JSON(http.StatusNotFound, gin.H{"error": err.Error()})

return

}

c.JSON(http.StatusOK, gin.H{"data": user})

}

// CreateUser ... Create User

func CreateUser(c \*gin.Context) {

var user model.User

if err := c.BindJSON(&user); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

err := model.CreateUser(&user)

if err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})

return

}

c.JSON(http.StatusOK, gin.H{"message": "success"})

}

// GetUserByID ... Get the user by id

func GetUserByID(c \*gin.Context) {

id := c.Params.ByName("id")

userID, err := model.StringToBinaryUUID(id)

if err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

var user model.User

err = model.GetUserByID(&user, userID)

if err != nil {

c.JSON(http.StatusNotFound, gin.H{"error": err.Error()})

return

}

c.JSON(http.StatusOK, gin.H{"data": user})

}

# **Integrate Swag to App**

## **General API info**

To integrate Swag into the App, we just need to write some annotations/ comments/docstring or whatever you want to call it. It’s really just bunch of comments before specific API function, which is used to generate the *Swagger* docs.

Before we get to describing individual API endpoints, we need to first write general description for our whole project. This part of annotations lives in the main package, right before the main function:

package main

...

// [@title](http://twitter.com/title) User API documentation

// [@version](http://twitter.com/version) 1.0.0

// [@host](http://twitter.com/host) localhost:5000

// [@BasePath](http://twitter.com/BasePath) /user

func main() {

....

}

**titile:** Document title

**version:** Version

**description**, **termsOfService**, **contact** … These are some statements, so don’t write them.

**host**, **BasePath:** If you want to directly swagger to debug the API, these two items need to be filled in correctly. The former is the port of the service document, ip. The latter is the base path, like mine is “/user”. BasePath is also not required.

In the original document there issecurityDefinitions.basic, securityDefinitions.apikey. These are all used for authentication.

## **API Operation annotations**

Now that we have added project-level documentation, let’s add documentation to each individual API.

// GetUsers ... Get all users

// [@Summary](http://twitter.com/Summary) Get all users

// [@Description](http://twitter.com/Description) get all users

// [@Tags](http://twitter.com/Tags) Users

// [@Success](http://twitter.com/Success) 200 {array} model.User

// [@Failure](http://twitter.com/Failure) 404 {object} object

// [@Router](http://twitter.com/Router) / [get]

func GetUsers(c \*gin.Context) {

...

}

// CreateUser ... Create User

// [@Summary](http://twitter.com/Summary) Create new user based on paramters

// [@Description](http://twitter.com/Description) Create new user

// [@Tags](http://twitter.com/Tags) Users

// [@Accept](http://twitter.com/Accept) json

// [@Param](http://twitter.com/Param) user body model.User true "User Data"

// [@Success](http://twitter.com/Success) 200 {object} object

// [@Failure](http://twitter.com/Failure) 400,500 {object} object

// [@Router](http://twitter.com/Router) / [post]

func CreateUser(c \*gin.Context) {

...

}

// GetUserByID ... Get the user by id

// [@Summary](http://twitter.com/Summary) Get one user

// [@Description](http://twitter.com/Description) get user by ID

// [@Tags](http://twitter.com/Tags) Users

// [@Param](http://twitter.com/Param) id path string true "User ID"

// [@Success](http://twitter.com/Success) 200 {object} model.User

// [@Failure](http://twitter.com/Failure) 400,404 {object} object

// [@Router](http://twitter.com/Router) /{id} [get]

func GetUserByID(c \*gin.Context) {

...

}

These comments will appear in the corresponding position of the API document. Here we mainly talk about the following parameters in detail:

**Tags**

Tags are used to group APIs.

**Accept**

The received parameter type, support form (mpfd) , JSON(json), etc., more in the table below.

**Produce**

The returned data structure is generallyjson, Other support is as follows:

**Param**

The parameters, from front to back are:

// @Param name body string true "Username" default(user)

// @Param email formData string true "Email"

@Param 1.Parameter name 2.Parameter type 3.Parameter data type 4.Required 5.Parameter description 6.Other attributes

**Success**

Specify the data for a successful response. The format is:

// @Success 1.HTTP response code {2.Response parameter type} 3.Response data type 4.Other description

**Failure**

Same as Success.

**Router**

Specify routing and HTTP method. The format is:

*// @Router /path/to/handle [HTTP method]*

No need to include a basic path.

# **Generate**

Finally, it’s time to generate the docs! All you need is one command —

swag init

This command needs to be ran from directory where main is. This command will create package called docs, which includes both *JSON* and *YAML* version of our docs.

If you need to update your API annotation or add more endpoints, all you need to is go for the command swag init . No need to delete or work on previous docs package. Everything will be updated by Swag itself.

We should see a similar output, if you are curious, you can navigate todocsCatalog and viewswagger.jsonfile.

# **Swagger UI**

This step is very simple. All we do here is importhttpSwaggerLibrary, and the huge documentation we generated. And remember, the import might be change as per your project requirements and package installations.

import (

\_ "Cyantosh0/go-swag/docs"

ginSwagger "github.com/swaggo/gin-swagger"

"github.com/swaggo/gin-swagger/swaggerFiles"

)

In addition to specifying routes for all APIs, we must also define a main route to use Swagger UI to serve thePathPrefixmethod.

r := route.SetupRouter()

r.GET("/swagger/\*any", ginSwagger.WrapHandler(swaggerFiles.Handler))

r.Run(":" + os.Getenv("SERVER\_PORT"))

Finally, once we are done with all the APIs, and it’s time take them for a spin. To run the app, navigate to your project directory, and run the following commands:

go run main.go

You can see your work coming to life by loading the swagger UI at <http://localhost:5000/swagger/index.html> [Here, my app is running in **PORT 5000**]

If everything goes well, we should be seeing a UI like below:

Here, we can check our API endpoints.

**Go Concurrency**

Go routin and channels are a lightweight built in features for managing concurrency and communication between several function executing at a same time.

This way once can execute the code that outside if the main program.

Go has below keywords like

go

chan

Concurrency with go routine

package main

import (

"fmt"

"time"

)

func timesThree(number int) {

fmt.Println(number \* 3)

}

func main() {

fmt.Println("We are executing a go routine")

go timesThree(3)

fmt.Println("Done!")

time.Sleep(time.Second)

}

PS F:\Training\Golang\Program> go run concurrency.go

We are executing a go routine

Done!

9

We have successfully run the concurrency execution

Main program will creates go routine for executing timesThree function

There for fmt.Println(“Done!”) will executes before go routine

But, what if we need some value returning from that function to continue with our main function.

Thats where channel comes and save the day.

package main

import (

"fmt"

)

func timesThree(number int, ch chan int) {

result := number \* 3

fmt.Println(number \* 3)

ch <- result

}

func main() {

fmt.Println("We are executing a goroutine")

ch := make(chan int)

go timesThree(3, ch)

result := <-ch

fmt.Printf("The result is: %v", result)

}

We are executing a goroutine

9

The result is: 9

Once the main program executes the goroutines, it waits for the channel to get some data before continuing, therefore fmt.Println("The result is: %v", result) is executed after the goroutine returns the result. This doesn’t mean that the main program will wait for the full goroutine to execute, just until the data is served to the channel.

**Developing a webservices**

* Download source code from <https://github.com/tsawler/bookings-udemy/releases/tag/v21>
* Unzipp into you workspace
* Go to visual studio
* File>openfolder
* Go to main directory
* Run the command go run main.go
* It will run the application on your browser



Open your browser and type URL

http://localhost:8080/



And you got a Hello World! on the browser

**Database operation**

Download postgresql

<https://www.enterprisedb.com/postgresql-tutorial-resources-training?uuid=db55e32d-e9f0-4d7c-9aef-b17d01210704&campaignId=7012J000001NhszQAC>

Download DBeaver

<https://dbeaver.io/download/>

Create a database called test in dbviewer

Code is as below & taken from

<https://att-c.udemycdn.com/2021-04-05_23-12-41-ccb5b133039198cd672e083dafee1c72/original.zip?response-content-disposition=attachment%3B+filename%3Dtest_connect.zip&Expires=1652886735&Signature=P-ADFLUqNG4i6xJIUwB2ukNITzHjJ3TTroisK2V7MOSsiKeC7eC7FxMw3dVCUEfeaMKi454dR~d~m~naWyZpuvFibWMU84GutwSxxIxpoDOg~EWY8lp~1l5ng6fJTHGKhu0kmKGdn8DJJJBDnNAlC5lfoGvVtsbg9VLnoHMPL24~56EEPjkqnNFteKOtm-GvSVaQz7lQZPwJQVnmCDRzv1oe0VaAozR4iKD1JQWgXYRER20ERT~50-5PycL73A~bj2rVh9qMVaOBqaFU1vLwaLTGguDm4LVV-jXSDI0blymO7mtRy4nW5QLOFC8gblmHhoOlL~UObnxe8o20uHtLhg__&Key-Pair-Id=APKAITJV77WS5ZT7262A>

package main

import (

"database/sql"

"fmt"

"log"

\_ "github.com/jackc/pgx/v4/stdlib"

)

func main() {

// connect to a database

conn, err := sql.Open("pgx", "host=localhost port=5432 dbname=test user=yagnik.pokal password=yagnik@2017")

if err != nil {

log.Fatal(fmt.Sprintf("Unable to connect: %v\n", err))

}

defer conn.Close()

log.Println("Connected to database!")

// test my connection

err = conn.Ping()

if err != nil {

log.Fatal("Cannot ping database!")

}

log.Println("Pinged database!")

// get rows from table

err = getAllRows(conn)

if err != nil {

log.Fatal(err)

}

// insert a row

query := `insert into users (first\_name, last\_name) values ($1, $2)`

\_, err = conn.Exec(query, "Jack", "Brown")

if err != nil {

log.Fatal(err)

}

log.Println("Inserted a row!")

// get rows from table again

err = getAllRows(conn)

if err != nil {

log.Fatal(err)

}

// update a row

stmt := `update users set first\_name = $1 where id = $2`

\_, err = conn.Exec(stmt, "Jackie", 5)

if err != nil {

log.Fatal(err)

}

log.Println("Updated one or more rows")

// get rows from table again

err = getAllRows(conn)

if err != nil {

log.Fatal(err)

}

// get one row by id

query = `select id, first\_name, last\_name from users where id = $1`

var firstName, lastName string

var id int

row := conn.QueryRow(query, 1)

err = row.Scan(&id, &firstName, &lastName)

if err != nil {

log.Fatal(err)

}

log.Println("QueryRow returns", id, firstName, lastName)

// delete a row

query = `delete from users where id = $1`

\_, err = conn.Exec(query, 6)

if err != nil {

log.Fatal(err)

}

log.Println("Deleted a row!")

// get rows from table again

err = getAllRows(conn)

if err != nil {

log.Fatal(err)

}

}

func getAllRows(conn \*sql.DB) error {

rows, err := conn.Query("select id, first\_name, last\_name from users")

if err != nil {

log.Println(err)

return err

}

defer rows.Close()

var firstName, lastName string

var id int

for rows.Next() {

err := rows.Scan(&id, &firstName, &lastName)

if err != nil {

log.Println(err)

return err

}

fmt.Println("Record is", id, firstName, lastName)

}

if err = rows.Err(); err != nil {

log.Fatal("Error scanning rows", err)

}

fmt.Println("---------------------------")

return nil

}





Build webaaplication

Dowbload code from below

<https://github.com/yagnikpokal/golang/tree/main/Baseapp>

Extract in a particular directory

Import extracted folder in VS

Import necessary packages

Debug and run the application in VS

Application will available on port 8080

Open your browser and type URL

<http://localhost:8080/>





Check on below link for demonstration

<https://youtu.be/bUwsVwwE8ZA>