Index

https://golanghero.com/
https://golang.cafe/
https://www.golangprojects.com/
https://www.welovegolang.com/
https://forum.golangbridge.org/c/jobs/8

What is golang?

=>It open-source developed by Google, it is simple syntax(inspired by C), fast compile(compile directly machine code), statically-type, Garbage collector(GC), rich standard Library , concurrency

What is mean by statically-typed?

=> variable types (e.g., integer, string, float) are checked and enforced at compile time

rather than at runtime. Type safety catches errors at compile time.

Return multiple value

==========

Go can have multiple value initialize

=> var a1, b2,c3 = 1,”Sagar” , 45.6

Also for short variable

```
x1, x2, x3 := 2221, "Sagar", 55.6
```

Go can define multiple variable in bracket

=> var (

Num1=10

Name =”Sagar”

)

how Switch is define ?

=> switch define by 1. Expression , like int string,

2. interface, like what interface it is

1. Expression

```
var day = "Monday"
    switch {
    case day == "Monday":
        fmt.Println("Its monday")
    case day == "Friday":
        fmt.Println("Its Friday")
    default:
        fmt.Println("invalid day")

}
```

2. interface,

```
var v interface{} = "string"
    switch switchType := v.(type) {
    case string:
        fmt.Println("its string", switchType)
    case int16:
        fmt.Println("its int16")
    default:
```

```
        fmt.Println("Invalid format")


    }
```

Array

Ellipses **(...)**

Package

In Go, a **package** is a collection of related Go files in the same directory .
Packages help organize code, promote reusability of the code.

Key Concepts:

1. **Package Declaration**:
   - Every Go file starts with a package declaration at the top.
   - The package name is usually the same as the folder name (except for main).
   - Executable programs **must** use package main.
   - For Custom package ,

     "your-module-name/utils" // Replace with your module name
1. **Imports**:
   - Use import to access code from other packages (e.g., import "fmt").
   - Import should from ur "myproject/mathutil " till ur last folder , used "pwd " command , all name lowwer case through your folder name "MyProject" use "myproject"
   - import "myproject/internal/auth"
1. **Visibility**:
   - Uppercase identifiers (e.g., Add, Calculate) are **exported** (public).
   - Lowercase identifiers (e.g., add, calculate) are **unexported** (private).

Directory Structure:

```
Copy
myproject/
├── go.mod
├── main.go
└── mathutils/
    └── mathutils.go
```

```
Copy
myproject/
├── go.mod
├── main.go
└── internal/
    └── auth/
        └── auth.go
```

Why Use Packages?

- **Reusability**: Share code across projects.

- **Encapsulation**: Hide internal logic (only expose what's needed).
- **Organization**: Break code into logical units (e.g., mathutils, logger).

NOTE : go.mod & main.go Should be at same level.
Package == project==workspace

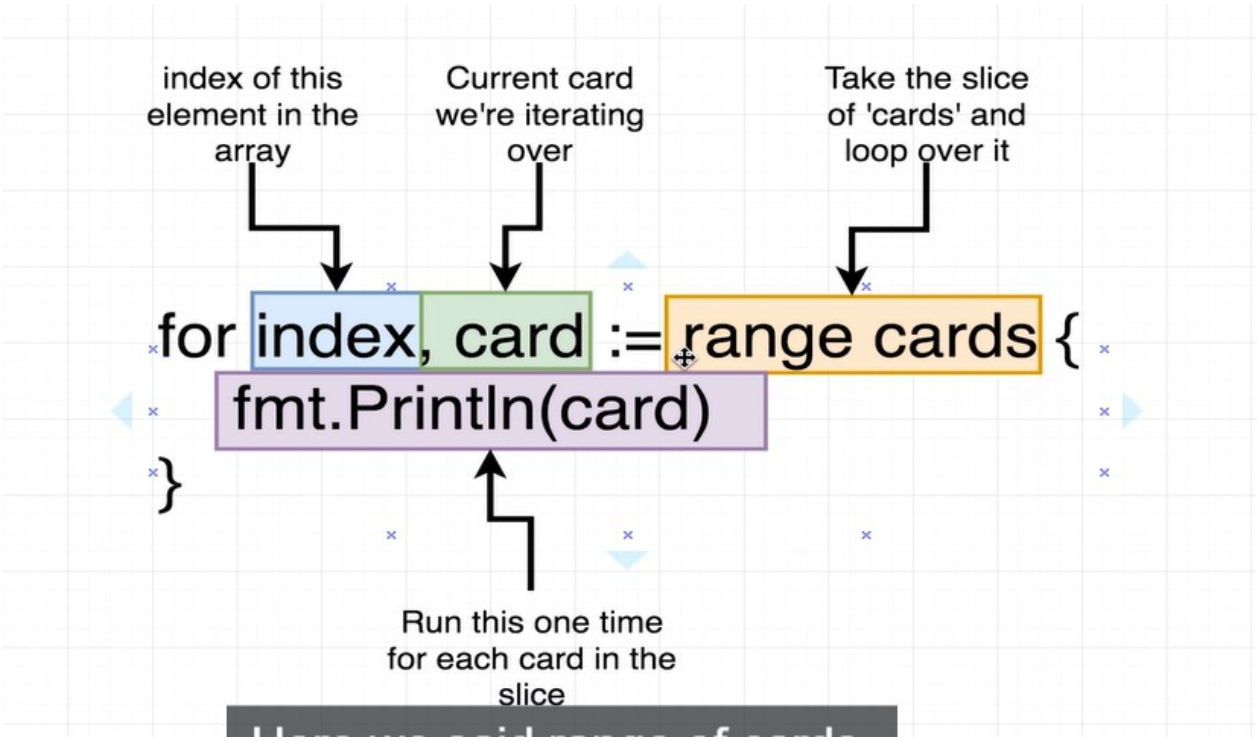
What is used of "go mod tidy"?
=> it add Missing Dependency **and** remove unused Dependency from sum.mod file.
if install new module like gin, then import not found dependency then use go mod tidy

"%+v"	Print struct value with its corresponding field .
Defer	delays the execution util function is over , it used file close, resource release it ensure that if there error, panic occurs it handle properly.

Panic	It is like ***throw*** in c++, after panic execution stop. panics are typically used for unrecoverable errors, so try to ovoid using panic.
	Catch exception like :
	`func foo() int {`
	`  defer fmt.Println("\n defer")`
	`  fmt.Println("inside foo")`
	`  panic(" foo throw")`
	`  fmt.Println("After foo") return 10`
	`}`
	`func main() {`
	`  defer func() {`
	`  ret := recover() if`
	`  ret != nil {`
	`  fmt.Println(" Recover ", ret)`
	`  }`
	`  }()`
	
	`  fmt.Printf("%d", foo())`
	`  fmt.Println("Hello World")`
	`}`

For index, value := range arr {

}




```
var arr [5]int = [5]int{1, 2, 3, 4, 5} for i, v := range
      arr {
            //fmt.Printf("\nindex=%d", i, "value=%d", v) fmt.Printf("\nindex=%d, value=%d",
            i, v)
      }


ar := [5]int{10, 20, 30, 40, 50}
      for i, v := range ar {
            //fmt.Printf("\nindex=%d", i, "value=%d", v) fmt.Printf("\nindex=%d, value=%d",
            i, v)
      }
```

No.	Array	Slice
Size	Fixed.	Dynamic size can grow shrink like vector.Slices are built on top of arrays and provide a more flexible way to work with collections of data.
Declaration Syntax	var arr **[5]**int	var slice **[]** int, OR slice := make([]int, 0, 5)
Passing Argument	Array pass by value	Slice by reference.

Usage	need a fixed-size collection of elements	more commonly used in Go because of their flexibility and dynamic nature. Support more operation like slicing ,appending
		make([]int, 0, 5) Len = 0, Capacity = 5

movies = append(...)

- The result of the append operation is assigned back to the movies slice, effectively updating it.

```
go
Copy
movies := []string{"A", "B", "C", "D", "E"}
index := 2
movies = append(movies[:index], movies[index+1:]...)
fmt.Println(movies) // Output: ["A", "B", "D", "E"]
```

Struct

Import

(“fmt”

“unsafe”)

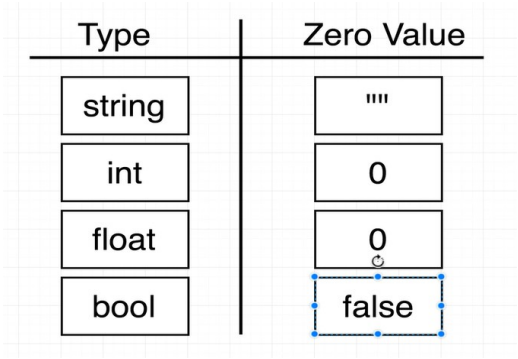
Type Emp struct
{ Id int
Name string
}

Func main() {
E:= Emp {id :1, Name:”Sagar”)
tempid := unsafe.Sizeof(e)

fmt.Printf("Emp id=%d, Name=%s", e.id, e.name)

}

Note: - When we just declared struct NOT initialized then by default value is zero .

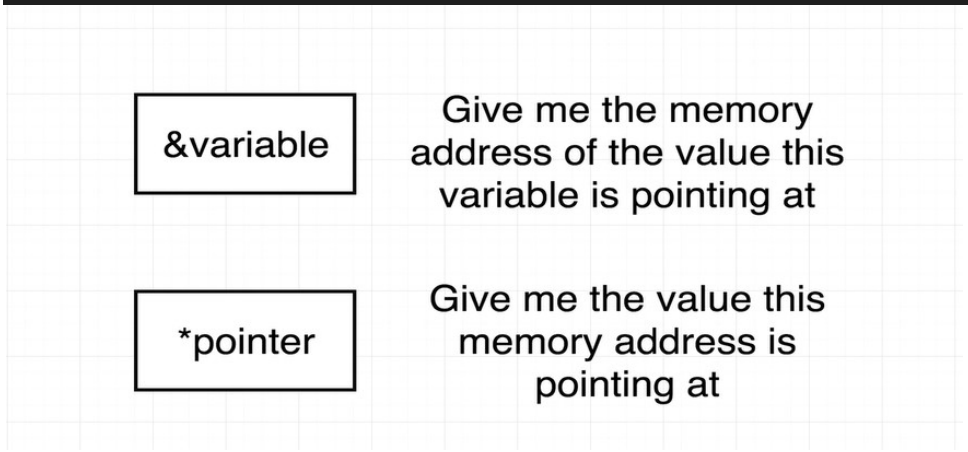


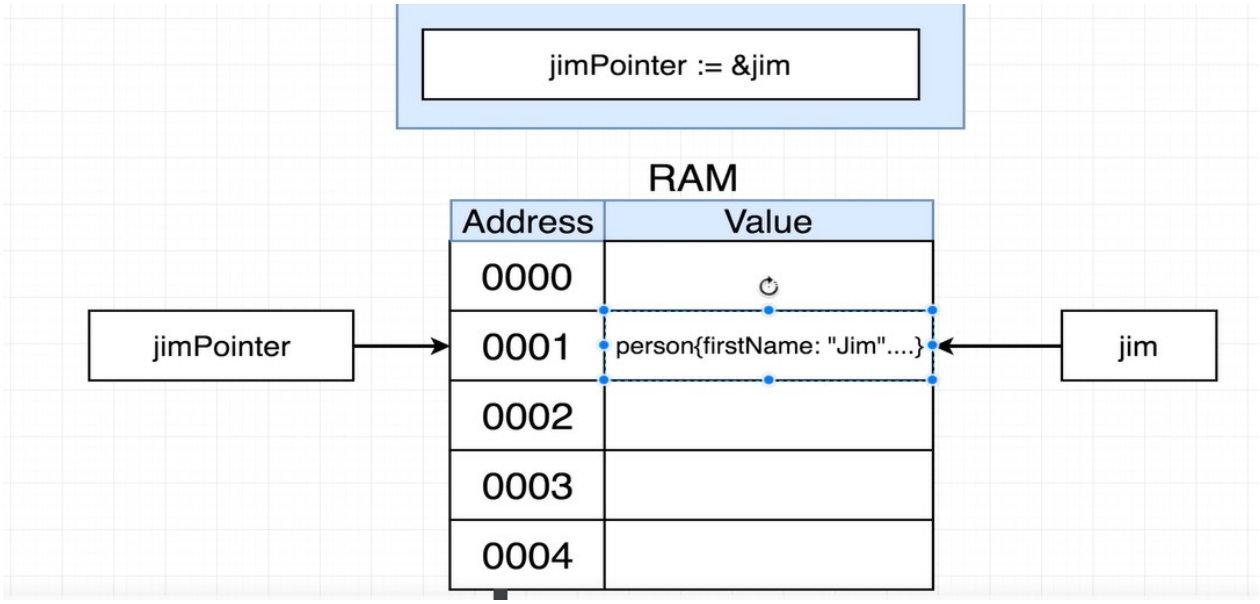
Type	Zero Value
string	""
int	0
float	0
bool	false

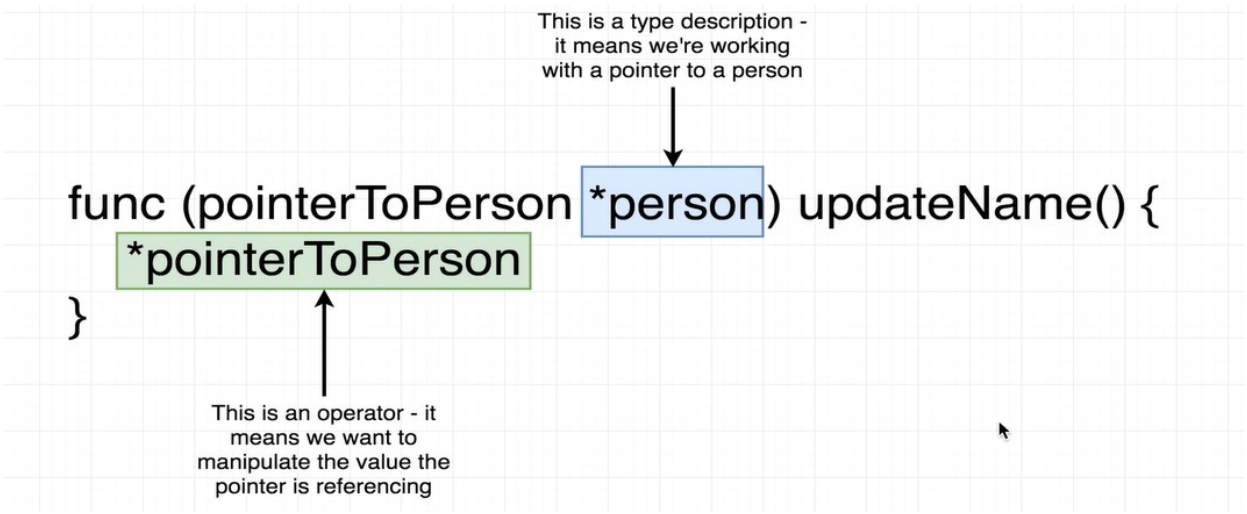
Struct using pointer, So its like reference pass to function.

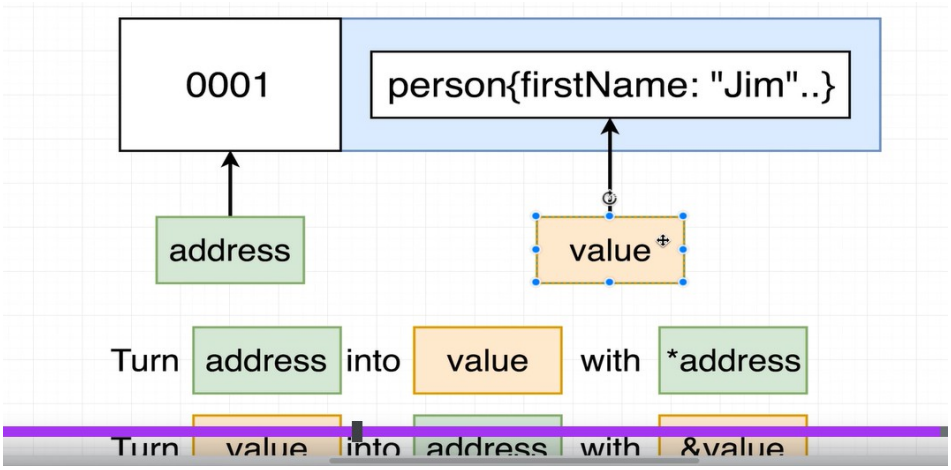
```
type Emp struct {
id              int
namestring
}

/*func(eEmp)update(){ e.id = 201
e.name ="Sagar"
}*/

func(e*Emp)update(){ (*e).id=201
(*e).name="Sam"
}

funcmain(){

e:=Emp{id:101,name:"Sagar"}eptr:= &e
fmt.Printf("\nEmp value id=%d, name=%s",e.id,e.name)
//e.update()
fmt.Printf("\nAfterupdateEmpvalueid=%d,name=%s",e.id,e.name)eptr.update()
fmt.Printf("\nAfterpointer update Emp value id=%d, name=%s",e.id,e.name)
}
```

```
    id          int
        namestring
}

/*func(eEmp)update(){ e.id = 201
        e.name ="Sagar"
}*/

func(e*Emp)update(){ (*e).id=201
        (*e).name="Sam"
}

funcmain(){

        e:=Emp{id:101,name:"Sagar"}eptr:= &e
        fmt.Printf("\nEmp value id=%d, name=%s",e.id,e.name)
        //e.update()
        fmt.Printf("\nAfterupdateEmpvalueid=%d,name=%s",e.id,e.name)eptr.update()
        fmt.Printf("\nAfterpointer update Emp value id=%d, name=%s",e.id,e.name)
}
```




jimPointer := &jim
RAM
Address
Value
0000
0001
0002
0003
0004
jimPointer
person{firstName: "Jim"....}
jim

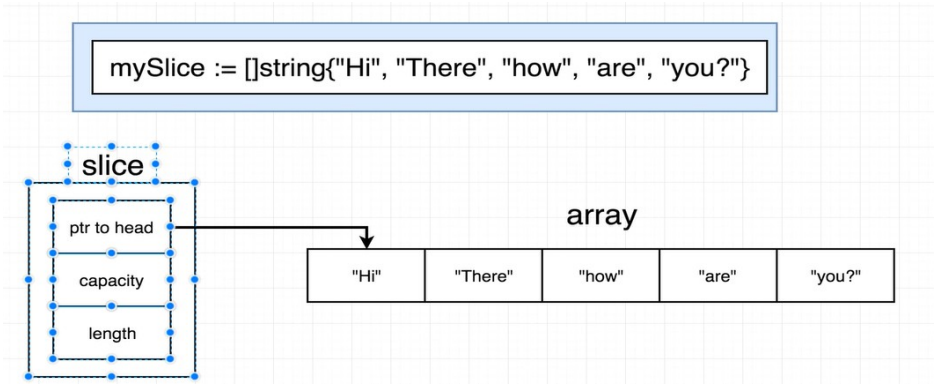

This is a type description - it means we're working with a pointer to a person
func (pointerToPerson *person) updateName() {
*pointerToPerson
}
This is an operator - it means we want to manipulate the value the pointer is referencing

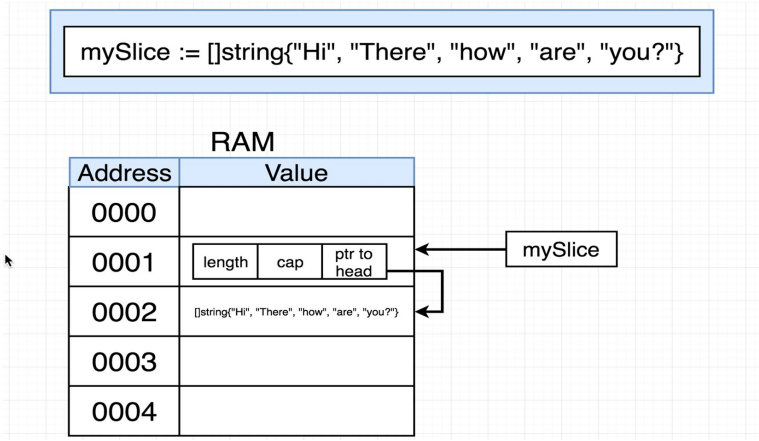

0001
person{firstName: "Jim"..}
address
value
Turn address into value with *address
Turn value into address with &value

IMP : Structure can pass as value OR it just pass with/Without pointer but receiver you have used pointer at receiver then it become pointer.

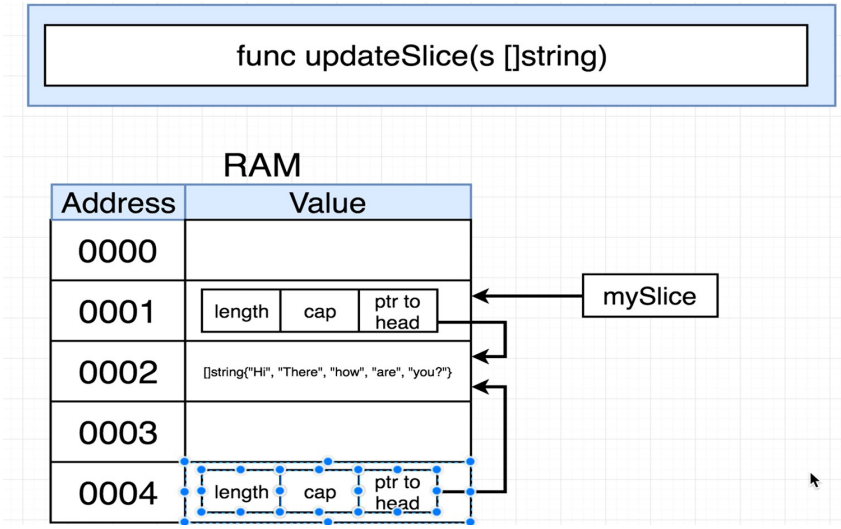
e :=Emp{id:111,name:"Sagar"} //initialize e object **e.update()** **func (epointer *emp)update() {}**	e :=Emp{id:111,name:"Sagar"} //initialize e object **eptr = &e** **eptr.update()** **func (epointer *emp)update() {}**
Above both type work	

Difference between slice and struct

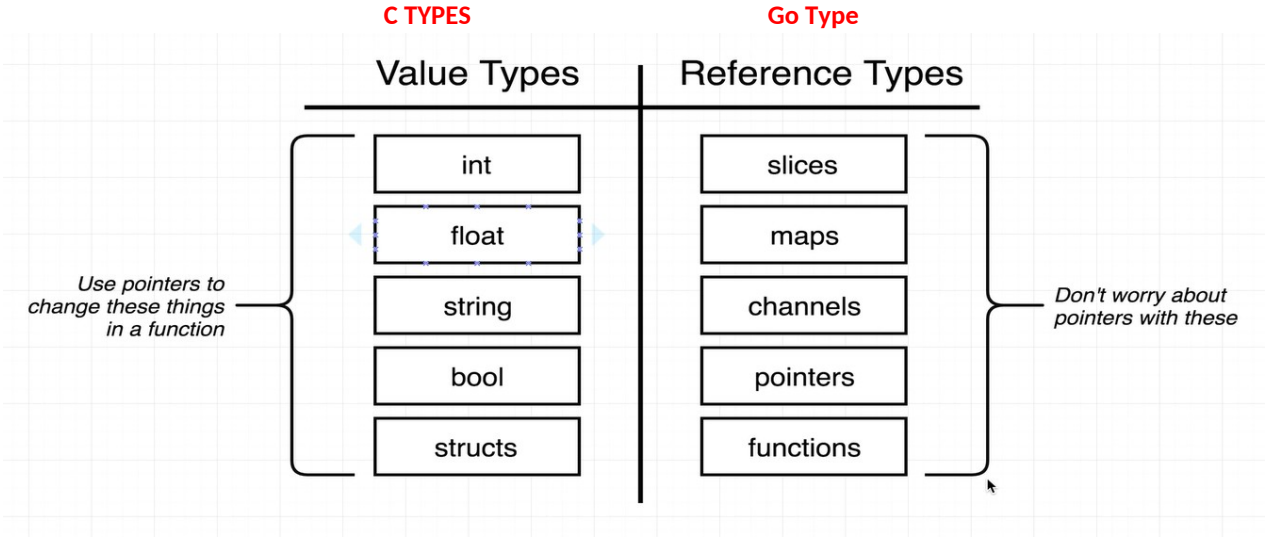




Note: Go is pass by value language



Here When pass slice as argument then slice will copy its value as shown above.



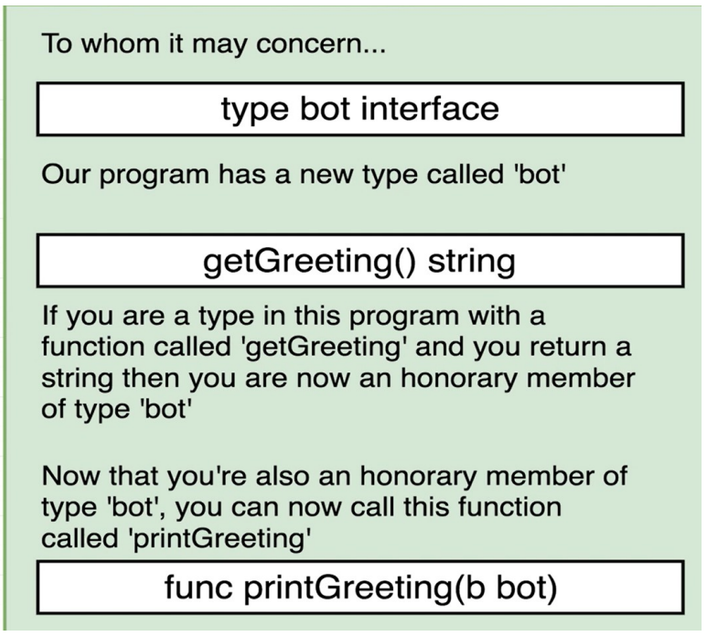
MAP

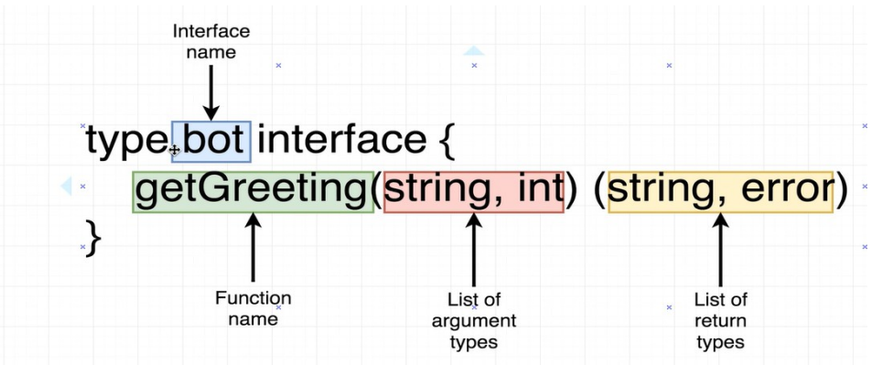
Mapname := map[key]value
myMap :=map[int]string
mymap :=make(map[int]string)
Maps are **unordered** collections, meaning that the order of key-value pairs is not guaranteed.

Interface

you can’t overload same function , that why interface is introduce.

```
packagemain

import(
	"fmt"
)

type Bot interface{
	getGreeting()string
}

type Englishbot struct{
}

func(Englishbot)getGreeting()string{//ThisismembermethodofthatstructreturN"English Hello"
                                                                    // So same name is allowed .
}

typeSpanishbotstruct{
}

func(Spanishbot)getGreeting()string{//ThisismembermethodofthatstructreturN"Spanish Hola"
                                                  // So same name is allowed .
}

funcprintGreeting(bBot){
	fmt.Println(b.getGreeting())
}

funcmain(){
	fmt.Printf("")

	e:=Englishbot{}s:=Sp
	anishbot{}printGreet
	ing(e)printGreeting(s
```





Interface automatically link with function . Q. How?

Empty Interface: Used for generic functions (e.g., fmt.Println).
The empty interface (interface{}) has no method signatures. This means **any type** satisfies it. It's Go's way of representing a generic type.

Ex.

```
Func main(){
 PrintAnything("Hello")
 PrintAnything(10)
 C := Circle {radius:20}
 PrintAnything(C)


}

    PrintAnything("Hello")
```

O/P =>
Type: string, Value: Hello
Type: int, Value: 10
Type: *main.Circle, Value: &{20}
Type: *main.Circle, Value: &{20}

GoRoutine and channel

What is difference between concurrency & parallel programing?
=> Parallel meaning multiple task at same time like eating & watching TV
But in case Concurrency meaning schedule task into timeslice to execute it some time

interval. Means it switched from one thread to another for CPU execution but they not executing together at same time.
In Parallel programming task is running in multiple CPU(Processor,Cores) but concurrency not given to CORE(CPU), it run within single process

1. Goroutines are light weight thread. It is manage by GoRuntime , it also manage memory i.e GC(garbage collector)
2. They are functions that run concurrently with other goroutines within the same address space.

Q => what is mean by within same address space .

3. It is very cheap for switch overhead and memory .
4. If main goroutine is terminated then all routines in same program also terminated.
5. Go routine always run in background.
6. It required less memory than OS thread.
7. Main Go routine don't have parent & childern

Go routine	Thread
It application level	It is OS level
Required less memory 2KB	Required less memory 2KB
It manage by Go run time	It manage OS.

No.	Goroutine	Thread
1	Goroutines are managed by the go runtime.	Operating system threads are managed by kernal.
2	Goroutine are not hardware dependent.	Threads are hardware dependent.
3	Goroutines have easy communication medium known as channel.	Thread does not have easy communication medium.
4	Due to the presence of channel one goroutine can communicate with other goroutine with low latency.	Due to lack of easy communication medium inter-threads communicate takes place with high latency.
5	Goroutine does not have ID because go does not have Thread Local Storage.	Threads have their own unique ID because they have Thread Local Storage.
6	Goroutines are cheaper than threads.	The cost of threads are higher than goroutine.
7	They are cooperatively scheduled.	They are preemptively scheduled.
8	They have fasted startup time than threads.	They have slow startup time than goroutines.
9	Goroutine has growable segmented stacks.	Threads does not have growable segmented stacks.

Go **Anonymous function/function literals**

1. No function name
2. Useful for define inline function
3. Ex

```
func(parameter_list)(return_type){
//code..

// Use return statement if return_type aregiven
// if return_type is not given, then donot
//usereturnstatementreturn
}()
```

```
func main() {
	fmt.Println("Hello, World!") f :=
	func() {
		fmt.Println(" Anonamous function with variable called")
	}
	f()

	func() {
		fmt.Println(" Anonamous function only")
```

```
    }()
}
```

Function closure

It is special type of anonymous function that can access and manipulate these outer variables (count) even after the outer function has finished executing.

In Go closure is a nested function that can access and modify variables declared in the outer function(Incr) where it was created. This allows the closure to remember and interact with those variables even after the outer function has finished running.

```
func counter() func() int { // nested
   count := 0
   // below is inner function is a closure:
   return func() int {
      count++ // Accesses "count" from the outer scope
      return count
   }
}

func main() {
   myCounter := counter()
   fmt.Println(myCounter()) // Output: 1
   fmt.Println(myCounter()) // Output: 2 (remembers "count")
}
```

Channels

1. Channel is way to communicate with other go routine., go routine is bidirectional communication.
2. Same type data allow to transferred, diff data NOT allowed

What is Buffered & unbuffered channel?

- **IMP** Channel is communicate between go routine, So Rule is first launch go routine then send message to channel.

```go
package main

import (
	"fmt"
)

func Test(c chan string) {
	fmt.Println(<-c) // Attempt to receive from the channel
}

func main() {
	ch := make(chan string)
	ch <- "This main" // Send to the channel
```

```
        go Test(ch)        // Launch goroutine
}
```

in the above example, **unbuffered channel**, meaning it can only hold one value at a time, and the send operation(main) will block until another goroutine receives the value.

- Since the Test goroutine is launched **after** the send operation, the main goroutine will block indefinitely, causing a **deadlock**.

Key Points About Closing Channels:

1. Sender should close channel.
2. Why it need to close => it send signal to receiver that no more value is send,
3. Where it is usefull => when we have for loop
4. If channel is close what happen if we send data => it cause panic
5. **Closing a channel multiple times will cause a panic**.
6. How to check channel is close or not => value , ok :=<-ch

```
package main


import "fmt"


func main() {
    ch := make(chan string)
    ch <- "This main"
    ret := <-ch
    fmt.Println("ret", ret)
}
```

What is output & why ?
=> 1. The channel ch is created as an **unbuffered channel (make(chan string)).**
The first operation i.e. **unbuffered channel** (ch <- "This main") is a **blocking** send:
Since the channel is unbuffered, it requires a receiver , but No receiver.
No other goroutine is running to receive the value.
2.The main goroutine gets **stuck waiting for a receive**r.
3.The program **never reaches the next line (ret := <-ch**) because it's **already blocked**.
Go detects this **deadlock and panics**:

Solution:

1. Make buffered channel

```
ch := make(chan string,1)
```

2. Another Solution is:
   Used **anonymous function** as below:

```
func main() {
    ch := make(chan string)
    go func() {
        ch <- "This main" // Runs in a separate goroutine, won't block main
    }()

    ret := <-ch // Now the main goroutine can receive the value
    fmt.Println("ret", ret)
}
```

RECV = READING (RR) ch <-chan Type
ex. Recv=Reading (RR) func foo(ch<- chan int)

- **chan :** bidirectional channel . both read and write
- **chan <- :** only writing to channel
- **<- chan :** only reading from the channel (input channel)
- *** chan :** channel pointer. both read and write

➢ **Only Send Channel**

Syntax:

```
func process (ch chan <- type ){
    Statements ...
}
```

➢ **Only Receive Channel**

Syntax:

```
func process (ch <- chan type ){
    Statements ...
}
```

➢ **Channel Pointer**

Syntax:

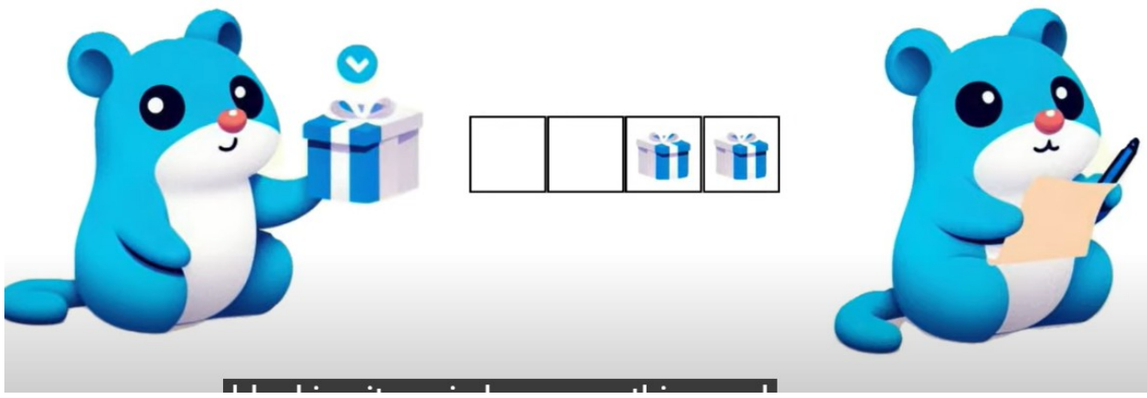
```
func process (ch *chan type ){
    Statements ...
}
```

Unbuffered `make(chan int)`

`make(chan int, 0)`

It is blocking until data is received.

Buffer channel



REST API

Rest API put & patch differnce

=> Put replacing it entirely

Patch partially by applying only the changes specified

Post –entirely create new record .

```
//create mux object
    r := mux.NewRouter()
r.HandleFunc("/Add", Add).Methods("Post")
```

=> HandlerFunc

"/Add" => used in postman => http://localhost:8000/Add

Add => This is function in go file

Post => select in postman

```
err := http.ListenAndServe(":8000", r) // this function is used after HandlerFunc
```

This above function is used create & Listen on port 8000.

API Function its descriptions

`r := mux.NewRouter()`	New object Mux library
r.HandleFunc("/movies", getmovies).Method("Get")	This URL endpoint(Query)–movies postman method – Get getmovies – go function
`err := http.ListenAndServe("8000", r)`	This function used to start API server
`func getmovies(w http.ResponseWriter, req *http.Request) {}`	DO NOT missed sequence, first responsewriter, Second is request pointer
`Param :=mux.Vars(req)`	Mux.Vars is used to get parameter like id from URL
`Body :=req.Body`	Get body of the request
json.NewDecoder(r.Body).Decode(&updatedMovie)	**1.json.NewDecoder(r.Body)**: Creates a new JSON decoder that reads from request Body. 2. **Decode(&updatedMovie)**: Decodes the JSON data from the request body into the updatedMovie variable. The & operator is used to pass a pointer to updatedMovie so that the decoder can populate it with the decoded data
json.NewEncoder(w).Encode(updatedMovie)	This line is used to **encode a Go struct into JSON and write it to the HTTP response**. **.Encode(updatedMovie)**: Encodes the updatedMovie struct into JSON and writes it to the response.

Key Differences Between Decode and Encode

Aspect	Decode	Encode
Purpose	Converts JSON data into a Go struct.	Converts a Go struct into JSON data.
Input	Reads from an io.Reader (e.g., r.Body).	Writes to an io.Writer (e.g., w).
Output	Populates a Go struct.	Writes JSON data to the response.
Common Use Case	Parsing JSON data from an HTTP request body.	Sending JSON data in an HTTP response.

Alphabetical order in Decode (D–Docode, J–JSon , S–Struct)

When to Use Which?

- Use **json.NewDecoder and json.NewEncoder** when working with streams (e.g., HTTP requests/responses).
- Use **json.Marshal and json.Unmarshal** when working with byte slices or in-memory data.

```
r := mux.NewRouter()
```

R.HandleFunc(“/movies”, getmovies).Method(“Get”)

```
err := http.ListenAndServe("8000", r)
```

Above function is used in rest Server to start

```
func getmovies(w http.ResponseWriter, req *http.Request) {}

//Get data from request browser
Param :=mux.vars[req]
```

Gorilla Mux:

```
go
Copy
Download
r := mux.NewRouter()
r.HandleFunc("/users/{id}", GetUser).Methods("GET")
http.ListenAndServe(":8080", r)
```

Gin:

```
go
Copy
Download
r := gin.Default()
r.GET("/users", Users)

r.POST("/newUser", AddUser)
r.Run(":8080")

Func Users(ctx *gin.Cpntext){

Ctx.JSON(http.StatusOk, users)

}
```

```
func AddUser(ctx *gin.Context) {
    var newusers User//here User is struct
    err := ctx.ShouldBindJSON(&newuser)// fetch data from POST
    if err != nil {
        ctx.JSON(http.StatusBadRequest, gin.H{"Error": err.Error()})
        return
    }
{ ur logic
}


func GetBooks(ctx *gin.Context) {
    ctx.JSON(http.StatusOK, Books)
}


Ctx.JSON (http.StausCreated, newuser)
```

Choose Gorilla Mux if you prefer minimalism and control, or Gin if you want a more complete, faster solution with less setup.

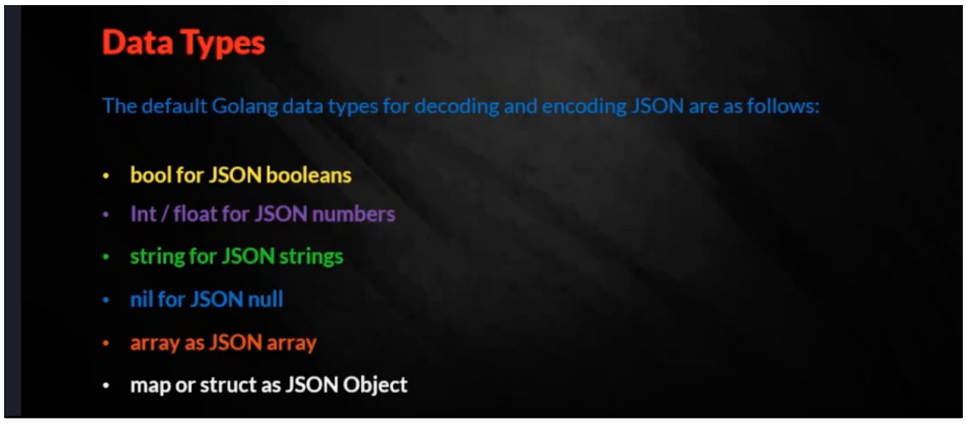
Comparison of Gorilla Mux and Thunder Client

```
    connStr := "user=postgres password=1234 dbname=postgres sslmode=disable"
```

In every database we have add

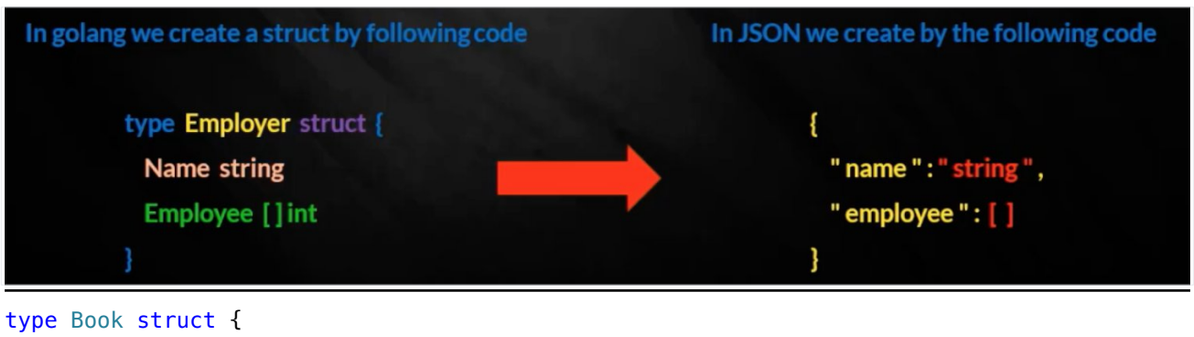
go get github.com/lib/pq command at command prompt

JSON



for accessing json data used `(dilda)

```
type Configuration struct {
    userName string `json:"user"`
}
```




```
type Book struct {
    ID     string `json:"id"`
```

```
    Title  string `json:"Title"`
    Auther string `json:"Auther"`
}
```

Marshal(Encode) => Convert golang Struct into JSON. (ME-SJ)
=> Marshal function return bytes, So it required to convert into string.

UnMarshal(Decode) => Convert JSON into golang struct (UDJS)

Protobuff

1. It is faster than JSON,XML for network transfer as it used byte stream.
2. It is used serialize & de serialize.
3.

How to create .proto file .

=>1. .proto file contain

Syntax =”proto3”

Option go_package= “location” // where you want **.proto.pb** file generated after compiling the code.

```
Message  NameofMessage {
    Datatype variable = id // here id should be unique for serialize & de serialize
}
```

```
syntax = "proto3";

message Book {
    string name = 1;
    int32 isbn = 2;
}
```

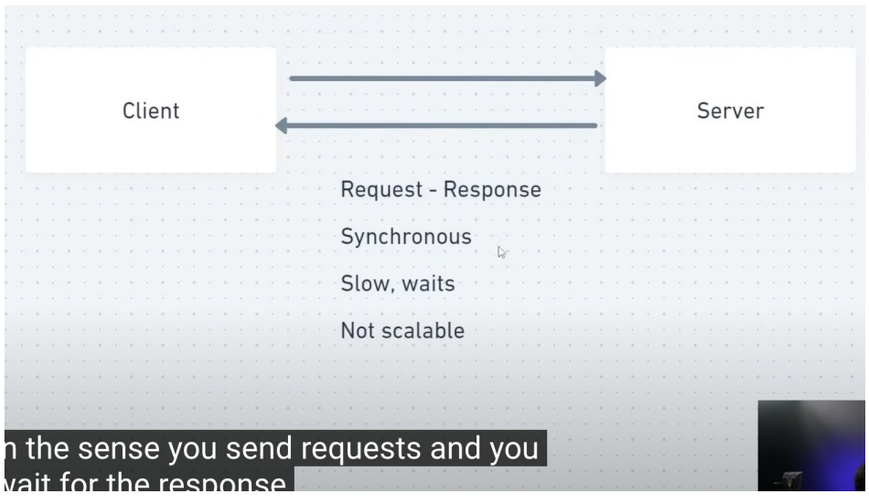
2. Compile proto file

Protoc —go_out=. Example.proto

gRPC

1. It required protobuf file, i.e. .proto &proto.pb.go

Below is client Server Architecture , to overcome we used GRPC



Comparison of gRPC Communication Types

Type	Client Request	Server Response	Use Case Example
Unary	Single	Single	Fetching a user profile.(CRUD), Simple Client Server
Server Streaming	Single	Stream	Streaming live stock prices.Server Send stream of data to client
Client Streaming	Stream	Single	Uploading a large file in chunks.
Bidirectional	Stream	Stream	Real-time chat or multiplayer gaming.

GRPC streaming	Code Changes

Unary	`service Greeter {`
	`  rpc SayHello(HelloRequest) returns (HelloResponse) {}`
	`}`

Server streaming	`service Greeter {`
	`  rpc SayHello(HelloRequest) returns (stream HelloResponse) {}`
	`}`

Client streaming	service Greeter { rpc SayHello(stream RequestMessage) returns (ResponseMessage) {} }

gRPC Client	gRPC Server
grpc.Dial() => it used make connection to grpc server, return connection object. Ex. Conn , err:=grpc.Dial(IPAdrres, option)	**grpc.NewServer() =>** it create grpc server object. Ex. newServer:=grpc.NewServer()
	net.Listen()=>server is listern to particular port Ex. net.Listen("tcp", "127.0.0.1:8085")

<service>_grpc.pb.go	<service>.pb.go
Client and Server Interfaces,	**Message Definitions(JSON Tag),**

Stub Code:(Client & server), Registration Functions	Serialization/Deserialization Code, Helper Functions
Used For : gRPC communication (client-server interaction).	**Used For :**Data representation and serialization/Deserialization

Commands	Description
protoc --go_out=. --go-grpc_out=. proto/greet.proto	This command genrate .pb.go & .grpc.pb.go file.
go get google.golang.org/grpc	Download grpc package

Error & Solution

Error	Solutions
could not import github.com/lib/pq (no required module provides package "github.com/lib/pq")compilerBrokenImport	go get github.com/lib/pq

flag

- the default
  flag.Parse(): Parses the command-line arguments.
- Ex. Define flags
  name := flag.String("name", "Guest", "Your name")
  age := flag.Int("age", 0, "Your age")

- func StringVar(p *string, name string, value string, usage string)
- Ex.
  var name string// Bind the flag to the variable
  flag.StringVar(&name, "name", "Guest", "Your name")

Ex.

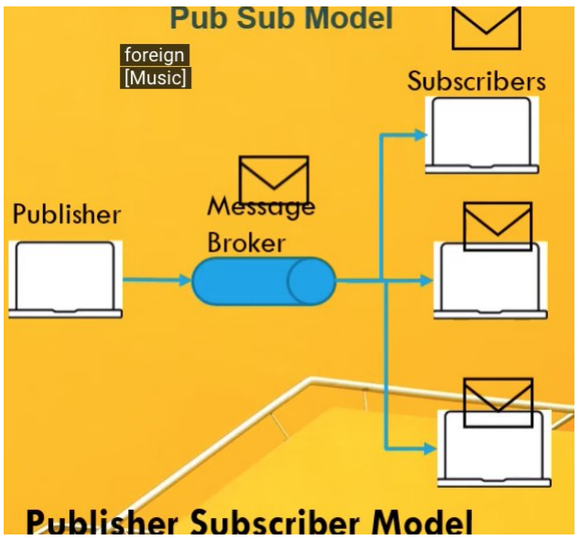
```
Name := flag.String("Name", "Sagar", " Name")
    flag.Parse() // whatever pass as command line like –Name Sham , it take this name if don't provied default is Sagar
    fmt.Println("Hello ", *Name)
```

ORM

In GORM, the default behavior for table naming is to **pluralize(Plural)** the struct name and convert it to **snake_case**. For example, if you have a struct named Student, GORM will map it to the students table by default.

student => students

Pub /sub Message



Sync Package

```go
package main
import (
    "fmt"
    "sync"
    "time"
)

func Worker(id int, wg *sync.WaitGroup) {
    defer wg.Done() //decrement waitgroup by 1

    fmt.Printf("Worker %d starting\n", id)
    time.Sleep(time.Second * 4) //sleep so, switch thread
    fmt.Println("woker task done ")
}

func main() {
    defer fmt.Println("main exit")
    var wg sync.WaitGroup

    for i := 0; i < 5; i++ {
        wg.Add(1)//increment waitgroup by 1.
        go Worker(i, &wg)
    }
    wg.Wait() // wait all worker finished
}
```

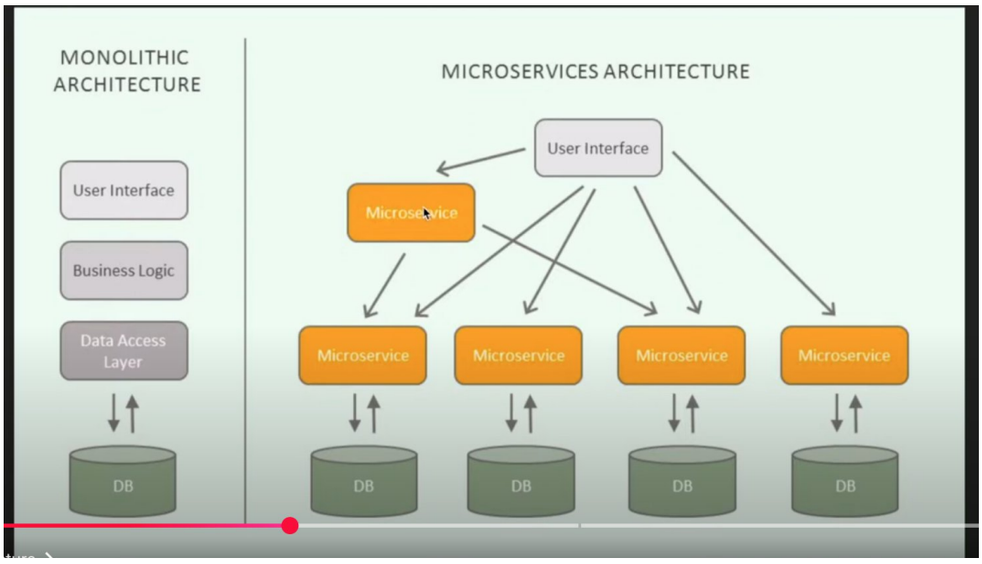
Key Methods:

- Add(delta int): Increments the counter for tasks to wait for.
- Done(): Decrements the counter when a task completes.
- Wait(): Blocks until the counter reaches zero.

```go
package main


import (
```

```
    "fmt"
    "sync"
)


const n = 0


var oddCh = make(chan bool)
var evenCh = make(chan bool)
var doneCh = make(chan bool)


func Odd(wg *sync.WaitGroup) {
    defer wg.Done()
   /*  below if block not working
  if n == 0 {
        //doneCh <- true
        //oddCh <- true
        evenCh <- true
    }*/
    for i := 1; i <= n; i += 2 {
        //wait odd channel
        <-oddCh //TODO , what this meaning
        fmt.Println(i)


        if i+1 <= n {
            evenCh <- true
        } else {
            doneCh <- true
        }


    }
}


func Even(wg *sync.WaitGroup) {
    defer wg.Done()

 /*  below if block not working
    if n == 0 {
        doneCh <- true
        //evenCh <- true
        oddCh <- true
    }*/
    for i := 2; i <= n; i += 2 {
        //wait even channel
        <-evenCh //TODO , what this meaning
        fmt.Println(i)


        if i+1 <= n {
            oddCh <- true
        } else {
            doneCh <- true
        }
    }
}


func main() {
    fmt.Println("Hello, World!")
if (n ==0){ return }
```

```
    var wg sync.WaitGroup


    go Even(&wg)
    wg.Add(1)
    go Odd(&wg)


    wg.Add(1)


    //starting odd channel
    oddCh <- true


    //done channel received
    <-doneCh


    wg.Wait() //wait for all thread
}
```



TODO LIST :

1. SQL Boiler (ORM)

2 .**Go (Programming Language), golang, go template, docker, REST APIs, web api, JavaScript**

- 3. **Develop and maintain OpenAPI specifications and implementations using go-openapi.**

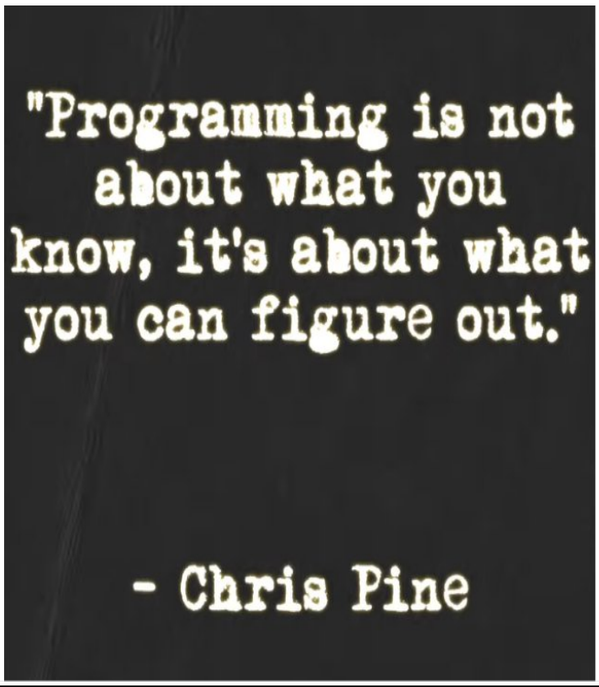
3.

1. Cloud Computing (Be Cloud Certified...AWS/Google/Azure)
2. Git/Version Control (learn basics on youtube)
3. Basic Database Knowledge
4. Basic Knowledge of Linux
5. Basic Docker Knowledge (learn docker in 1hour)
6. Basic API knowledge (API for each language)

7. Pipeline Familiarity (github actions)

brilliant.org

4.


"Programming is not
about what you
know, it's about what
you can figure out."
- Chris Pine