

Go



Presented by
VAISHALI TAPASWI

FANDS INFONET Pvt.Ltd.
www.fandsindia.com



Ground Rules

- Turn off cell phone. If you cannot please keep it on silent mode. You can go out and attend your call.
- If you have any questions or issues please let me know immediately.
- Let us be punctual.

www.fandsindia.com



Agenda

www.fandsindia.com

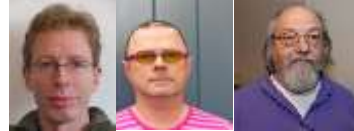


Go or Golang

- The language is called Go. The "golang" name arose because the web site is golang.org, not go.org (not available). Many use the golang name, though, and it is handy as a label. For instance, the Twitter tag for the language is "#golang". The language's name is just plain Go, regardless.
- Go is a compiled systems-oriented programming language started by Google in 2007. Go can be considered the result of a rather conservative language evolution from languages such as C and C++

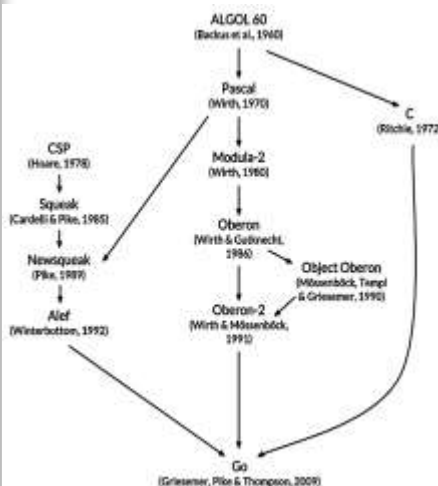
Go

- Developed ~2007 at Google by Robert Griesemer, Rob Pike, Ken Thompson
- Open Source
- C-like syntax
- Compiled, Statically Typed
 - Very Fast Compilation
- Garbage Collection
- Built-in Concurrency
- No classes or Type Inheritance or Overloading or Generics
 - unusual interface mechanism instead of inheritance



Go Influences

□ Positive



□ Negative

"When the three of us got started, it was pure research. The three of us got together and decided that we hated C++. We started off with the idea that all three of us had to be talked into every feature in the language, so there was no extraneous garbage put into the language for any reason."
(Ken Thompson)



C Influence with simplicity & safety

- A syntax and environment adopting patterns more common in dynamic languages:
 - Optional concise variable declaration and initialization through type inference (`x := 0` not `int x = 0`; or `var x = 0`);.
 - Fast compilation times
 - Remote package management (`go get`) and online package documentation.
- Distinctive approaches to particular problems:
 - Built-in concurrency primitives: light-weight processes (goroutines), channels, and the `select` statement.
 - An interface system in place of virtual inheritance, and type embedding instead of non-virtual inheritance.
 - A toolchain that, by default, produces statically linked native binaries without external dependencies.
- A desire to keep the language specification simple, by omitting features which are common in similar languages.



Go Tools

- | | |
|---|--|
| <ul style="list-style-type: none">□ <code>go build</code><ul style="list-style-type: none">– which builds Go binaries using only information in the source files themselves, no separate makefiles□ <code>go test</code><ul style="list-style-type: none">– for unit testing and microbenchmarks□ <code>go fmt</code><ul style="list-style-type: none">– for formatting code□ <code>go get</code><ul style="list-style-type: none">– for retrieving and installing remote packages□ <code>go vet</code><ul style="list-style-type: none">– a static analyzer looking for potential errors in code | <ul style="list-style-type: none">□ <code>go run</code><ul style="list-style-type: none">– a shortcut for building and executing code□ <code>Go doc</code><ul style="list-style-type: none">– for displaying documentation or serving it via HTTP□ <code>Gorename</code><ul style="list-style-type: none">– for renaming variables, functions, and so on in a type-safe way□ <code>go generate</code><ul style="list-style-type: none">– a standard way to invoke code generators |
|---|--|

Also includes profiling and debugging support, runtime instrumentation (track GC pauses), and a race condition tester.



HelloWorld

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World")
}
```

```
go run hello.go # to compile and run
go build hello.go # to create a binary
go fmt hello.go # for more
```



Lab 1

- Create demo1.go
- Compile/Run
- Check go fmt for demo1.go
- Modify package name and watch error message
- Modify main method signature
- Go doc fmt.Println
- Check golang documentation



Program Consists of

- Package Declaration
- Import Packages
- Functions
- Variables
- Statements and Expressions
- Comments
 - // , /*..*/



Package Declaration

- Go code is organized into packages
- Naming
 - Good package names are short and clear.
 - They are lower case
 - No under_scores or mixedCaps
 - Just simple nouns
- Package paths

```
import (  
    "context"           // package context  
    "golang.org/x/time/rate" // package rate  
    "os/exec"           // package exec
```
- Nested packages are supported



Functions

□ Unusual features

- Multiple return values
- Named result parameters
 - `func nextInt(b []byte, pos int) (value, nextPos int)`



Variables

```
var c1, c2 rune
```

```
var a, b, c = 0, 1.23, false
```

```
x := 0; y := 1.23; z := false
```

- Go infers the type from the type of the initializer
- Assignment between items of different type requires an explicit conversion, e.g., `int(float_expression)`



Statements and Expressions

- If
- If ..Else
- Switch
- For
- Defer
- For like while



Basic Go Syntax

<https://go.dev/tour>



Packages, Variables and Functions



Lab - Packages

- Use `Os.Args` to print all the command line arguments and print sum of string length of all the arguments
 - `Len(..)`
- Check OS documentation to print the same of current executable.



Lab - Functions

```
func add(x int, y int) int {  
    return x + y  
}
```

```
}  
func add(x, y int) int {  
    return x + y  
}
```

- Create a go file to create two functions add and divide
- Invoke those functions from main method



Lab – different Code files

- Create two go files
 - Helper.go – add, divide
 - Lab.go – main to invoke add and divide
- Run lab.go
 - See undefined error
- Run
 - Go run lab.go helper.go ...



Lab – Function multiple result

A function can return any number of results

```
func swap(x, y string) (string, string) {  
    return y, x  
}
```

- Write a calc method to return addition, subtraction



Lab - Named return values

A return statement without arguments returns the named return values. This is known as a "naked" return.

```
func split(sum int) (x, y int) {  
    x = sum * 4 / 9  
    y = sum - x  
    return  
}
```

- Write a function to return all calculations like +, -, *, /



Lab - Exported names

A name is exported if it begins with a capital letter

- Create a calc.go file with different package and write these two functions in the same
- Invoke from main.main method
- Understand GOROOT and GOPATH



Variables

A var statement can be at package or function level

- Create variables in different scopes and check
- Declare same variable name at package and function level and observe
 - Scope precedence



Data Types

- bool
- string
- int int8 int16 int32 int64
- uint uint8 uint16 uint32 uint64 uintptr
- byte // alias for uint8
- rune // alias for int32
- // represents a Unicode code point
- float32 float64
- complex64 complex128
- Variables declared without an explicit initial value are given their zero value.
- The zero value is:
 - 0 for numeric types,
 - false for the boolean type, and
 - "" (the empty string) for strings.



Constants

- Constants are declared like variables, but with the const keyword.
- Constants can be character, string, boolean, or numeric values.
- Constants cannot be declared using the := syntax.



Lab

- Asgn1
 - Accept a number from user and print Fibonacci series till that number
- Asgn2
 - Accept 5 strings from user and sort and print the same



Flow Control Statement



For

- Init, Condition and Post
 - for $i := 0; i < 10; i++$
- For is Go's while
 - for $\text{sum} < 1000 \{$
- Infinite Loop
 - for $\{$



If

- parentheses () optional but the braces { } are required.
 - if $x < 0$
- If with a short statement
 - if $v := \text{math.Pow}(x, n); v < \text{lim}$
- If and else
 - if $v := \text{math.Pow}(x, n); v < \text{lim} \{$
 return v
 } else {
 fmt.Printf("%g >= %g\n", v, lim)
 }



Switch

- A switch statement is a shorter way to write a sequence of if - else statements. It runs the first case whose value is equal to the condition expression.
- No fallback

```
switch os := runtime.GOOS; os {  
    case "darwin":  
        fmt.Println("OS X.")  
    case "linux":  
        fmt.Println("Linux.")  
    default:  
        // freebsd, openbsd,  
        // plan9, windows...  
        fmt.Printf("%s.\n", os)  
}
```



Switch with no condition

- Switch without a condition is the same as switch true.
- This construct can be a clean way to write long if-then-else chains.

```
switch {  
    case t.Hour() < 12:  
        fmt.Println("Good  
morning!")  
    case t.Hour() < 17:  
        fmt.Println("Good  
afternoon.")  
    default:  
        fmt.Println("Good  
evening.")  
}
```




Closures

A closure is a function value that references variables from outside its body. The function may access and assign to the referenced variables; in this sense the function is "bound" to the variables.

```
package main
import "fmt"
func adder() func(int) int {
    sum := 0
    return func(x int) int
    {
        sum += x
        return sum
    }
}
```



Defer, Panic and Recover

Defer

- A defer statement pushes a function call onto a list. The list of saved calls is executed after the surrounding function returns. Defer is commonly used to simplify functions that perform various clean-up actions.

```
func do(srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()
    ...other operations
}
```

Defer, Stacking Defers

- A defer statement defers the execution of a function until the surrounding function returns.
- The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.

```
i := 10;
defer fmt.Println("world" , i)
i = 20;
fmt.Println("hello" , i)
-----
fmt.Println("counting")
for i := 0; i < 10; i++ {
    defer fmt.Println(i)
}
fmt.Println("done")
```



Panic

- Panic is a built-in function that stops the ordinary flow of control and begins panicking. When the function *F* calls panic, execution of *F* stops, any deferred functions in *F* are executed normally, and then *F* returns to its caller. To the caller, *F* then behaves like a call to panic. The process continues up the stack until all functions in the current goroutine have returned, at which point the program crashes. Panics can be initiated by invoking panic directly. They can also be caused by runtime errors, such as out-of-bounds array accesses.



Recover

- Recover is a built-in function that regains control of a panicking goroutine. Recover is only useful inside deferred functions. During normal execution, a call to recover will return nil and have no other effect. If the current goroutine is panicking, a call to recover will capture the value given to panic and resume normal execution.



Panic and Recover Example

```
package main
import "fmt"
func main() {
    f()
    fmt.Println("Returned normally from f.")
}
func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    fmt.Println("Calling g.")
    g(0)
    fmt.Println("Returned normally from g.")
}
```

```
func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v",
            i))
    }
    defer fmt.Println("Defer in
g", i)
    fmt.Println("Printing in g",
i)
    g(i + 1)
}
```



Structs, Slices and Maps

Pointers

- A pointer holds the memory address of a value.
- The type `*T` is a pointer to a `T` value. Its zero value is `nil`.
 - `var p *int`
- The `&` operator generates a pointer to its operand.
 - `i := 42`
 - `p = &i`
- The `*` operator denotes the pointer's underlying value.
 - `fmt.Println(*p) // read i through the pointer p`
 - `*p = 21 // set i through the pointer p`
- This is known as "dereferencing" or "indirecting".

Structs

- A struct is a collection of fields.


```
type Vertex struct {
    X int
    Y int
}
```
- Struct fields are accessed using a dot.
- Initialize Options
 - `v1 = Vertex{1, 2} // has type Vertex`
 - `v2 = Vertex{X: 1} // Y:0 is implicit`
 - `v3 = Vertex{} // X:0 and Y:0`
 - `p = &Vertex{1, 2} // has type *Vertex`



Arrays

- An array's length is part of its type, so arrays cannot be resized.
 - var a [2]string
 - a[0] = "Hello"
 - primes := [6]int{2, 3, 5, 7, 11, 13}



Slices

- An array has a fixed size. A slice, on the other hand, is a dynamically-sized, flexible view into the elements of an array
 - a[low : high]
 - includes the first element, but excludes the last one
 - primes := [6]int{2, 3, 5, 7, 11, 13}
 - var s []int = primes[1:4]



Slices = References to Arrays

- A slice does not store any data, it just describes a section of an underlying array.
- Changing the elements of a slice modifies the corresponding elements of its underlying array.
- Other slices that share the same underlying array will see those changes.



Slice Syntax Variation

- Slice Literal
 - A slice literal is like an array literal without the length.
 - `[3]bool{true, true, false}`
 - `[]bool{true, true, false}`
- Slice defaults (for `var a [10]int`)
 - `a[0:10]`
 - `a[:10]`
 - `a[0:]`
 - `a[:]`



Slice Length and Capacity

- The length of a slice is the number of elements it contains. `len(x)`
 - You can extend a slice's length by re-slicing it, provided it has sufficient capacity
- The capacity of a slice is the number of elements in the underlying array, counting from the first element in the slice. `cap(x)`
- Nil slices - The zero value of a slice is nil.
 - A nil slice has a length and capacity of 0 and has no underlying array.
 - `var s []int`



Make for Dynamically Sized Arrays

- Slices can be created with the built-in `make` function
 - The `make` function allocates a zeroed array and returns a slice that refers to that array:
 - `a := make([]int, 5) // len(a)=5`
- To specify a capacity, 3rd argument
 - `b := make([]int, 0, 5) // len(b)=0, cap(b)=5`
 - `b = b[:cap(b)] // len(b)=5, cap(b)=5`
 - `b = b[1:] // len(b)=4, cap(b)=4`



Appending to a slice

- `func append(s []T, vs ...T) []T`
 - The resulting value of `append` is a slice containing all the elements of the original slice plus the provided values.
 - If the backing array of `s` is too small to fit all the given values a bigger array will be allocated. The returned slice will point to the newly allocated array.
 - Immutable



Range

- The range form of the for loop iterates over a slice or map.
 - When ranging over a slice, two values are returned for each iteration, index and element
- ```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}
func main() {
 for i, v := range pow {
 fmt.Printf("2**%d = %d\n", i, v)
 }
}
```
- Can skip any of these with `_`.



## Maps

- A map maps keys to values.
- The zero value of a map is nil. A nil map has no keys, nor can keys be added.
- The make function returns a map of the given type, initialized and ready for use.

```
var m map[string]int;
m = make(map[string]int)
m["a"]=100
m["b"]=200
m["a"]=300
fmt.Println(m["a"]);
```

```
var m = map[string]int{
 "a":10,
 "b":20,
 "c":30,
}
```



## Working with Maps

- Insert or update an element in map m:
  - m[key] = elem
- Retrieve an element:
  - elem = m[key]
- delete an element:
  - delete(m, key)
- Test that a key is present with a two-value assignment:
  - elem, ok = m[key]
  - If key is in m, ok is true. If not, ok is false.
  - If key is not in the map, then elem is the zero value for the map's element type.



# Methods and Interfaces



## Methods

- ❑ Go does not have classes. However, you can define methods on types.
- ❑ A method is a function with a special receiver argument.
- ❑ The receiver appears in its own argument list between the func keyword and the method name.

```
func (v Vertex) Abs() float64 {
 return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

## Methods

```
type MyFloat float64
func (f MyFloat) Abs() float64 {
 if f < 0 {
 }
 return float64(f) }
```

- You can declare a method on non-struct types, too.
- In this example we see a numeric type `MyFloat` with an `Abs` method.
- Can only declare a method with a receiver whose type is defined in the same package as the method. You cannot declare a method with a receiver whose type is defined in another package (which includes the built-in types such as `int`).

## Pointer Receivers

- Methods with pointer receivers can modify the value to which the receiver points (as `Scale` does here). Since methods often need to modify their receiver, pointer receivers are more common than value receivers.

```
type Vertex struct {
 X, Y float64
}

func (v Vertex) Abs() float64 {
 return math.Sqrt(...)
}

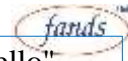
func (v *Vertex) Scale(f float64)
{
 v.X = v.X * f
 v.Y = v.Y * f
}
```



## Interfaces

- An interface type is defined as a set of method signatures.
- A value of interface type can hold any value that implements those methods.
- No implements keyword

```
type tostr interface {
 Convert() string
}
type Emp struct {
 empno int
 ename string
}
func (e Emp) Convert() string {
 str := "Emp Details[Empno = "+
 strconv.Itoa(e.empno)+ " , Name = "
 + e.ename + "]" ;
 return str
}
func main(){
 var a tostr;
 e := Emp{10, "aaa"}
 a = e
 fmt.Println(a.Convert())
}
```



## Type Assertions

```
var i interface{ } = "hello"
s := i.(string)
s, ok := i.(string)
```

- A type assertion provides access to an interface value's underlying concrete value.
- `t := i.(T)`
- This statement asserts that the interface value `i` holds the concrete type `T` and assigns the underlying `T` value to the variable `t`.
- If `i` does not hold a `T`, the statement will trigger a panic



## Type Switches

- A *type switch* is a construct that permits several type assertions in series.
- A type switch is like a regular switch statement, but the cases in a type switch specify types (not values), and those values are compared against the type of the value held by the given interface value.

```
func do(i interface{}) {
 switch v := i.(type) {
 case int:
 fmt.Printf("Twice %v is
%v\n", v, v*2)
 case string:
 fmt.Printf("%q is %v
bytes long\n", v, len(v))
 default:
 fmt.Printf("I don't know
about type %T!\n", v)
 }
}
```



## Stringer

- One of the most ubiquitous interfaces is *Stringer* defined by the `fmt` package.
- A *Stringer* is a type that can describe itself as a string. The `fmt` package (and many others) look for this interface to print values.

```
type Person struct {
 Name string
 Age int
}

func (p Person) String() string {
 return fmt.Sprintf("%v (%v
years)", p.Name, p.Age)
}

func main() {
 a := Person{"Arthur Dent", 42}
 z := Person{"Zaphod
Beeblebrox", 9001}
 fmt.Println(a, z)
}
```



## Http Get

```
package main
import ("fmt" "net/http" "io/ioutil")
func main() {
 url := "https://reqres.in/api/users/2"
 var client = http.Client{}
 resp, err := client.Get(url);
 if err != nil {
 fmt.Println("Error "); }
 else {
 fmt.Println("resp" , resp);
 data, _ := ioutil.ReadAll(resp.Body)
 fmt.Println("\n\n\nbody " ,string(data))
 }
}
```



## Http Package

```
package main
import (
 "fmt" io
 "log"
 http "net/http")

func main() {
 helloHandler := func(w http.ResponseWriter, req
 *http.Request) {
 io.WriteString(w, "<h1>Index Page</h1>")
 http.HandleFunc("/", helloHandler)
 fmt.Printf("sever starting on 8080")
 log.Fatal(http.ListenAndServe(":8080", nil))
 }
```

A vertical decorative bar on the left side of the slide, composed of many thin, horizontal stripes in various shades of blue, black, and yellow.

# Go Popular Utilities

A vertical decorative bar on the left side of the slide, composed of many thin, horizontal stripes in various shades of blue, black, and yellow.

## Reading Properties File

```
package main

import "fmt"
import "github.com/magiconair/properties"

func main() {
 fmt.Println("Hello, World")
 p := properties.MustLoadFile("sim.properties",
properties.UTF8)
 if port, ok := p.Get("port"); ok {
 fmt.Println(port)
 }
}
```





## Logger

```
main() {
 f, err := os.OpenFile("testlogfile.log",
 os.O_RDWR | os.O_CREATE | os.O_APPEND,
 0666)
 if err != nil {
 log.Fatalf("error opening file: %v", err)
 }
 defer f.Close()
 log.SetOutput(f)
 log.Println("This is a test log entry")
}
```



## JSON $\leftrightarrow$ Struct

## XML $\leftrightarrow$ Struct

### □ "encoding/json"

```
type MyJsonObject struct {
 Page int `json:"page"`
 PerPage int `json:"per_page"`
}
```

### □ Marshal

– json.Marshal()

### □ UnMarshal

– str := `{"page": 1, PerPage:4}`  
– res := MyJsonObject{}  
– json.Unmarshal([]byte(str), &res)



# GoRoutines



## Goroutine

- A goroutine is a lightweight thread managed by the Go runtime.
- `go f(x, y, z)`
  - starts a new goroutine running
- The evaluation of `f`, `x`, `y`, and `z` happens in the current goroutine and the execution of `f` happens in the new goroutine.
- Goroutines run in the same address space, so access to shared memory must be synchronized. The `sync` package provides useful primitives, although you won't need them much in Go as there are other primitives.



## Channels

- Channels are a typed conduit through which you can send and receive values with the channel operator, <-
  - ch <- v // Send v to channel ch.
  - v := <-ch // Receive from ch, and assign value to v.
- Like maps and slices, channels must be created before use:
  - ch := make(chan int)
- By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.
- The example code sums the numbers in a slice, distributing the work between two goroutines. Once both goroutines have completed their computation, it calculates the final result.



## Channels

```
func reader(c chan string) {
 for msg := range c {
 fmt.Println("in reader ", msg)
 time.Sleep(100)
 }
}

func writer(str string, c chan string)
{
 for i := 1; i <= 5; i++ {
 fmt.Println("#####in count ", i)
 c <- str + strconv.Itoa(i);
 time.Sleep(time.Millisecond * 100)
 }
}
```

```
func main() {
 c := make(chan string,10)
 go writer("sheep", c)
 go reader(c)
 for msg := range c {
 fmt.Println("in main ", msg)
 time.Sleep(time.Millisecond * 300)
 }
 i := 10;
 fmt.Scanln(&i);
}
```



## Buffered Channels

- Channels can be buffered. Provide the buffer length as the second argument to make to initialize a buffered channel:
- `ch := make(chan int, 100)`
- Sends to a buffered channel block only when the buffer is full. Receives block when the buffer is empty.

```
package main
import "fmt"
func main() {
 ch := make(chan
int, 2)
 ch <- 1
 ch <- 2
 fmt.Println(<-ch)
 fmt.Println(<-ch)
}
```



## Range and Close

- A sender can close a channel to indicate that no more values will be sent.
- Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression

```
func fibonacci(n int, c chan
int) {
 x, y := 0, 1
 for i := 0; i < n; i++ {
 c <- x
 x, y = y, x+y
 }
 close(c)
}
```



## 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.

```
func fibonacci(c, quit chan int) {
 x, y := 0, 1
 for {
 select {
 case c <- x:
 x, y = y, x+y
 case <-quit:
 fmt.Println("quit")
 return
 }
 }
}
```



## sync.Mutex

```
type SafeCounter struct {
 v map[string]int
 mux sync.Mutex
}
```

- Channels are great for communication among goroutines.
- What if we just want to make sure only one goroutine can access a variable at a time to avoid conflicts?
- This concept is called mutual exclusion, and the conventional name for the data structure that provides it is mutex.
- Go's standard library provides mutual exclusion with sync.Mutex and its two methods:
  - Lock
  - Unlock
- We can define a block of code to be executed in mutual exclusion by surrounding it with a call to Lock and Unlock
- We can also use defer to ensure the mutex will be unlocked as in the Value method.



## WaitGroups

WaitGroup is actually a type of counter which blocks the execution of function (or might say A goroutine) until its internal counter become 0.

```
func main() {
 var wg sync.WaitGroup
 wg.Add(2)
 go func() {
 deposit()
 wg.Done()
 }()
 go func() {
 widraw()
 wg.Done()
 }()
 fmt.Println("before
wait...")
 wg.Wait()
 fmt.Println("in main after
deposit and widraw ")
}
```



## Handling Race Conditions

### ❑ Race Detector

- Go's race detector enables instrumenting memory accesses in order to determine if memory is ever being acted on concurrently. The go test framework exposes the race detector through the -race flag

### ❑ Explicit Synchronization

- Explicit synchronization is where variables accesses are protected through synchronization primitives such as a mutex

### ❑ Static Analysis (go vet)

- Static analysis (specifically mutex detection) helps with misuse of mutex and is another supportive reactionary detection. It doesn't help to directly detect when a variable needs a mutex, but only if a mutex isn't being used correctly.



# Database Communication



## RDBMS - MySQL

- ❑ Import \_ "github.com/go-sql-driver/mysql"
- ❑ db, err :=  
sql.Open("mysql","username:password  
@tcp(hostname:port)/dbname")
- ❑ rows, err1 := db.Query("insert into emp  
values (11,'AAA',11000)")



# MongoDB

<https://www.mongodb.com/blog/post/mongodb-go-driver-tutorial-part-1-connecting-using-bson-and-crud-operations>

## □ Import

- "context"
- "go.mongodb.org/mongo-driver/mongo"
- "go.mongodb.org/mongo-driver/mongo/options"

## □ Significance of context

- Package context defines the Context type, which carries deadlines, cancelation signals, and other request-scoped values across API boundaries and between processes.
- TODO returns a non-nil, empty Context. Code should use context.TODO when it's unclear which Context to use or it is not yet available (because the surrounding function has not yet been extended to accept a Context parameter).



# MongoDB

## □ Connect to MongoDB

```
clientOptions :=
options.Client().ApplyURI("mongodb://localhost:27017")
client, err := mongo.Connect(context.TODO(), clientOptions)
if err != nil {
 log.Fatal(err)
}
err = client.Ping(context.TODO(), nil)
if err != nil {
 log.Fatal(err)
}
fmt.Println("Connected to MongoDB!")
```





## MongoDB

### □ Insert

```
emp1 := Emp{10, "Vaishali", 11000}
collection := client.Database("test").Collection("emp")
inResult, err := collection.InsertOne(context.TODO(),
emp1)
if err != nil {
log.Fatal(err)}
fmt.Println("Inserted a single document: ",
inResult.InsertedID)
```



## DynamoDB

### □ Create session with aws cli config files

```
import(
 "github.com/aws/aws-sdk-go/aws"
 "github.com/aws/aws-sdk-go/aws/session"
)

sess, err := session.NewSession(&aws.Config{
 Region: aws.String("us-east-1")})

if err != nil {
 fmt.Println("In err", err)
} else {
 fmt.Println("Session Created Successfully")
}
```



## Create Service and Invoke

### □ Create Services with session parameter

```
import(
 "github.com/aws/aws-sdk-go/service/dynamodb"
)
svc := dynamodb.New(sess)
listtablesoutput, err :=
 svc.ListTables(&dynamodb.ListTablesInput{ })
```



## Redis (<https://tutorialedge.net/golang/go-redis-tutorial/>)

```
package main
import (
 "fmt"
 "github.com/go-redis/redis"
)
func main() {
 fmt.Println("Go Redis Tutorial")
 client := redis.NewClient(&redis.Options{
 Addr: "localhost:6379",
 Password: "",
 DB: 0, })
 pong, err := client.Ping().Result()
 fmt.Println(pong, err) }
```



## Processing

### □ Setting Values

- Simple
  - `err = client.Set("name", "Elliot", 0).Err()`
- Composite
  - `json, err := json.Marshal(Author{Name: "Elliot", Age: 25})`
  - `err = client.Set("id1234", json, 0).Err()`

### □ Getting Values

- `val, err := client.Get("name").Result()`



## Testing in Go



## Unit Testing

- Unit components
  - functions, structs, methods and pretty much anything that end-user might depend on
- Unit Testing
  - test the integrity of these unit components by creating unit tests. A unit test is a program that tests a unit component by all possible means and compares the result to the expected output.

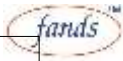


## What can we test?

- If we have a module or a package, we can test whatever exports are available in the package (because they will be consumed by the end-user).
- If we have an executable package, whatever units we have available within the package scope, we should test it.



## How




```
import "testing"
func TestAbc(t *testing.T) {
 t.Error() // to indicate test failed
}
```

- ❑ The built-in testing package is provided by the Go's standard library.
- ❑ A unit test is a function that accepts the argument of type `*testing.T` and calls the `Error` (or any other error methods which we will see later) on it.
- ❑ Function must start with `Test` keyword and the latter name should start with an uppercase letter
- ❑ Go test filename (-v)



## Coverage

- 
- ❑ Test Coverage is the percentage of your code covered by test suit. In layman's language, it is the measurement of how many lines of code in your package were executed when you ran your test suit (compared to total lines in your code). Go provide built-in functionality to check your code coverage.
  - ❑ Go test .. -cover



## Analyze Coverage

- Create a coverage output file
  - go test -v ../test.go -coverprofile tmp.txt
- Read tmp.txt
- Go tools to convert tmp.txt in readable format
  - go tool cover -html=tmp.txt -o tmp.html
- Open tmp.html in browser

## Benchmarks

```
// from fib_test.go
func BenchmarkFib10(b *testing.B) {
 // run the Fib function b.N times
 for n := 0; n < b.N; n++ {
 Fib(10)
 }
}
```

- The Go testing package contains a benchmarking facility that can be used to examine the performance of your Go code.
- Benchmarks are placed inside `_test.go` files and follow the rules of their Test counterparts except name
- The value of `b.N` will increase each time until the benchmark runner is satisfied with the stability of the benchmark. This has some important ramifications which we'll investigate later in this article.
- Each benchmark must execute the code under test `b.N` times. The for loop in `BenchmarkFib10` will be present in every benchmark function.
- Run Benchmarks
  - go test Lab2\_test.go Lab2.go -bench=.
  - go test Lab6\_test.go Lab6.go -bench=, -v -benchtime=20s



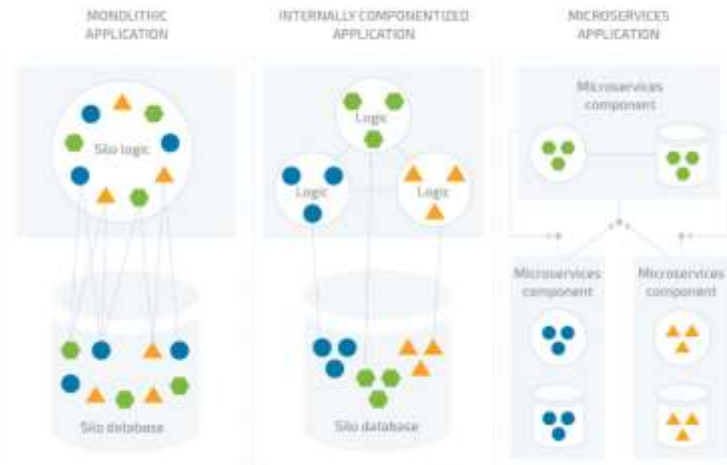
## Mocking

- Why Mocking
- Go mocking
  - Multiple options
    - Testify
    - GoMock
- Testify
  - Testify is more popular and active on github
  - Used by more packages according to goDoc
  - Testify provides better error messages on unexpected calls and calls with unexpected parameter values, including argument types, a helpful stack trace, and closest matching calls



## Micro Services Development

## Monolith to Micro Services



## Monolith to Micro Services

| Category          | Monolithic architecture                                                             | Microservices architecture                                                                              |
|-------------------|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| Code              | Single code base                                                                    | Multiple code base. Each microservice has its own code base                                             |
| Understandability | Often confusing and hard to understand                                              | Much better readability<br>And easier to maintain                                                       |
| Deployment        | Complex deployments with maintenance windows and scheduled downtimes.               | Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime. |
| Language          | Typically entirely developed in one programming language.                           | Each microservice can be developed in a different programming language.                                 |
| Scaling           | Requires you to scale the entire application even though bottlenecks are localized. | Enables you to scale bottle-necked services without scaling the entire application.                     |

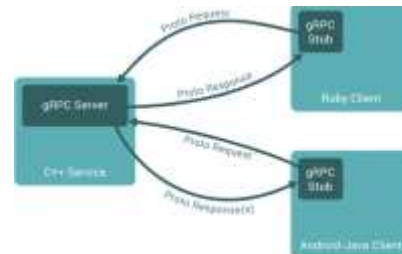


## Go Support

### □ Format

- XML, JSON, protobuf/gRPC
- gRPC is a light-weight binary based RPC communication protocol brought out by Google
- gRPC uses the new HTTP 2.0 spec
- Allows for the use of binary data.
- Allows bi-directional streaming
- gRPC has an interchange DSL called protobuf.
- Protobuf allows you to define an interface to your service using a developer friendly format.

## gRPC



- A client application can directly call a method on a server application on a different machine as if it were a local object, making it easier for you to create distributed applications and services.
- gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types.
- On the server side, the server implements this interface and runs a gRPC server to handle client calls. On the client side, the client has a stub (referred to as just a client in some languages) that provides the same methods as the server.

# REST vs gRPC

| Capability / Architecture                                                                         | REST                             | GraphQL                                          | gRPC                                   |
|---------------------------------------------------------------------------------------------------|----------------------------------|--------------------------------------------------|----------------------------------------|
| Compressed Payload                                                                                | 1: gzip                          | 2: gzip + requested fields                       | 3: Binary                              |
| Asynch Requests                                                                                   | 0: HTTP                          | 2: Requires AMQP                                 | 3: HTTP/2                              |
| Reduced Requests                                                                                  | 1: Requires a Composite API      | 3: Retrieves just what is specified in the query | 2: Write your function                 |
| Market standards                                                                                  | 3: (Lots of Tools and Standards) | 1: Experimental Use                              | 1: Experimental Use                    |
| Reuse                                                                                             | 3: Based on Business Domains     | 1: Specific function                             | 1: Specific function                   |
| Compatibility with Event Architecture or Reactive                                                 | 1: Synch Requests                | 2: Mutation Syntax and Asynch Calls              | 3: Asynch Calls and Syntax flexibility |
| Parser de trama optimizada                                                                        | 1: JSON Serialization            | 1: JSON Serialization                            | 3: Binary Serialization                |
| <b>SCORE (Example)</b>                                                                            | <b>10</b>                        | <b>12</b>                                        | <b>16</b>                              |
| Scale: 0 - Does not Apply; 1- Low applicability; 2 - Average applicability; 3: High applicability |                                  |                                                  |                                        |

## Go Support for

- Containerization
  - Docker
- Different Patterns involved
  - Service Registry
  - Service Discovery
  - Load Balancer
- Frameworks
  - Go-micro (Service Discovery)
  - Gorm - Go + ORM



## JSON and Get/Post

```
func handleRequests() {
 http.HandleFunc("/", homePage)
 http.HandleFunc("/emp", process)
 log.Fatal(http.ListenAndServe(":8080", nil))
}

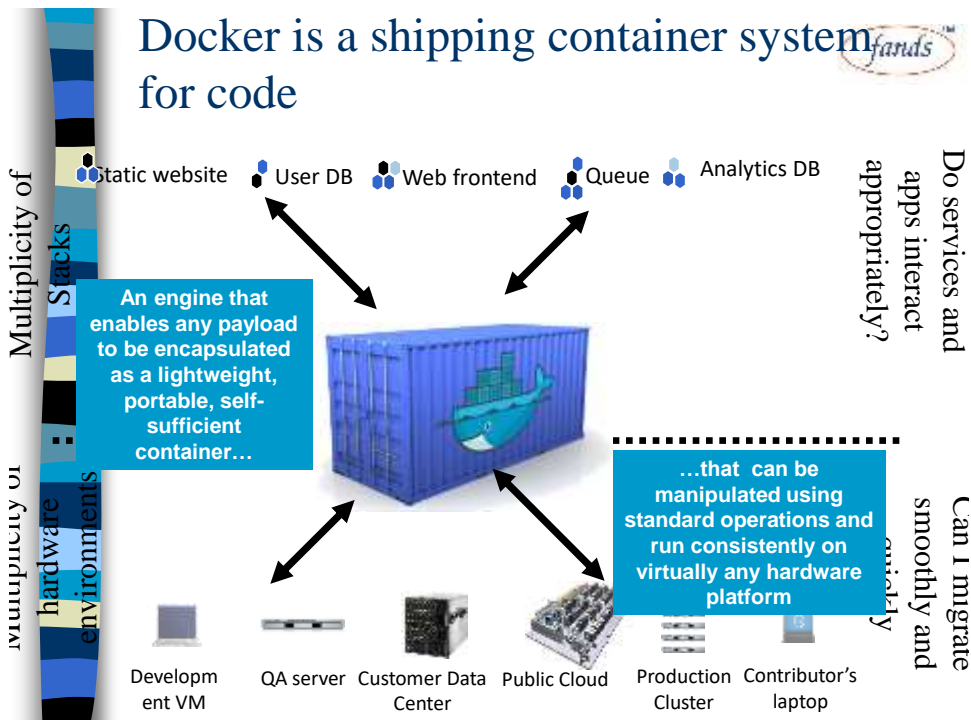
func process(w http.ResponseWriter, r *http.Request) {
 switch r.Method {
 case "GET":
 returnallemps(w, r)
 case "POST":
 reqBody, _ := ioutil.ReadAll(r.Body)
 var emp Emp
 json.Unmarshal(reqBody, &emp)
 EmpArr = append(EmpArr, emp)
 json.NewEncoder(w).Encode(emp)
 }
}
```



## Lab Write REST API for emp table of DynamoDb/RDS

- Write Http Server
  - Create Emp struct with json tags
  - Write main to start http server
  - Write Get/Post methods for /emps
  - Test Code
- Modify current code to include separate file for EmpHandler
- Modify EmpHandler to connect to DynamoDb/MySQL for insert and retrieve

# Introduction to Docker and Kubernetes



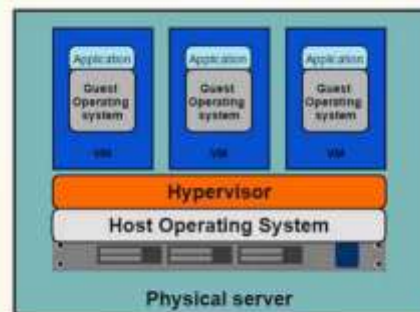
## What is Docker?

- Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux, Mac OS and Windows.
  - Wikipedia

[www.fandsindia.com](http://www.fandsindia.com)

## Hypervisor-based Virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine



[www.fandsindia.com](http://www.fandsindia.com)

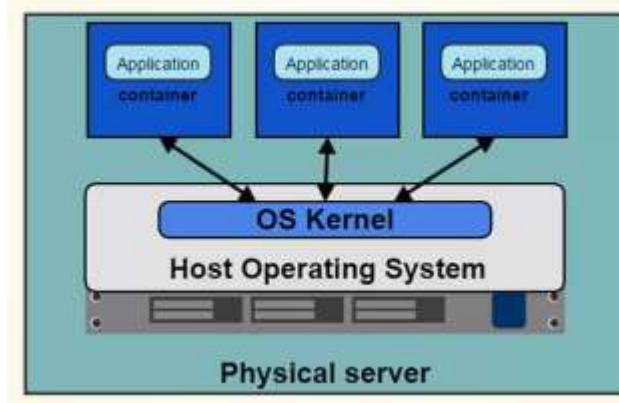
## Introducing Containers

*Container based virtualization uses the kernel on the host's operating system to run multiple guest instances*

- Each guest instance is called a container
- Each container has its own
  - Root filesystem
  - Processes
  - Memory
  - Network ports

[www.fandsindia.com](http://www.fandsindia.com)

## Containers



[www.fandsindia.com](http://www.fandsindia.com)



## Containers Vs VMs

- Containers are more lightweight
- No need to install guest OS
- Less CPU, RAM, storage space required
- More containers per machine than VMs
- Greater portability

[www.fandsindia.com](http://www.fandsindia.com)



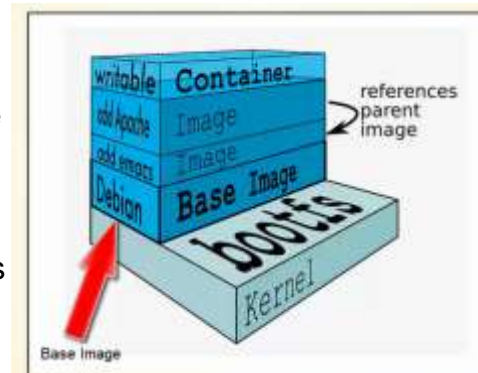
## Images and Containers

- |                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>□ Images<ul style="list-style-type: none"><li>– Read only template used to create containers</li><li>– Built by you or other Docker users</li><li>– Stored in the Docker Hub or your local Registry</li></ul></li></ul> | <ul style="list-style-type: none"><li>□ Containers<ul style="list-style-type: none"><li>– Isolated application platform</li><li>– Contains everything needed to run your application</li><li>– Based on images</li></ul></li></ul> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

[www.fandsindia.com](http://www.fandsindia.com)

## Image Layers

- Images are comprised of multiple layers
- A layer is also just another image
- Every image contains a base layer
- Docker uses a copy on write system
- Layers are read only



[www.fandsindia.com](http://www.fandsindia.com)

## Create Image

- Option 1
  - Create a container
  - Modify container
  - Commit as new image
- Option 2
  - Create dockerfile
    - Base Image
    - Compile or copy exe
    - Run (in container)
  - Build to create image
  - Create container and test

```
FROM golang:1.11
COPY web.go .
RUN go build ./web.go
EXPOSE 8080
CMD ./web
```

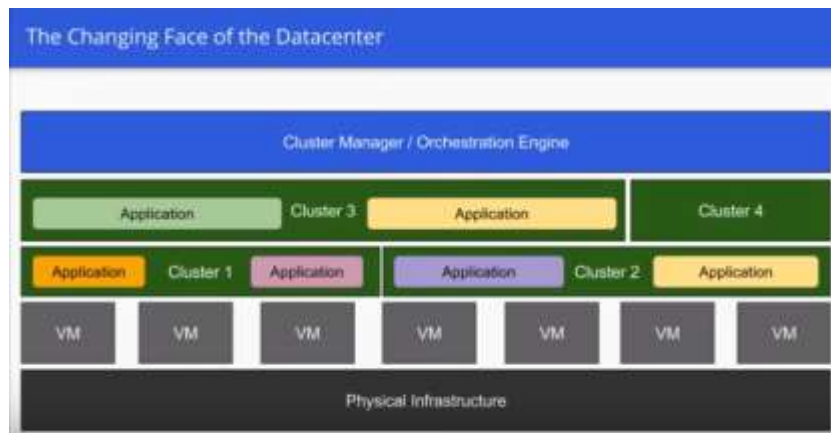


## Docker Compose

- With a single command
  - Build Images
  - Launch multiple containers
  - Configure port and network parameters
  - Stop multiple containers

```
version: "3"
services:
 web:
 build: ./web
 ports:
 - "8080:8080"
 networks:
 webnet:
 myrdbms:
 image: mysql:5.5
 ports:
 - "3306:3306"
 environment:
 MYSQL_ROOT_PASSWORD: mypass
 networks:
 webnet:
networks:
 webnet:
```

## Why Kubernetes?





## What Kubernetes?

- Kubernetes is a production-ready, open source platform designed with Google's accumulated experience in container orchestration
- Kubernetes orchestrates the placement (scheduling) and execution of application containers within and across computer clusters

[www.fandsindia.com](http://www.fandsindia.com)



## Kubernetes

- With modern web services, users expect applications to be available 24/7, and developers expect to deploy new versions of those applications several times a day. Containerization helps package software to serve these goals, enabling applications to be released and updated in an easy and fast way without downtime. Kubernetes helps you make sure those containerized applications run where and when you want, and helps them find the resources and tools they need to work

[www.fandsindia.com](http://www.fandsindia.com)

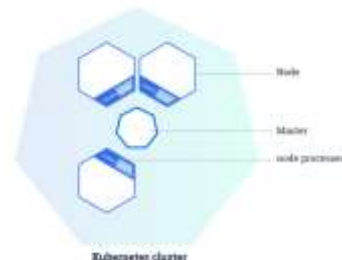
## Kubernetes Clusters

- Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit.
- Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way

[www.fandsindia.com](http://www.fandsindia.com)

## Kubernetes Cluster

- A Kubernetes cluster consists of two types of resources:
  - The Master coordinates the cluster
  - Nodes are the workers that run applications



[www.fandsindia.com](http://www.fandsindia.com)



## Master and Nodes

- The Master
  - responsible for managing the cluster.
  - Co-ordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.
- A node
  - is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster.
  - Each node has a Kubelet, which is an agent for managing the node and communicating with the Kubernetes master.
- A node is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster.
- The nodes communicate with the master using the Kubernetes API

[www.fandsindia.com](http://www.fandsindia.com)

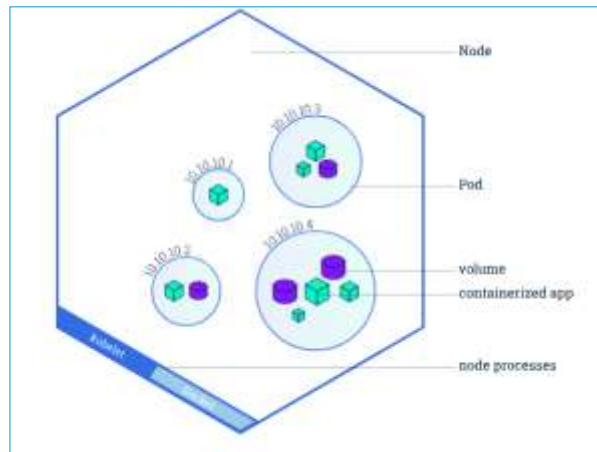
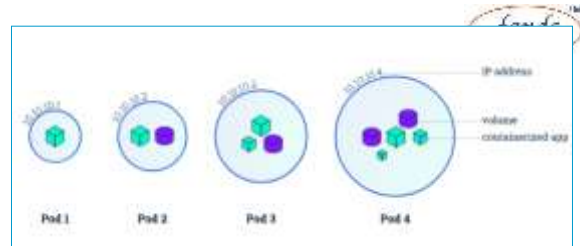


## Kubernetes Pods and Nodes

- A Pod is a Kubernetes abstraction that represents a group of one or more application containers (such as Docker or rkt), and some shared resources for those containers. Those resources include:
  - Shared storage, as Volumes
  - Networking, as a unique cluster IP address
  - Information about how to run each container, such as the container image version or specific ports to use

[www.fandsindia.com](http://www.fandsindia.com)

## Pods and Nodes



## Kubernetes Pods and Nodes

- A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster.
- Pod always runs on a **Node**.
- Each Node is managed by the Master. A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster. The Master's automatic scheduling takes into account the available resources on each Node.



## Node

- Every Kubernetes Node runs at least:
  - Kubelet, a process responsible for communication between the Kubernetes Master and the Node; it manages the Pods and the containers running on a machine.
  - A container runtime (like Docker, rkt) responsible for pulling the container image from a registry, unpacking the container, and running the application.

[www.fandsindia.com](http://www.fandsindia.com)



## Using kubectl to Create a Deployment

- Kubernetes Deployments
  - Once you have a running Kubernetes cluster, you can deploy your containerized applications on top of it. To do so, you create a **Deployment** configuration. The Deployment instructs Kubernetes how to create and update instances of your application. Once you've created a Deployment, the Kubernetes master schedules mentioned application instances onto individual Nodes in the cluster.

[www.fandsindia.com](http://www.fandsindia.com)



## Kubernetes Deployments

- Once the application instances are created, a Kubernetes Deployment Controller continuously monitors those instances. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces it. **This provides a self-healing mechanism to address machine failure or maintenance.**

[www.fandsindia.com](http://www.fandsindia.com)

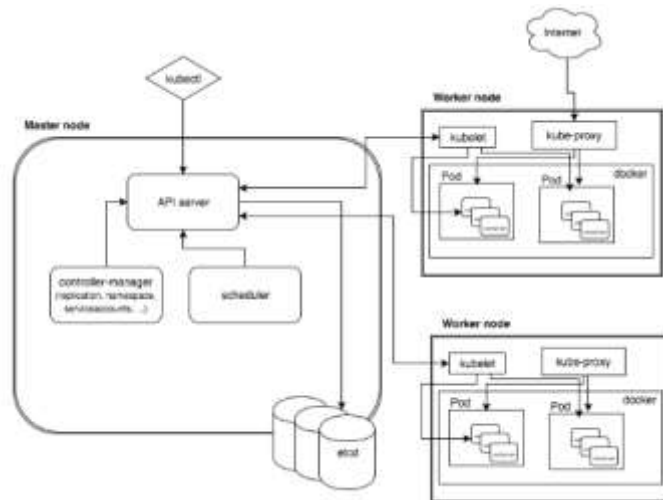


## Kubernetes Features

- Replication of components
- Auto-scaling
- Load balancing
- Rolling updates
- Logging across components
- Monitoring and health checking
- Service discovery
- Authentication

[www.fandsindia.com](http://www.fandsindia.com)

## High Level Kubernetes Architecture



## Design Patterns





## A Short History

- Christopher Alexander - 1970s
  - *A Pattern Language*
  - *A Timeless Way of Building*
- The Gang of Four (GoF): Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides - 1995
- *Design Patterns: Elements of Reusable Object-Oriented Software*

[www.fandsindia.com](http://www.fandsindia.com)



## What is Design Pattern?

- A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it. [Buschmann et al., 1996]
- *A design pattern is a well worn and known good solution to a common problem.*

[www.fandsindia.com](http://www.fandsindia.com)



## Categorization of Patterns

- **Creational Patterns:** Used to create objects instead of direct instantiation
- **Structural Patterns:** Compose objects into larger structures
- **Behavioural Patterns:** Communication between objects

[www.fandsindia.com](http://www.fandsindia.com)



## Why Design Patterns ?

- Reuse solutions that have worked in the past.
- Basing new designs on prior experience
- Once you know the pattern, a lot of design decisions follow automatically
- Makes successful design solutions more accessible to developers of new systems
- Help you identify less-obvious abstractions and the objects that can capture them

[www.fandsindia.com](http://www.fandsindia.com)



## What are Design Patterns ?

- Generically: Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
- Specifically: Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

[www.fandsindia.com](http://www.fandsindia.com)



## Design Themes

*Think differently about Types Vs. Classes  
Program to an interface (Type), not an  
implementation (Class).*

*Favor object composition over class inheritance,  
black box over white box reuse.*

*Design relationships between  
objects and their types to achieve good run-time  
structure. Don't limit yourself to compile time  
structures.*

[www.fandsindia.com](http://www.fandsindia.com)



## Principles of OO Class Design

- The Single Responsibility Principle
- The Open-Closed Principle (OCP)
- The Dependency Inversion Principle (DIP)
- Principle of Least Knowledge or Law of Demeter (LoD)
- The Interface Segregation Principle (ISP)

[www.fandsindia.com](http://www.fandsindia.com)



## The Single Responsibility Principle

- *THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.*
- *This means*
  - *Class should have only a single purpose to live and all its methods should work together to help achieve this goal*
- *More than one responsibilities attached to the class make it*
  - *Difficult to understand (needlessly complex)*
  - *Difficult to change (rigid)*
  - *Difficult to reuse*

[www.fandsindia.com](http://www.fandsindia.com)



## The Open Closed Principle

- *A module should be open for extension but closed for modification.*
- Most important principle
- It says that software entities should be designed so that they would not allow for changes to old code later on. New code can be added via extensions provided during design.

[www.fandsindia.com](http://www.fandsindia.com)



## The Dependency Inversion Principle

- *Depend upon Abstractions. Do not depend upon concretions.*
- High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.

[www.fandsindia.com](http://www.fandsindia.com)



## Principle of Least Knowledge / Law of Demeter

- Any object receiving a message in a given method must be one of a restricted set of objects.
- When applied to object-oriented programs, the Law of Demeter can be more precisely called the “Law of Demeter for Functions/Methods” (LoD-F).
- An object A can call a method of an object instance B, but object A cannot “reach through” object B to access yet another object, C, to request its services. Doing so would mean that object A implicitly requires greater knowledge of object B’s internal structure.

[www.fandsindia.com](http://www.fandsindia.com)



## The Interface Segregation Principle

- Clients should not be forced to depend upon interfaces that they do not use.
- Many client specific interfaces are better than one general purpose interface.
- Impact of changes to one interface is relatively smaller.

[www.fandsindia.com](http://www.fandsindia.com)



# Patterns



## Categorization of Patterns

- **Creational Patterns:** Used to create objects instead of direct instantiation
- **Structural Patterns:** Compose objects into larger structures
- **Behavioural Patterns:** Communication between objects



## Creational Pattern

Abstract the Instantiation Process.

Makes a system independent of how its objects are created, composed and represented.

A *Class* creational pattern uses inheritance to vary the class that has been instantiated.

An *Object* creational pattern delegates instantiation to another object.

[www.fandsindia.com](http://www.fandsindia.com)



## Creational Patterns

- **Factory** – depending on the data provided the Factory pattern returns an instance of a particular class
- **Abstract Factory** – a further abstraction of Factory that determines a group of instances to return
- **Singleton** – maintains only a single instance of a class
- **Builder** – a more complex pattern than Factory that returns an entire user interface based on the context of the data
- **Prototype** – clones an existing instance rather than creating new instances of a class

[www.fandsindia.com](http://www.fandsindia.com)





## Structural Pattern

These are concerned with how classes and objects are composed to form larger structures.

Structural *Class* patterns use inheritance to compose interfaces or implementations.

Structural *Object* patterns describe ways to compose objects to realize new functionality.

[www.fandsindia.com](http://www.fandsindia.com)



## Structural Patterns

- **Adapter** - changes the interface of one class to that of another one.
- **Bridge** – keeps the program's interface constant while changing the actual class that is displayed or being used. The interface and the underlying class can be changed separately.
- **Composite** – is a collection of objects which may be either a Composite or just a primitive object
- **Decorator** - a class that surrounds a given class, adds new capabilities to it, and passes all the unchanged methods to the underlying class.
- **Facade** - groups a complex object hierarchy together and provides a simpler interface to the underlying data.
- **Flyweight** - limits the proliferation of small, similar class instances by moving some of the class data outside the class and passing it in during various execution methods.
- **Proxy** - provides a simple place-holder class for a more complex class which is expensive to instantiate.

[www.fandsindia.com](http://www.fandsindia.com)



## Behavioral Patterns

These are concerned with algorithms and the assignment of responsibility between objects. These describe not just patterns of objects or classes but also the patterns of communication between them. Behavioral *Class* Patterns use inheritance to distribute behavior between classes. Behavioral *Object* patterns use object composition.

[www.fandsindia.com](http://www.fandsindia.com)














## Behavioral Patterns

- **Observer** - defines the way a number of classes can be notified of a change,
- **Mediator** - defines how communication between classes can be simplified by using another class to keep all classes from having to know about each other.
- **Chain of Responsibility** - allows an even further decoupling between classes, by passing a request between classes until it is recognized.
- **Template** - provides an abstract definition of an algorithm
- **Interpreter** - defines how to include language elements in a program.
- **Strategy** - encapsulates an algorithm inside a class,
- **Visitor** - adds function to a class
- **State** - provides a memory for a class's instance variables.
- **Command** - provides a simple way to separate execution of a command from the interface environment that produced it
- **Iterator** - formalizes the way to move through a list of data within a class.
- **Memento** - Define an object that encapsulates how a set of objects interact

[www.fandsindia.com](http://www.fandsindia.com)

# Open Source

## Open Source and ....

|                                 |  Free software                                                                                                                                                           |  Open-source software |  Freeware                                                                             |  Public-domain software |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <b>Definition</b>               | "FREE" is a matter of liberty, not price                                                                                                                                                                                                                    | "OPEN" doesn't just mean access to the source code                                                       | "FREE" refers to price, while freedom of the use is restricted by creator                                                                                                | "PUBLIC DOMAIN" belongs to the public as a whole                                                             |
| <b>Ground philosophy</b>        | Social movement                                                                                                                                                                                                                                             | Development methodology                                                                                  | Marketing goals                                                                                                                                                          | Copyright disclamation                                                                                       |
| <b>Ground rules</b>             | Four Freedoms<br><a href="https://www.gnu.org/philosophy/free-sw.html">https://www.gnu.org/philosophy/free-sw.html</a>                                                                                                                                      | Open Software initiative<br><a href="https://opensource.org/osd">https://opensource.org/osd</a>          |                                                                                                                                                                          | Creative Common Organization<br><a href="https://creativecommons.org">https://creativecommons.org</a>        |
| <b>Free of charge</b>           | Not necessary                                                                                                                                                                                                                                               | Not necessary                                                                                            | ✓ YES                                                                                                                                                                    | ✓ YES                                                                                                        |
| <b>Covered by copyright law</b> | ✓ YES                                                                                                                                                                                                                                                       | ✓ YES                                                                                                    | ✓ YES                                                                                                                                                                    | ✗ NO                                                                                                         |
| <b>Examples</b>                 |    |                       |   |                         |



## Why and What

- Why?
  - Ubiquitous
  - Runs key infrastructure
  - Eclipsing proprietary software
  - Polar opposite of intellectual property perspective
- Open Source is two things
  - A licensing model
  - A development model



## Licenses by OSI (Open Source Initiative)

- Types
  - GPL (General Public License)
  - BSD
  - MIT
  - Apache 2.0
  - LGPL (Lesser General public License)

# Types

## Copyleft vs. Permissive

### Copyleft

If you are using a component with this kind of open source license, then you too must make your code open for use by others as well.

### Permissive:

Guarantees the freedom to use, modify, and redistribute, while also permitting proprietary derivative works.

## Five Types of Software Licenses

### Public Domain License

Anyone is free to use and modify the software.

### LGPL

You can link to open source libraries within your own software.

Resulting code can be licensed under any.

### Permissive

Few restrictions or requirements for the distribution or modifications of the software.

### Copyleft

Restrictive - known as reciprocal licenses.

### Proprietary

Most restrictive. Ineligible for copying, modifying or distribution.



RESTRICTIVENESS



|                                             | © Copyright           | ⌋ Copyleft                                                              | 🔑 Permissive                                                                     | CC Creative Commons                                                              |
|---------------------------------------------|-----------------------|-------------------------------------------------------------------------|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| What is a user allowed to do with the code? | What creator dictates | What user wants under certain rules                                     | What user wants with a few restrictions                                          | What user wants without restrictions                                             |
| Clause of the use                           | As creator dictates   | Derivative work must be attributed to creator, open-source and copyleft | Derivative work must be attributed to a creator                                  | Derivative work must be attributed to a creator                                  |
| Source code                                 | As creator dictates   | Must be open                                                            | Don't have to be open                                                            | No specific terms about the distribution of source code                          |
| Is creator liable for bugs?                 | ✓ YES                 | ✓ YES                                                                   | ✗ NO                                                                             | ✗ NO                                                                             |
| Re-licensing                                | As creator dictates   | Derivative work cannot be released as proprietary software              | Derivative work can be released under another license or as proprietary software | Derivative work can be released under another license or as proprietary software |
| Commercial restrictions                     | As creator dictates   | Permitted                                                               | Permitted                                                                        | Permitted                                                                        |



## Open Source and Go

- 87% of Go code in GitHub is licensed under a permissive license such as MIT, Apache, and BSD 3-clause
- Go is a statically-linked programming language
  - you almost always use different packages. Be it part of standard library, libraries developed by the community or by your team and create standalone binary



## OSS Review Toolkit (ORT)

- Analyzer
  - The analyzer is a Software Composition Analysis (SCA) tool that determines the dependencies of software projects
- Advisor
  - The advisor retrieves security advisories from configured services.
- Downloader
  - Taking an ORT result file with an analyzer result as the input (-i), the downloader retrieves the source code
- Scanner
  - Wraps underlying license / copyright scanners with a common API so all supported scanners can be used
- Evaluator
  - The evaluator is used to perform custom license policy checks on scan results. The rules to check against are
- Reporter
  - The reporter generates a wide variety of documents in different formats



## go-licenses

- go-licenses analyzes the dependency tree of a Go package/binary. It can output a report on the libraries used and under what license they can be used. It can also collect all of the license documents, copyright notices and source code into a directory in order to comply with license terms on redistribution.
- This is not an officially supported Google product.



## Go Plugins



## Plugins

- Build loosely coupled modular programs using packages compiled as shared object libraries that can be loaded and bound to dynamically at runtime.
- A Go plugin is a package compiled using the `-buildmode=plugin` build flag to produce a shared object (`.so`) library file.



## Plugin

- A plugin is a Go main package with exported functions and variables that has been built with: `go build -buildmode=plugin`. When a plugin is first opened, the init functions of all packages not already part of the program are called. The main function is not run. A plugin is only initialized once, and cannot be closed.





## Go variables

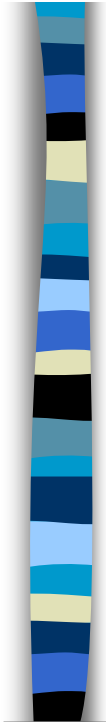
- GOOS
- GOARCH



## QUESTION / ANSWERS



[www.fandsindia.com](http://www.fandsindia.com)



THANKING YOU !



[www.fandsindia.com](http://www.fandsindia.com)