fands™

A Fast AND Steady Approach

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.

Agenda

Go or Golang

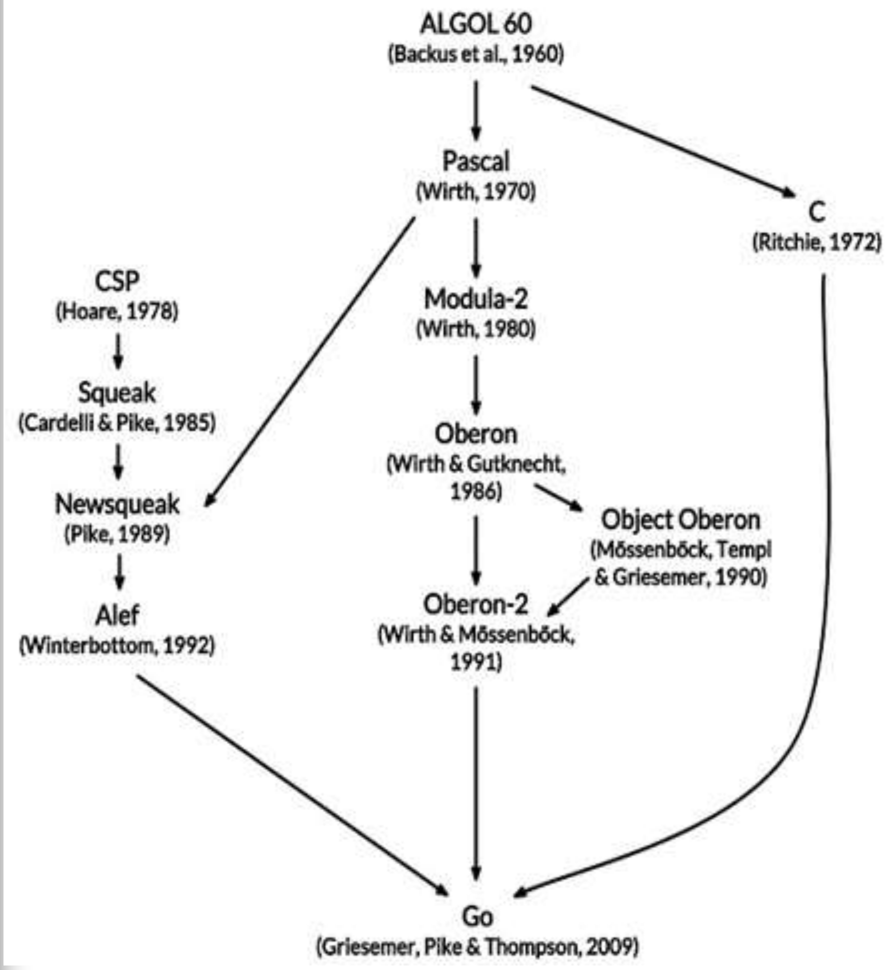
- The language is called Go. The "golang" name arose because the web site is golang.org, not go.org(not available). Many use the golang name, though, and it is handy as a label. For instance, the Twitter tag for the language is "#golang". The language's name is just plain Go, regardless.
- Go is a compiled 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

- go build
  - which builds Go binaries using only information in the source files themselves, no separate makefiles
- go test
  - for unit testing and microbenchmarks
- go fmt
  - for formatting code
- go get
  - for retrieving and installing remote packages
- go vet
  - a static analyzer looking for potential errors in code
- go run
  - a shortcut for building and executing code
- Godoc
  - for displaying documentation or serving it via HTTP
- Gorename
  - for renaming variables, functions, and so on in a type-safe way
- go generate
  - 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.org 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

go get 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 name of current executable.

Lab - Functions

```
func add(x int, 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 sum < 1000 {
- Infinite Loop
  - for {

If

- parentheses ( ) but the braces { } are required.
  - if x < 0
- If with a short statement
  - if v := math.Pow(x, n); v < lim
- If and else

```
if v := math.Pow(x, n); v < 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.")
}
```

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")
```

Closures

- A closure is a special type of anonymous function that references variables declared outside of the function itself. It is similar to accessing global variables which are available before the declaration of the function.

```
package main

import "fmt"


func main(){
  sum := 0
  counter := func(cnt int)
int {
     sum += cnt
     return sum
  }

  fmt.Println(counter(1))
  fmt.Println(counter(2))
}
```

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))}
```

Go Popular Utilities

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 ←→Struct XML ←→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(.)
}}

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.Millisec
ond * 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()

Defer, Panic and Recover

Defer Recap

- 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
```

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)
}
```

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

fands™

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)
        }}
```

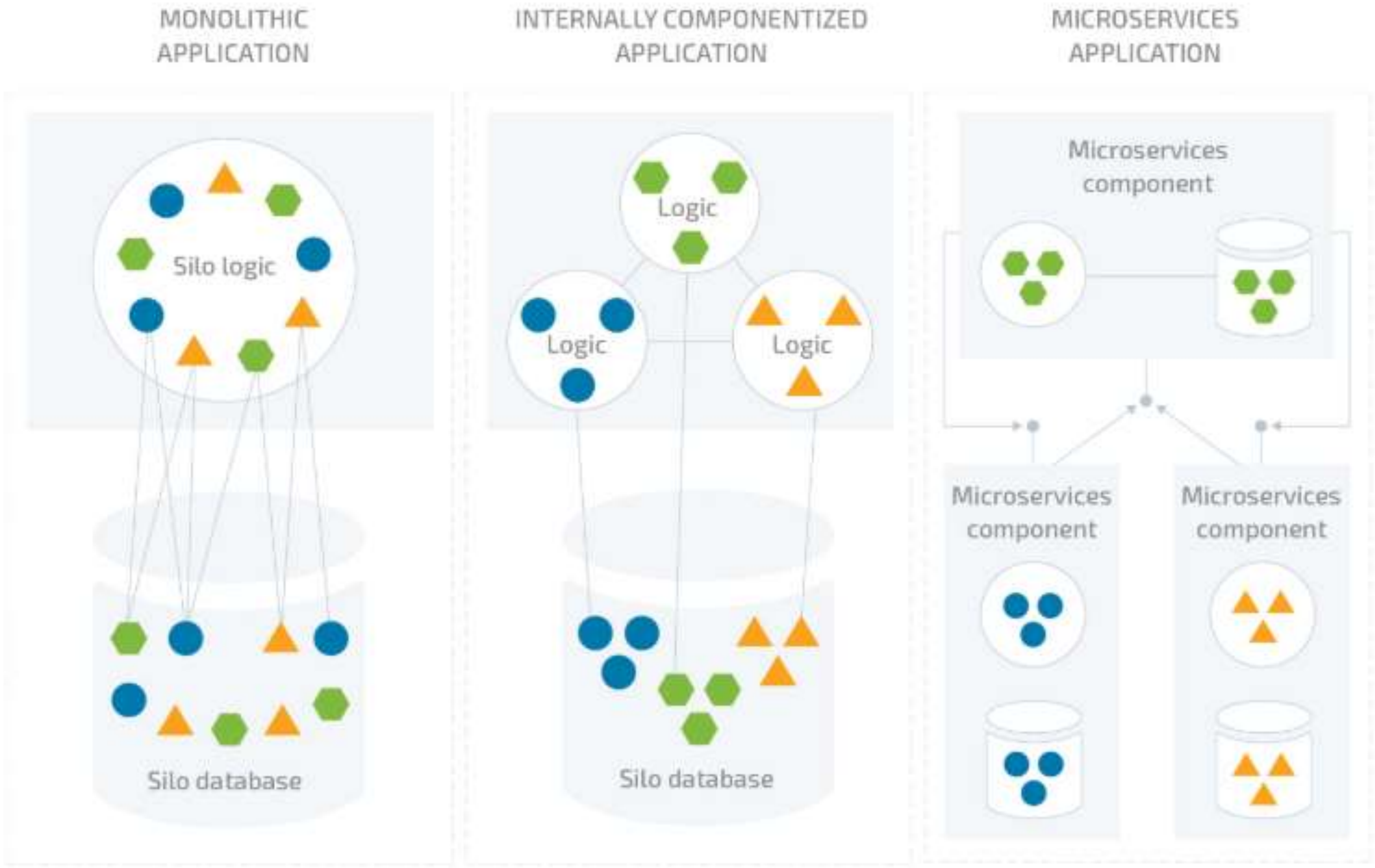
- The Go testing package contains a benchmarking facility that can be used to examine the performance of your Go code.
- Benchmarks are placed inside _test.go files and follow the rules of their Test counterparts except name
- The value of b.N will increase each time until the benchmark runner is satisfied 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

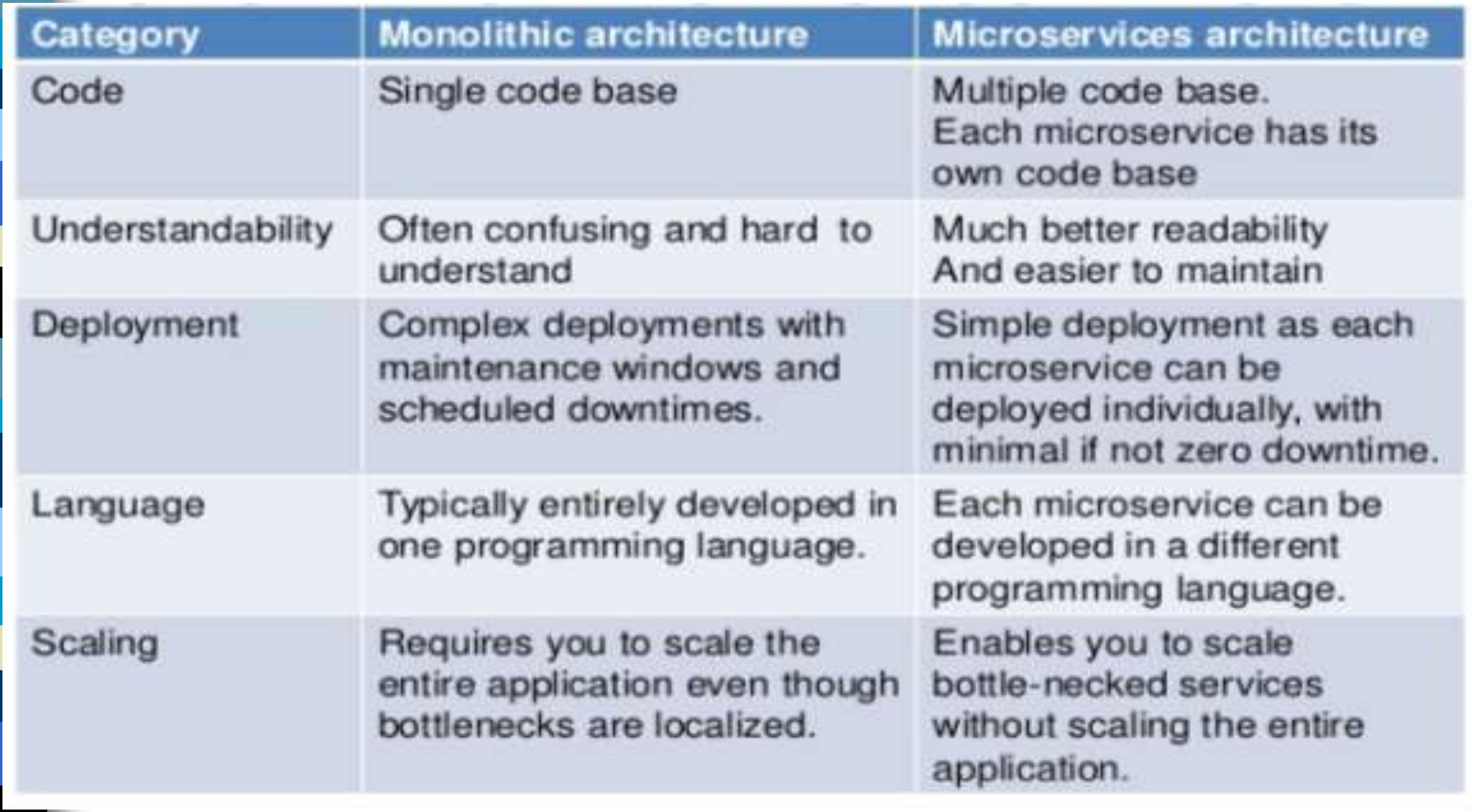
- Why Mocking
- Go mocking
  - Multiple options
    - Testify
    - GoMock
- Testify
  - Testify is more popular and active on github
  - Used by more packages according to goDoc
  - Testify provides better error messages on unexpected calls and calls with unexpected parameter values, including argument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

Micro Services Development

Monolith to Micro Services



Monolith to Micro Services

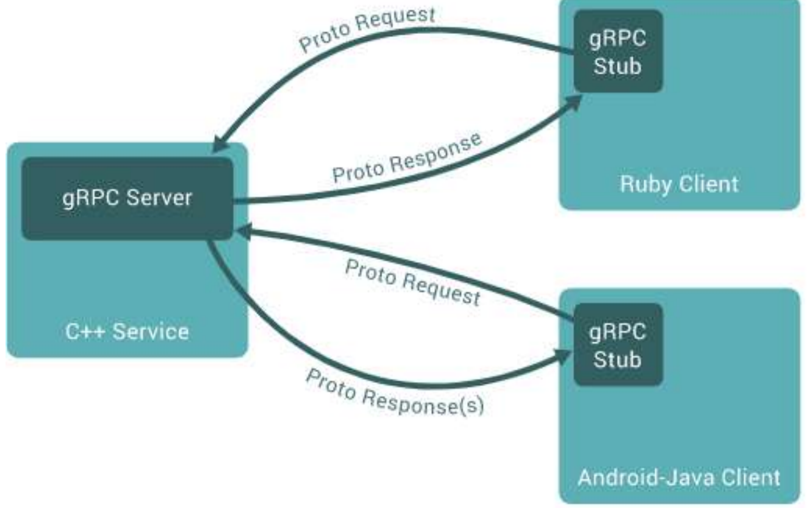


Category	Monolithic architecture	Microservices architecture
Code	Single code base	Multiple code base. Each microservice has its own code base
Understandability	Often confusing and hard to understand	Much better readability And easier to maintain
Deployment	Complex deployments with maintenance windows 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

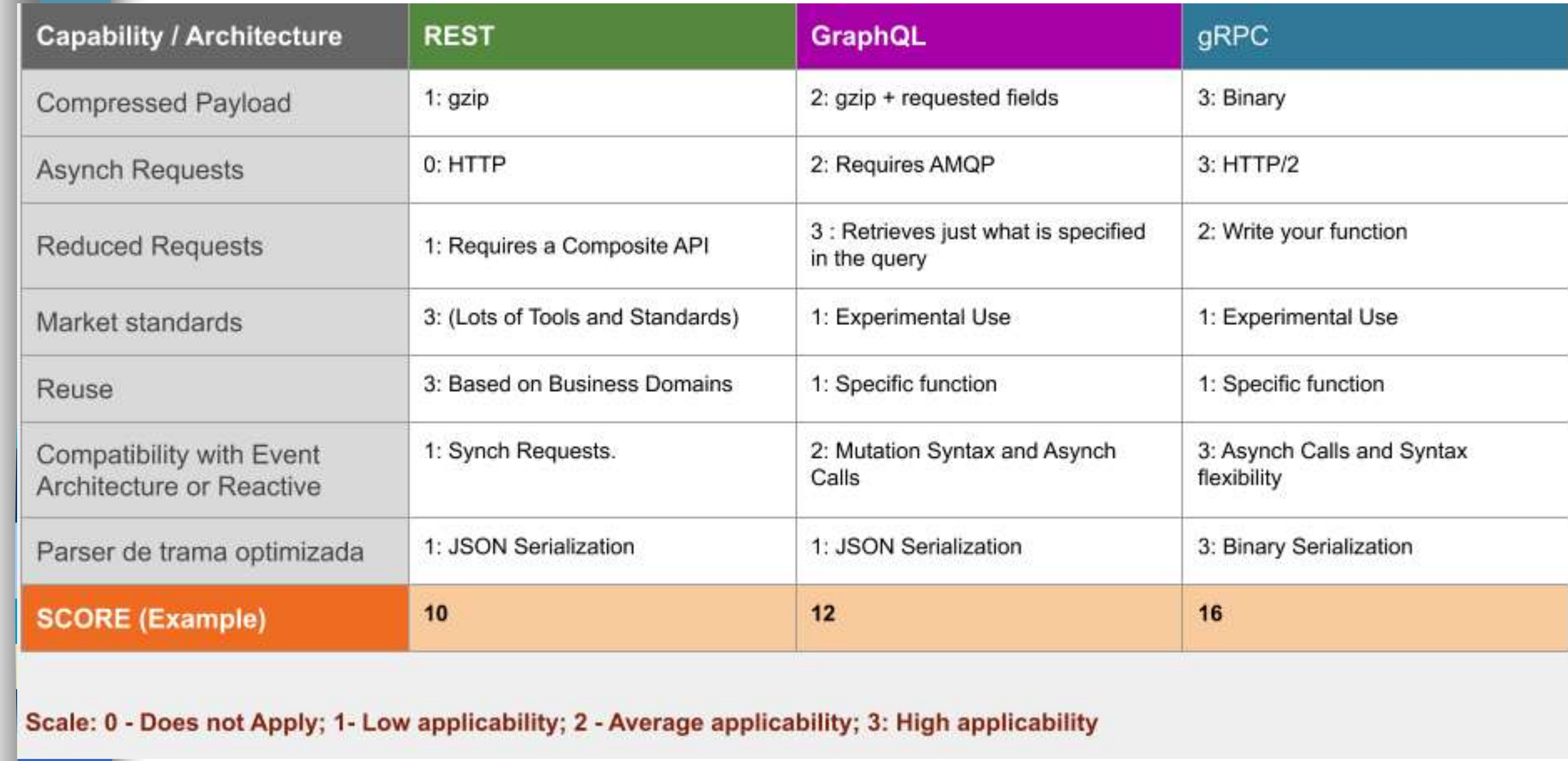
- Format
  - XML, JSON, protobuf/gRPC
  - gRPC is a light-weight binary based RPC communication protocol brought out by Google
  - gRPC uses the new HTTP 2.0 spec
  - Allows for the use of binary data.
  - Allows bi-directional streaming
  - gRPC has an interchange DSL called protobuf.
  - Protobuf allows 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
SCORE (Example)	**10**	**12**	**16**

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

- 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 for insert and retrieve

Introduction to Docker and Kubernetes

Docker is a shipping container system for code

Multiplicity of Stacks

Do services and apps interact appropriately?

Static website

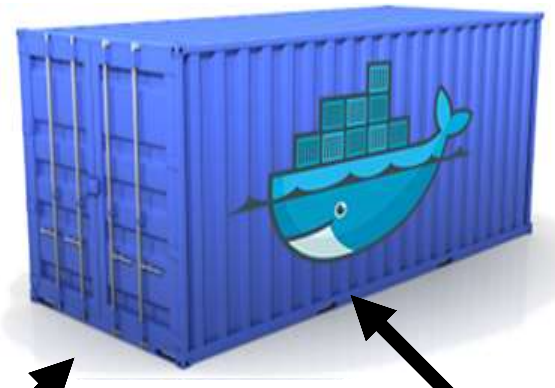
User DB

Web frontend

Queue

Analytics DB

An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…



…that can be manipulated using standard operations and run consistently on virtually any hardware platform

Multiplicity of hardware environments

Can I migrate smoothly and quickly

Development VM

QA server

Customer Data Center

Public Cloud

Production Cluster

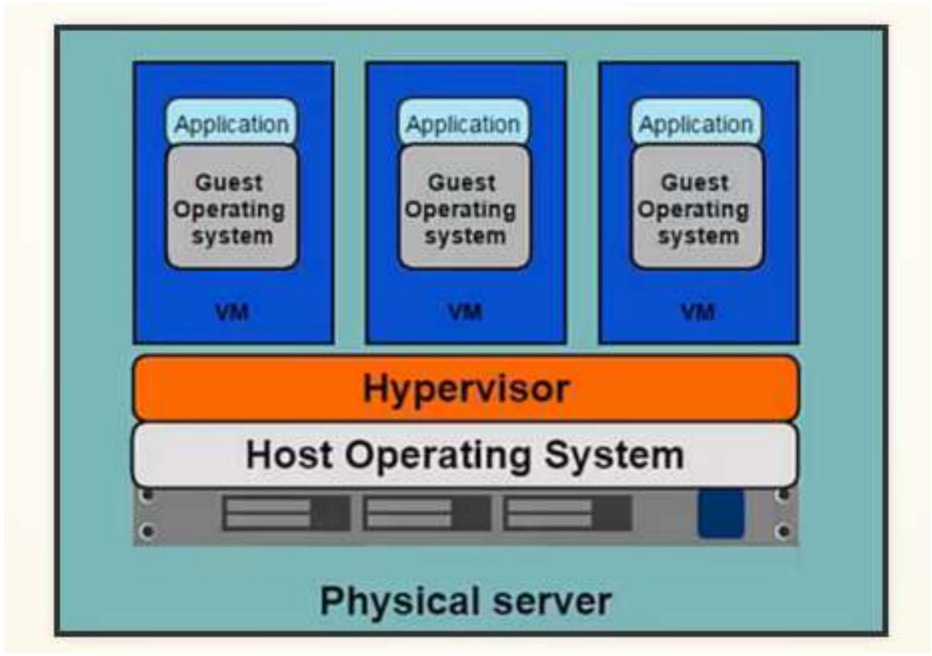
Contributor's laptop

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

Hypervisor-based Virtualization

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

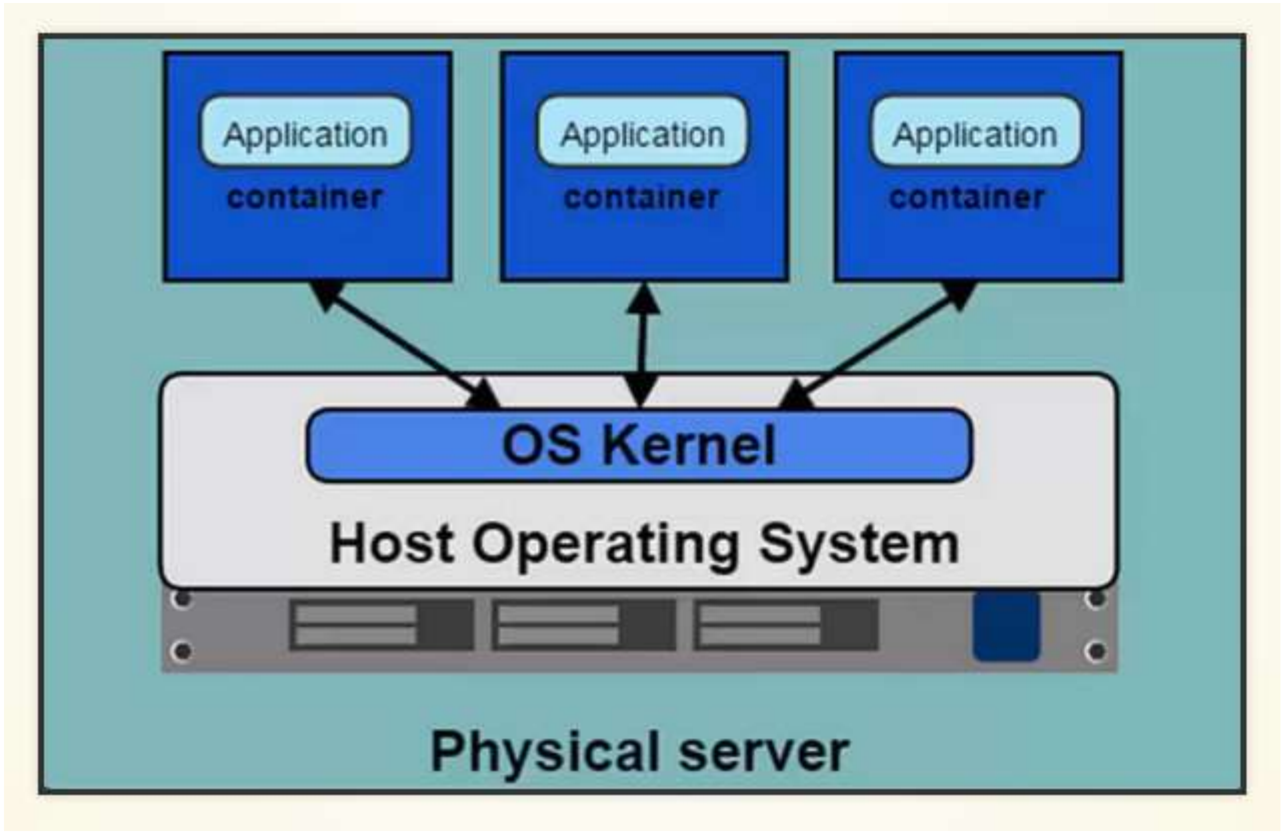


Introducing Containers

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

Containers


Application
container
Application
container
Application
container
OS Kernel
Host Operating System
Physical server

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

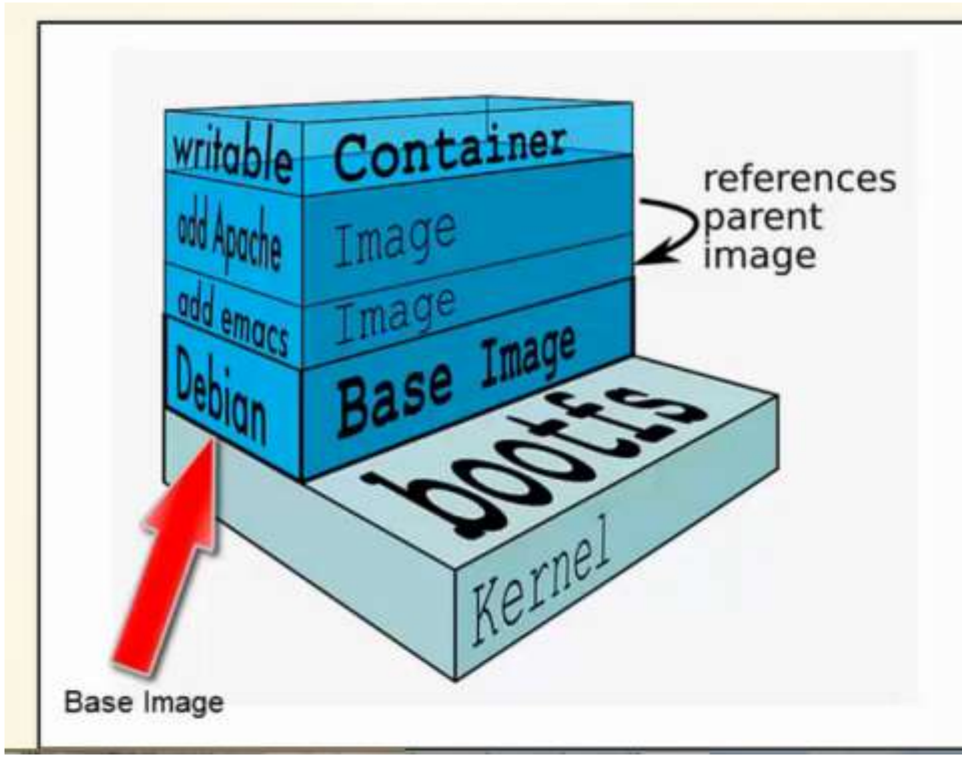
Images and Containers

- Images
  - Read only template used to create containers
  - Built by you or other Docker users
  - Stored in the Docker Hub or your local Registry

- Containers
  - Isolated application platform
  - Contains everything needed to run your application
  - Based on images

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



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
```

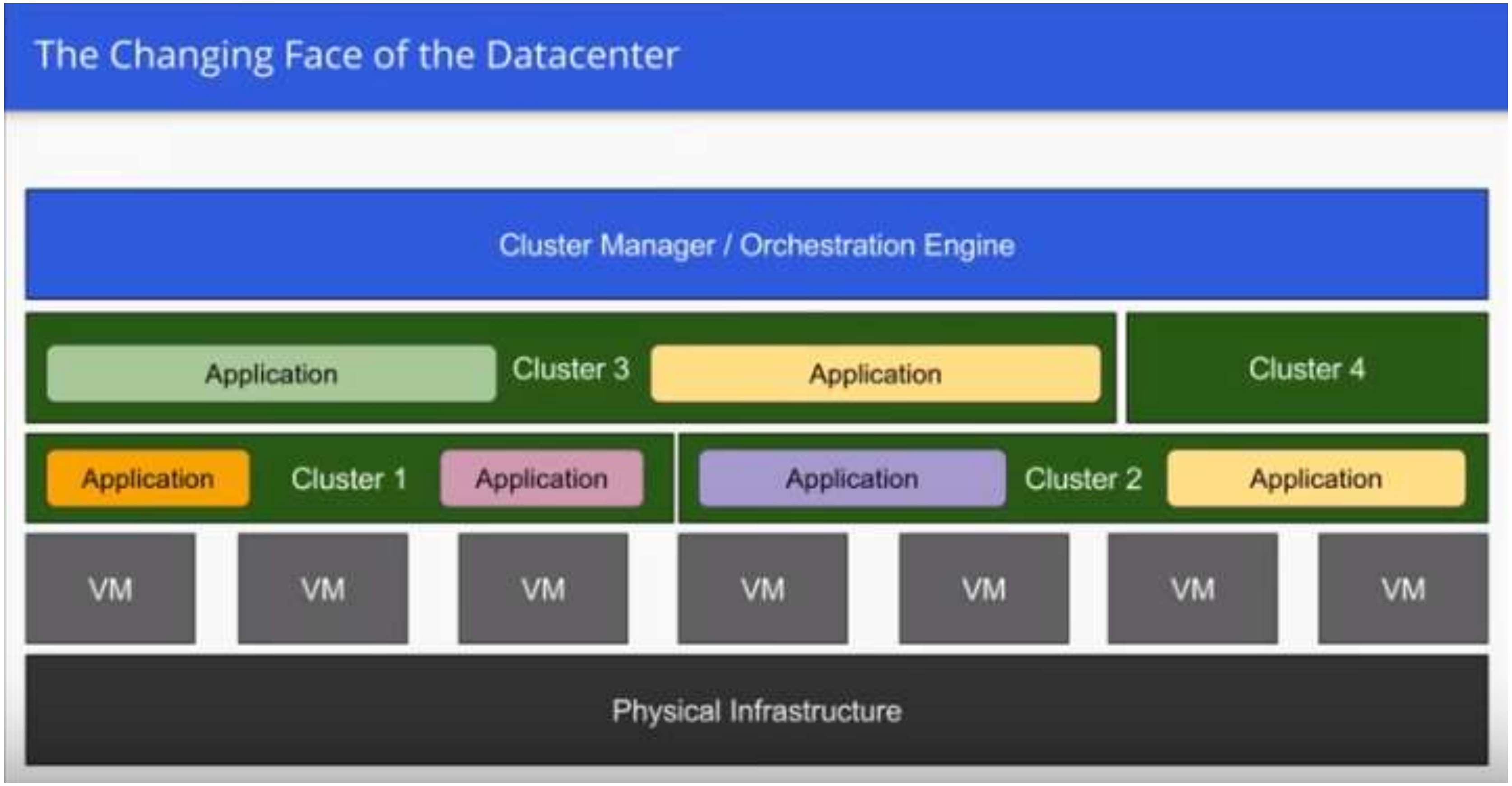
Docker Compose

- 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:
```

fands™

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

Kubernetes

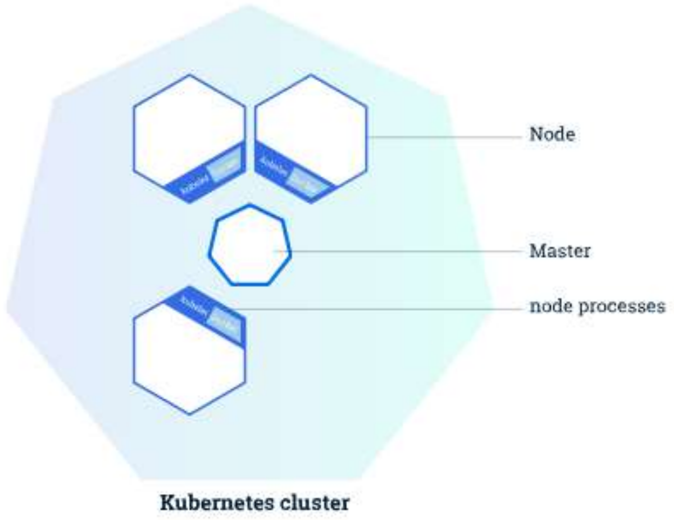
- With modern web services, users expect applications to be available 24/7, and developers expect to deploy new versions of those applications several times a day. Containerization helps package software to serve these goals, enabling applications to be released and updated in an easy and fast way without 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

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

Kubernetes Cluster

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



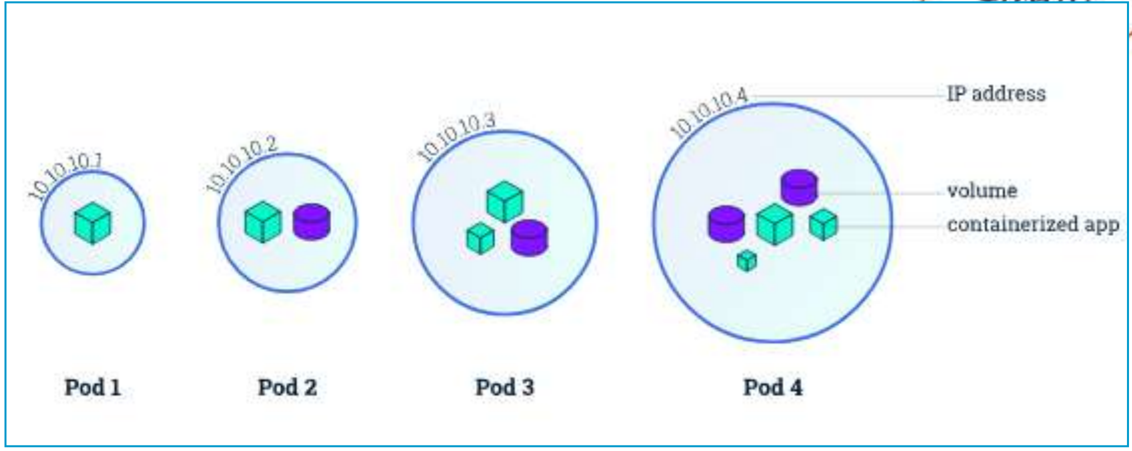
Master and Nodes

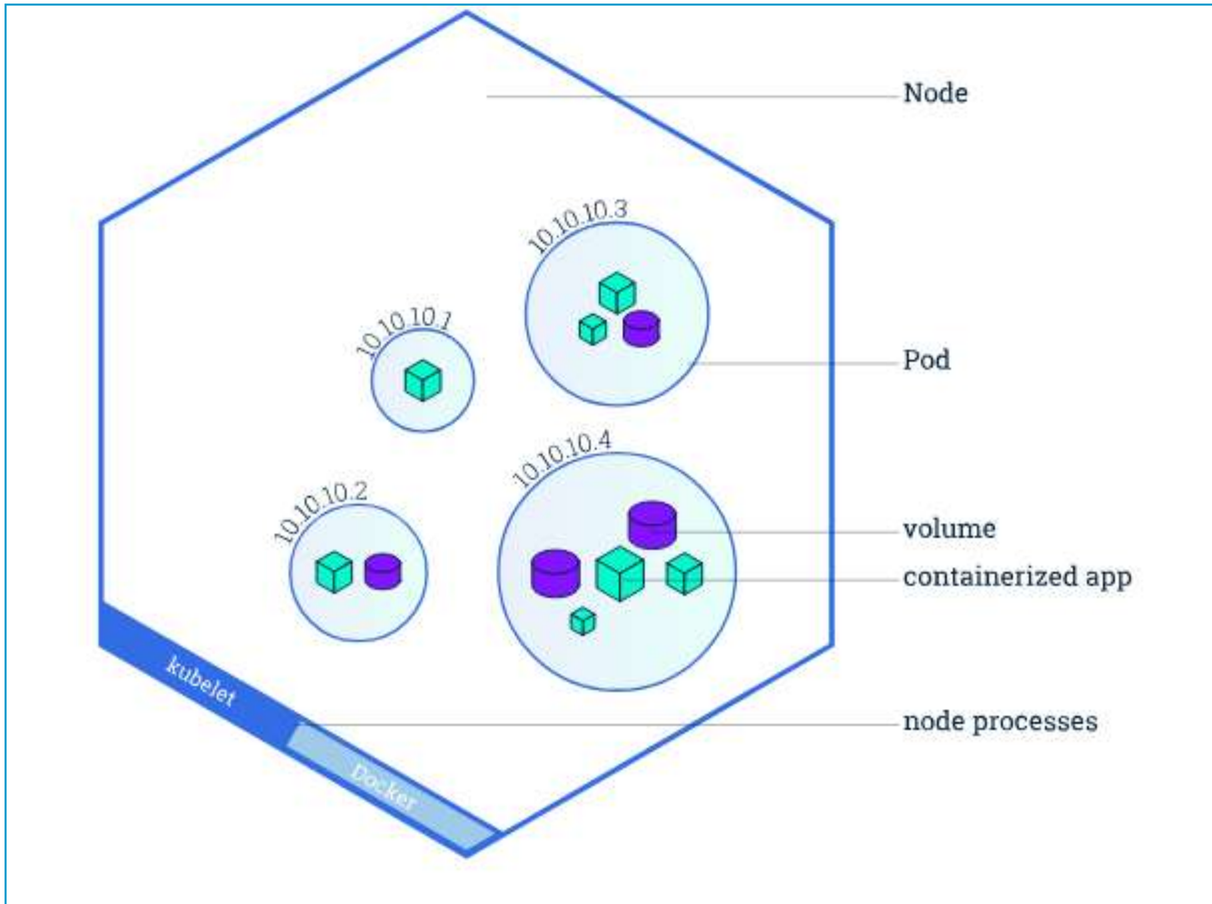
- The Master
  - responsible for managing the cluster.
  - Co-ordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.
- A node
  - is a VM or a physical computer that serves as a worker machine 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

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

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.

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.

Kubernetes Deployments

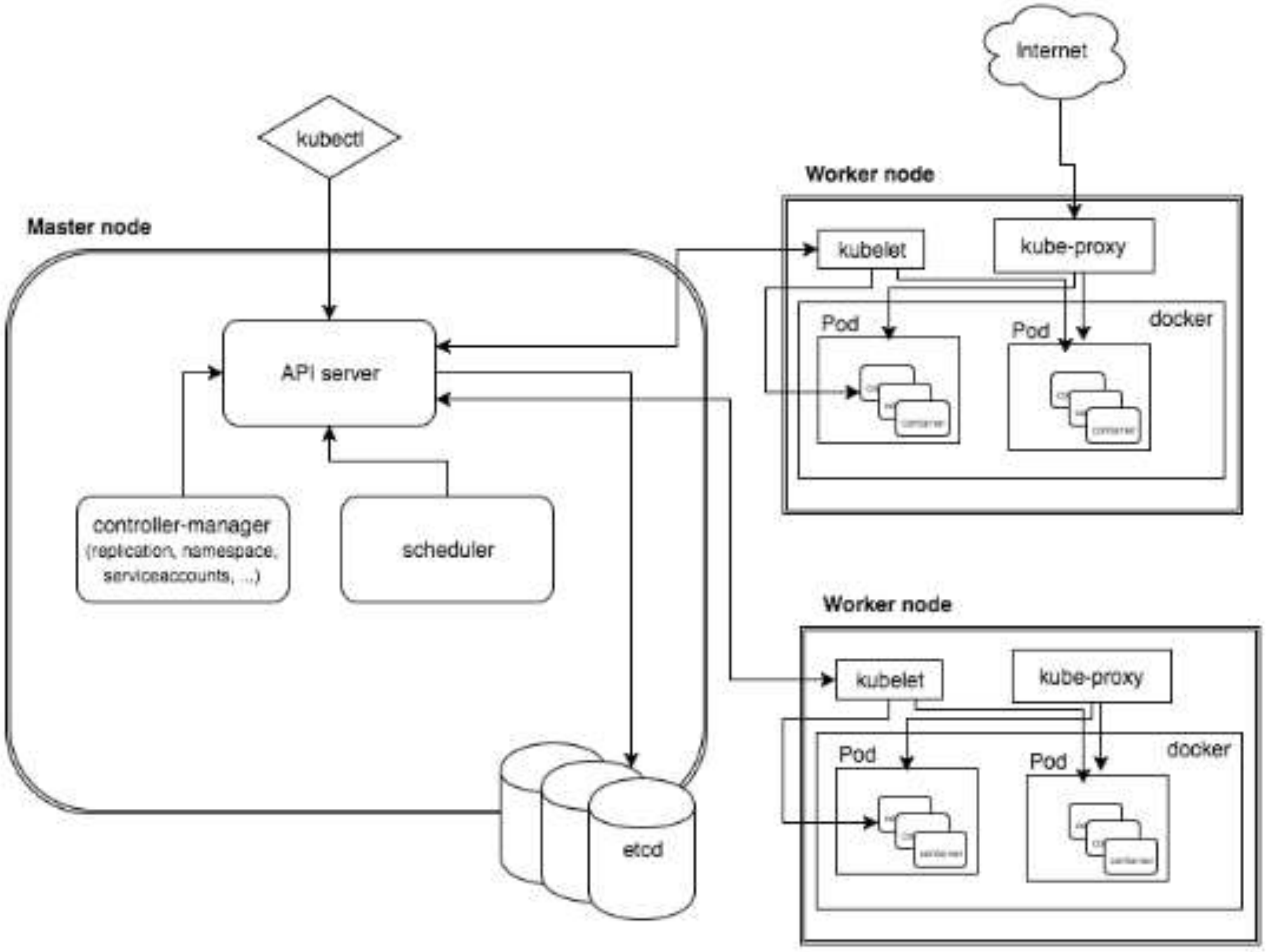
- Once the application instances are created, a Kubernetes Deployment Controller continuously monitors those instances. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces it. **This provides a self-healing mechanism to address machine failure or maintenance.**

fands™

Kubernetes Features

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

High Level Kubernetes Architecture



Main Components

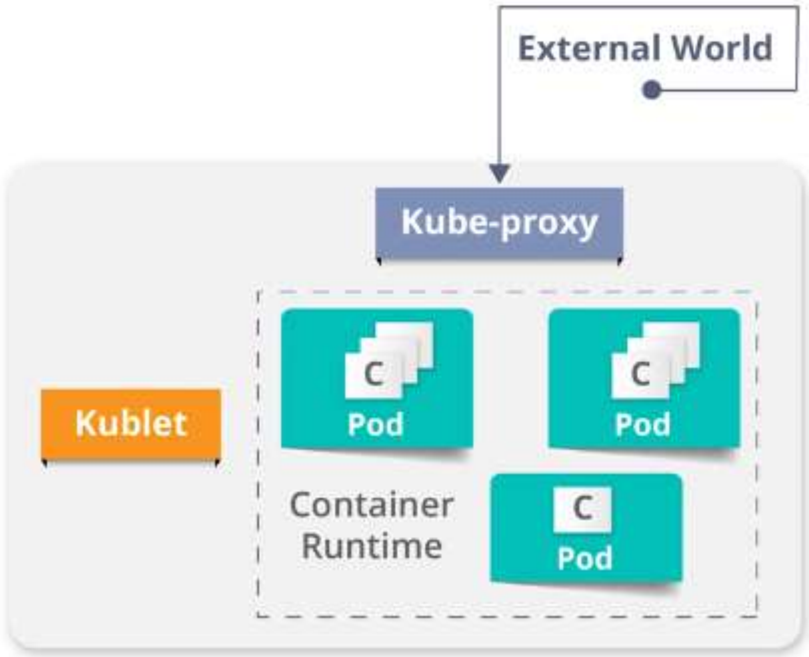
- Master Node(s)
- Worker Nodes
- Distributed Key-Value Store like etcd

Master Node

- The master node
  - is responsible for the management of Kubernetes cluster
  - is the entry point of all administrative tasks.
  - is the one taking care of orchestrating the worker nodes, where the actual services are running.
  - Can be more than one master nodes in the cluster
  - Only one of them will be the leader

Worker Node

- The pods are run here, so the worker node contains all the necessary services to manage the networking between the containers, communicate with the master node, and assign resources to the containers scheduled.



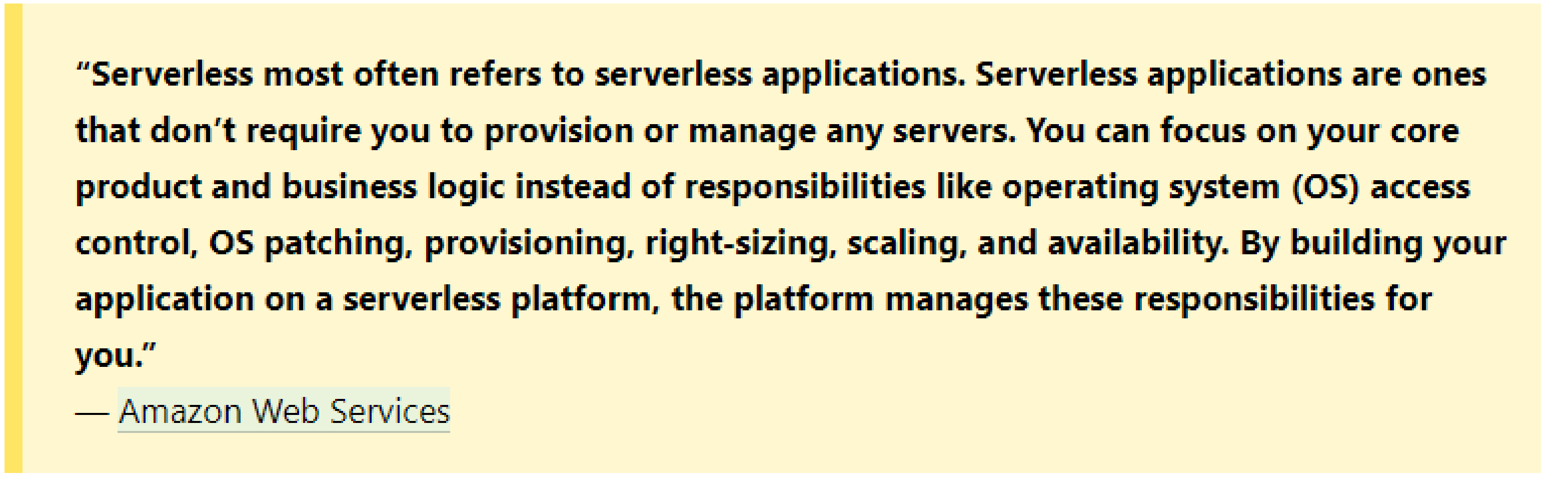
etcd storage

- etcd is a simple, distributed, consistent key-value store. It’s mainly used for shared configuration and service discovery.
- It provides a REST API for CRUD operations as well as an interface to register watchers on specific nodes, which enables a reliable way to notify the rest of the cluster about configuration changes.
- An example of data stored by Kubernetes in etcd is jobs being scheduled, created and deployed, pod/service details and state, namespaces and replication information, etc.

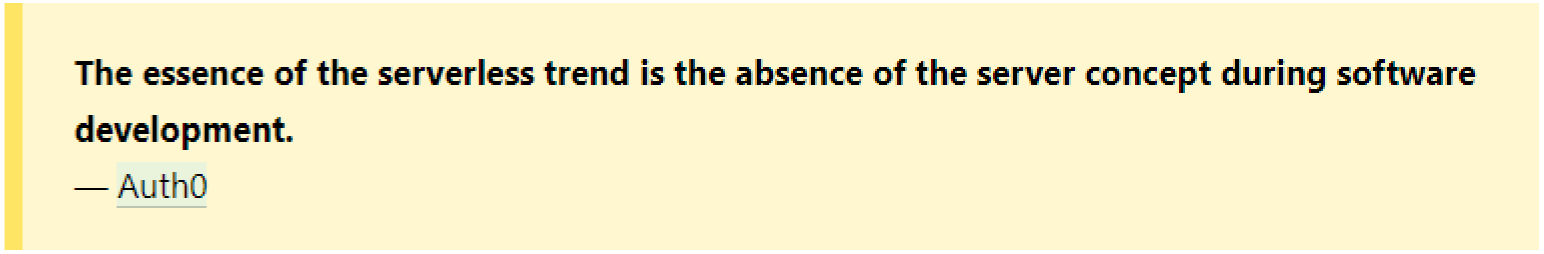
Cloud API Calls

https://docs.aws.amazon.com/sdk-for-go/v1/developer-guide/s3-example-basic-bucket-operations.html

Serverless



> “Serverless most often refers to serverless applications. Serverless applications are ones that don’t require you to provision or manage any servers. You can focus on your core product and business logic instead of responsibilities like operating system (OS) access control, OS patching, provisioning, right-sizing, scaling, and availability. By building your application on a serverless platform, the platform manages these responsibilities for you.”
> — Amazon Web Services



> The essence of the serverless trend is the absence of the server concept during software development.
> — Auth0

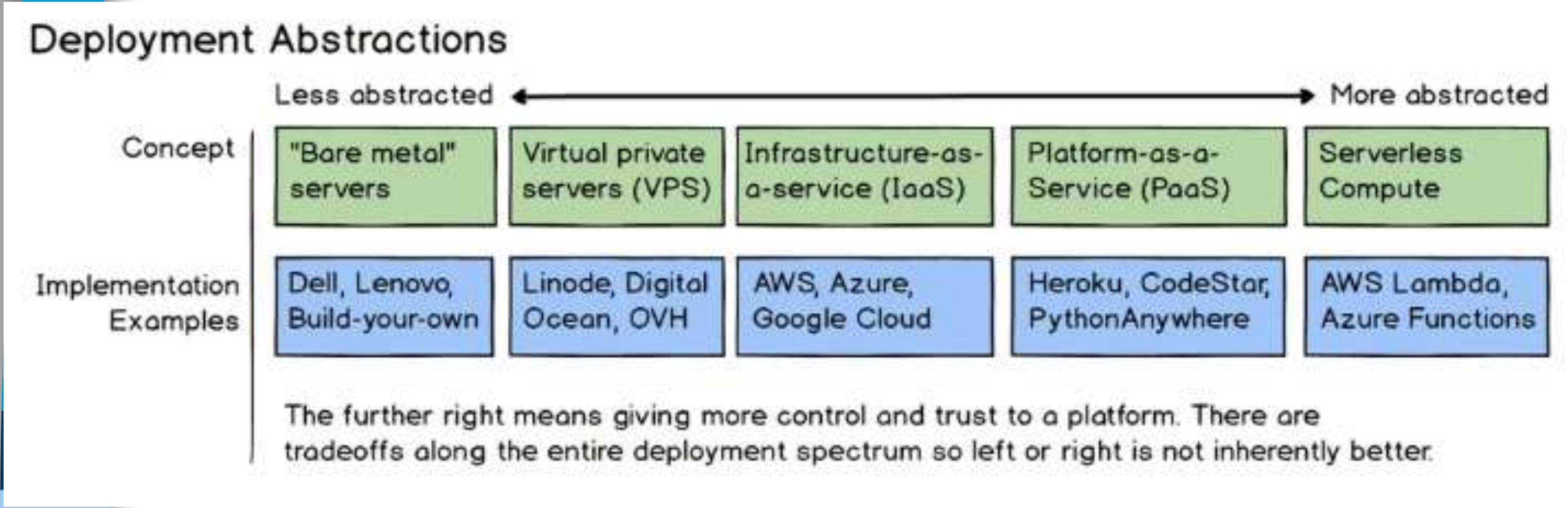
Serverless Architecture

- Serverless Architecture (also known as serverless computing or function as a service, FaaS) is a software design pattern where applications are hosted by a third-party service, eliminating the need for server software and hardware management by the developer.

Serverless Architecture

- Serverless architectures are application designs that incorporate third-party “Backend as a Service” (BaaS) services, and/or that include custom code run in managed, ephemeral containers on a “Functions as a Service” (FaaS) platform. By using these ideas, and related ones like single-page applications, such architectures remove much of the need for a traditional always-on server component. Serverless architectures may benefit from significantly reduced operational cost, complexity, and engineering lead time, at a cost of increased reliance on vendor dependencies and comparatively immature supporting services.

Deployment



Deployment Abstractions

Less abstracted ⟷ More abstracted

Concept	"Bare metal" servers	Virtual private servers (VPS)	Infrastructure-as-a-service (IaaS)	Platform-as-a-Service (PaaS)	Serverless Compute
Implementation Examples	Dell, Lenovo, Build-your-own	Linode, Digital Ocean, OVH	AWS, Azure, Google Cloud	Heroku, CodeStar, PythonAnywhere	AWS Lambda, Azure Functions

The further right means giving more control and trust to a platform. There are tradeoffs along the entire deployment spectrum so left or right is not inherently better.

Main Pillars of Serverless

- No server management
  - You don’t know how many and how they are configured
- Flexible scaling
  - If you need more resources, they will be allocated for you
- High availability
  - Redundancy and fault tolerance are built in
- Never pay for idle
  - Unused resources cost $0

Lambda Lab

- set GOOS=linux
- set GOARCH=amd64
- set CGO_ENABLED=0
- set GOPATH=c:\siemensGo\...
- go get
- go build -o main
- go.exe get -u github.com/aws/aws-lambda-go/cmd/build-lambda-zip
- %USERPROFILE%\Go\bin\build-lambda-zip.exe -o main.zip main

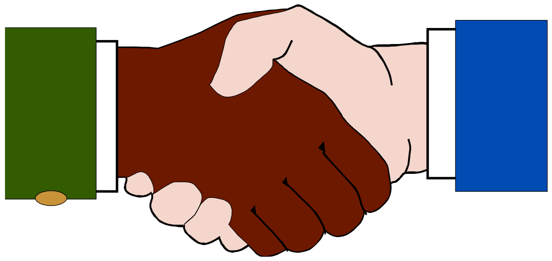
fands™

QUESTION / ANSWERS

www.fandsindia.com

fands™

THANKING YOU !



www.fandsindia.com