Golang

Parameswari Ettiappan

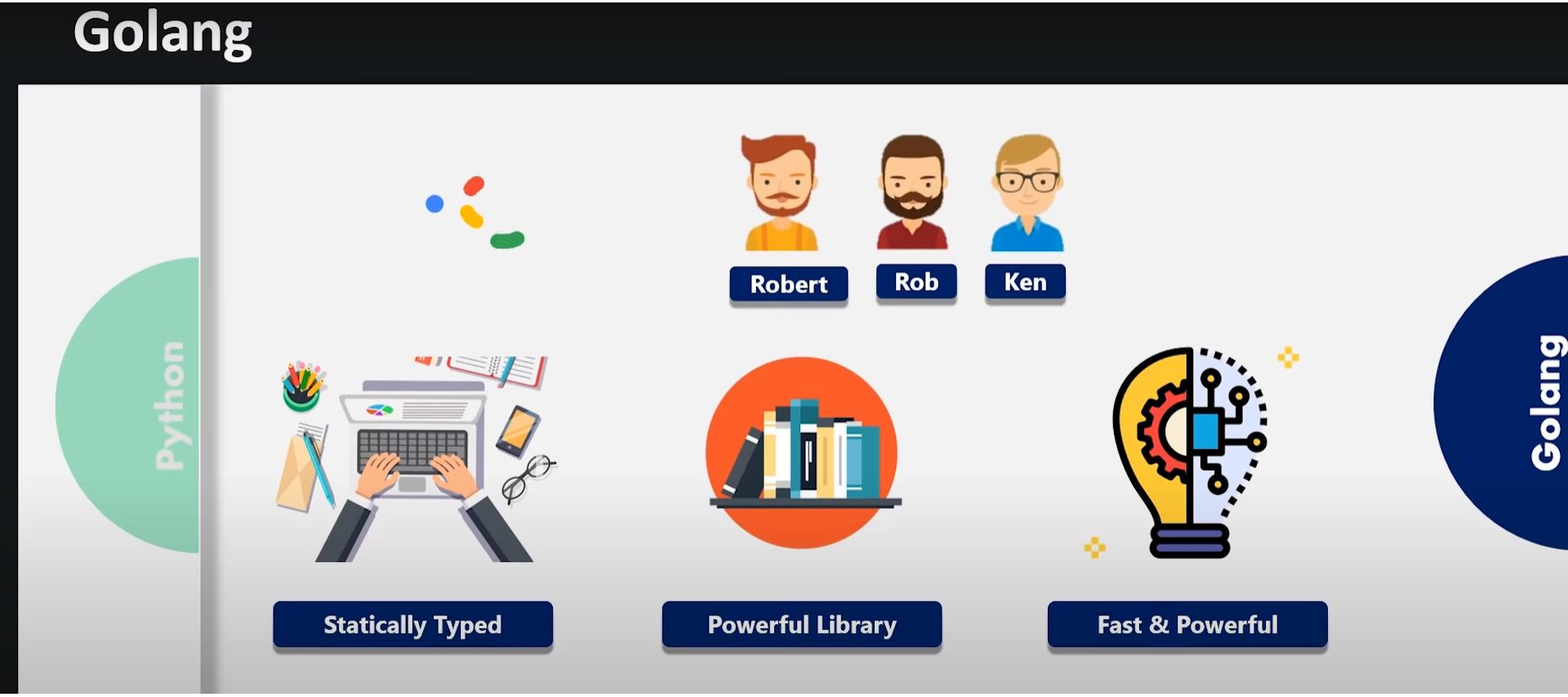
Goals

- Capture core Go language Concepts lab setup, Data Types and Kick Start Coding
- Understanding Functions, Recursion Error Handling and Recover
- Capturing Methods and Interfaces to design modules More on Bit Vector,Port interface and Http Handling
- Designing concurrent applications like Chat Server
- Go Tool Introduction

Software Requirements

- Ubuntu or Windows 10 or Mac
- Golang zip/tar
- Goland / Nano / Sublime/Notepad++
- MySQL 5 or above

Golang


Golang
Python
Robert
Rob
Ken
Statically Typed
Powerful Library
Fast & Powerful
Golang

Introduction

- Go is a programming language that was born out of frustration at Google.
- Developers continually had to pick a language that executed efficiently but took a long time to compile, or to pick a language that was easy to program but ran inefficiently in production.
- Go was designed to have all three available at the same time: fast compilation, ease of programming, and efficient execution in production.

Introduction

- While Go is a versatile programming language that can be used for many different programming projects, it's particularly well suited for networking/distributed systems programs.
- It has earned a reputation as "the language of the cloud".
- It focuses on helping the modern programmer do more with a strong set of tooling and making deployment easy by compiling to a single binary.
- Go is easy to learn, with a very small set of keywords, which makes it a great choice for beginners and experienced developers alike.

Why Go

- Go is an open-source but backed up by a large corporation
- Automatic memory management (garbage collection)
- Strong focus on support for concurrency
- Fast compilation and execution
- Statically type, but feels like dynamically typed
- Good cross-compiling (cross-platform) support
- Go compiles to native machine code
- Rapid development and growing community (Docker/Kubernetes)

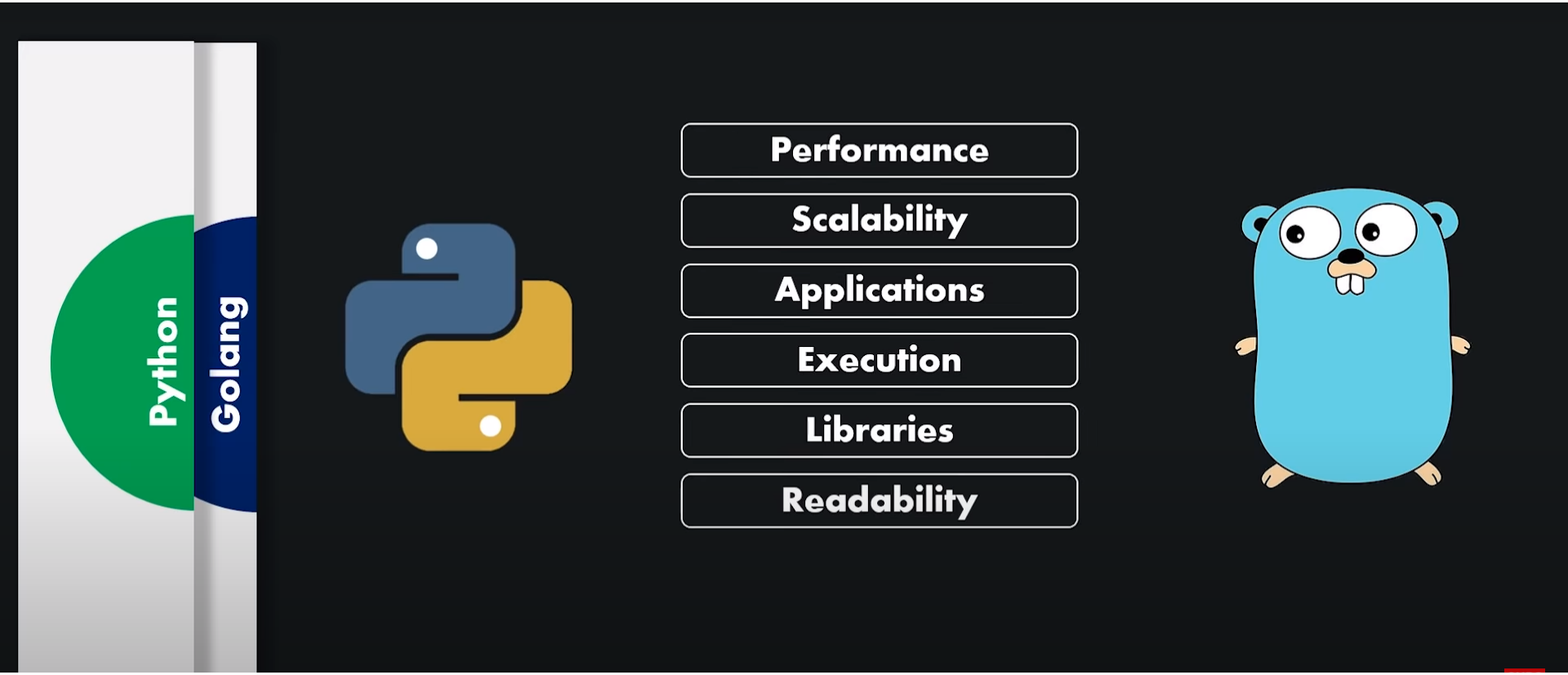
Does Golang have a future?

- It has a very bright future. In the 6 short years since its birth, Go has skyrocketed to the Top 20 of all language ranking indices:
- Go is an absolutely minimalist language with only a handful of programming concept
- Go has superb built-in support for concurrency

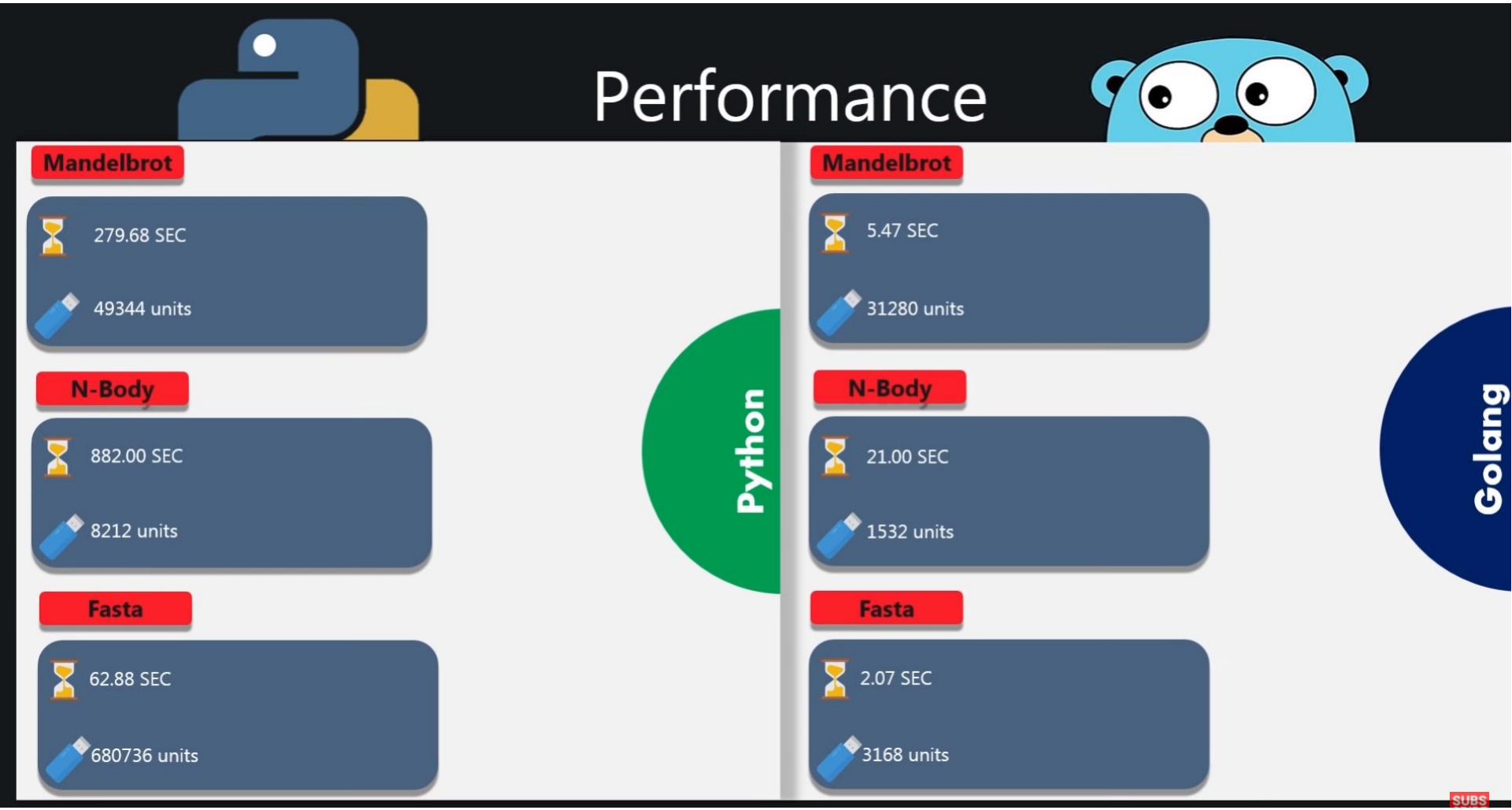
Is Golang better than Python and C++?

- For the readability of code, Golang definitely has the upper hand in most cases and trumps Python as a programming language.
- Go is much easier to learn and code in than C++ because it is simpler and more compact. Go is significantly faster to compile over C++.

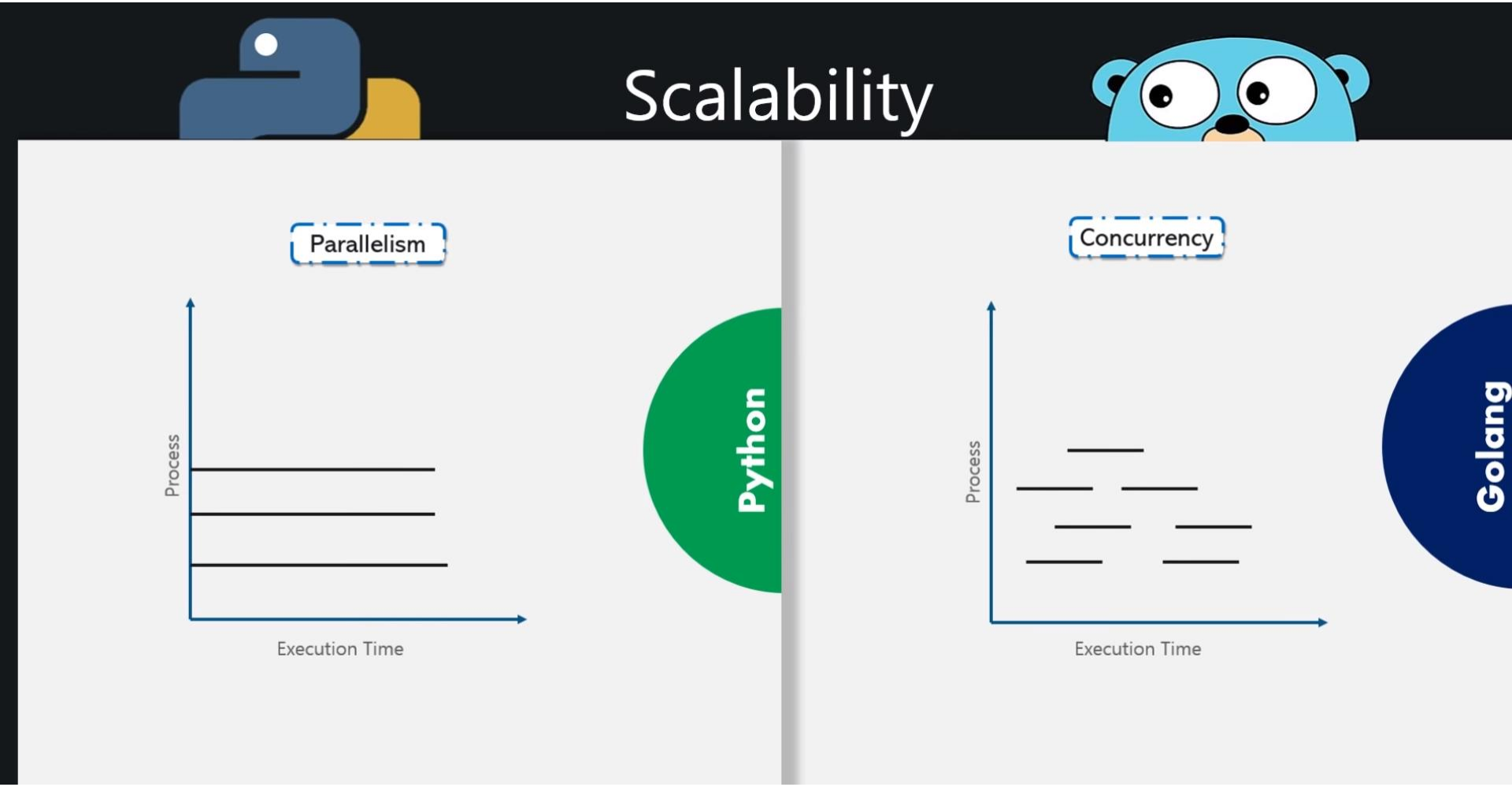
Golang vs Python



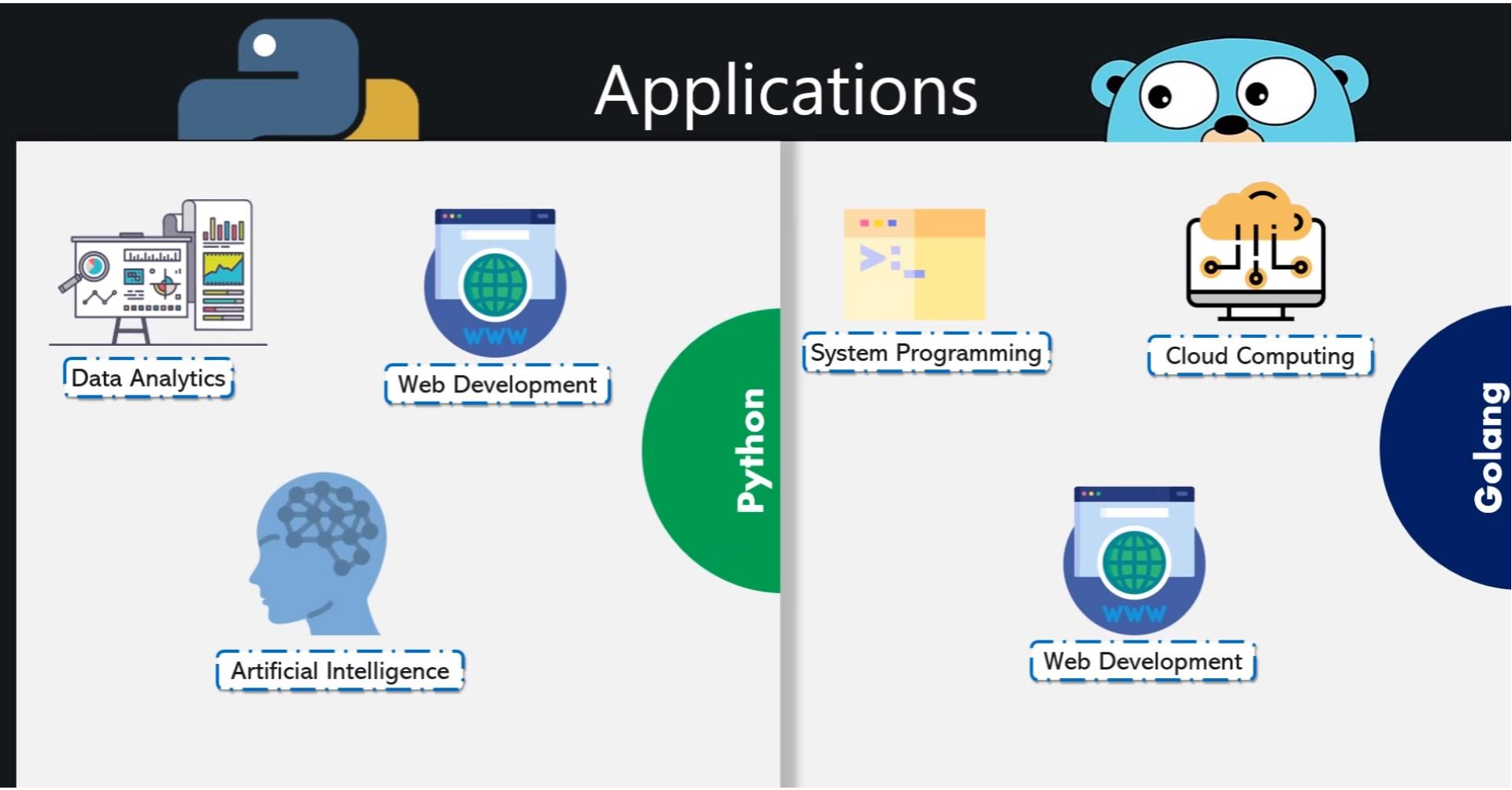
Golang vs Python



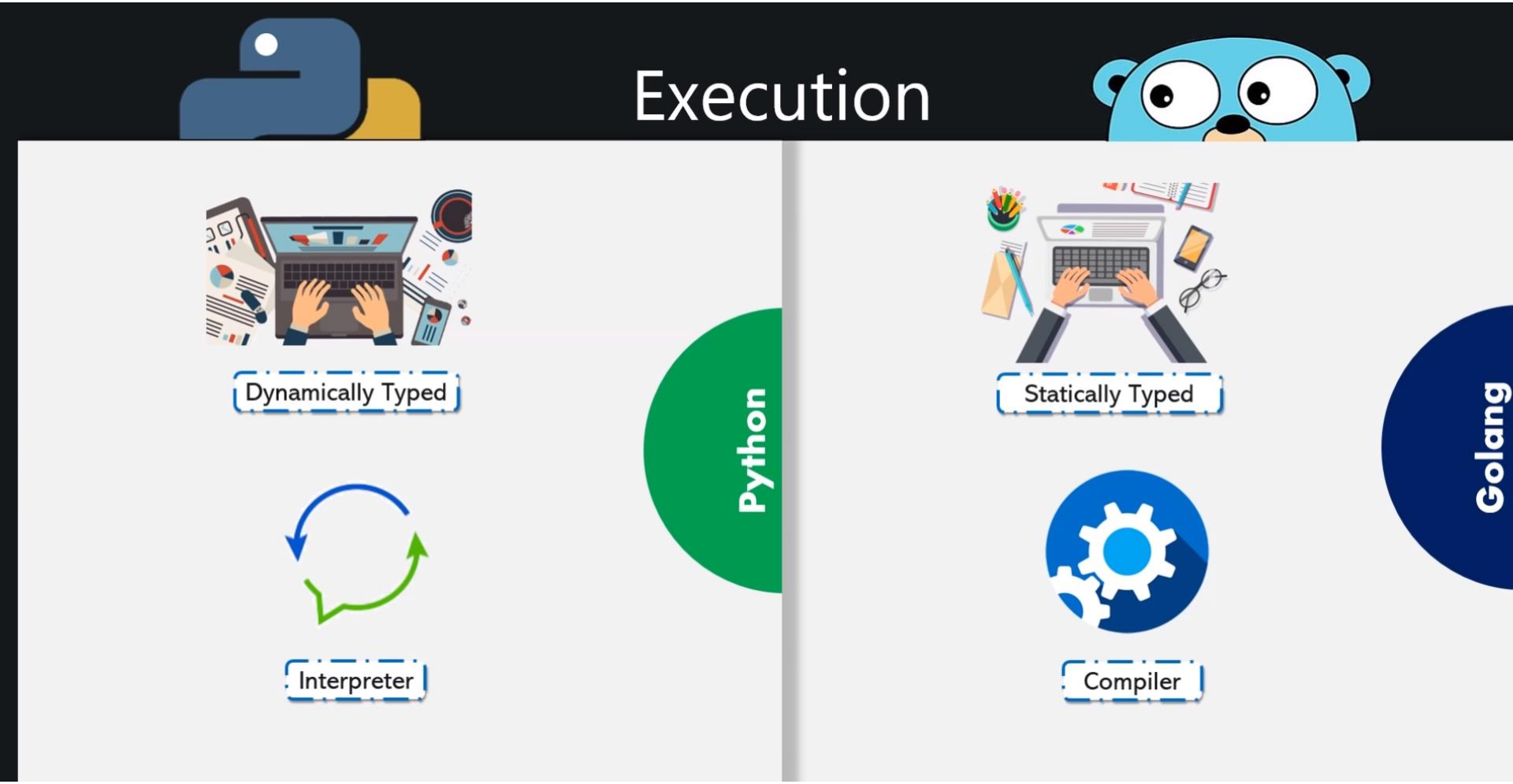
Golang vs Python



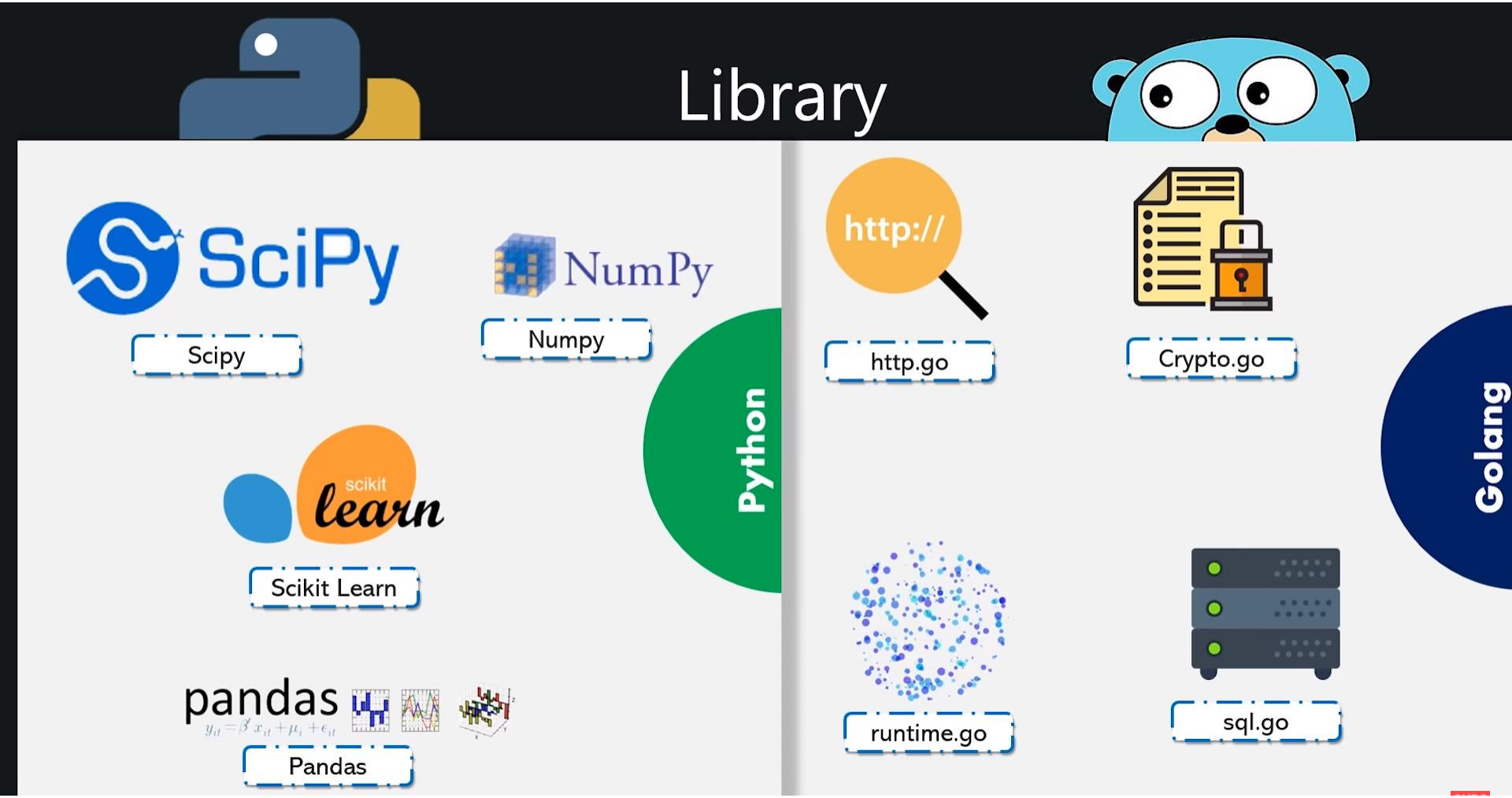
Golang vs Python



Golang vs Python



Golang vs Python



Golang vs Python

Readability

```
print("hello world")
```

- **Excellent Readability**
- **There are multiple ways to write the same thing, which may lead to confusion when someone else reads your code**

Python

```
package main

import "fmt"

func main () {

  fmt.Println("Hello World")

}
```

Golang

- **Excellent Readability**
- **Go is strict about how you write your code which means once you write something it is understood by everone**

What is Clean Architecture?

- Independent of Frameworks. The architecture does not depend on the existence of some library of feature laden software.
- This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
- Testable. The business rules can be tested without the UI, Database, Web Server, or any other external element.

What is Clean Architecture?

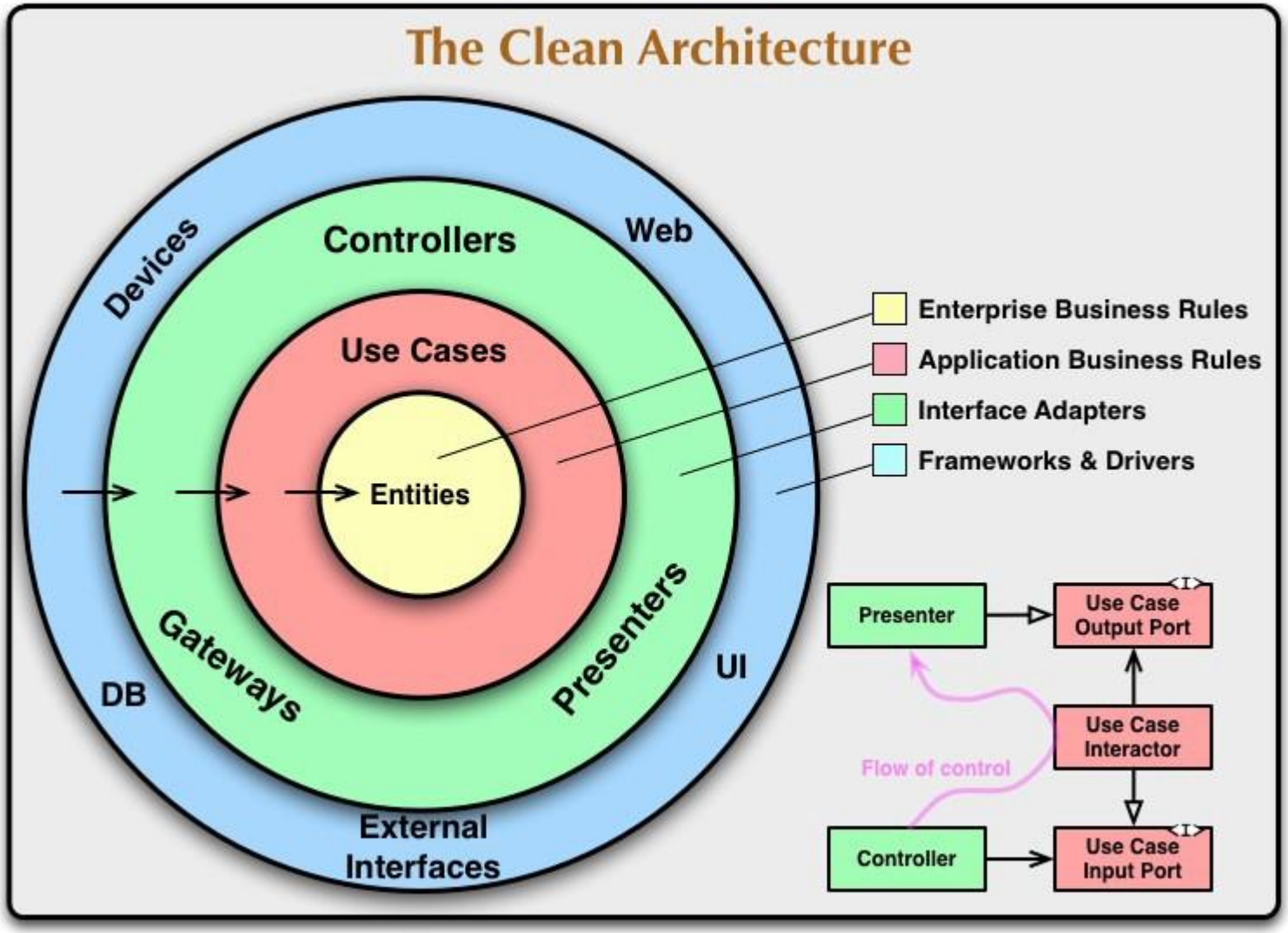
- Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
- Independent of Database. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
- Independent of any external agency. In fact your business rules simply don’t know anything at all about the outside world.

What is Clean Architecture?

- From Uncle Bob’s Architecture we can divide our code in 4 layers :
- Entities: encapsulate enterprise wide business rules. An entity in Go is a set of data structures and functions.
- Use Cases: the software in this layer contains application specific business rules. It encapsulates and implements all of the use cases of the system.

What is Clean Architecture?

- Controller: the software in this layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the Database or the Web
- Framework & Driver: this layer is generally composed of frameworks and tools such as the Database, the Web Framework, etc.


The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port

Golang Architecture

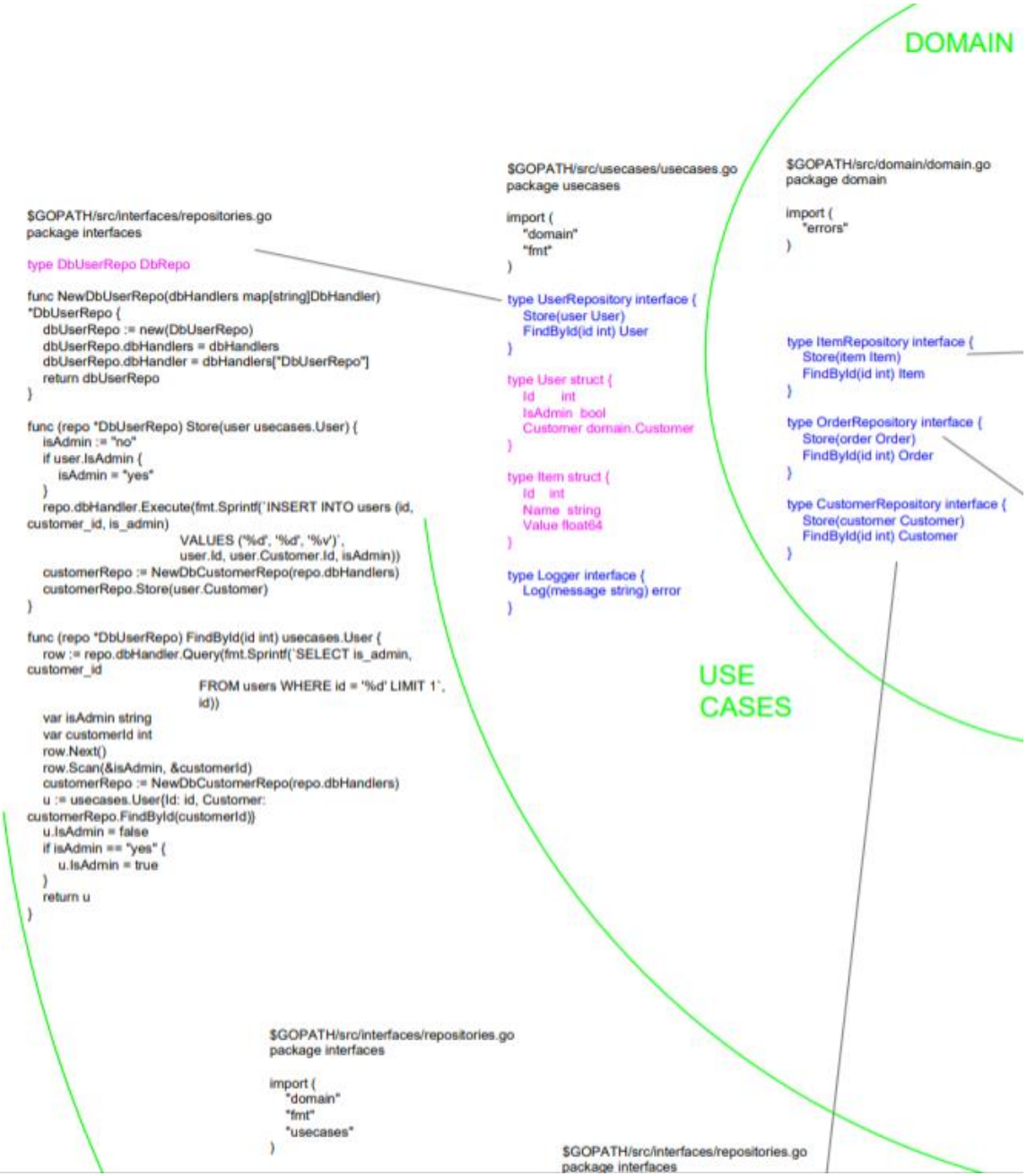
- The top directory contains three directories:
  - app: application package root directory
  - cmd: main package directory
  - vendor: several vendor packages directory

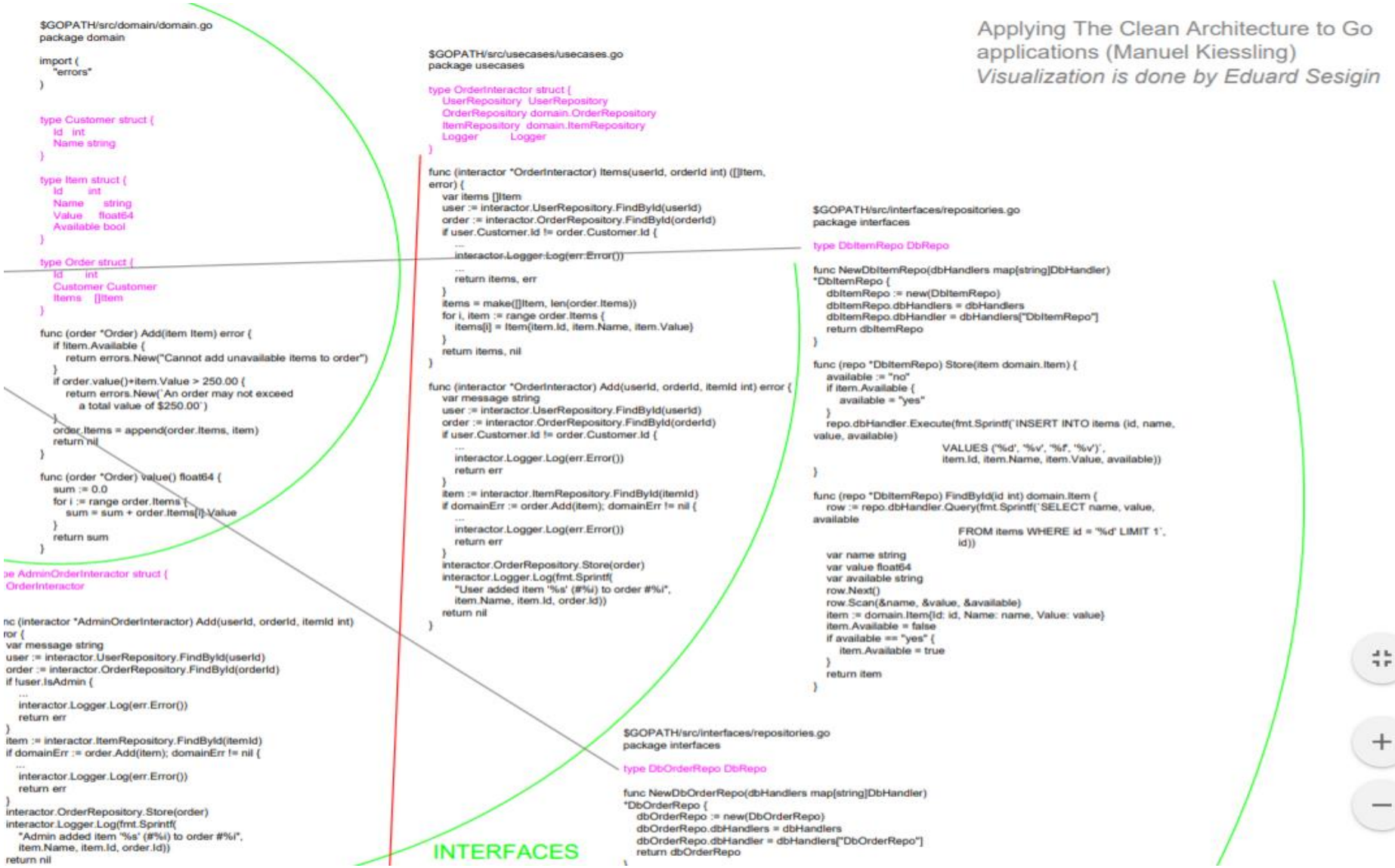
```
% tree
.
├── Makefile
├── README.md
├── app
│   ├── domain
│   │   ├── model
│   │   ├── repository
│   │   └── service
│   ├── interface
│   │   ├── persistence
│   │   └── rpc
│   ├── registry
│   └── usecase
├── cmd
│   └── 8am
│       └── main.go
└── vendor
    ├── vendor packages
    |...
```

Golang Architecture

- There are 4 layers, blue, green, red and yellow layers there in order from the outside. App directory has 3 layers except blue:
  - interface: the green layer
  - usecase: the red layer
  - domain: the yellow layer

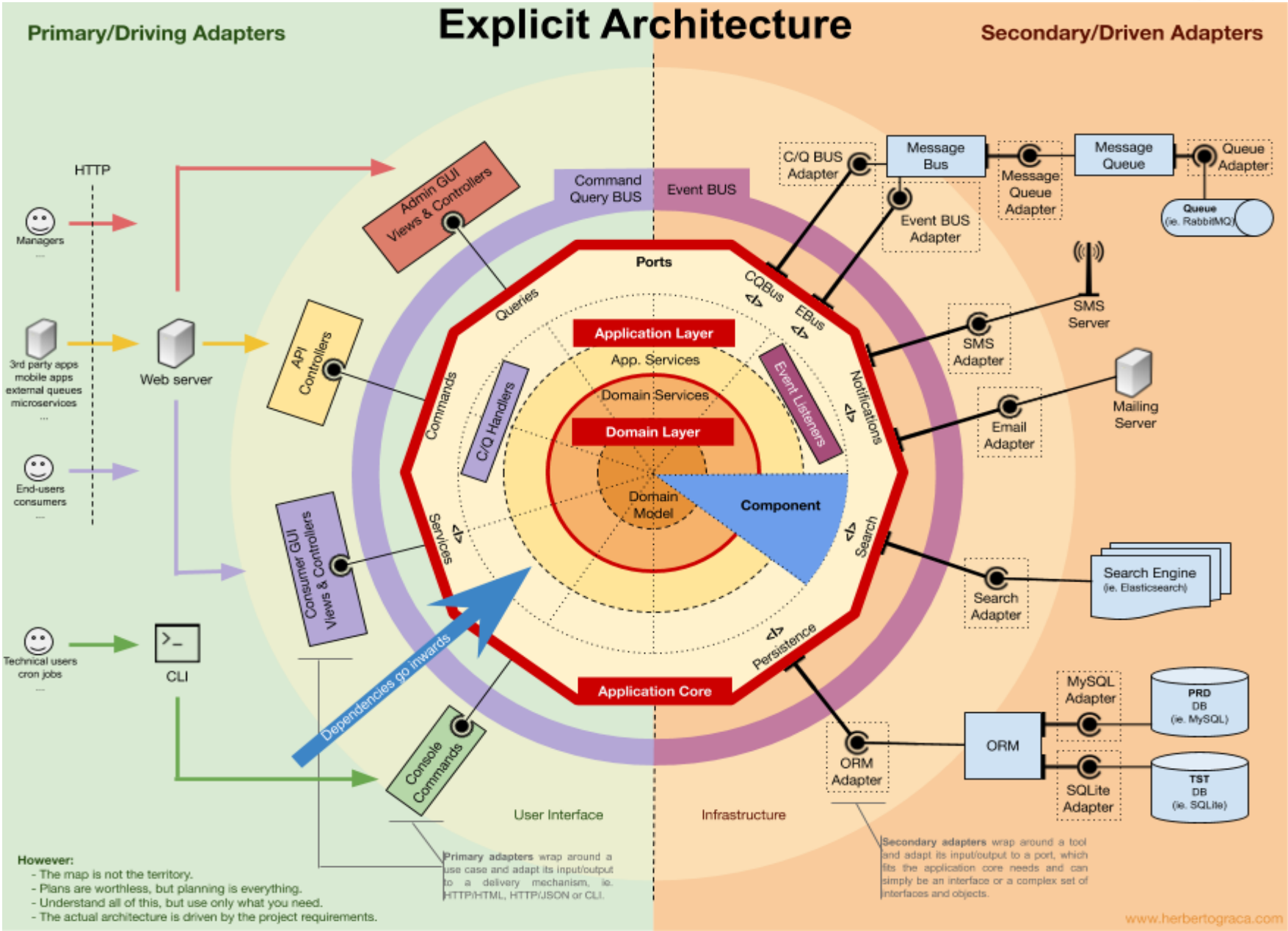
```
% tree
.
├── Makefile
├── README.md
├── app
│   ├── domain
│   │   ├── model
│   │   ├── repository
│   │   └── service
│   ├── interface
│   │   ├── persistence
│   │   └── rpc
│   ├── registry
│   └── usecase
├── cmd
│   └── 8am
│       └── main.go
└── vendor
    ├── vendor packages
    |...
```


DOMAIN
USE CASES
$GOPATH/src/usecases/usecases.go
package usecases
import (
"domain"
"fmt"
)
type UserRepository interface {
Store(user User)
FindById(id int) User
}
type User struct {
Id int
IsAdmin bool
Customer domain.Customer
}
type Item struct {
Id int
Name string
Value float64
}
type Logger interface {
Log(message string) error
}
$GOPATH/src/domain/domain.go
package domain
import (
"errors"
)
type ItemRepository interface {
Store(item Item)
FindById(id int) Item
}
type OrderRepository interface {
Store(order Order)
FindById(id int) Order
}
type CustomerRepository interface {
Store(customer Customer)
FindById(id int) Customer
}
$GOPATH/src/interfaces/repositories.go
package interfaces
type DbUserRepo DbRepo
func NewDbUserRepo(dbHandlers map[string]DbHandler)
*DbUserRepo {
dbUserRepo := new(DbUserRepo)
dbUserRepo.dbHandlers = dbHandlers
dbUserRepo.dbHandler = dbHandlers["DbUserRepo"]
return dbUserRepo
}
func (repo *DbUserRepo) Store(user usecases.User) {
isAdmin := "no"
if user.IsAdmin {
isAdmin = "yes"
}
repo.dbHandler.Execute(fmt.Sprintf(`INSERT INTO users (id,
customer_id, is_admin)
VALUES ('%d', '%d', '%v')`,
user.Id, user.Customer.Id, isAdmin))
customerRepo := NewDbCustomerRepo(repo.dbHandlers)
customerRepo.Store(user.Customer)
}
func (repo *DbUserRepo) FindById(id int) usecases.User {
row := repo.dbHandler.Query(fmt.Sprintf(`SELECT is_admin,
customer_id
FROM users WHERE id = '%d' LIMIT 1`,
id))
var isAdmin string
var customerId int
row.Next()
row.Scan(&isAdmin, &customerId)
customerRepo := NewDbCustomerRepo(repo.dbHandlers)
u := usecases.User{Id: id, Customer:
customerRepo.FindById(customerId)}
u.IsAdmin = false
if isAdmin == "yes" {
u.IsAdmin = true
}
return u
}
$GOPATH/src/interfaces/repositories.go
package interfaces
import (
"domain"
"fmt"
"usecases"
)
$GOPATH/src/interfaces/repositories.go
package interfaces



Applying The Clean Architecture to Go applications (Manuel Kiessling)

Visualization is done by Eduard Sesigin


Explicit Architecture
Primary/Driving Adapters
Secondary/Driven Adapters
HTTP
Managers
3rd party apps
mobile apps
external queues
microservices
End-users
consumers
Technical users
cron jobs
Web server
CLI
Admin GUI
Views & Controllers
API
Controllers
Consumer GUI
Views & Controllers
Console
Commands
Command
Query BUS
Event BUS
Ports
Queries
Commands
Application Layer
App. Services
Domain Services
Domain Layer
Domain
Model
Component
C/Q Handlers
Event Listeners
CQBus
EBus
Notifications
Services
Search
Persistence
Application Core
Dependencies go inwards
User Interface
Infrastructure
C/Q BUS
Adapter
Message
Bus
Message
Queue
Adapter
Message
Queue
Queue
Adapter
Queue
(ie. RabbitMQ)
Event BUS
Adapter
SMS
Server
SMS
Adapter
Mailing
Server
Email
Adapter
Search Engine
(ie. Elasticsearch)
Search
Adapter
MySQL
Adapter
PRD
DB
(ie. MySQL)
ORM
ORM
Adapter
SQLite
Adapter
TST
DB
(ie. SQLite)
However:
- The map is not the territory.
- Plans are worthless, but planning is everything.
- Understand all of this, but use only what you need.
- The actual architecture is driven by the project requirements.
Primary adapters wrap around a use case and adapt its input/output to a delivery mechanism, ie. HTTP/HTML, HTTP/JSON or CLI.
Secondary adapters wrap around a tool and adapt its input/output to a port, which fits the application core needs and can simply be an interface or a complex set of interfaces and objects.
www.herbertograca.com

Go Installation

Download and install

1. Go download.
2. Go install.
3. Go code.

Download and install Go quickly with the steps described here.

For other content on installing, you might be interested in:

- Managing Go installations -- How to install multiple versions and uninstall.
- Installing Go from source -- How to check out the sources, build them on your own machine, and run them.

1. Go download.

Click the button below to download the Go installer.

Download Go for Windows
go1.15.6.windows-amd64.msi (115 MB)

Go Installation

Linux | Mac | Windows

If you have a previous version of Go installed, be sure to remove it before installing another.

1. Download the archive and extract it into /usr/local, creating a Go tree in /usr/local/go.

   For example, run the following as root or through `sudo`:

   ```
   tar -C /usr/local -xzf go1.15.6.linux-amd64.tar.gz
   ```

2. Add /usr/local/go/bin to the `PATH` environment variable.

   You can do this by adding the following line to your $HOME/.profile or /etc/profile (for a system-wide installation):

   ```
   export PATH=$PATH:/usr/local/go/bin
   ```

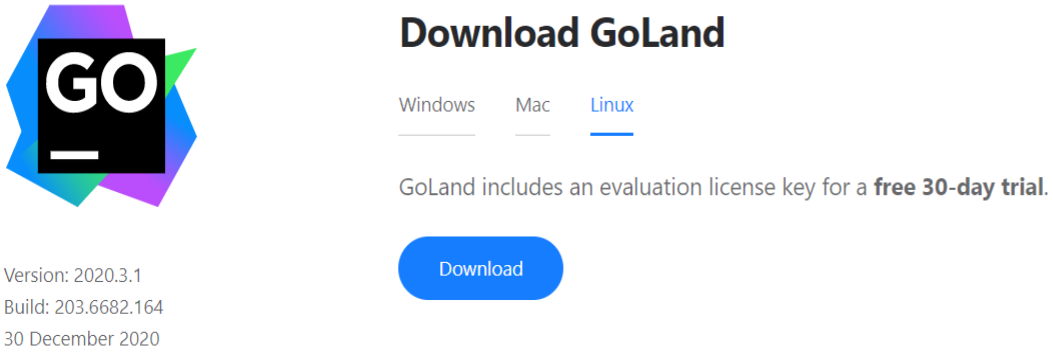
   Note: Changes made to a profile file may not apply until the next time you log into your computer. To apply the changes immediately, just run the shell commands directly or execute them from the profile using a command such as `source $HOME/.profile`.

3. Verify that you've installed Go by opening a command prompt and typing the following command:

   ```
   $ go version
   ```

4. Confirm that the command prints the installed version of Go.

Go Installation



Installation Instructions

1. Unpack the goland-2020.3.1.tar.gz file to an empty directory using the following command: tar -xzf goland- 2020.3.1.tar.gz
2. Note: A new instance MUST NOT be extracted over an existing one. The target folder must be empty.
3. Run goland.sh from the bin subdirectory

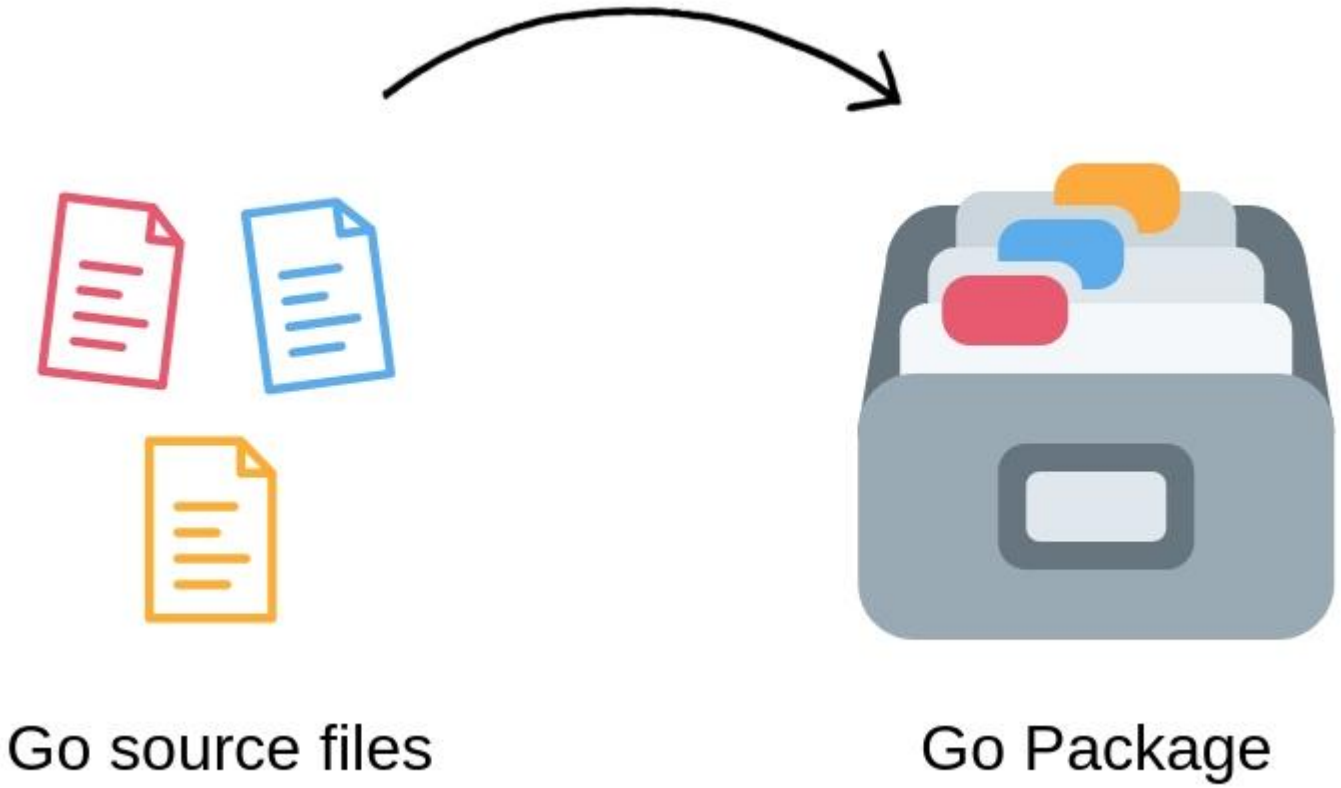
Golang Structure

- Every Go Program Contains the following Parts
  - Declaration of Packages
  - Package Importing
  - Functions
  - Variables
  - Expression and Statements
  - Comments

Golang Execution Steps

- Step1) Write the Go Program in a Text Editor.
- Step2) Save the program with ".go" as the program file extension.
- Step3) Go to command Prompt.
- Step4) In the command prompt, we have to open the directory where we have saved our Program.
- Step5) After opening the directory, we have to open our file and click enter to compile our code.
- Step6) If no errors are present in our code, then our program is executed, and the following output is displayed:

Packages



Packages

Standard library ▾

Name	Synopsis
archive	
tar	Package tar implements access to tar archives.
zip	Package zip provides support for reading and writing ZIP archives.
bufio	Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.
builtin	Package builtin provides documentation for Go's predeclared identifiers.
bytes	Package bytes implements functions for the manipulation of byte slices.
compress	
bzip2	Package bzip2 implements bzip2 decompression.
flate	Package flate implements the DEFLATE compressed data format, described in RFC 1951.
gzip	Package gzip implements reading and writing of gzip format compressed files, as specified in RFC 1952.
lzw	Package lzw implements the Lempel-Ziv-Welch compressed data format, described in T. A. Welch, "A Technique for High-Performance Data Compression", Computer, 17(6) (June 1984), pp 8-19.
zlib	Package zlib implements reading and writing of zlib format compressed data, as specified in RFC 1950.
container	
heap	Package heap provides heap operations for any type that implements heap.Interface.
list	Package list implements a doubly linked list.
ring	Package ring implements operations on circular lists.
context	Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.
crypto	Package crypto collects common cryptographic constants.
aes	Package aes implements AES encryption (formerly Rijndael), as defined in U.S. Federal Information Processing Standards Publication 197.
cipher	Package cipher implements standard block cipher modes that can be wrapped around low-level block cipher implementations.
des	Package des implements the Data Encryption Standard (DES) and the Triple Data Encryption Algorithm (TDEA) as defined in U.S. Federal Information Processing Standards Publication 46-3.
dsa	Package dsa implements the Digital Signature Algorithm, as defined in FIPS 186-3.
ecdsa	Package ecdsa implements the Elliptic Curve Digital Signature Algorithm, as defined in FIPS 186-3.
ed25519	Package ed25519 implements the Ed25519 signature algorithm.

Packages

database	
sql	Package sql provides a generic interface around SQL (or SQL-like) databases.
driver	Package driver defines interfaces to be implemented by database drivers as used by package sql.
debug	
dwarf	Package dwarf provides access to DWARF debugging information loaded from executable files, as defined in the DWARF 2.0 Standard at http://dwarfstd.org/doc/dwarf-2.0.0.pdf
elf	Package elf implements access to ELF object files.
gosym	Package gosym implements access to the Go symbol and line number tables embedded in Go binaries generated by the gc compilers.
macho	Package macho implements access to Mach-O object files.
pe	Package pe implements access to PE (Microsoft Windows Portable Executable) files.
plan9obj	Package plan9obj implements access to Plan 9 a.out object files.
encoding	Package encoding defines interfaces shared by other packages that convert data to and from byte-level and textual representations.
ascii85	Package ascii85 implements the ascii85 data encoding as used in the btoa tool and Adobe's PostScript and PDF document formats.
asn1	Package asn1 implements parsing of DER-encoded ASN.1 data structures, as defined in ITU-T Rec X.690.
base32	Package base32 implements base32 encoding as specified by RFC 4648.
base64	Package base64 implements base64 encoding as specified by RFC 4648.
binary	Package binary implements simple translation between numbers and byte sequences and encoding and decoding of varints.
csv	Package csv reads and writes comma-separated values (CSV) files.
gob	Package gob manages streams of gobs - binary values exchanged between an Encoder (transmitter) and a Decoder (receiver).
hex	Package hex implements hexadecimal encoding and decoding.
json	Package json implements encoding and decoding of JSON as defined in RFC 7159.
pem	Package pem implements the PEM data encoding, which originated in Privacy Enhanced Mail.
xml	Package xml implements a simple XML 1.0 parser that understands XML name spaces.
errors	Package errors implements functions to manipulate errors.
expvar	Package expvar provides a standardized interface to public variables, such as operation counters in servers.
flag	Package flag implements command-line flag parsing.
fmt	Package fmt implements formatted I/O with functions analogous to C's printf and scanf.
go	
ast	Package ast declares the types used to represent syntax trees for Go packages.
build	Package build gathers information about Go packages.
constant	Package constant implements Values representing untyped Go constants and their corresponding operations.
doc	Package doc extracts source code documentation from a Go AST.

Your First Program


```
package main

import "fmt"

// this is a comment

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

package main

- This is known as a “package declaration”.
- Every Go program must start with a package declaration.
- Packages are Go's way of organizing and reusing code.
- There are two types of Go programs: executables and libraries.
- **Executable** applications are the kinds of programs that we can run directly from the terminal. (in Windows they end with .exe)
- **Libraries** are collections of code that we package together so that we can use them in other programs.

Packages and imports Every Go


```
package main

func main() {
        print("Hello, World!\n")
}
```

```
import "fmt"
import "math/rand"
```

```
import (
    "fmt"
    "math/rand"
)
```

Code Location


```
import "github.com/mattetti/goRailsYourself/crypto"
```

```
$ go get github.com/mattetti/goRailsYourself/crypto
```

Import fmt

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

Fmt package

Printing

The verbs:

General:

```
%v      the value in a default format
        when printing structs, the plus flag (%+v) adds field names
%#v     a Go-syntax representation of the value
%T      a Go-syntax representation of the type of the value
%%      a literal percent sign; consumes no value
```

Boolean:

```
%t      the word true or false
```

Integer:

```
%b      base 2
%c      the character represented by the corresponding Unicode code point
%d      base 10
%o      base 8
%O      base 8 with 0o prefix
%q      a single-quoted character literal safely escaped with Go syntax.
%x      base 16, with lower-case letters for a-f
%X      base 16, with upper-case letters for A-F
%U      Unicode format: U+1234; same as "U+%04X"
```

Fmt package

Floating-point and complex constituents:

```
%b      decimalless scientific notation with exponent a power of two,
        in the manner of strconv.FormatFloat with the 'b' format,
        e.g. -123456p-78
%e      scientific notation, e.g. -1.234456e+78
%E      scientific notation, e.g. -1.234456E+78
%f      decimal point but no exponent, e.g. 123.456
%F      synonym for %f
%g      %e for large exponents, %f otherwise. Precision is discussed below.
%G      %E for large exponents, %F otherwise
%x      hexadecimal notation (with decimal power of two exponent), e.g. -0x1.23abcp+20
%X      upper-case hexadecimal notation, e.g. -0X1.23ABCP+20
```

String and slice of bytes (treated equivalently with these verbs):

```
%s      the uninterpreted bytes of the string or slice
%q      a double-quoted string safely escaped with Go syntax
%x      base 16, lower-case, two characters per byte
%X      base 16, upper-case, two characters per byte
```

Slice:

```
%p      address of 0th element in base 16 notation, with leading 0x
```

Pointer:

```
%p      base 16 notation, with leading 0x
The %b, %d, %o, %x and %X verbs also work with pointers,
formatting the value exactly as if it were an integer.
```

Fmt package

Index

func Errorf(format string, a ...interface{}) error
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
func Fscan(r io.Reader, a ...interface{}) (n int, err error)
func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)
func Fscanln(r io.Reader, a ...interface{}) (n int, err error)
func Print(a ...interface{}) (n int, err error)
func Printf(format string, a ...interface{}) (n int, err error)
func Println(a ...interface{}) (n int, err error)
func Scan(a ...interface{}) (n int, err error)
func Scanf(format string, a ...interface{}) (n int, err error)
func Scanln(a ...interface{}) (n int, err error)
func Sprint(a ...interface{}) string
func Sprintf(format string, a ...interface{}) string
func Sprintln(a ...interface{}) string
func Sscan(str string, a ...interface{}) (n int, err error)
func Sscanf(str string, format string, a ...interface{}) (n int, err error)
func Sscanln(str string, a ...interface{}) (n int, err error)
type Formatter
type GoStringer
type ScanState
type Scanner
type State
type Stringer

How To Write Comments in Go


```
// This is a comment


/*
Everything here
will be considered
a block comment
*/
```

Naming Variables: Rules and Style

- The naming of variables is quite flexible, but there are some rules to keep in mind:
- Variable names must only be one word (as in no spaces).
- Variable names must be made up of only letters, numbers, and underscores (_).
- Variable names cannot begin with a number.

Valid	Invalid	Why Invalid
`userName`	`user-name`	Hyphens are not permitted
`name4`	`4name`	Cannot begin with a number
`user`	`$user`	Cannot use symbols
`userName`	`user name`	Cannot be more than one word

Basic Types


```
bool
string

Numeric types:

uint         either 32 or 64 bits
int          same size as uint
uintptr      an unsigned integer large enough to store the uninterpreted bits of
             a pointer value
uint8        the set of all unsigned  8-bit integers (0 to 255)
uint16       the set of all unsigned 16-bit integers (0 to 65535)
uint32       the set of all unsigned 32-bit integers (0 to 4294967295)
uint64       the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8         the set of all signed  8-bit integers (-128 to 127)
int16        the set of all signed 16-bit integers (-32768 to 32767)
int32        the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64        the set of all signed 64-bit integers
             (-9223372036854775808 to 9223372036854775807)

float32      the set of all IEEE-754 32-bit floating-point numbers
float64      the set of all IEEE-754 64-bit floating-point numbers

complex64    the set of all complex numbers with float32 real and imaginary parts
complex128   the set of all complex numbers with float64 real and imaginary parts

byte         alias for uint8
rune         alias for int32 (represents a Unicode code point)
```

Variables & inferred typing

The var statement declares a list of variables with the type declared last.

```
var (
        name     string
        age      int
        location string
)
```

Or even

```
var (
        name, location  string
        age             int
)
```

Variables & inferred typing

Variables can also be declared one by one:

```
var name     string
var age      int
var location string
```

A var declaration can include initializers, one per variable.

```
var (
        name     string = "Prince Oberyn"
        age      int    =  32
        location string = "Dorne"
)
```

If an initializer is present, the type can be omitted, the variable will take the type of the initializer (inferred typing).

```
var (
        name     = "Prince Oberyn"
        age      =  32
        location = "Dorne"
)
```

Variables & inferred typing

You can also initialize variables on the same line:

```
var (
        name, location, age = "Prince Oberyn", "Dorne", 32
)
```

Inside a function, the `:=` short assignment statement can be used in place of a var declaration with implicit type.

```
func main() {
        name, location := "Prince Oberyn", "Dorne"
        age := 32
        fmt.Printf("%s (%d) of %s", name, age, location)
}
```

A variable can contain any type, including functions:

```
func main() {
        action := func() {
                //doing something
        }
        action()
}
```

Constants


```
const Pi = 3.14
const (
        StatusOK                   = 200
        StatusCreated              = 201
        StatusAccepted             = 202
        StatusNonAuthoritativeInfo = 203
        StatusNoContent            = 204
        StatusResetContent         = 205
        StatusPartialContent       = 206
)
```

```
const (
        Pi     = 3.14
        Truth  = false
        Big    = 1 << 62
        Small  = Big >> 61
)

func main() {
        const Greeting = ""
        fmt.Println(Greeting)
        fmt.Println(Pi)
        fmt.Println(Truth)
        fmt.Println(Big)
}
```

Basic Types


```
package main

import (
        "fmt"
        "math/cmplx"
)

var (
        goIsFun bool       = true
        maxInt  uint64     = 1<<64 - 1
        complex complex128 = cmplx.Sqrt(-5 + 12i)
)

func main() {
        const f = "%T(%v)\n"
        fmt.Printf(f, goIsFun, goIsFun)
        fmt.Printf(f, maxInt, maxInt)
        fmt.Printf(f, complex, complex)
}
```

```
bool(true)
uint64(18446744073709551615)
complex128((2+3i))
```

Type conversion

The expression `T(v)` converts the value `v` to the type `T`. Some numeric conversions:

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

```
b := 125.0
c := 390.8

fmt.Println(int(b))
fmt.Println(int(c))
```

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    a := strconv.Itoa(12)
    fmt.Printf("%q\n", a)
}
```

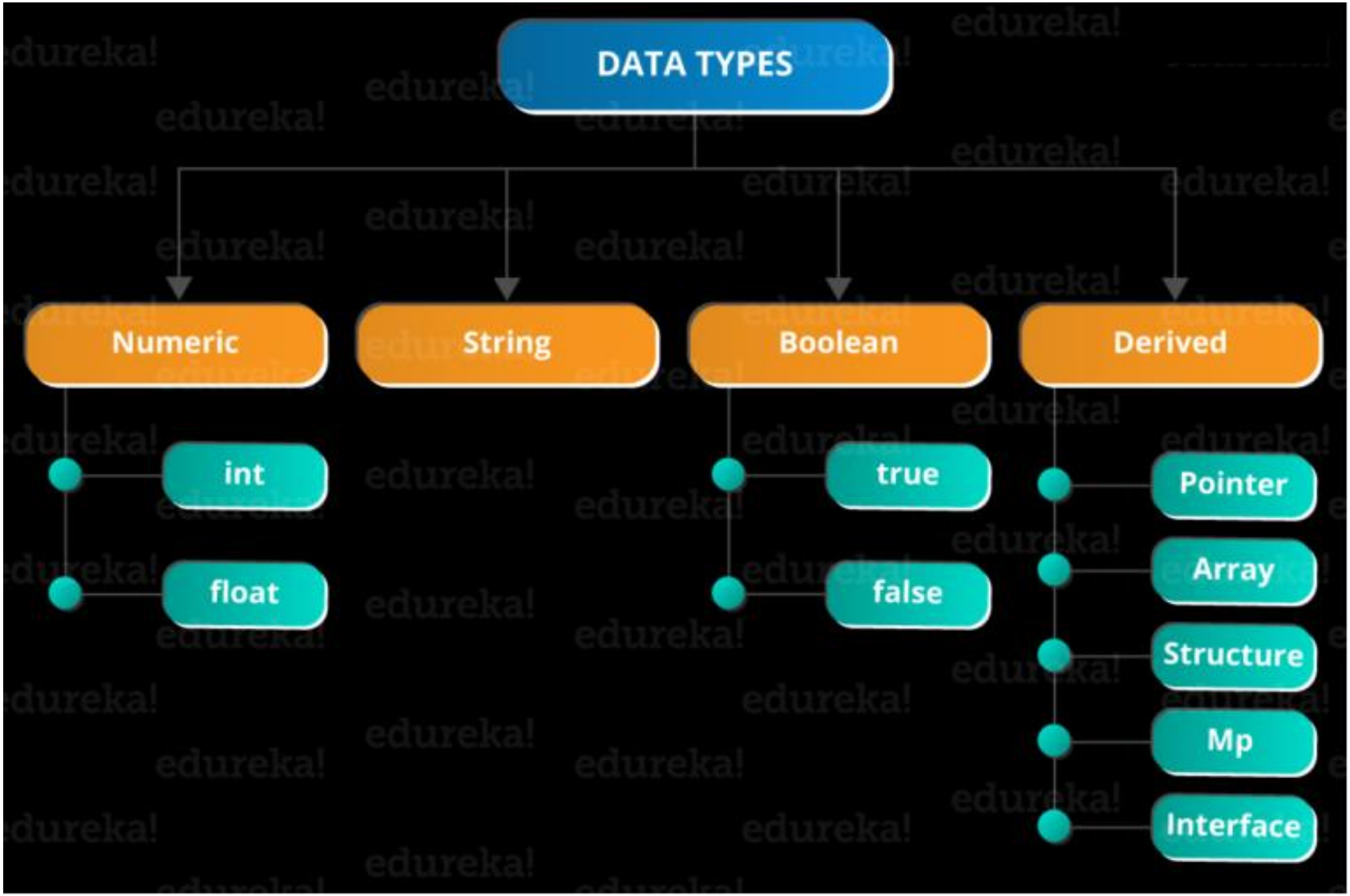
Type conversion


```
a := "not a number"
b, err := strconv.Atoi(a)
fmt.Println(b)
fmt.Println(err)
```

Global and Local Variables

```
package main

import "fmt"


var g = "global"

func printLocal() {
    l := "local"
    fmt.Println(l)
}

func main() {
    printLocal()
    fmt.Println(g)
}
```

Data Types


DATA TYPES
Numeric
String
Boolean
Derived
int
float
true
false
Pointer
Array
Structure
Mp
Interface

Data Types

Data Type	Range
uint8	0 to 255
uint16	0 to 65535
uint32	0 to 4294967295
uint64	0 to 18446744073709551615

Data Type	Range
int8	-128 to 127
int16	-32768 to 32767
int32	-2147483648 to 2147483647
int64	-9223372036854775808 to 9223372036854775808

Data Types

Go has the following architecture-independent integer types:

```
uint8       unsigned  8-bit integers (0 to 255)
uint16      unsigned 16-bit integers (0 to 65535)
uint32      unsigned 32-bit integers (0 to 4294967295)
uint64      unsigned 64-bit integers (0 to 18446744073709551615)
int8        signed  8-bit integers (-128 to 127)
int16       signed 16-bit integers (-32768 to 32767)
int32       signed 32-bit integers (-2147483648 to 2147483647)
int64       signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
```

Floats and complex numbers also come in varying sizes:

```
float32     IEEE-754 32-bit floating-point numbers
float64     IEEE-754 64-bit floating-point numbers
complex64   complex numbers with float32 real and imaginary parts
complex128  complex numbers with float64 real and imaginary parts
```

There are also a couple of alias number types, which assign useful names to specific data types:

```
byte        alias for uint8
rune        alias for int32
```

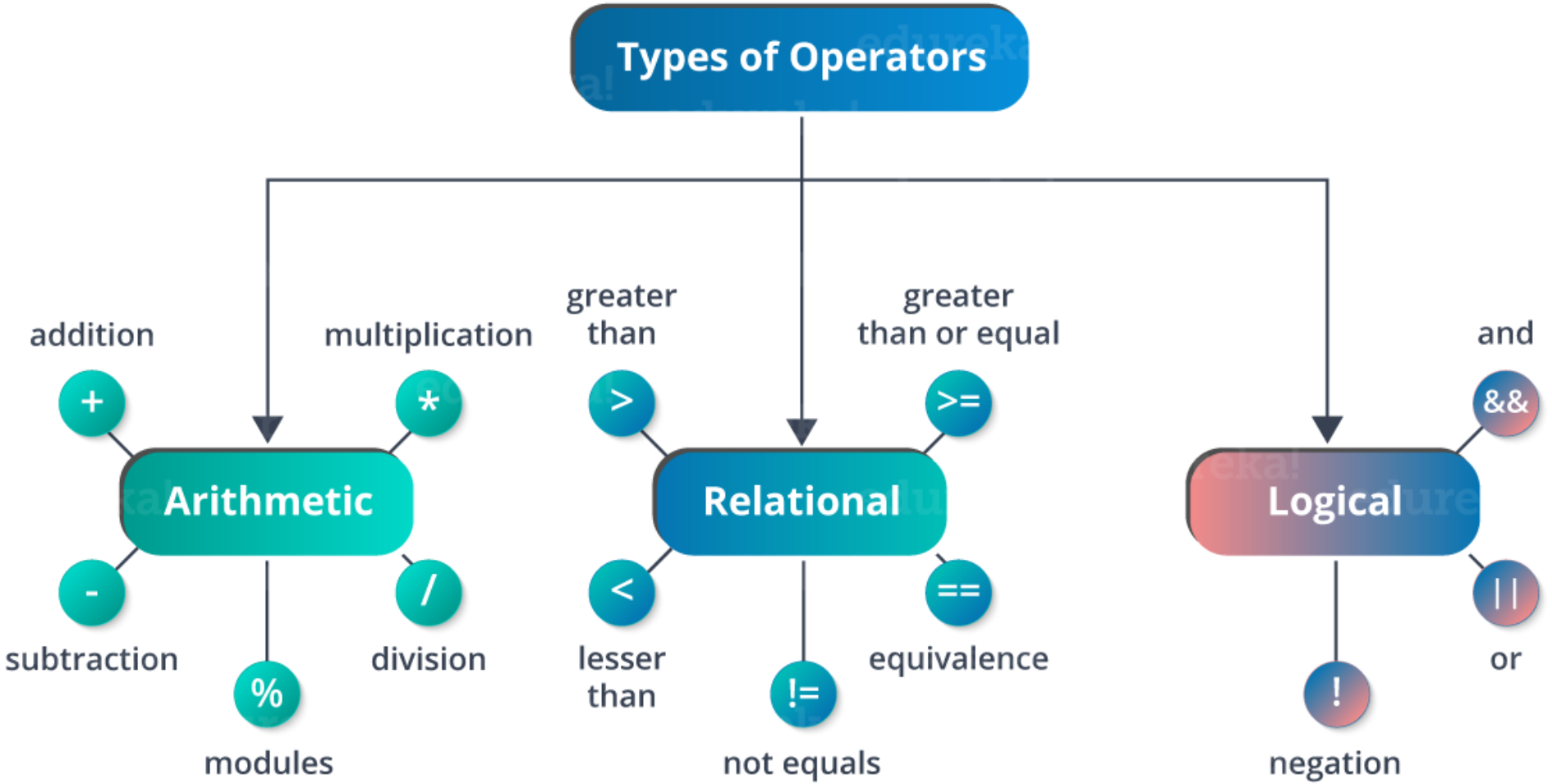
Data Types


```
uint     unsigned, either 32 or 64 bits
int      signed, either 32 or 64 bits
uintptr  unsigned integer large enough to store the uninterpreted bits of a pointer value
```

Keywords

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	If	range	type
continue	for	import	return	var

Operators


Types of Operators
Arithmetic
addition
+
multiplication
*
subtraction
-
division
/
modules
%
Relational
greater than
>
greater than or equal
>=
lesser than
<
not equals
!=
equivalence
==
Logical
and
&&
or
||
negation
!

Arrays

- Define Array
  - *[capacity]data_type{element_values}*
  - *[4]string{"blue coral", "staghorn coral", "pillar coral", "elkhorn coral"}*

- `coral := [4]string{"blue coral", "staghorn coral", "pillar coral", "elkhorn coral”}`
- `fmt.Println(coral)`

Arrays


```
regNos :=make([]int32,100)
for r:=range regNos{

        regNos[r] = rand.Int31n(1000)

        }

        for index, val := range regNos{
                fmt.Printf("%d = %d\n", index, val)
        }
```

Pointers

- A data type called a pointer holds the memory address of the data, but not the data itself.
- The memory address tells the function where to find the data, but not the value of the data.
- You can pass the pointer to the function instead of the data, and the function can then alter the original variable in place.
- This is called passing by reference, because the value of the variable isn’t passed to the function, just its location.

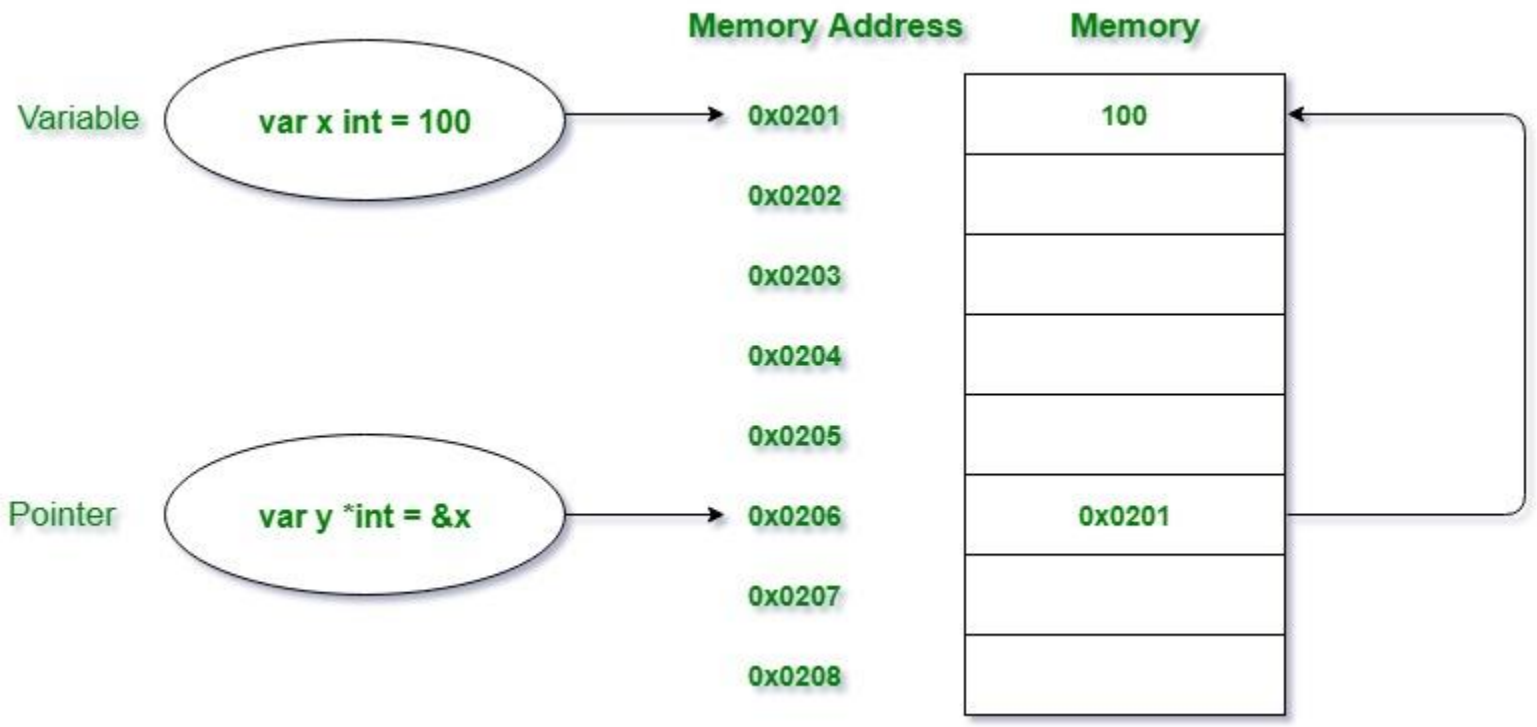
Pointers

- An ampersand in front of a variable name is to get the address, or a pointer to that variable.
- Asterisk (*) or dereferencing operator prefixed with an * creates variable and initializes the pointer with the address.
- **var myPointer *int32 = &someint**

What is the need of the pointers?

- Variables are the names given to a memory location where the actual data is stored.
- To access the stored data we need the address of that particular memory location.
- To remember all the memory addresses(Hexadecimal Format) manually is an overhead that’s why we use variables to store data and variables can be accessed just by using their name.
- Golang also allows saving a hexadecimal number into a variable using the literal expression i.e. number starting from 0x is a hexadecimal number.

What is the need of the pointers?



Pointers

main.go

```
package main

import "fmt"

func main() {
    var creature string = "shark"
    var pointer *string = &creature

    fmt.Println("creature =", creature)
    fmt.Println("pointer =", pointer)
}
```

Structures

- A structure or struct in Golang is a user-defined type that allows to group/combine items of possibly different types into a single type.
- Any real-world entity which has some set of properties/fields can be represented as a struct.
- This concept is generally compared with the classes in object-oriented programming.
- It can be termed as a lightweight class that does not support inheritance but supports composition.
- For Example, an address has a name, street, city, state, Pincode. It makes sense to group these three properties into a single structure address

Declaring Structure

Declaring a structure:

```
type Address struct {
      name string
      street string
      city string
      state string
      Pincode int
}
```

Refer OOPsDemo and CompositionDemo

Inline Structure


```
package main

import "fmt"

func main() {
    c := struct {
        Name string
        Type string
    }{
        Name: "Sammy",
        Type: "Shark",
    }
    fmt.Println(c.Name, "the", c.Type)
}
```

Methods in GO

func(reciver_name Type) method_name(parameter_list)(return_type){
// Code
}

```
type Creature struct {
    Name     string
    Greeting string
}

func (c Creature) Greet() {
    fmt.Printf("%s says %s", c.Name, c.Greeting)
}
```

Method vs Function

Method	Function
It contain receiver.	It does not contain receiver.
It can accept both pointer and value.	It cannot accept both pointer and value.
Methods of the same name but different types can be defined in the program.	Functions of the same name but different type are not allowed to define in the program.

Method with Pointer and Value

- Refer StructAuthorDemo

Map

- 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.

Interface

- Go language interfaces are different from other languages.
- In Go language, the interface is a custom type that is used to specify a set of one or more method signatures and the interface is abstract, so you are not allowed to create an instance of the interface.
- But you are allowed to create a variable of an interface type and this variable can be assigned with a concrete type value that has the methods the interface requires.
- Interface is a collection of methods as well as it is a custom type.

Interface


```
type Modify interface {
        changeName(name *string)
}

func receiveInterface(m Modify){
        var name string ="Parameswari"
        var ptrname *string=&name
        m.changeName(ptrname)
```

How To Define and Call Functions in Go

- A function is defined by using the func keyword. This is then followed by a name of your choosing and a set of parentheses that hold any parameters the function will take (they can be empty).
- The lines of function code are enclosed in curly brackets {}.
- In this case, we'll define a function named hello():
- **func hello() {}**
- This sets up the initial statement for creating a function.
- From here, we'll add a second line to provide the instructions for what the function does. In this case, we'll be printing Hello, World! to the console:
- Our function is now fully defined, but if we run the program at this point, nothing will happen since we didn't call the function.
- So, inside of our main() function block, let's call the function with hello():

How To Define and Call Functions in Go


```
import (
    "fmt"
    "strings"
)

func main() {
    names()
}

func names() {
    fmt.Println("Enter your name:")

    var name string
    fmt.Scanln(&name)
    // Check whether name has a vowel
    for _, v := range strings.ToLower(name) {
        if v == 'a' || v == 'e' || v == 'i' || v == 'o' || v == 'u' {
            fmt.Println("Your name contains a vowel.")
            return
        }
    }
    fmt.Println("Your name does not contain a vowel.")
}
```

How To Define and Call Functions in Go

repeat.go

```
package main

import "fmt"

func main() {
    repeat("Sammy", 5)
}

func repeat(word string, reps int) {
    for i := 0; i < reps; i++ {
        fmt.Print(word)
    }
}
```

How To Define and Call Functions in Go

double.go

```
package main

import "fmt"

func main() {
    result := double(3)
    fmt.Println(result)
}

func double(x int) int {
    y := x * 2
    return y
}
```

We can run the program and see the output:

```
go run double.go
```

How To Define and Call Functions in Go

repeat.go

```
package main

import "fmt"

func main() {
    val, err := repeat("Sammy", -1)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(val)
}

func repeat(word string, reps int) (string, error) {
    if reps <= 0 {
        return "", fmt.Errorf("invalid value of %d provided for reps. value mu
    }
    var value string
    for i := 0; i < reps; i++ {
        value = value + word
    }
    return value, nil
}
```

Variadic Function

hello.go

```
package main

import "fmt"

func main() {
    sayHello()
    sayHello("Sammy")
    sayHello("Sammy", "Jessica", "Drew", "Jamie")
}

func sayHello(names ...string) {
    for _, n := range names {
        fmt.Printf("Hello %s\n", n)
    }
}
```

Variadic Function

hello.go

```go
package main

import "fmt"

func main() {
    sayHello()
    sayHello("Sammy")
    sayHello("Sammy", "Jessica", "Drew", "Jamie")
}

func sayHello(names ...string) {
    if len(names) == 0 {
        fmt.Println("nobody to greet")
        return
    }
    for _, n := range names {
        fmt.Printf("Hello %s\n", n)
    }
}
```

Variadic Function with ordered Arguments

join.go

```
package main

import "fmt"

func main() {
    var line string

    names := []string{"Sammy", "Jessica", "Drew", "Jamie"}

    line = join(",", names)
    fmt.Println(line)
}

func join(del string, values ...string) string {
    var line string
    for i, v := range values {
        line = line + v
        if i != len(values)-1 {
            line = line + del
        }
    }
    return line
}
```

Recursive Function

- Recursion is the process of repeating items in a self-similar way.
- The same concept applies in programming languages as well.
- If a program allows to call a function inside the same function, then it is called a recursive function call.

Anonymous Function

- Go language provides a special feature known as an anonymous function.
- An anonymous function is a function which doesn’t contain any name.
- It is useful when you want to create an inline function.
- In Go language, an anonymous function can form a closure.

Understanding init in Go

- In Go, the predefined init() function sets off a piece of code to run before any other part of your package.
- This code will execute as soon as the package is imported.
- It can be used when you need your application to initialize in a specific state, such as when you have a specific configuration or set of resources with which your application needs to start.
- It is also used when importing a side effect, a technique used to set the state of a program by importing a specific package.
- This is often used to register one package with another to make sure that the program is considering the correct code for the task.
- Although init() is a useful tool, it can sometimes make code difficult to read, since a hard-to-find init() instance will greatly affect the order in which the code is run.
- Because of this, it is important for developers who are new to Go to understand the facets of this function, so that they can make sure to use init() in a legible manner when writing code.

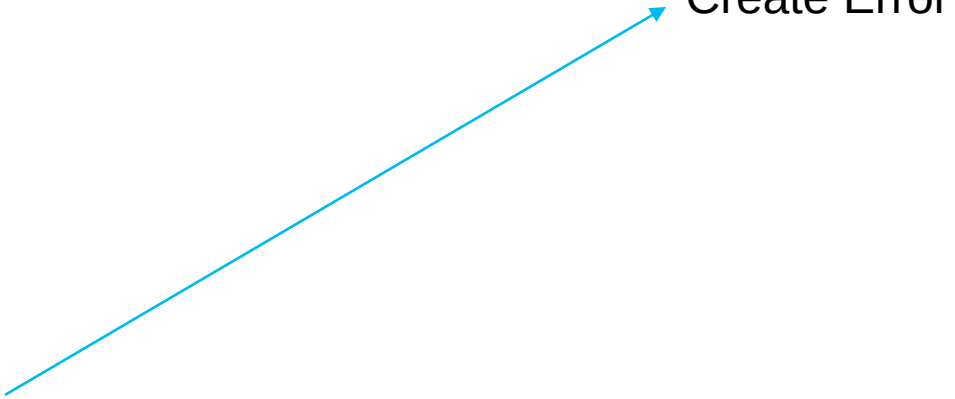
Understanding init in Go

main.go

```
package main

import (
    "fmt"
    "time"
)

var weekday string

func init() {
    weekday = time.Now().Weekday().String()
}

func main() {
    fmt.Printf("Today is %s", weekday)
}
```

Handling Errors

```
package main

import (
    "errors"
    "fmt"
)

func main() {
    err := errors.New("barnacles")


    fmt.Println("Sammy says:", err)
}
```



Dynamic Error Message


```
package main

import (
    "fmt"
    "time"
)

func main() {
    err := fmt.Errorf("error occurred at: %v", time.Now())
    fmt.Println("An error happened:", err)
}
```

Output

```
An error happened: Error occurred at: 2019-07-11
16:52:42.532621 -0400 EDT m=+0.000137103
```

Error Nil


```
package main

import (
    "errors"
    "fmt"
)

func boom() error {
    return errors.New("barnacles")
}

func main() {
    err := boom()

    if err != nil {
        fmt.Println("An error occurred:", err)
        return
    }
    fmt.Println("Anchors away!")
}
```

Error Along Value


```
package main

import (
    "errors"
    "fmt"
    "strings"
)

func capitalize(name string) (string, error) {
    if name == "" {
        return "", errors.New("no name provided")
    }
    return strings.ToTitle(name), nil
}
```

Error Along Value


```
func main() {
    name, err := capitalize("sammy")
    if err != nil {
        fmt.Println("Could not capitalize:", err)
        return
    }


     fmt.Println("Capitalized name:", name)
}
```

Handling Panics in Go

- Errors that a program encounters fall into two broad categories: those the programmer has anticipated and those the programmer has not.
- The error largely deal with errors that we expect as we are writing Go programs.
- The error interface even allows us to acknowledge the rare possibility of an error occurring from function calls, so we can respond appropriately in those situations.

Handling Panics in Go

- Panics fall into the second category of errors, which are unanticipated by the programmer.
- These unforeseen errors lead a program to spontaneously terminate and exit the running Go program.
- Common mistakes are often responsible for creating panics.

Handling Panics in Go

- There are certain operations in Go that automatically return panics and stop the program.
- Common operations include indexing an array beyond its capacity, performing type assertions, calling methods on nil pointers, incorrectly using mutexes, and attempting to work with closed channels.
- Most of these situations result from mistakes made while programming that the compiler has no ability to detect while compiling your program.
- Since panics include detail that is useful for resolving an issue, developers commonly use panics as an indication that they have made a mistake during a program's development.

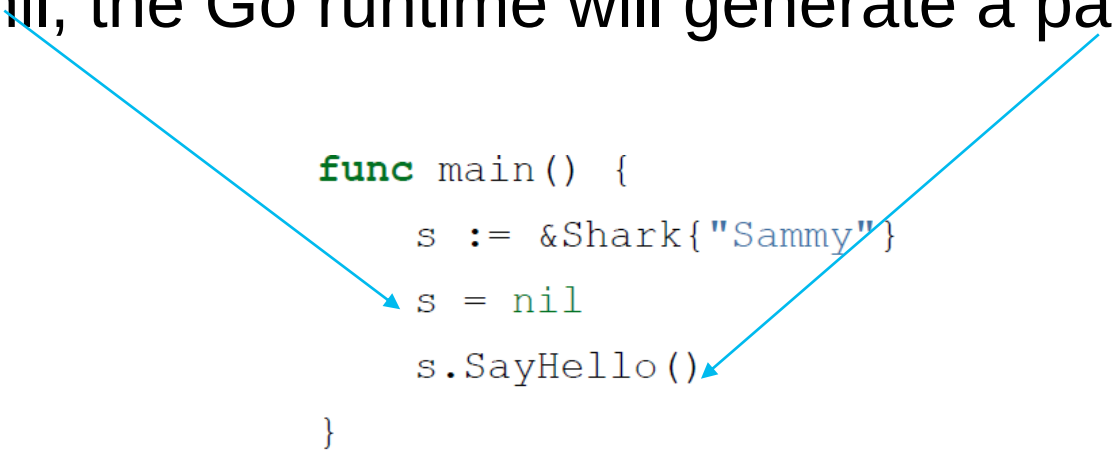
Handling Panics in Go

- **Out of Bounds Panics**
- When you attempt to access an index beyond the length of a slice or the capacity of an array, the Go runtime will generate a panic.

```
func main() {
    names := []string{
        "lobster",
        "sea urchin",
        "sea cucumber",
    }
    fmt.Println("My favorite sea creature is:", names[len(names)])
}
```

Nil Receivers

- The Go programming language has pointers to refer to a specific instance of some type existing in the computer's memory at runtime.
- Pointers can assume the value nil indicating that they are not pointing at anything.
- When we attempt to call methods on a pointer that is nil, the Go runtime will generate a panic.



```
func main() {
    s := &Shark{"Sammy"}
    s = nil
    s.SayHello()
}
```

Deferred Functions

- Your program may have resources that it must clean up properly, even while a panic is being processed by the runtime.
- Go allows you to defer the execution of a function call until its calling function has completed execution.
- Deferred functions run even in the presence of a panic, and are used as a safety mechanism to guard against the chaotic nature of panics.
- Functions are deferred by calling them as usual, then prefixing the entire statement with the defer keyword, as in defer sayHello().

Deferred Function

Syntax:

```
// Function
defer func func_name(parameter_list Type)return_type{
// Code
}

// Method
defer func (receiver Type) method_name(parameter_list){
// Code
}

defer func (parameter_list)(return_type){
// code
}()
```

File Handling

- Intro
  - Everything is a File
- Basic Operations
  - Create Empty File
  - Truncate a File
  - Get File Info
  - Rename and Move a File
  - Delete Files
  - Open and Close Files
  - Check if File Exists
  - Check Read and Write Permissions
  - Change Permissions, Ownership, and Timestamps
  - Create Hard Links and Symlinks

File Handling

- Reading and Writing
  - Copy a File
  - Seek Positions in File
  - Write Bytes to a File
  - Quick Write to File
  - Use Buffered Writer
  - Read up to n Bytes from File
  - Read Exactly n Bytes
  - Read At Least n Bytes
  - Read All Bytes of File

File Handling

- Quick Read Whole File to Memory
  - Use Buffered Reader
  - Read with a Scanner
  - Archiving(Zipping)
  - Archive(Zip) Files
  - Extract(Unzip) Archived Files
  - Compressing
  - Compress a File
  - Uncompress a File
- Misc
  - Temporary Files and Directories
  - Downloading a File Over HTTP
  - Hashing and Checksums

Embedding interfaces in Go

- In Go language, the interface is a collection of method signatures and it is also a type means you can create a variable of an interface type.
- As Go language does not support inheritance, but the Go interface fully supports embedding.
- In embedding, an interface can embed other interfaces or an interface can embed other interface's method signatures in it.
- Any number of interfaces can be embedded in a single interface.
- Day3/Embddingdemo.go

Bit Vector Type

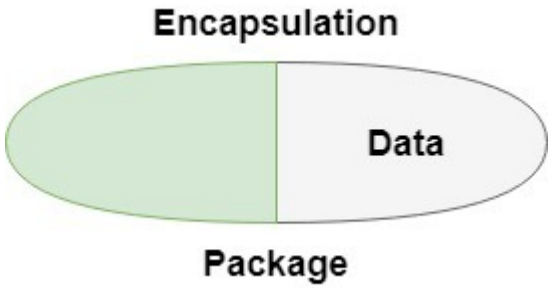
- A bit vector is an array data structure that compactly stores bits.
- This library is based on 5 static different data structures:
- 8-bit vector: relies on an internal uint8
- 16-bit vector: relies on an internal uint16
- 32-bit vector: relies on an internal uint32
- 64-bit vector: relies on an internal uint64
- 128-bit vector: relies on two internal uint64 (for ASCII problems)
- The rationale of using a static integer compared to a dynamic []byte is first of all to save memory. There is no structure and/or slice overhead.

Bit Vector Type

- import "github.com/dropbox/godropbox/container/bitvector"
- Package bitvector provides the implementation of a variable sized compact vector of bits which supports lookups, sets, appends, insertions, and deletions.
- type BitVector
  - func NewBitVector(data []byte, length int) *BitVector
  - func (vector *BitVector) Append(bit byte)
  - func (vector *BitVector) Bytes() []byte
  - func (vector *BitVector) Delete(index int)
  - func (vector *BitVector) Element(i int) byte
  - func (vector *BitVector) Insert(bit byte, index int)
  - func (vector *BitVector) Length() int
  - func (vector *BitVector) Set(bit byte, index int)

Encapsulation in Golang

- Encapsulation is defined as the wrapping up of data under a single unit.
- It is the mechanism that binds together code and the data it manipulates.
- In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.



Encapsulation in Golang

- Go language, encapsulation is achieved by using packages.
- Go provides two different types of identifiers, i.e. exported and unexported identifiers.
- Encapsulation is achieved by exported elements(variables, functions, methods, fields, structures) from the packages, it helps to control the visibility of the elements(variables, functions, methods, fields, structures).
- The elements are visible if the package in which they are defined is available in your program.

Exported Identifiers

- Exported identifiers are those identifiers which are exported from the package in which they are defined.
- The first letter of these identifiers is always in capital letter.
- This capital letter indicates that the given identifier is exported identifier.
- Exported identifiers are always limited to the package in which they are defined.
- When you export the specified identifier from the package you simple just export the name not the implementation of that identifier.
- This mechanism is also applicable for function, fields, methods, and structures.
- **// Exported Method**
- **res := strings.ToUpper(slc[x])**

Unexported Identifiers

- Unexported identifiers are those identifiers which are not exported from any package.
- They are always in lowercase.
- **// Unexported function**

```
func addition(val ...int) int {
    s := 0
    for x := range val {
        s += val[x]
    }
    fmt.Println("Total Sum: ", s)
    return s
}
```

What Are Interface Types?

- An interface type specifies a collection of method prototypes.
- In other words, each interface type defines a method set.
- In fact, we can view an interface type as a method set. For any of the method prototype specified in an interface type, its name can't be the blank identifier _.
- We also often say that each interface type specifies a behavior set (represented by the method set specified by that interface type).

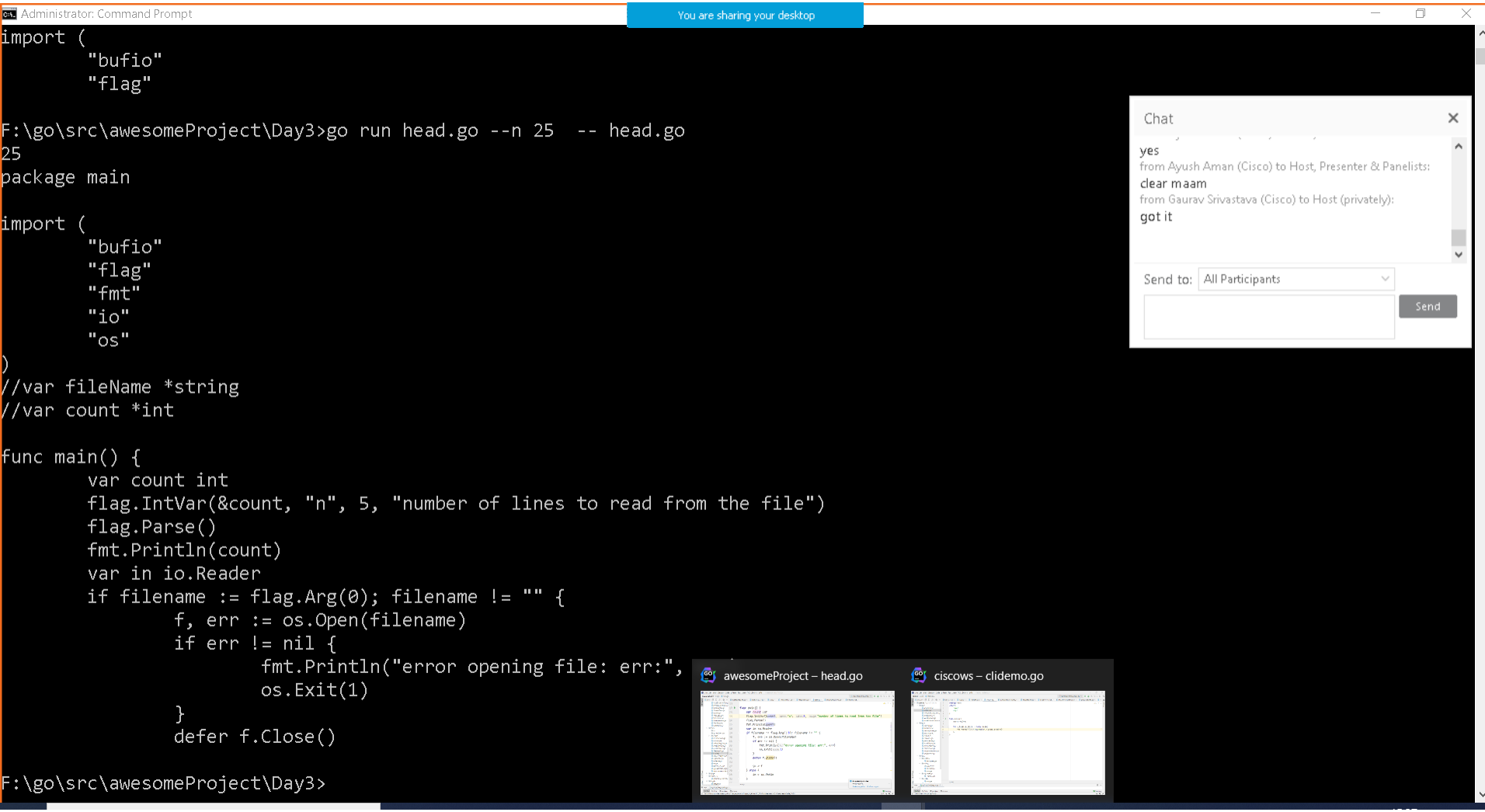
Interface Satisfaction

- A type satisfies an interface if it possesses all the methods the interface requires.
- For example, an *os.File satisfies io.Reader, Writer, Closer, and ReadWriter.
- A *bytes.Buffer satisfies Reader, Writer, and ReadWriter, but does not satisfy Closer because it does not have a Close method.
- As a shorthand, Go programmers often say that a concrete type “is a” particular interface type, meaning that it satisfies the interface.
- For example, a *bytes.Buffer is an io.Writer; an *os.File is an io.ReadWriter.

Parsing flags by flag value

- Using the flag package involves three steps:
- Define variables to capture flag values.
- Define the flags your Go application will use
- Finally, parse the flags provided to the application upon execution.
- Most of the functions within the flag package are concerned with defining flags and binding them to variables that you have defined.
- The parsing phase is handled by the Parse() function.
- go run colortext.go -color

Parsing flags by flag value



Parsing flags by flag value


```
F:\go\src\awesomeProject\Day3>go run head.go -- head.go
package main

import (
        "bufio"
        "flag"

F:\go\src\awesomeProject\Day3>go run flagstringint.go
env: development
port: 3000

F:\go\src\awesomeProject\Day3>go run flagstringint.go --port 8000
env: development
port: 8000

F:\go\src\awesomeProject\Day3>
```

The 3 ways to sort in Go

- Slice of ints, float64s or strings
- Custom comparator
- Custom data structures
- Bonus: Sort a map by key or value
- Performance and implementation

Assertions

- Package assert provides a set of comprehensive testing tools for use with the normal Go testing system.
- Assertions allow you to easily write test code, and are global funcs in the `assert` package.
- All assertion functions take, as the first argument, the `*testing.T` object provided by the testing framework.
- This allows the assertion funcs to write the failings and other details to the correct place.
- Every assertion function also takes an optional string message as the final argument, allowing custom error messages to be appended to the message the assertion method outputs.

Type Assertions

- Type assertions in Golang provide access to the exact type of variable of an interface.
- If already the data type is present in the interface, then it will retrieve the actual data type value held by the interface.
- A type assertion takes an interface value and extracts from it a value of the specified explicit type.
- Basically, it is used to remove the ambiguity from the interface variables.

Type Assertions

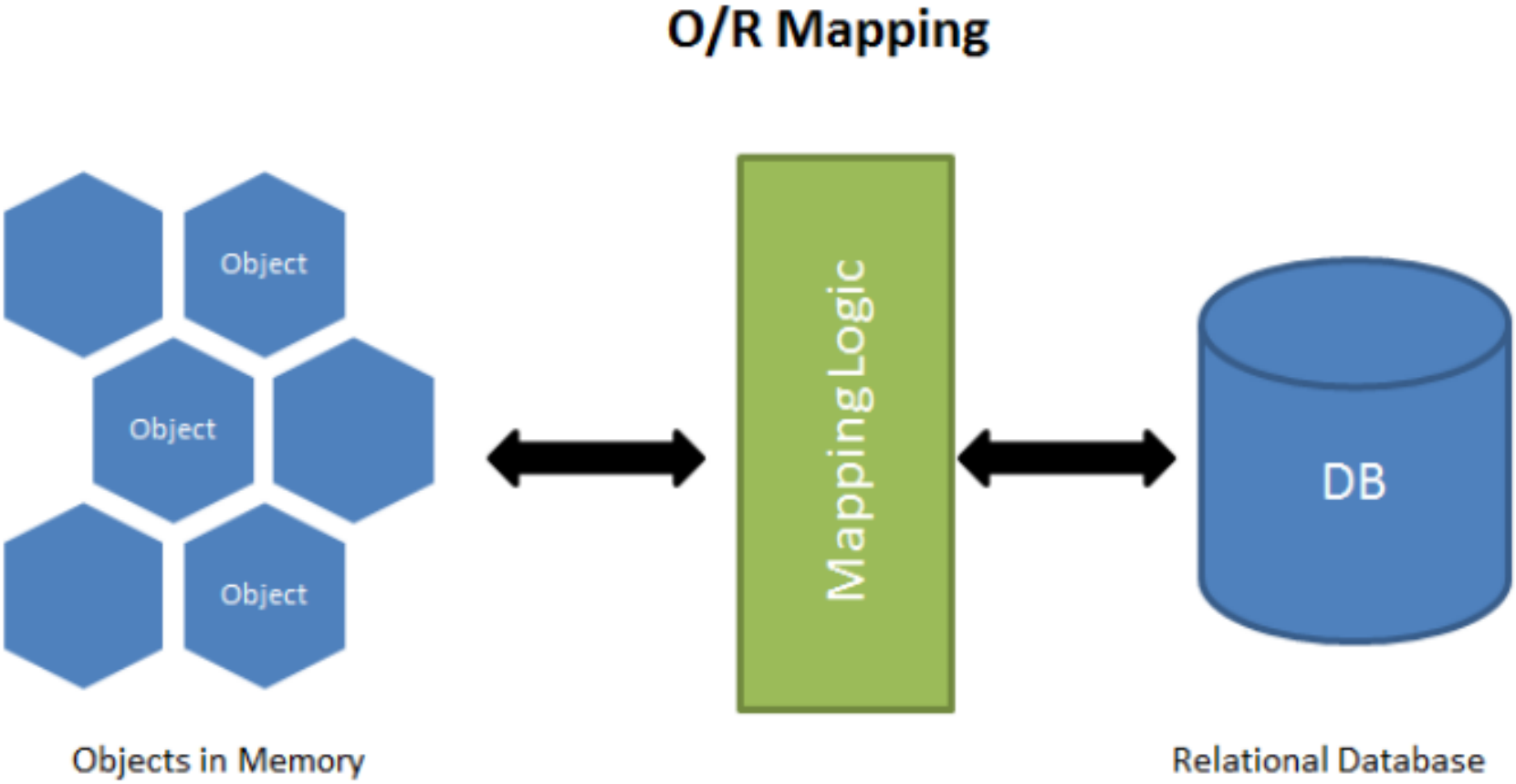
- Syntax:
- t := value.(typeName)
- where value is a variable whose type must be an interface.
- typeName is the concrete type we want to check and underlying typeName value is assigned to variable.

Type Switch

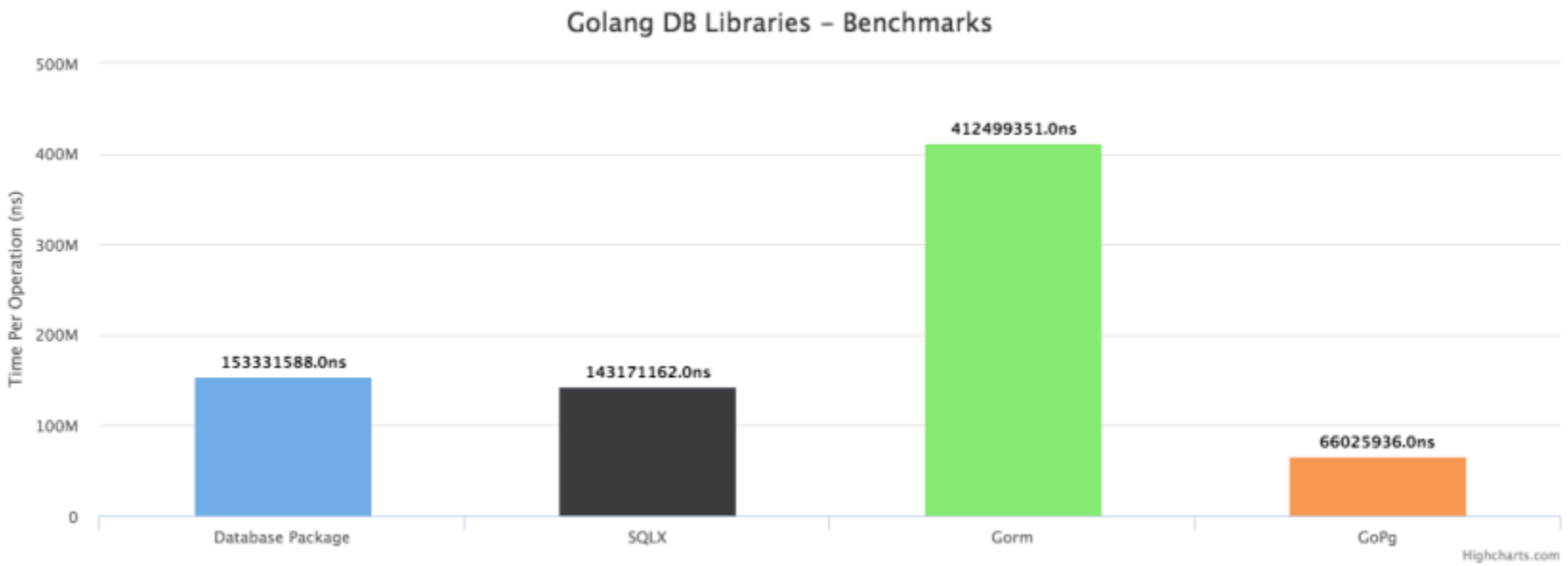
- type switch which makes use of type assertion to determine the type of variable, and do the operations accordingly.

Go with back end mysql

Go ORM


O/R Mapping
Object
Object
Object
Mapping Logic
DB
Objects in Memory
Relational Database

Go ORM


Golang DB Libraries – Benchmarks
Time Per Operation (ns)
500M
400M
300M
200M
100M
0
153331588.0ns
143171162.0ns
412499351.0ns
66025936.0ns
Database Package
SQLX
Gorm
GoPg
Highcharts.com

Regular Expressions

Choice and grouping

Regexp	Meaning
xy	x followed by y
x|y	x or y, prefer x
xy|z	same as (xy)|z
xy*	same as x(y*)

Repetition (greedy and non-greedy)

Regexp	Meaning
x*	zero or more x, prefer more
x*?	prefer fewer (non-greedy)
x+	one or more x, prefer more
x+?	prefer fewer (non-greedy)
x?	zero or one x, prefer one
x??	prefer zero
x{n}	exactly n x

Regular Expressions

Character classes

Expression	Meaning
`.`	any character
`[ab]`	the character a or b
`[^ab]`	any character except a or b
`[a-z]`	any character from a to z
`[a-z0-9]`	any character from a to z or 0 to 9
`\d`	a digit: `[0-9]`
`\D`	a non-digit: `[^0-9]`
`\s`	a whitespace character: `[\t\n\f\r ]`
`\S`	a non-whitespace character: `[^\t\n\f\r ]`
`\w`	a word character: `[0-9A-Za-z_]`
`\W`	a non-word character: `[^0-9A-Za-z_]`
`\p{Greek}`	Unicode character class*
`\pN`	one-letter name
`\P{Greek}`	negated Unicode character class*
`\PN`	one-letter name

Regular Expressions

Special characters

To match a **special character** `\^$.|?*+-[]{}()` literally, escape it with a backslash. For example `\{` matches an opening brace symbol.

Other escape sequences are:

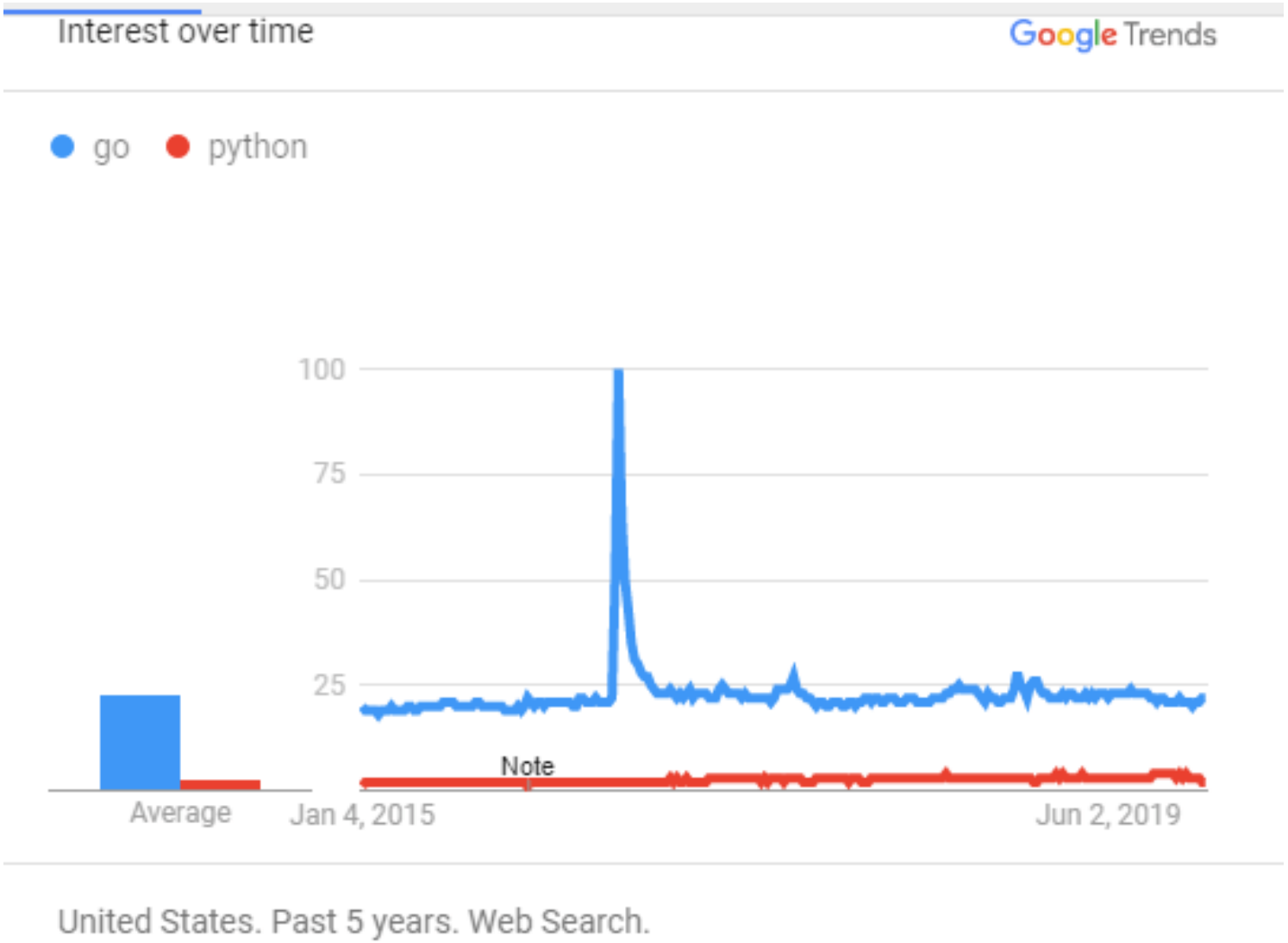
Symbol	Meaning
`\t`	horizontal tab = `\011`
`\n`	newline = `\012`
`\f`	form feed = `\014`
`\r`	carriage return = `\015`
`\v`	vertical tab = `\013`
`\123`	octal character code (up to three digits)
`\x7F`	hex character code (exactly two digits)

Regular Expressions

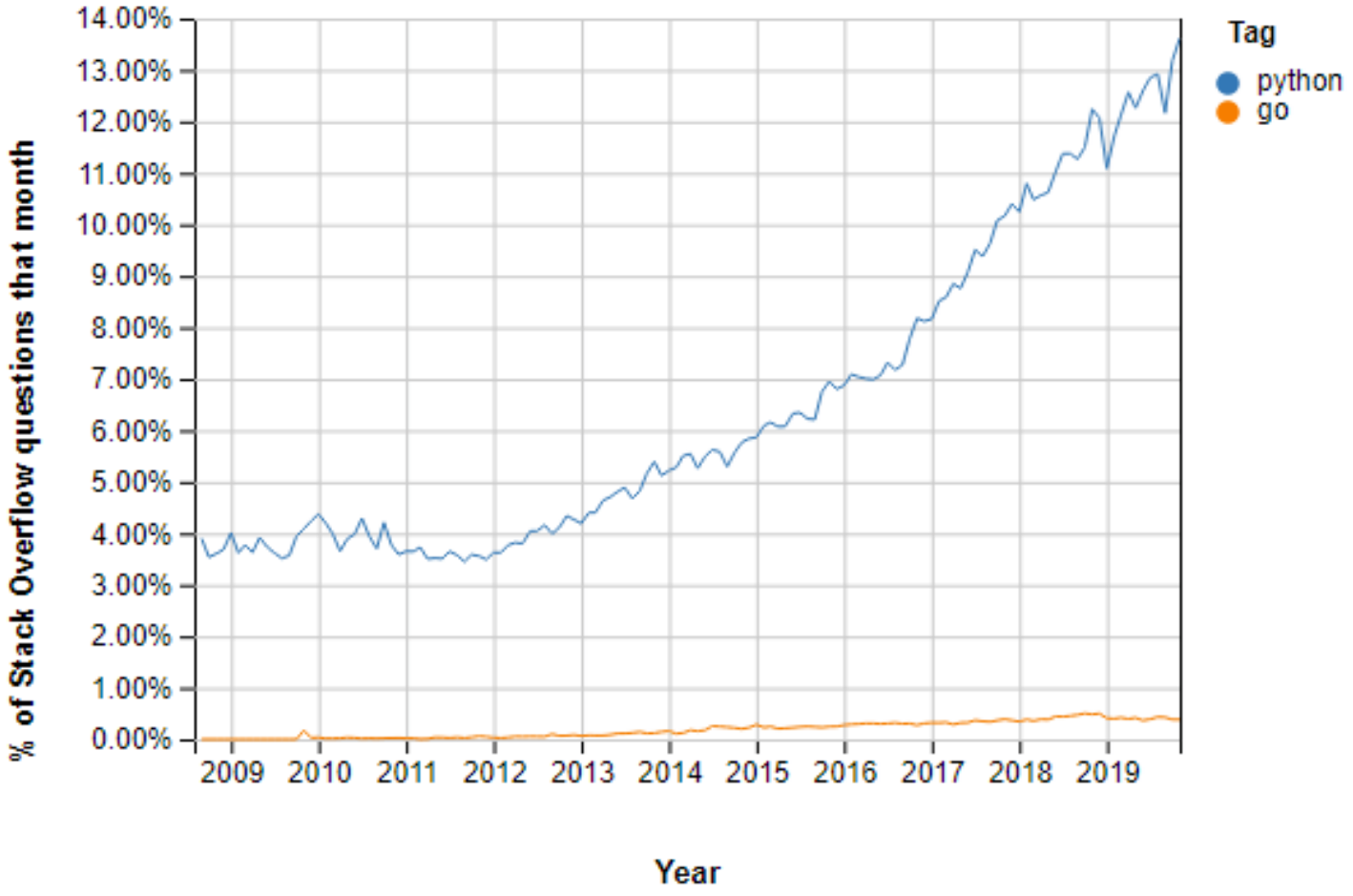
Text boundary anchors

Symbol	Matches
\A	at beginning of text
^	at beginning of text or line
$	at end of text
\z	
\b	at ASCII word boundary
\B	not at ASCII word boundary

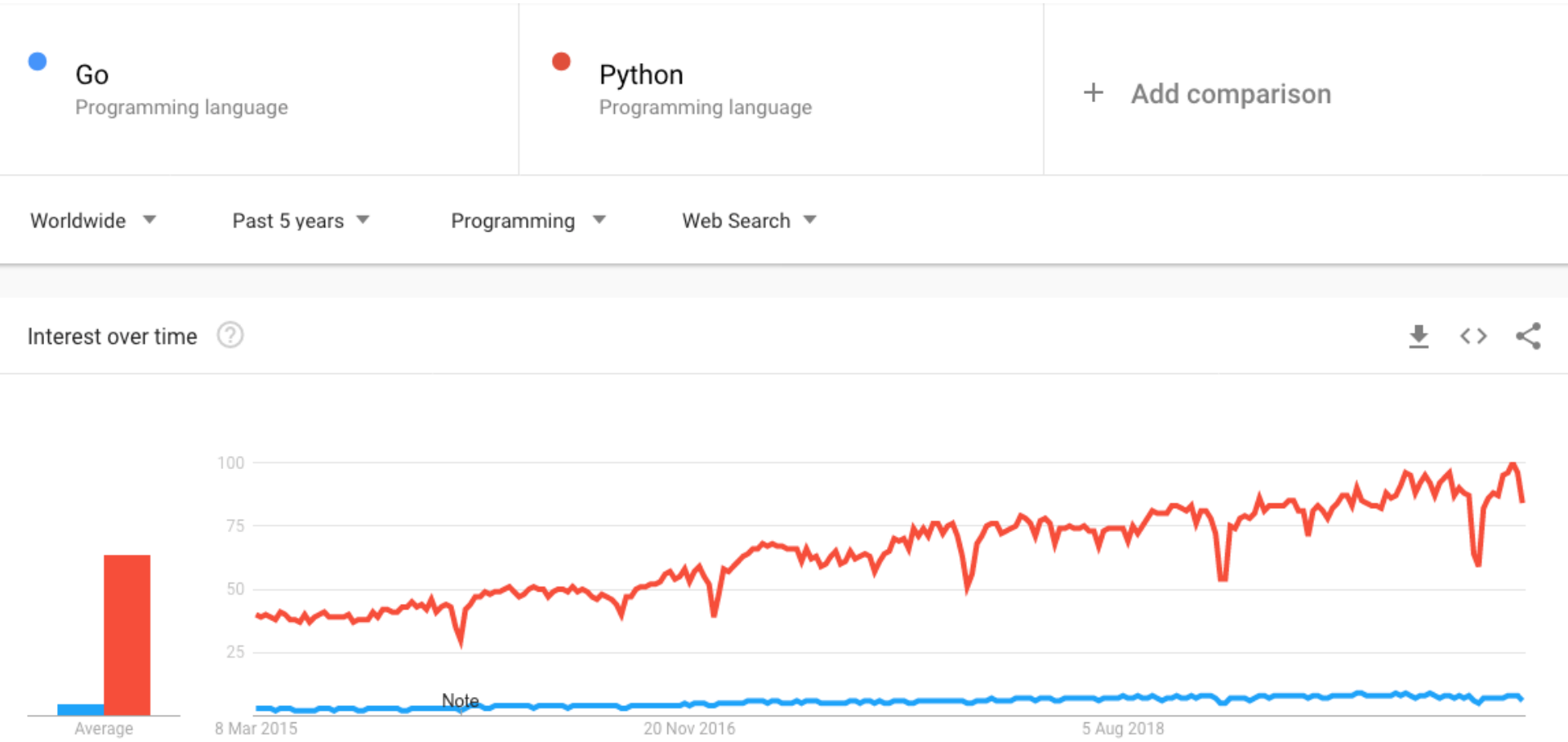
GO Community vs Python Community



GO Community vs Python Community

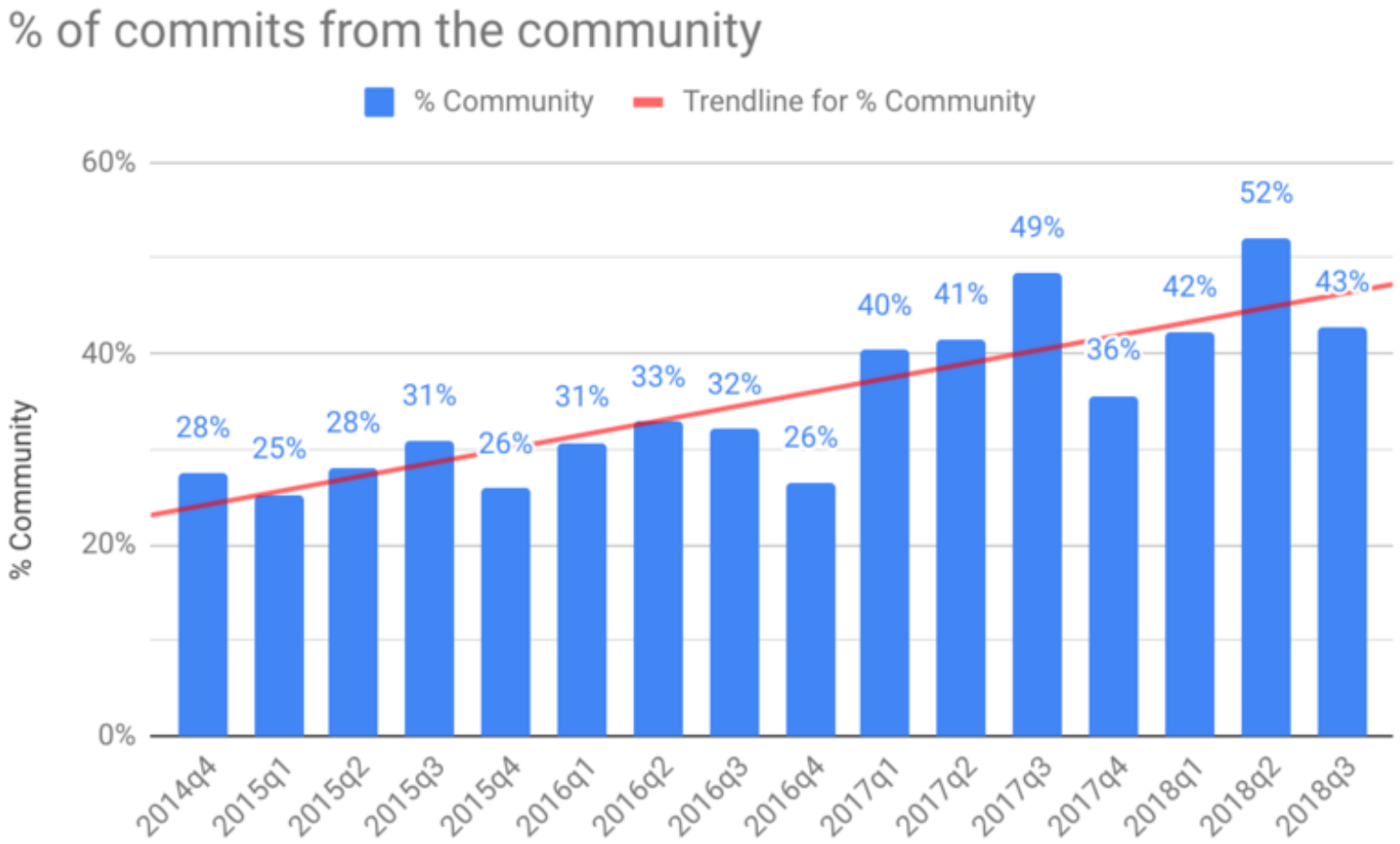


GO Community vs Python Community



GO Community vs Python Community

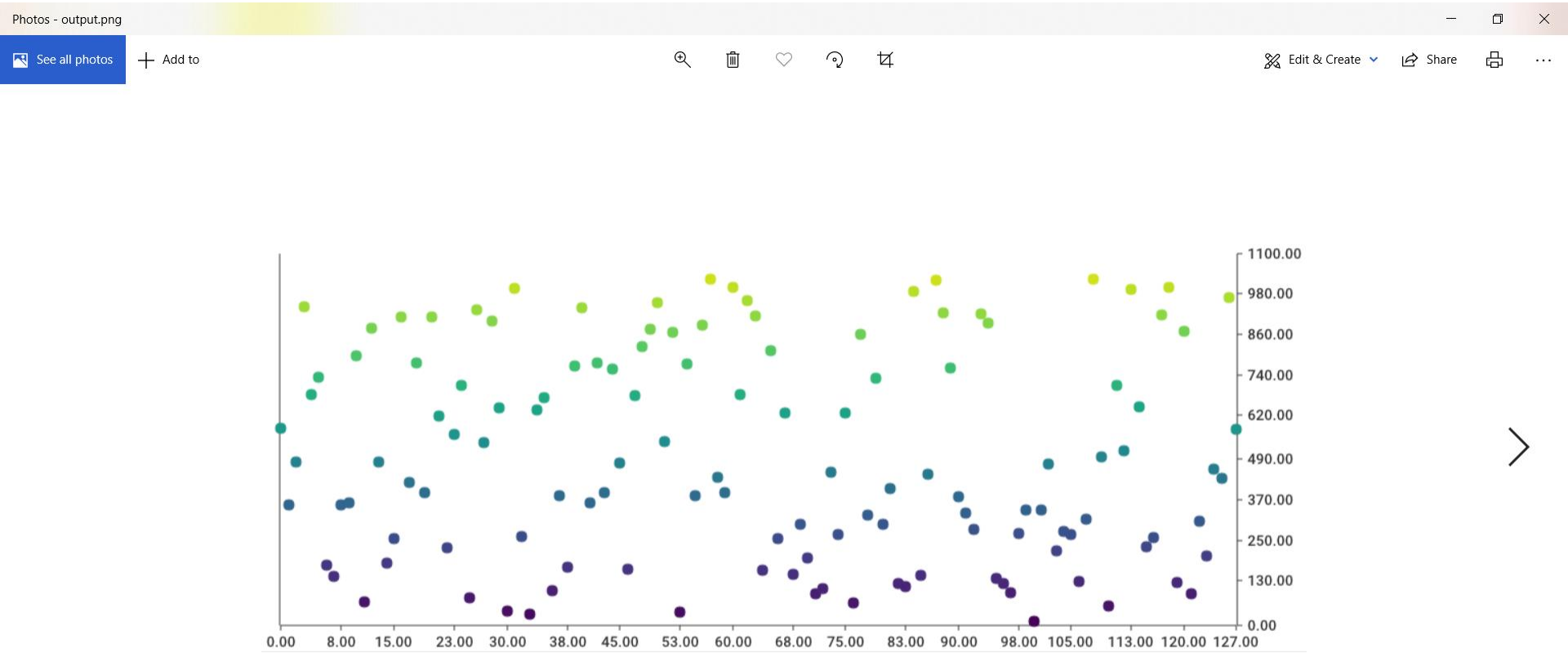
GO Community vs Python Community



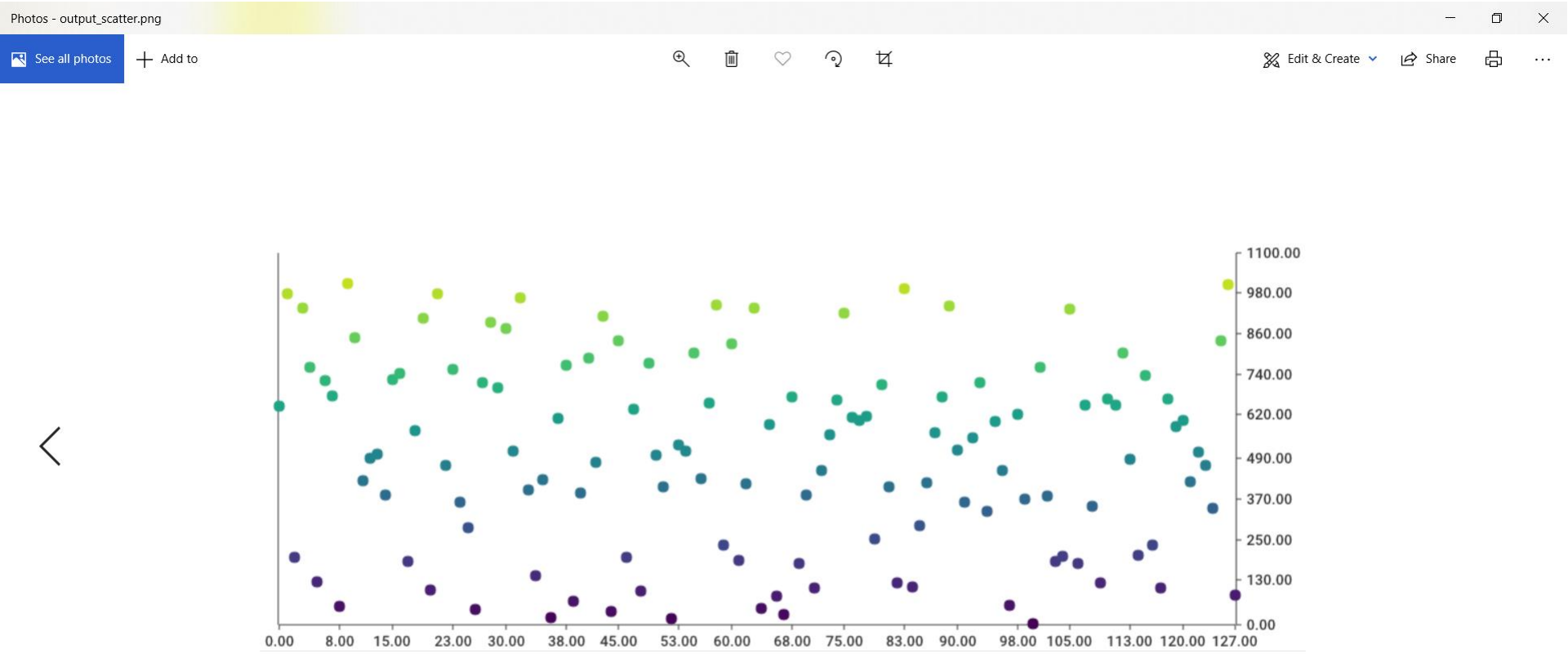
Developers can easily support Go apps

- Automatic documentation GoDoc
- Static code analysis GoMetaLinter is a meta tool
- Embedded testing environment
  - Go provides developers with a simple API that you can use for testing, profiling, and even adding your own code samples. You can easily start testing, run parallel tests, skip tests, and do much more.
- Race condition detection

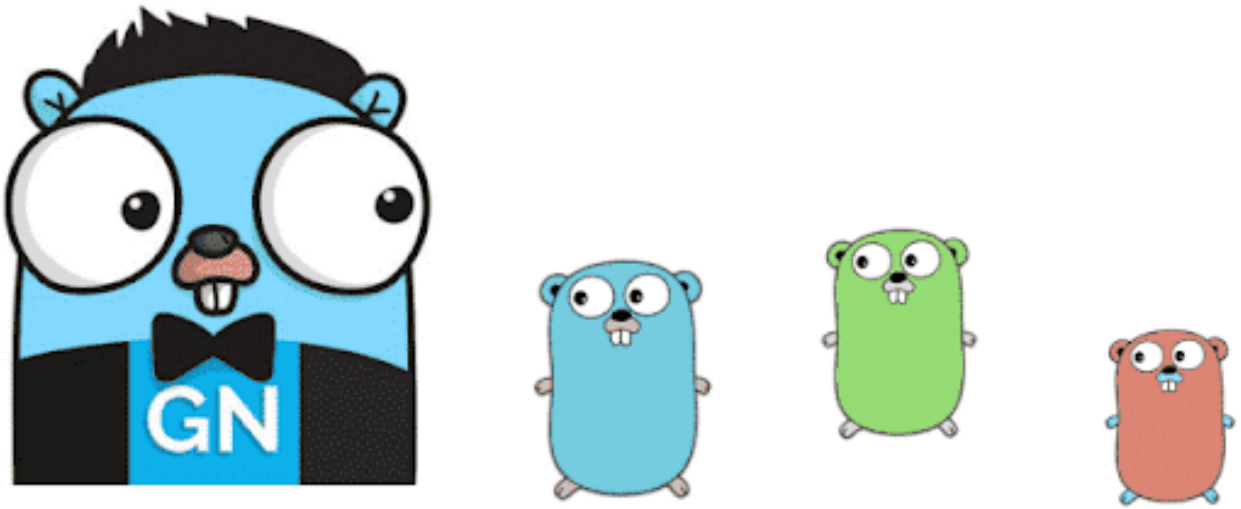
Go charts

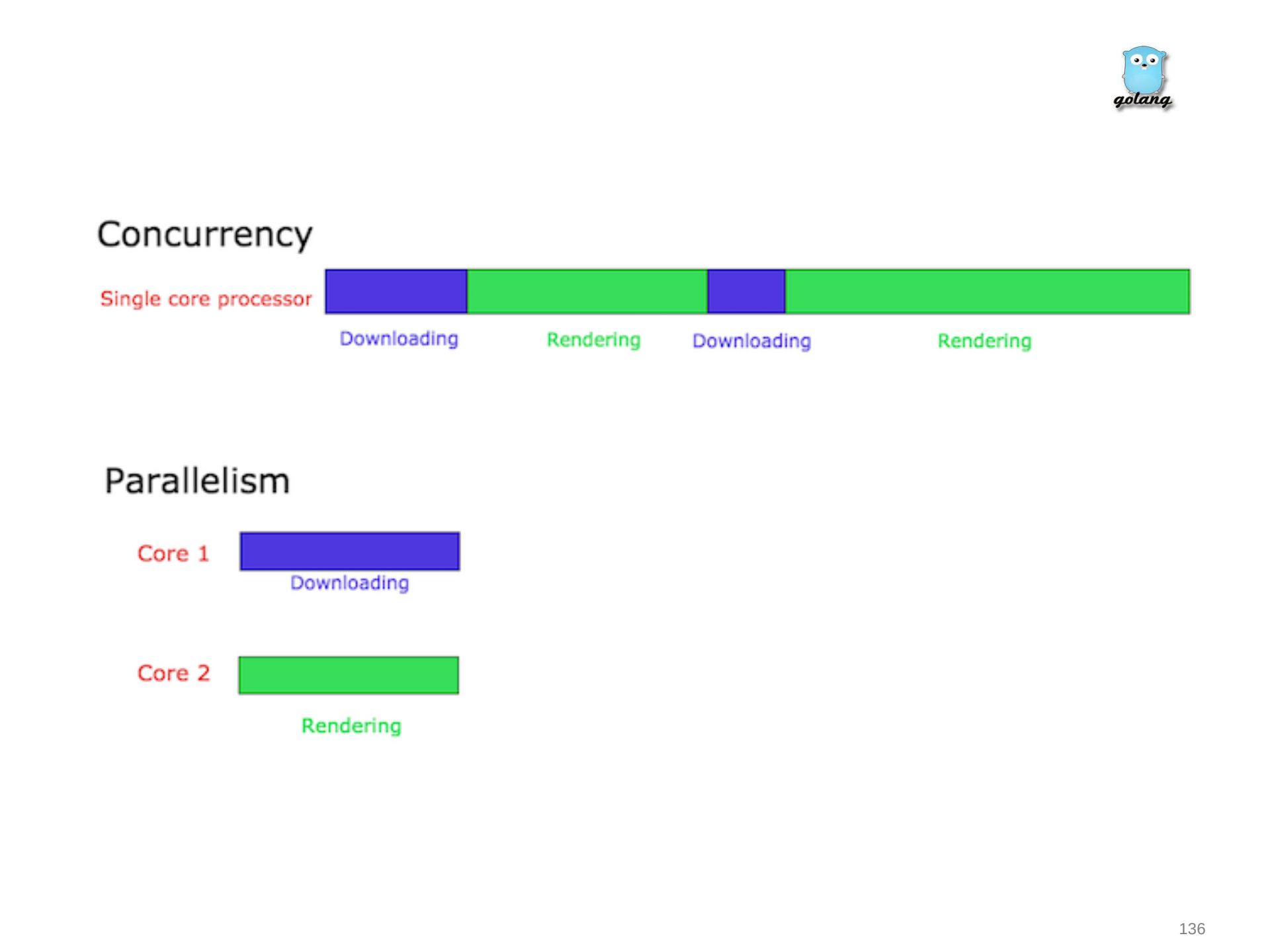

Photos - output.png
See all photos
Add to
Edit & Create
Share
1100.00
980.00
860.00
740.00
620.00
490.00
370.00
250.00
130.00
0.00
0.00
8.00
15.00
23.00
30.00
38.00
45.00
53.00
60.00
68.00
75.00
83.00
90.00
98.00
105.00
113.00
120.00
127.00

Go charts


Photos - output_scatter.png
See all photos
Add to
Edit & Create
Share
1100.00
980.00
860.00
740.00
620.00
490.00
370.00
250.00
130.00
0.00
0.00 8.00 15.00 23.00 30.00 38.00 45.00 53.00 60.00 68.00 75.00 83.00 90.00 98.00 105.00 113.00 120.00 127.00
Type here to search
ENG
22:43
20/01/2021

Go routines





Concurrency

Parallelism

Go Routines

- Goroutines are incredibly lightweight “threads” managed by the go runtime.
- They enable us to create asynchronous parallel programs that can execute some tasks far quicker than if they were written in a sequential manner.
- Goroutines are far smaller that threads, they typically take around 2kB of stack space to initialize compared to a thread which takes 1Mb
- Goroutines are typically multiplexed onto a very small number of OS threads which typically mean concurrent go programs require far less resources in order to provide the same level of performance as languages such as Java.

Go Routines

- Creating a thousand goroutines would typically require one or two OS threads at most, whereas if we were to do the same thing in java it would require 1,000 full threads each taking a minimum of 1Mb of Heap space.
- By mapping hundreds or thousands of goroutines onto a single thread we don’t have to worry about the performance hit when creating and destroying threads in our application.
- It’s incredibly in-expensive to create and destroy new goroutines due to their size and the efficient way that go handles them.

Goroutines and Threads - the differences

- Go uses goroutines while a language like Java uses threads.
- 3 factors - memory consumption, setup and teardown and switching time differentiates it from other languages.

Memory consumption

- The creation of a goroutine does not require much memory - only 2kB of stack space.
- They grow by allocating and freeing heap storage as required.
- Threads on the other hand start out at 1Mb (500 times more), along with a region of memory called a guard page that acts as a guard between one thread's memory and another.
- A server handling incoming requests can therefore create one goroutine per request without a problem, but one thread per request will eventually lead to the dreaded OutOfMemoryError.
- This isn't limited to Java - any language that uses OS threads as the primary means of concurrency will face this issue.

Setup and teardown costs

- Threads have significant setup and teardown costs because it has to request resources from the OS and return it once its done.
- The workaround to this problem is to maintain a pool of threads.
- In contrast, goroutines are created and destroyed by the runtime and those operations are pretty cheap.
- The language doesn’t support manual management of goroutines.

Switching Costs

- When a thread blocks, another has to be scheduled in its place.
- Threads are scheduled preemptively, and during a thread switch, the scheduler needs to save/restore ALL registers, that is, 16 general purpose registers, PC (Program Counter), SP (Stack Pointer), segment registers, 16 XMM registers, FP coprocessor state, 16 AVX registers, all MSRs etc.
- This is quite significant when there is rapid switching between threads.
- Goroutines are scheduled cooperatively and when a switch occurs, only 3 registers need to be saved/restored - Program Counter, Stack Pointer and DX.
- The cost is much lower.
- The number of goroutines is generally much higher, but that doesn't make a difference to switching time for two reasons.
- Only runnable goroutines are considered, blocked ones aren't.
- Also, modern schedulers are O(1) complexity, meaning switching time is not affected by the number of choices (threads or goroutines).

How goroutines are executed

- The runtime manages the goroutines throughout from creation to scheduling to teardown.
- The runtime is allocated a few threads on which all the goroutines are multiplexed.
- At any point of time, each thread will be executing one goroutine.
- If that goroutine is blocked, then it will be swapped out for another goroutine that will execute on that thread instead

How goroutines are executed

- As the goroutines are scheduled cooperatively, a goroutine that loops continuously can starve other goroutines on the same thread.
- In Go 1.2, this problem is somewhat alleviated by occasionally invoking the Go scheduler when entering a function, so a loop that includes a non-inlined function call can be prempted.

Anonymous Goroutine Functions

- package main

```
import "fmt"

func main() {
    // we make our anonymous function concurrent using `go`
    go func() {
        fmt.Println("Executing my Concurrent anonymous function")
    }()
    // we have to once again block until our anonymous goroutine
    // has finished or our main() function will complete without
    // printing anything
    fmt.Scanln()
}
```

Waiting for Goroutines to Finish Execution

- The WaitGroup type of sync package, is used to wait for the program to finish all goroutines launched from the main function.
- It uses a counter that specifies the number of goroutines, and Wait blocks the execution of the program until the WaitGroup counter is zero.
- The Add method is used to add a counter to the WaitGroup.
- The Done method of WaitGroup is scheduled using a defer statement to decrement the WaitGroup counter.
- The Wait method of the WaitGroup type waits for the program to finish all goroutines.
- The Wait method is called inside the main function, which blocks execution until the WaitGroup counter reaches the value of zero and ensures that all goroutines are executed.

Channels

- Channels provide a way for two goroutines to communicate with one another and synchronize their execution.
- A channel allows one goroutine to signal another goroutine about a particular event.
- Signaling is at the core of everything you should be doing with channels.

Channels

- Channels are type safe message queues that have the intelligence to control the behavior of any goroutine attempting to receive or send on it.
- A channel acts as a conduit between two goroutines and will synchronize the exchange of any resource that is passed through it.
- It is the channel's ability to control the goroutines interaction that creates the synchronization mechanism.
- When a channel is created with no capacity, it is called an unbuffered channel.
- In turn, a channel created with capacity is called a buffered channel.

Channel Direction

- We can specify a direction on a channel type, thus restricting it to either sending or receiving. For example, pinger's function signature can be changed to this:
- **func pinger(c chan<- string)**
- Now pinger is only allowed to send to c. Attempting to receive from c will result in a compile-time error. Similarly, we can change printer to this:
- **func printer(c <-chan string)**
- A channel that doesn't have these restrictions is known as bidirectional.
- A bidirectional channel can be passed to a function that takes send-only or receive-only channels, but the reverse is not true.

Channel Select

- Go has a special statement called select that works like a switch but for channels.

```
go func() {
    for {
        select {
        case msg1 := <- c1:
            fmt.Println(msg1)
        case msg2 := <- c2:
            fmt.Println(msg2)
        }
    }
}()
```

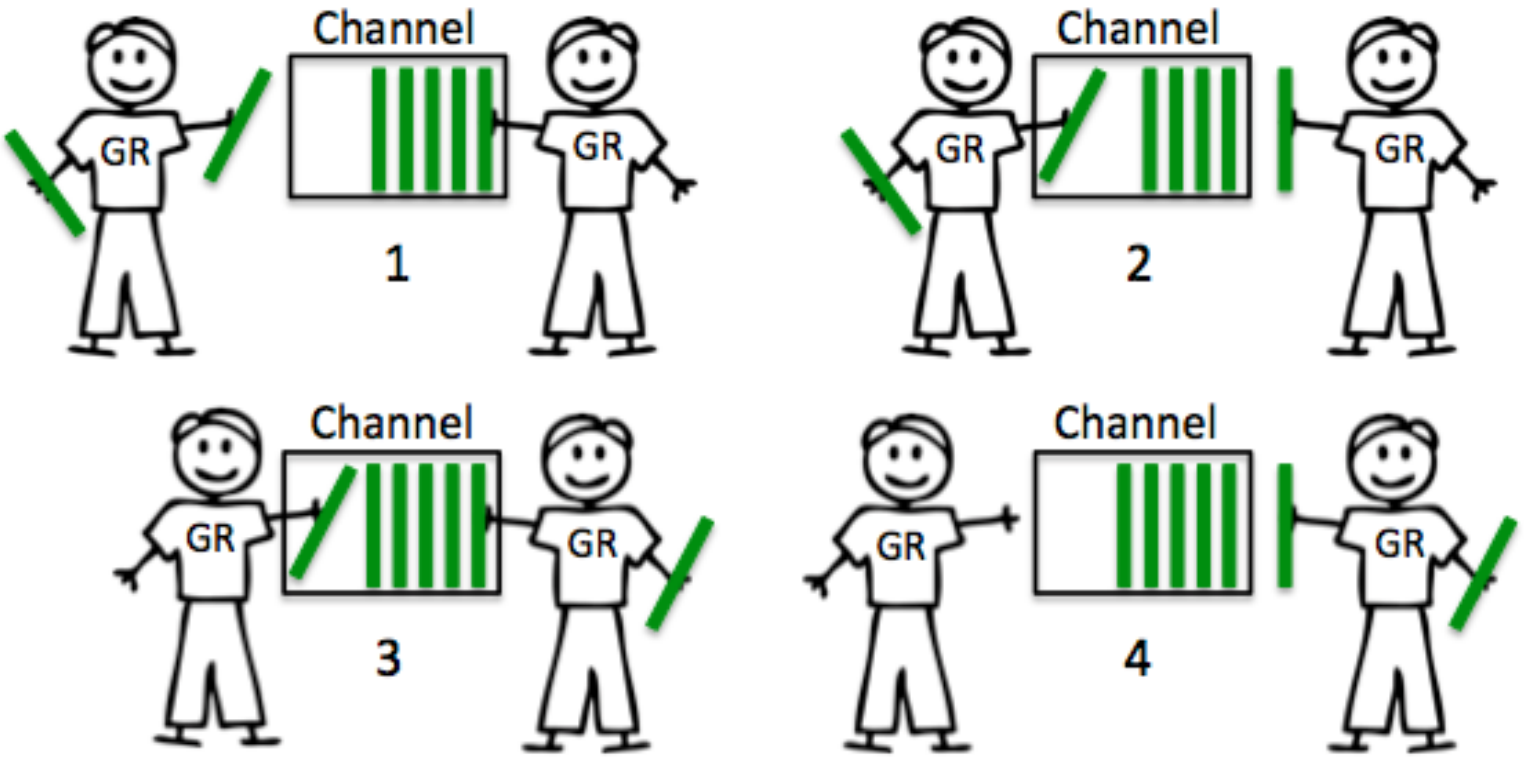
Buffered Channel

- Buffered channels have capacity and therefore can behave a bit differently.
- When a goroutine attempts to send a resource to a buffered channel and the channel is full, the channel will lock the goroutine and make it wait until a buffer becomes available.
- If there is room in the channel, the send can take place immediately and the goroutine can move on.
- When a goroutine attempts to receive from a buffered channel and the buffered channel is empty, the channel will lock the goroutine and make it wait until a resource has been sent.

Buffered Channels

- It’s also possible to pass a second parameter to the make function when creating a channel:
- c := make(chan int, **10**)
- This creates a buffered channel with a capacity of 10.
- Normally, channels are synchronous; both sides of the channel will wait until the other side is ready.
- A buffered channel is asynchronous; sending or receiving a message will not wait unless the channel is already full.
- If the channel is full, then sending will wait until there is room for at least one more int.

Buffered Channel



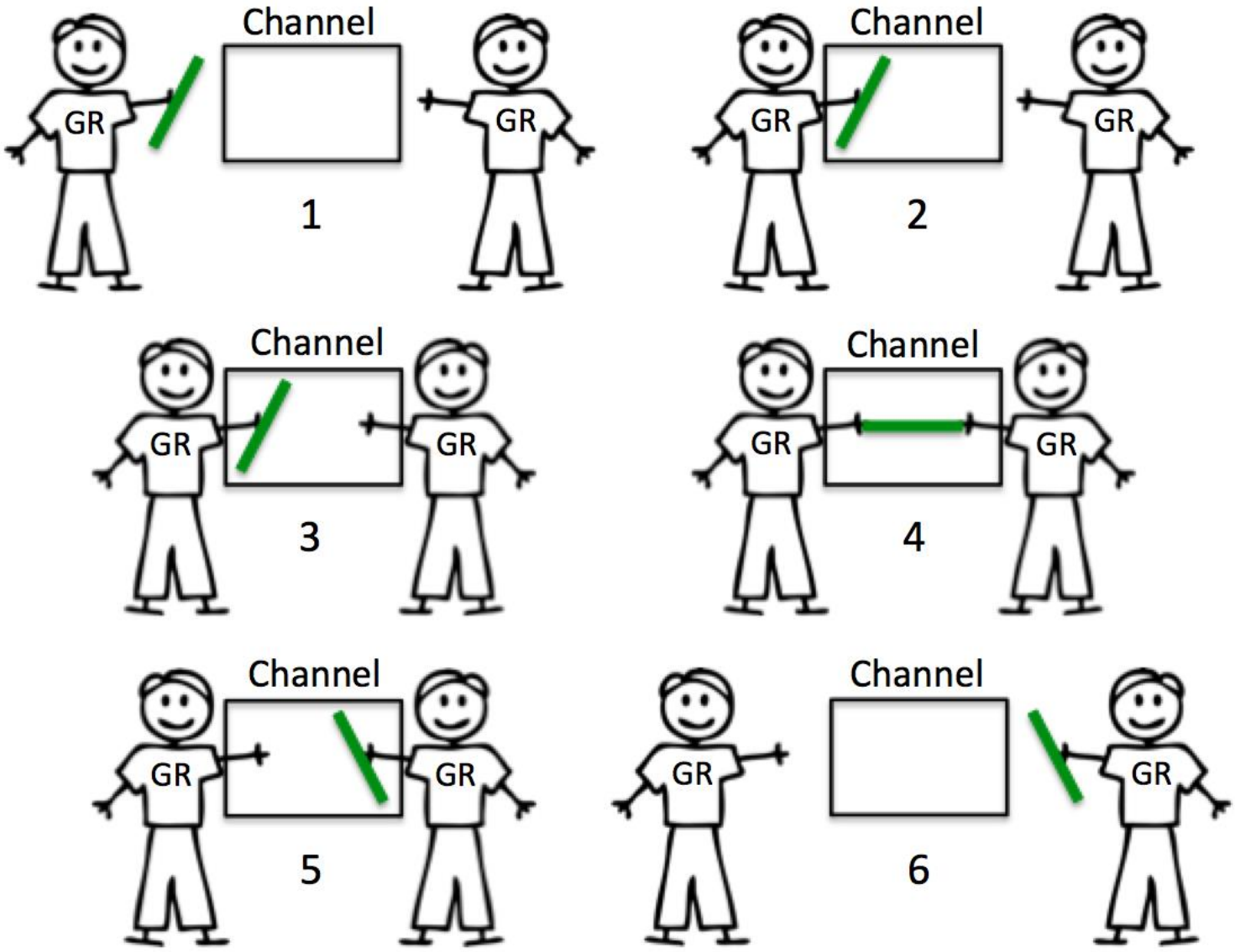
Buffered Channels

- **goroutine1 := make(chan string, 5) // Buffered channel of strings.**
- **goroutine1 <- "Australia" // Send a string through the channel.**
- **A goroutine1** channel of type string that contains a buffer of 5 values.
- Then we send the string "Australia" through the channel.

UnBuffered Channel

- An unbuffered channel is a channel that needs a receiver as soon as a message is emitted to the channel.
- To declare an unbuffered channel, you just don’t declare a capacity.
- Unbuffered channels have no capacity and therefore require both goroutines to be ready to make any exchange.
- When a goroutine attempts to send a resource to an unbuffered channel and there is no goroutine waiting to receive the resource, the channel will lock the sending goroutine and make it wait.
- When a goroutine attempts to receive from an unbuffered channel, and there is no goroutine waiting to send a resource, the channel will lock the receiving goroutine and make it wait.

UnBuffered Channel



Concurrent clock server

- Networking is a natural domain to use concurrency since servers typically handle many connections from their clients at once, each client being essentially independent of the others.
- Net package, which provides the components for building networked client and server programs that communicate over TCP, UDP, or Unix domain sockets.

Concurrent clock server

- The Listen function creates a net.Listener, an object that listens for incoming connections on a network port, in this case TCP port localhost:8000.
- The listener's Accept method blocks until an incoming connection request is made, then returns a net.Conn object representing the connection.
- The handleConn function handles one complete client connection.
- In a loop, it writes the current time, time.Now(), to the client.
- Since net.Conn satisfies the io.Writer interface, we can write directly to it.
- The loop ends when the write fails, most likely because the client has disconnected, at which point handleConn closes its side of the connection using a deferred call to Close and goes back to waiting for another connection request.

Concurrent clock server

- The time.Time.Format method provides a way to format date and time information by example.
- Its argument is a template indicating how to format a reference time, specifically Mon Jan 2 03:04:05PM 2006 UTC-0700.
- The reference time has eight components.
- Any collection of them can appear in the Format string in any order and in a number of formats; the selected components of the date and time will be displayed in the selected formats.
- This example uses the hour, minute, and second of the time.
- The time package defines templates for many standard time formats, such as time.RFC1123. The same mechanism is used in reverse when parsing a time using time.Parse.

Concurrent clock server

Administrator: Command Prompt - clockserver

```
F:\go\src\awesomeProject\Day4>cd clockserver

F:\go\src\awesomeProject\Day4\clockserver>go build

F:\go\src\awesomeProject\Day4\clockserver>dir
 Volume in drive F is New Volume
 Volume Serial Number is 5641-E892

 Directory of F:\go\src\awesomeProject\Day4\clockserver

21/01/2021  08:31 AM    <DIR>          .
21/01/2021  08:31 AM    <DIR>          ..
21/01/2021  08:31 AM         2,674,688 clockserver.exe
21/01/2021  08:28 AM               612 concurrentclockserver.
               2 File(s)      2,675,300 bytes
               2 Dir(s)  43,784,691,712 bytes free

F:\go\src\awesomeProject\Day4\clockserver>clockserver

F:\go\src\awesomeProject\Day4\clockserver>clockserver
```

Telnet localhost

```
08:32:45
        08:32:46
```

Concurrent Echo Server

Administrator: Command Prompt - echoserver

```
F:\go\src\awesomeProject\Day4\clockserver>cd..

F:\go\src\awesomeProject\Day4>cd echoserver

F:\go\src\awesomeProject\Day4\echoserver>go build

F:\go\src\awesomeProject\Day4\echoserver>dir
 Volume in drive F is New Volume
 Volume Serial Number is 5641-E892

 Directory of F:\go\src\awesomeProject\Day4\echoserver

21/01/2021  08:44 AM    <DIR>          .
21/01/2021  08:44 AM    <DIR>          ..
21/01/2021  08:44 AM               832 concurrentechoserver.go
21/01/2021  08:44 AM         2,711,552 echoserver.exe
               2 File(s)      2,712,384 bytes
               2 Dir(s)  43,774,881,792 bytes free

F:\go\src\awesomeProject\Day4\echoserver>echoserver
```

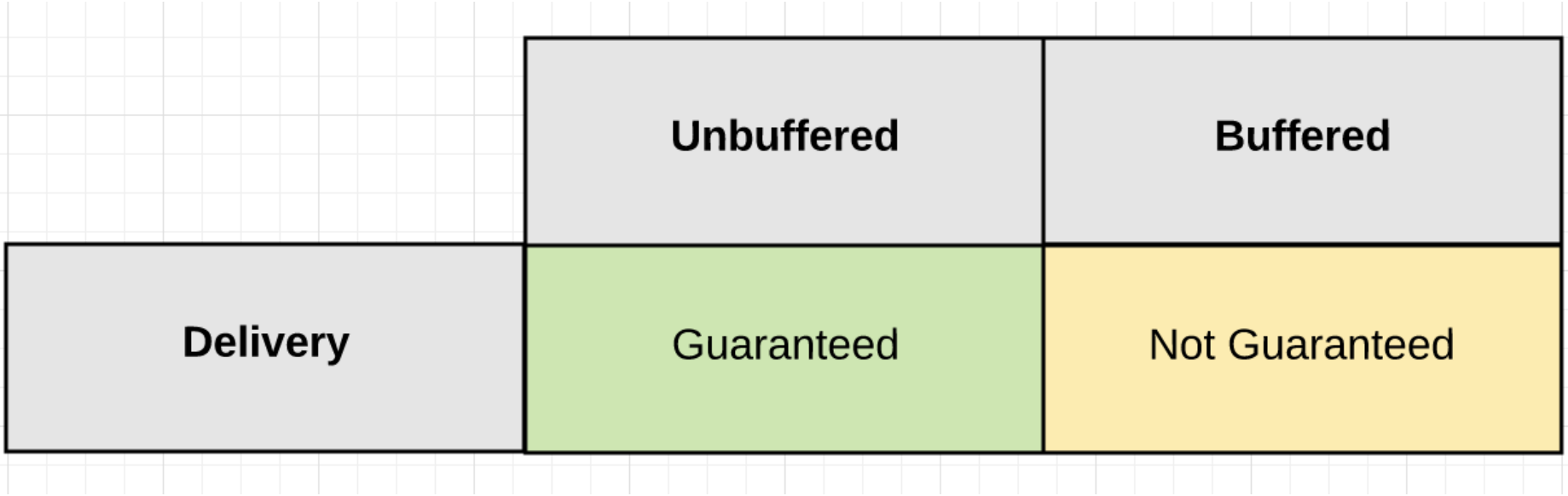
Telnet localhost

```
        HELLO
                Hello
                        hello
                             Hi
        HI
                Hi
                  ho    hi
                          w are you
        HOW ARE YOU
                        how are you
                                        how are you
                                                    done
        DONE
                done
                        done
```

Channels

- Three channel attributes
  - Guarantee Of Delivery
  - State
  - With or Without Data

Guarantee of Delivery



	Unbuffered	Buffered
Delivery	Guaranteed	Not Guaranteed

State

	NIL	Open	Closed
Send	Blocked	Allowed	Panic
Receive	Blocked	Allowed	Allowed

With and Without Data

Signaling With Data

	Guarantee	No Guarantee	Delayed Guarantee
Channel	Unbuffered	Buffered >1	Buffered =1

With and Without Data

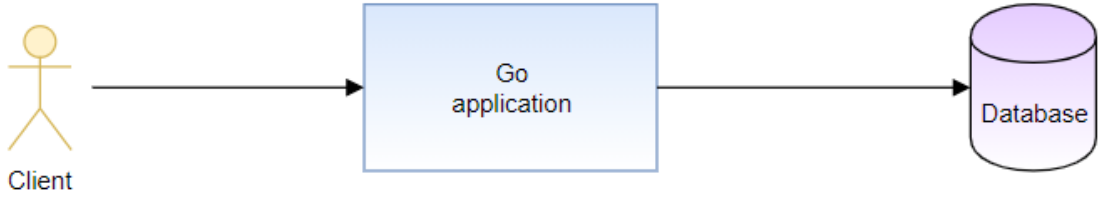
Signaling Without Data

	First Choice	Second Choice	Smell
Channel	context.Context	Unbuffered	Buffered

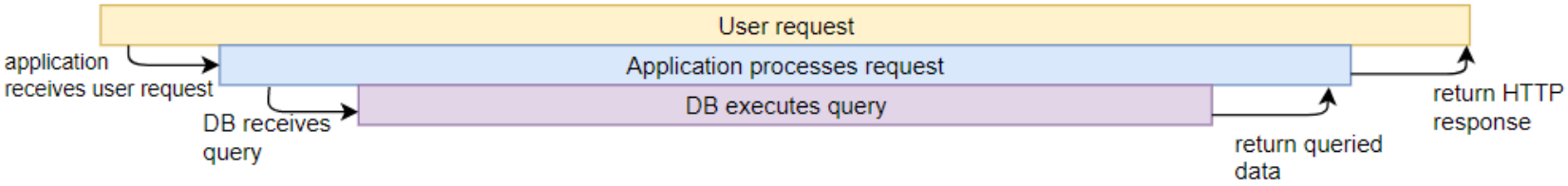
Cancellation

- Most use context with downstream operations, like making an HTTP call, or fetching data from a database, or while performing async operations with go-routines.
- It’s most common use is to pass down common data which can be used by all downstream operations.
- However, a lesser known, but highly useful feature of context is it’s ability to cancel, or halt an operation mid-way.

Cancellation

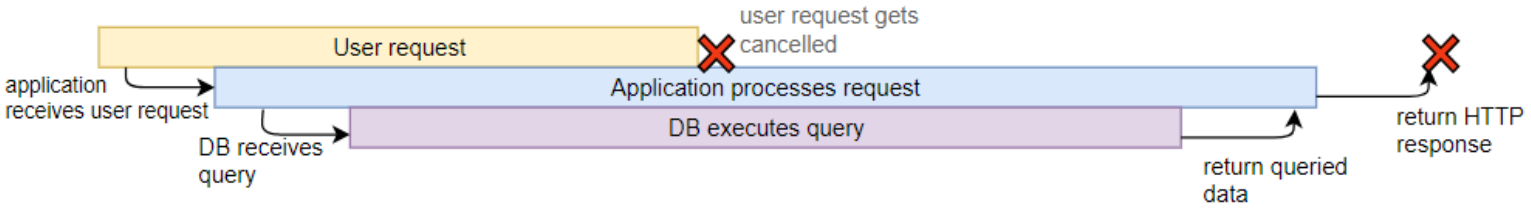


The timing diagram, if everything worked perfectly, would look like this:

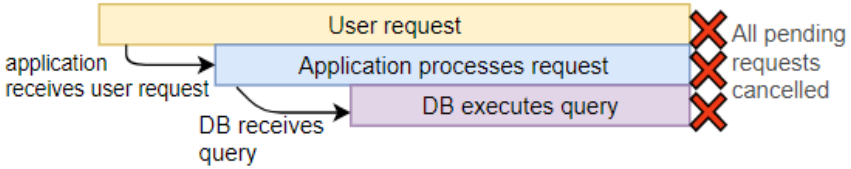


Cancellation

- if the client cancelled the request in the middle? This could happen if, for example, the client closed their browser mid-request.
- Without cancellation, the application server and database would continue to do their work, even though the result of that work would be wasted:



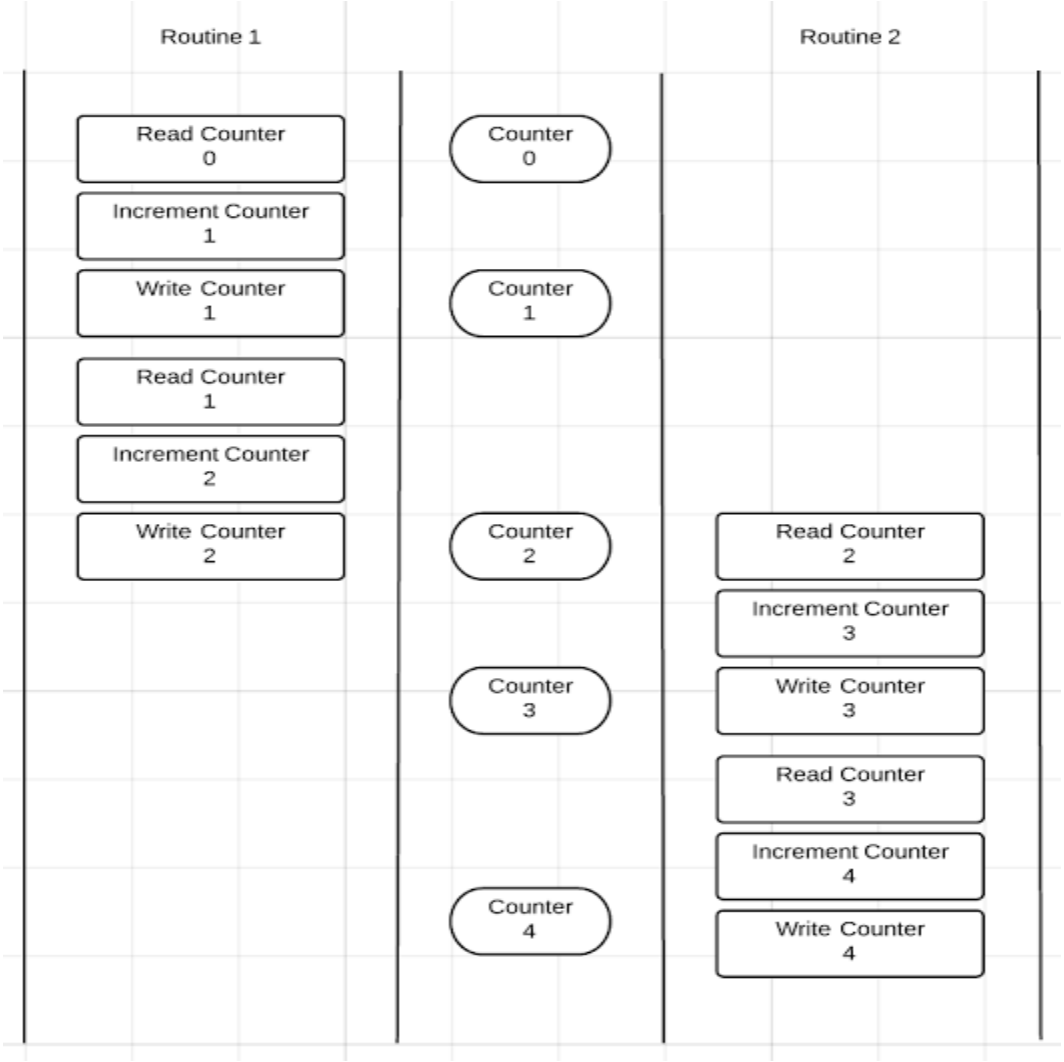
Ideally, we would want all downstream components of a process to halt, if we *know* that the process (in this example, the HTTP request) halted:



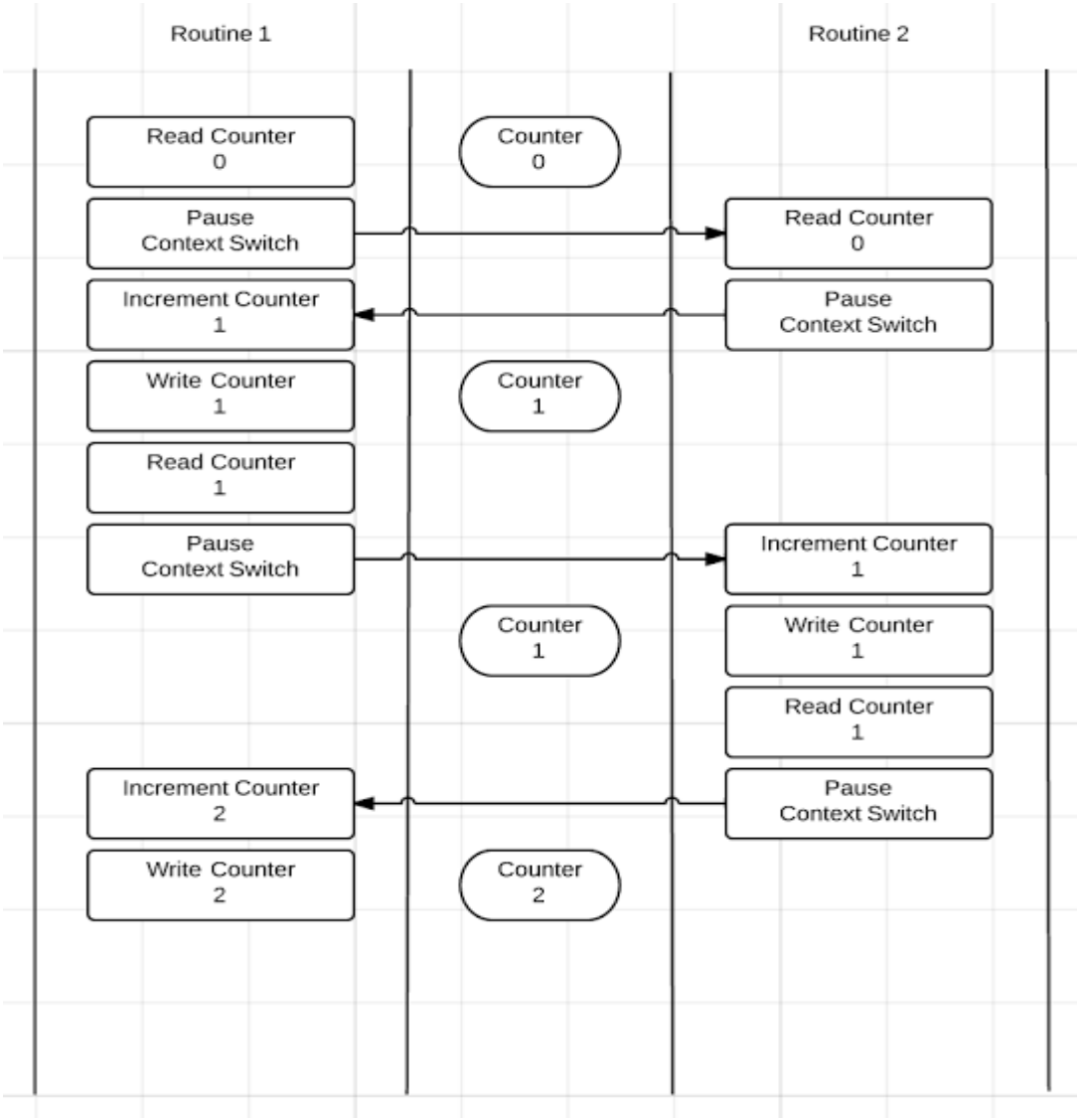
Cancellation


```
F:\go\src\awesomeProject\day5\cancellation>go run cancel_listen.go
processing request
request cancelled
exit status 2

F:\go\src\awesomeProject\day5\cancellation>
```

Race Conditions



Race Conditions



Race Condition Detector

- A race condition is when two or more routines have access to the same resource, such as a variable or data structure and attempt to read and write to that resource without any regard to the other routines.
- This type of code can create the craziest and most random bugs you have ever seen.
- It usually takes a tremendous amount of logging and luck to find these types of bugs.
- **go build -race**

Race Condition Detector

Administrator: Command Prompt

```
F:\go\src\awesomeProject\day5\raceconditions>go build --race

F:\go\src\awesomeProject\day5\raceconditions>raceconditions
==================
WARNING: DATA RACE
Read at 0x0000011d45e8 by goroutine 8:
  main.Routine()
      F:/go/src/awesomeProject/day5/raceconditions/racecondition_v1.go:27 +0x4e

Previous write at 0x0000011d45e8 by goroutine 7:
  main.Routine()
      F:/go/src/awesomeProject/day5/raceconditions/racecondition_v1.go:29 +0x6a

Goroutine 8 (running) created at:
  main.main()
      F:/go/src/awesomeProject/day5/raceconditions/racecondition_v1.go:16 +0x7c

Goroutine 7 (finished) created at:
  main.main()
      F:/go/src/awesomeProject/day5/raceconditions/racecondition_v1.go:16 +0x7c
==================
Final Counter: 4
Found 1 data race(s)

F:\go\src\awesomeProject\day5\raceconditions>
```

Race detector has pulled out the two lines of code that is reading and writing to the global Counter variable. It also identified the point in the code where the routine was spawned.

Understanding Mutexes

- Mutual exclusion is a property of concurrency control which states that " NO TWO PROCESS SHOULD ACCESS THE SAME RESOURCE AT THE SAME TIME ".

Understanding Mutexes

Instruction	Goroutine 1	Goroutine 2	Bank Balance
1	Read balance ⇐ £50		£50
2		Read balance ⇐ £50	£50
3	Add £100 to balance		—
4		Add £50 to balance	—
5	Write balance ⇒ £150		£150
6		Write balance ⇒ £100	£100

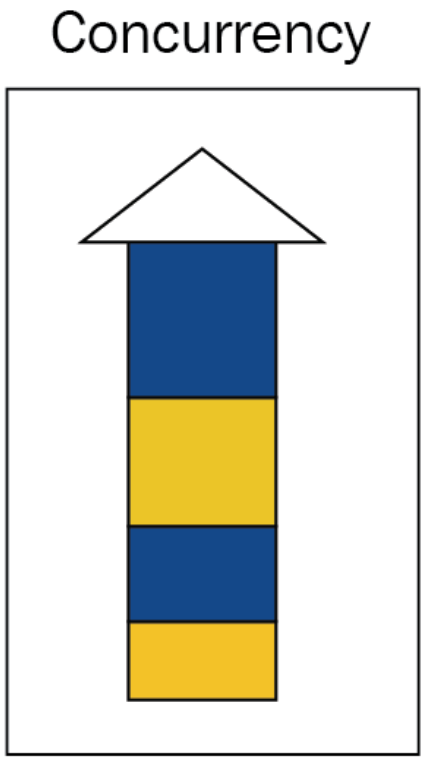
How Concurrency and Parallelism works in Golang

- Concurrency: Concurrency is about dealing with lots of things at once.
- This means that we manage to get multiple things done at once in a given period of time.
- However, we will only be doing a single thing at a time.
- This tends to happen in programs where one task is waiting and the program decides to run another task in the idle time.
- In diagram, this is denoted by running the yellow task in idle periods of the blue task.

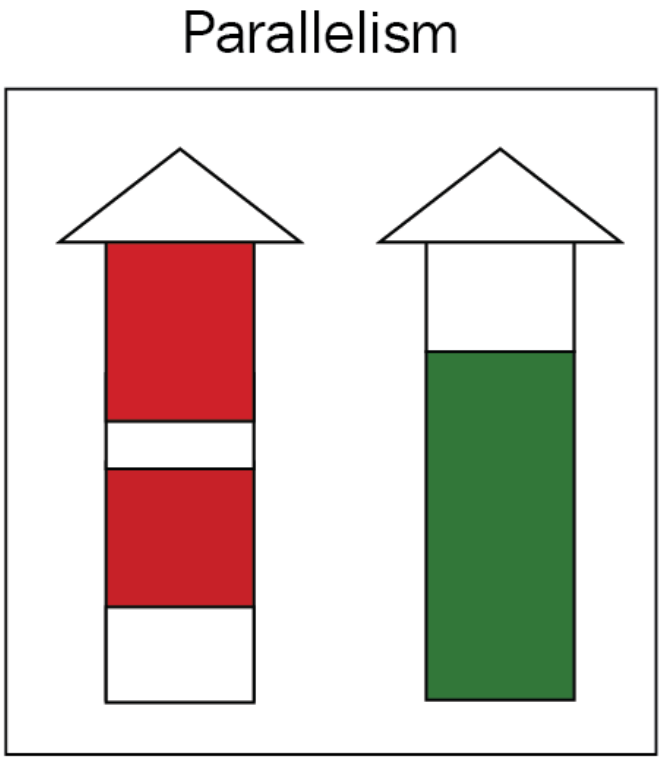
How Concurrency and Parallelism works in Golang

- Parallelism: Parallelism is about doing lots of things at once.
- This means that even if we have two tasks, they are continuously working without any breaks in between them.
- In the diagram, this is shown by the fact that the green task is running independently and is not influenced by the red task in any manner:

How Concurrency and Parallelism works in Golang



Concurrency is about *dealing with* lots of things at once



Parallelism is about *doing* lots of things at once

How Concurrency and Parallelism works in Golang

- Concurrency
- Imagine you start your day and need to get six things done:
  - Make hotel reservation
  - Book flight tickets
  - Order a dress
  - Pay credit card bills
  - Write an email
  - Listen to an audiobook

How Concurrency and Parallelism works in Golang

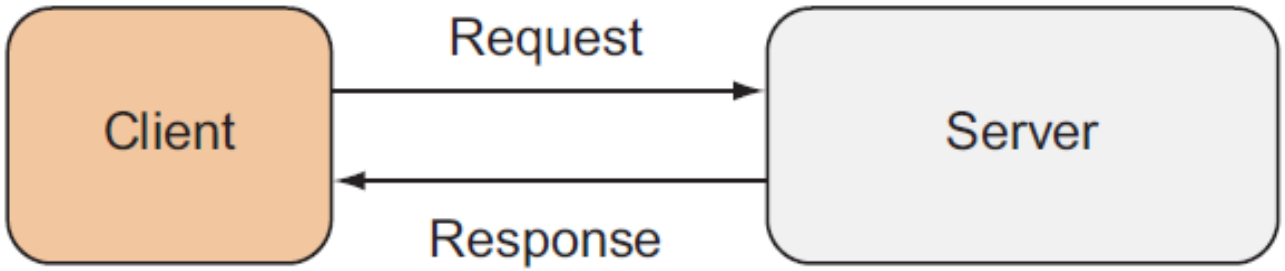
- The order in which they are completed doesn't matter, and for some of the tasks, such as writing an email or listening to an audiobook, you need not complete them in a single sitting. Here is one possible way to complete the tasks:
  - Order a dress.
  - Write one-third of the email.
  - Make hotel reservation.
  - Listen to 10 minutes of audiobook.
  - Pay credit card bills.
  - Write another one-third of the email.
  - Book flight tickets.
  - Listen to another 20 minutes of audiobook.
  - Complete writing the email.
  - Continue listening to audiobook until you fall asleep.

Web Development

Large-scale web applications typically need to be

- Scalable
- Modular
- Maintainable
- High-performance

Web Development



Parts of Web

- Handler
  - A handler receives and processes the HTTP request sent from the client.
  - It also calls the template engine to generate the HTML and finally bundles data into the HTTP response to be sent back to the client.
- Template engine
  - A template is code that can be converted into HTML that's sent back to the client in an HTTP response message.
  - Templates can be partly in HTML or not at all.
  - A template engine generates the final HTML using templates and data.

Parts of Web

- There are two types of templates with different design philosophies:
- Static templates or logic-less templates are HTML interspersed with placeholder tokens.
  - A static template engine will generate the HTML by replacing these tokens with the correct data.
  - Examples of static template engines are CTemplate and Mustache

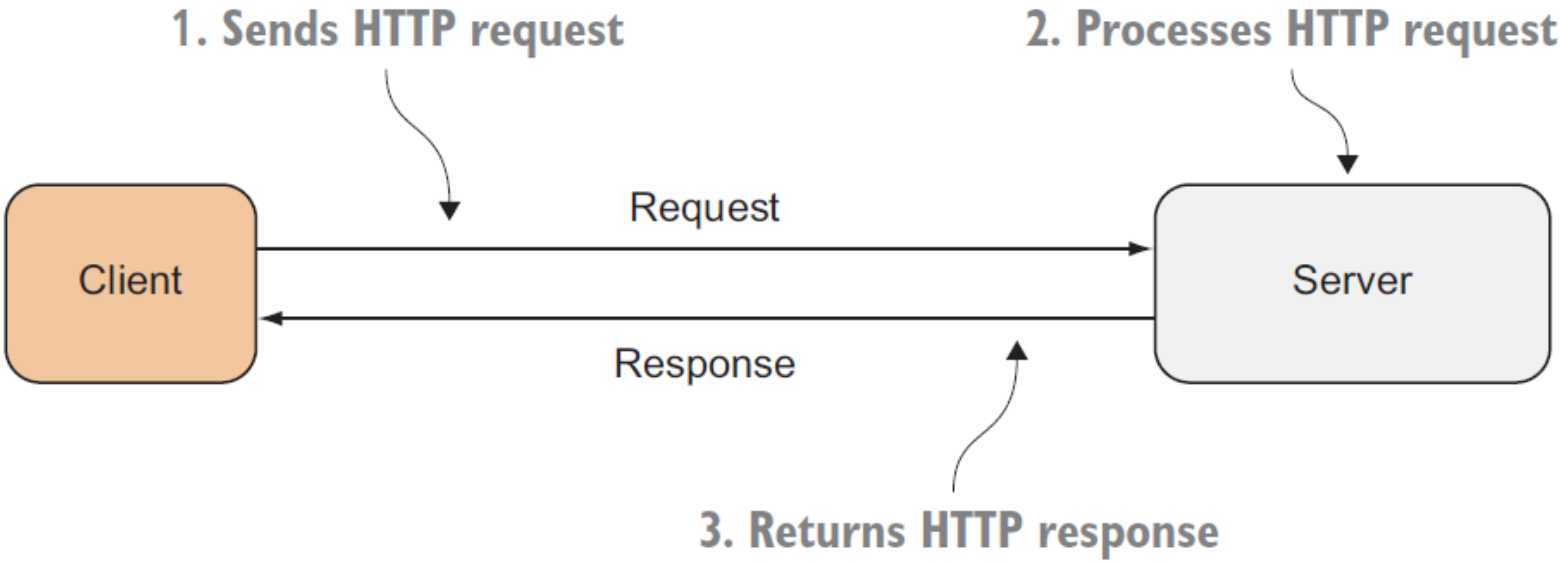
Parts of Web

- Active Templates
  - Active templates often contain HTML too, but in addition to placeholder tokens, they contain other programming language constructs like conditionals, iterators, and variables.
  - Examples of active template engines are Java ServerPages (JSP), Active Server Pages (ASP), and Embedded Ruby (ERB).
  - PHP started off as a kind of active template engine and has evolved into its own programming language.

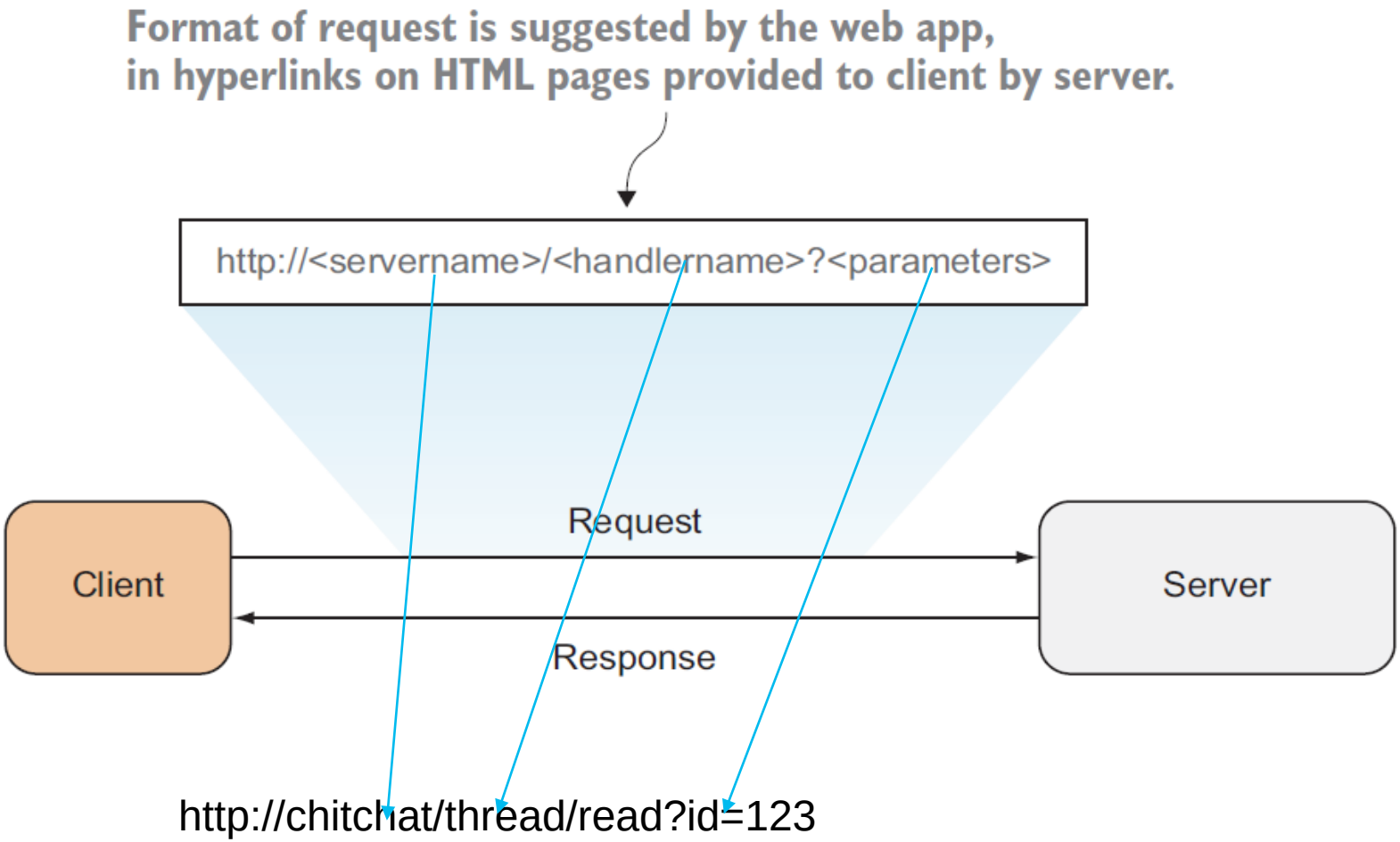
Sample App


```
package main

import (
    "fmt"
    "net/http"
)

func handler(writer http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(writer, "Hello World, %s!", request.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

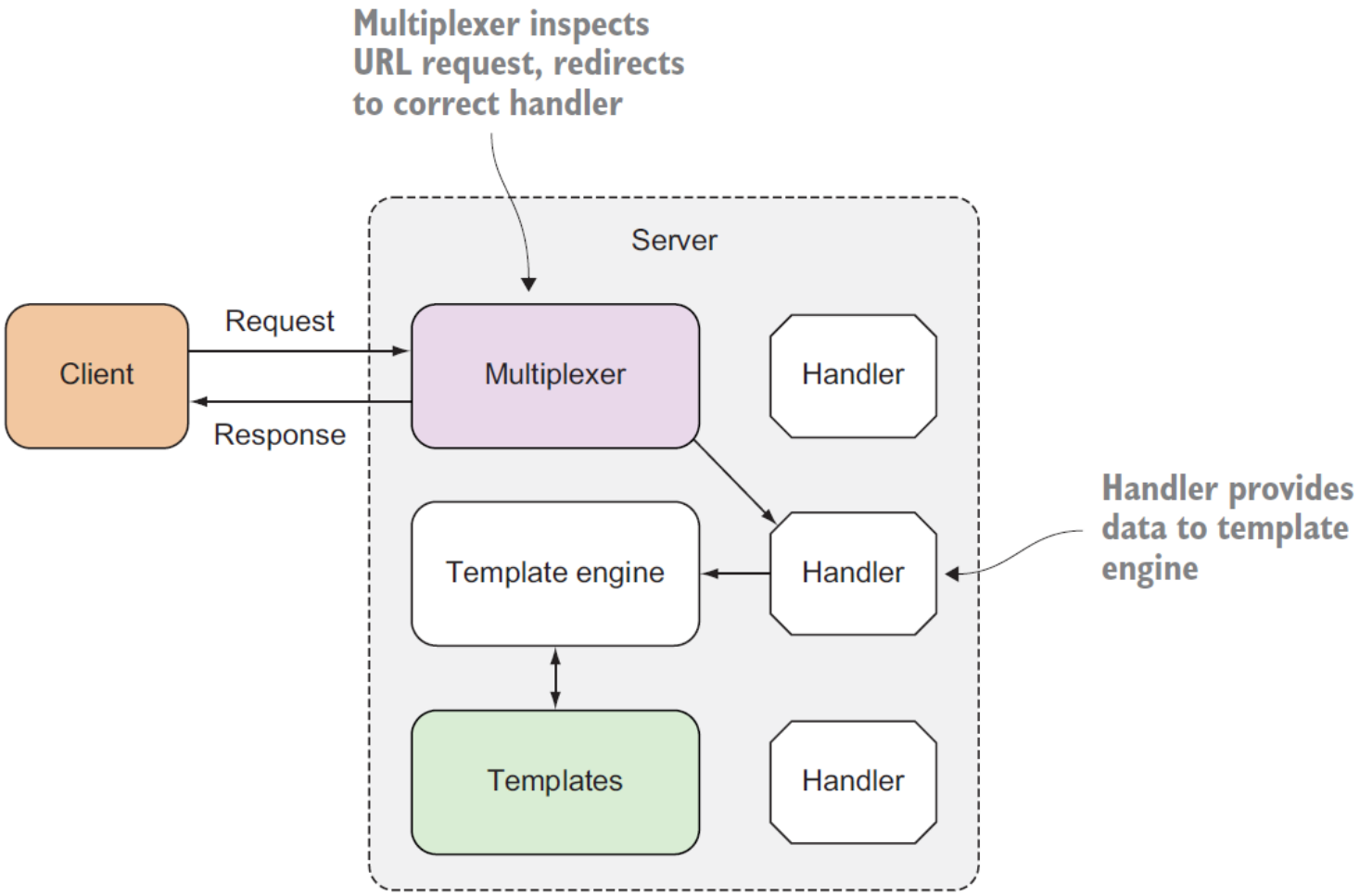
Application design



Application design



Application design



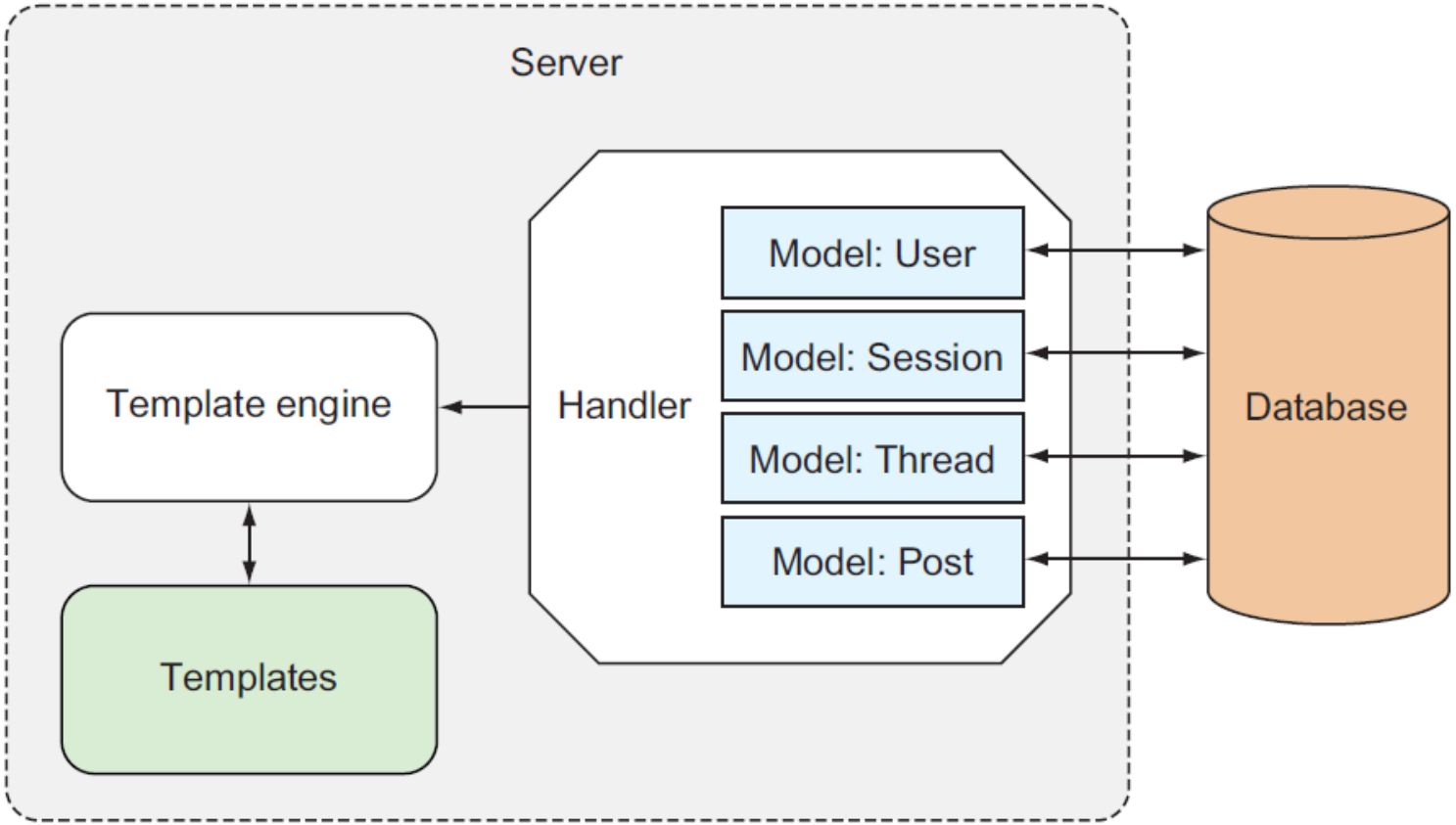
Data model

ChitChat's data model is simple and consists of only four data structures, which in turn map to a relational database.

The four data structures are

- User—Representing the forum user's information
- Session—Representing a user's current login session
- Thread—Representing a forum thread (a conversation among forum users)
- Post—Representing a post (a message added by a forum user) within a thread

Data model


Server
Handler
Model: User
Model: Session
Model: Thread
Model: Post
Template engine
Templates
Database

Basic Authentication Http Server


```
func BasicAuth(handler http.HandlerFunc, realm string) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request)
  {
    user, pass, ok := r.BasicAuth()
    if !ok || subtle.ConstantTimeCompare([]byte(user),
    []byte(ADMIN_USER)) != 1||subtle.ConstantTimeCompare([]byte(pass),
    []byte(ADMIN_PASSWORD)) != 1
    {
      w.Header().Set("WWW-Authenticate", `Basic realm="`+realm+`"`)
      w.WriteHeader(401)
      w.Write([]byte("You are Unauthorized to access the
      application.\n"))
      return
    }
    handler(w, r)
  }
}
```

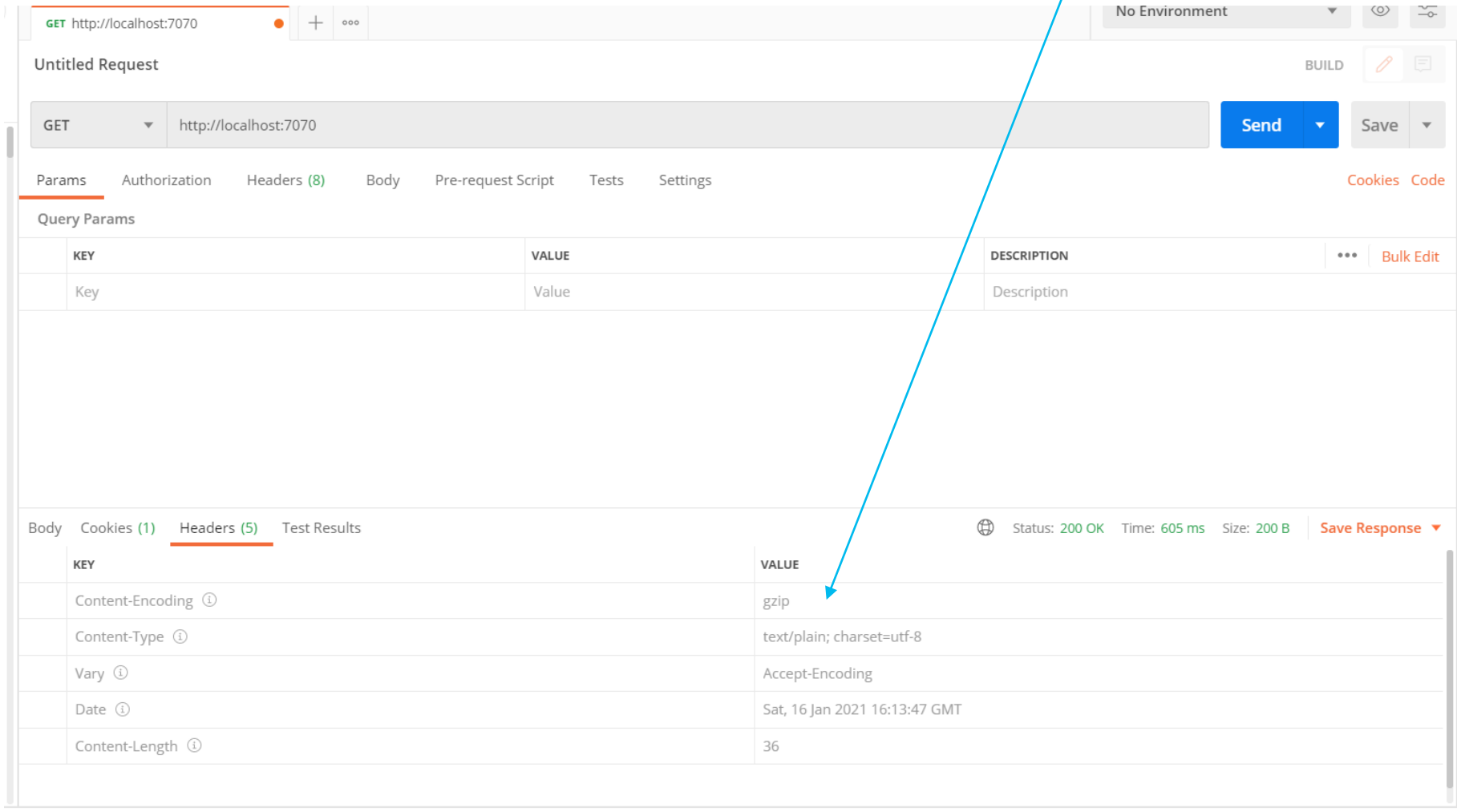
Optimizing HTTP server responses with GZIP compression

- GZIP compression means sending the response to the client from the server in a .gzip format rather than sending a plain response and it's always a good practice to send compressed responses if a client/browser supports it.
- By sending a compressed response we save network bandwidth and download time eventually rendering the page faster.
- What happens in GZIP compression is the browser sends a request header telling the server it accepts compressed content (.gzip and .deflate) and if the server has the capability to send the response in compressed form then sends it.

Optimizing HTTP server responses with GZIP compression


```
package main
import
(
  "io"
  "net/http"
  "github.com/gorilla/handlers"
)
const
(
  CONN_HOST = "localhost"
  CONN_PORT = "8080"
)
func helloWorld(w http.ResponseWriter, r *http.Request)
{
  io.WriteString(w, "Hello World!")
}
func main()
{
  mux := http.NewServeMux()
  mux.HandleFunc("/", helloWorld)
  err := http.ListenAndServe(CONN_HOST+":"+CONN_PORT,
  handlers.CompressHandler(mux))
  if err != nil
  {
    log.Fatal("error starting http server : ", err)
    return
  }
}
```

Optimizing HTTP server responses with GZIP compression



Creating a simple TCP server

- Whenever you have to build high performance oriented systems then writing a TCP server is always the best choice over an HTTP server, as TCP sockets are less hefty than HTTP.
- Go supports and provides a convenient way of writing TCP servers using a net package.

Creating a simple TCP server


```
package main
import
(
  "log"
  "net"
)
const
(
  CONN_HOST = "localhost"
  CONN_PORT = "8080"
  CONN_TYPE = "tcp"
)
func main()
{
  listener, err := net.Listen(CONN_TYPE, CONN_HOST+":"+CONN_PORT)
  if err != nil
  {
    log.Fatal("Error starting tcp server : ", err)
  }
  defer listener.Close()
  log.Println("Listening on " + CONN_HOST + ":" + CONN_PORT)
  for
  {
    conn, err := listener.Accept()
    if err != nil
    {
      log.Fatal("Error accepting: ", err.Error())
    }
    log.Println(conn)
  }
}
```

TCP server read data from incoming connections


```
func handleRequest(conn net.Conn)
{
  message, err := bufio.NewReader(conn).ReadString('\n')
  if err != nil
  {
    fmt.Println("Error reading:", err.Error())
  }
  fmt.Print("Message Received from the client: ", string(message))
  conn.Close()
}
```

Implementing HTTP request routing


```
func login(w http.ResponseWriter, r *http.Request)
{
  fmt.Fprintf(w, "Login Page!")
}
func logout(w http.ResponseWriter, r *http.Request)
{
  fmt.Fprintf(w, "Logout Page!")
}
func main()
{
  http.HandleFunc("/", helloWorld)
  http.HandleFunc("/login", login)
  http.HandleFunc("/logout", logout)
  err := http.ListenAndServe(CONN_HOST+":"+CONN_PORT, nil)
  if err != nil
  {
    log.Fatal("error starting http server : ", err)
    return
  }
}
```

Implementing HTTP request routing using Gorilla Mux


```
var GetRequestHandler = http.HandlerFunc
(
  func(w http.ResponseWriter, r *http.Request)
  {
    w.Write([]byte("Hello World!"))
  }
)
var PostRequestHandler = http.HandlerFunc
(
  func(w http.ResponseWriter, r *http.Request)
  {
    w.Write([]byte("It's a Post Request!"))
  }
)
var PathVariableHandler = http.HandlerFunc
(
  func(w http.ResponseWriter, r *http.Request)
  {
    vars := mux.Vars(r)
    name := vars["name"]
    w.Write([]byte("Hi " + name))
  }
)
```

Working with Templates, Static Files, and HTML Forms

- Creating your first template
- Serving static files over HTTP
- Serving static files over HTTP using Gorilla Mux
- Creating your first HTML form
- Reading your first HTML form
- Validating your first HTML form
- Uploading your first file

Template file Parsing


```
Microsoft Windows [Version 10.0.19041.746]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd F:\go\src\awesomeProject\webmaster\first-template

C:\WINDOWS\system32>f:

F:\go\src\awesomeProject\webmaster\first-template>dir
 Volume in drive F is New Volume
 Volume Serial Number is 5641-E892

 Directory of F:\go\src\awesomeProject\webmaster\first-template

16/01/2021  11:33 PM    <DIR>          .
16/01/2021  11:33 PM    <DIR>          ..
16/01/2021  11:33 PM               675 first-template.go
16/01/2021  11:21 PM    <DIR>          templates
               1 File(s)            675 bytes
               3 Dir(s)  43,791,962,112 bytes free

F:\go\src\awesomeProject\webmaster\first-template>go run first-template.go
```

VEB DENTAL CARE PROJECT

Go run .

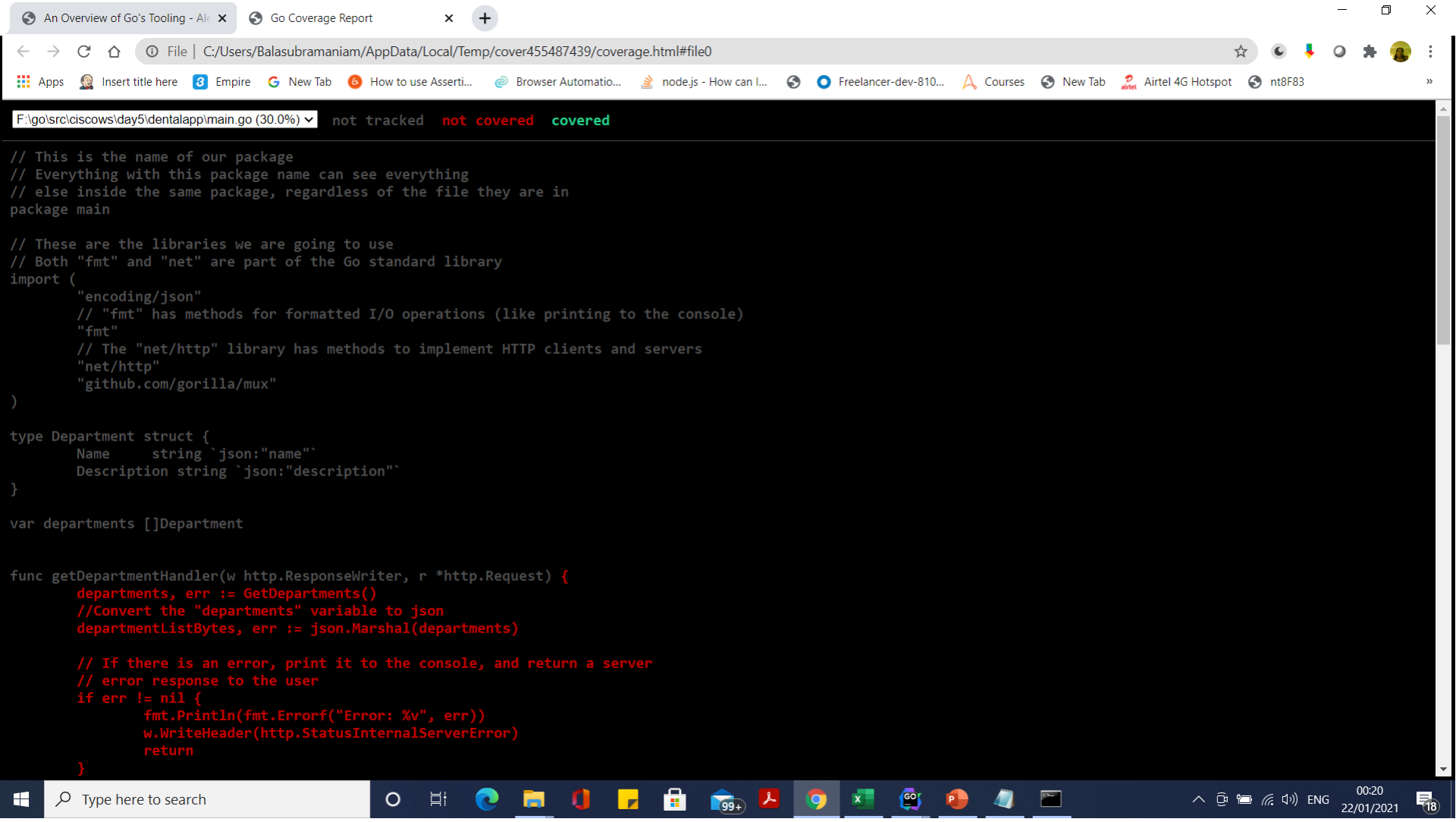
Go coverage

- go test -cover ./...
- go test -coverprofile=profile.out ./...
- go tool cover –html=profile.out
- If you want you can go a step further and set the -covermode=count flag to make the coverage profile record the exact number of times that each statement is executed during the tests.
- go test -covermode=count -coverprofile=profile.out ./...
- go tool cover -html=profile.out

Go Tool

Go Tool cheatsheet

Go coverage


An Overview of Go's Tooling - Al
Go Coverage Report
File | C:/Users/Balasubramaniam/AppData/Local/Temp/cover455487439/coverage.html#file0
Apps
Insert title here
Empire
New Tab
How to use Asserti...
Browser Automatio...
node.js - How can I...
Freelancer-dev-810...
Courses
New Tab
Airtel 4G Hotspot
nt8F83
F:\go\src\ciscows\day5\dentalapp\main.go (30.0%)
not tracked
not covered
covered
// This is the name of our package
// Everything with this package name can see everything
// else inside the same package, regardless of the file they are in
package main
// These are the libraries we are going to use
// Both "fmt" and "net" are part of the Go standard library
import (
"encoding/json"
// "fmt" has methods for formatted I/O operations (like printing to the console)
"fmt"
// The "net/http" library has methods to implement HTTP clients and servers
"net/http"
"github.com/gorilla/mux"
)
type Department struct {
Name string `json:"name"`
Description string `json:"description"`
}
var departments []Department
func getDepartmentHandler(w http.ResponseWriter, r *http.Request) {
departments, err := GetDepartments()
//Convert the "departments" variable to json
departmentListBytes, err := json.Marshal(departments)
// If there is an error, print it to the console, and return a server
// error response to the user
if err != nil {
fmt.Println(fmt.Errorf("Error: %v", err))
w.WriteHeader(http.StatusInternalServerError)
return
}
Type here to search
ENG
00:20
22/01/2021

Go Test Tool


```
Microsoft Windows [Version 10.0.19041.746]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd F:\go\src\ciscows\day5\dentalapp

C:\WINDOWS\system32>f:

F:\go\src\ciscows\day5\dentalapp>go test ./
ok      _/F_/go/src/ciscows/day5/dentalapp      1.040s

F:\go\src\ciscows\day5\dentalapp>
```

Test all dependencies before release

Administrator: Command Prompt

```
F:\go\src\ciscows\day5\dentalapp>go test all
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\apexlogs\apexlogs.go:13:2: cannot find package "github.com/apex/logs" in a
ny of:
        D:\Go\src\github.com\apex\logs (from $GOROOT)
        C:\Users\Balasubramaniam\go\src\github.com\apex\logs (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\apexlogs\apexlogs.go:10:2: cannot find package "github.com/tj/go-buffer" i
n any of:
        D:\Go\src\github.com\tj\go-buffer (from $GOROOT)
        C:\Users\Balasubramaniam\go\src\github.com\tj\go-buffer (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\cli\cli.go:12:2: cannot find package "github.com/fatih/color" in any of:
        D:\Go\src\github.com\fatih\color (from $GOROOT)
        C:\Users\Balasubramaniam\go\src\github.com\fatih\color (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\delta\delta.go:13:2: cannot find package "github.com/aybabtme/rgbterm" in
any of:
        D:\Go\src\github.com\aybabtme\rgbterm (from $GOROOT)
        C:\Users\Balasubramaniam\go\src\github.com\aybabtme\rgbterm (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\delta\delta.go:14:2: cannot find package "github.com/tj/go-spin" in any of
:
        D:\Go\src\github.com\tj\go-spin (from $GOROOT)
        C:\Users\Balasubramaniam\go\src\github.com\tj\go-spin (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\es\es.go:11:2: cannot find package "github.com/tj/go-elastic/batch" in any
 of:
        D:\Go\src\github.com\tj\go-elastic\batch (from $GOROOT)
        C:\Users\Balasubramaniam\go\src\github.com\tj\go-elastic\batch (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\graylog\graylog.go:6:2: cannot find package "github.com/aphistic/golf" in
any of:
        D:\Go\src\github.com\aphistic\golf (from $GOROOT)
        C:\Users\Balasubramaniam\go\src\github.com\aphistic\golf (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\kinesis\kinesis.go:8:2: cannot find package "github.com/aws/aws-sdk-go/aws
" in any of:
        D:\Go\src\github.com\aws\aws-sdk-go\aws (from $GOROOT)
        C:\Users\Balasubramaniam\go\src\github.com\aws\aws-sdk-go\aws (from $GOPATH)
C:\Users\Balasubramaniam\go\src\github.com\apex\log\handlers\kinesis\kinesis.go:9:2: cannot find package "github.com/aws/aws-sdk-go/aws
```

Type here to search

ENG 00:23 22/01/2021

Performing Static Analysis

Distribution List

Administrator: Command Prompt

```
F:\go\src\ciscows\day5\dentalapp>go tool dist list
aix/ppc64
android/386
android/amd64
android/arm
android/arm64
darwin/amd64
darwin/arm64
dragonfly/amd64
freebsd/386
freebsd/amd64
freebsd/arm
freebsd/arm64
illumos/amd64
js/wasm
linux/386
linux/amd64
linux/arm
linux/arm64
linux/mips
linux/mips64
linux/mips64le
linux/mipsle
linux/ppc64
linux/ppc64le
linux/riscv64
linux/s390x
netbsd/386
netbsd/amd64
netbsd/arm
netbsd/arm64
openbsd/386
```

Diagnosing Problems and Making Optimizations

- A nice feature of Go is that it makes it easy to benchmark your code.
- If you're not familiar with the general process for writing benchmarks there are good guides here and here.
- To run benchmarks you'll need to use the go test tool, with the -bench flag set to a regular expression that matches the benchmarks you want to execute

Diagnosing Problems and Making Optimizations

Administrator: Command Prompt

```
F:\go\src\ciscows\day5\dentalapp>go test -bench=. ./...
PASS
ok      _/F_/go/src/ciscows/day5/dentalapp      1.251s

F:\go\src\ciscows\day5\dentalapp>
```

Administrator: Command Prompt

```
F:\go\src\ciscows\day5\dentalapp>go test -bench=. ./...
PASS
ok      _/F_/go/src/ciscows/day5/dentalapp      1.251s

F:\go\src\ciscows\day5\dentalapp>go test -bench=. -benchmem ./...
PASS
ok      _/F_/go/src/ciscows/day5/dentalapp      0.251s

F:\go\src\ciscows\day5\dentalapp>
```

-benchmem flag, which forces memory allocation statistics to be included in the output.

Profiling

- Go makes it possible to create diagnostic profiles for CPU use, memory use, goroutine blocking and mutex contention.
- You can use these to dig a bit deeper and see exactly how your application is using (or waiting on) resources.

Profiling

- There are three ways to generate profiles:
- If you have a web application you can import the net/http/pprof package.
- This will register some handlers with the http.DefaultServeMux which you can then use to generate and download profiles for your running application.
- This post provides a good explanation and some sample code.
- For other types of applications, you can profile your running application using the pprof.StartCPUProfile() and pprof.WriteHeapProfile() functions. See the runtime/pprof documentation for sample code.
- Or you can generate profiles while running benchmarks or tests by using the various -***profile flags like so:

Profiling

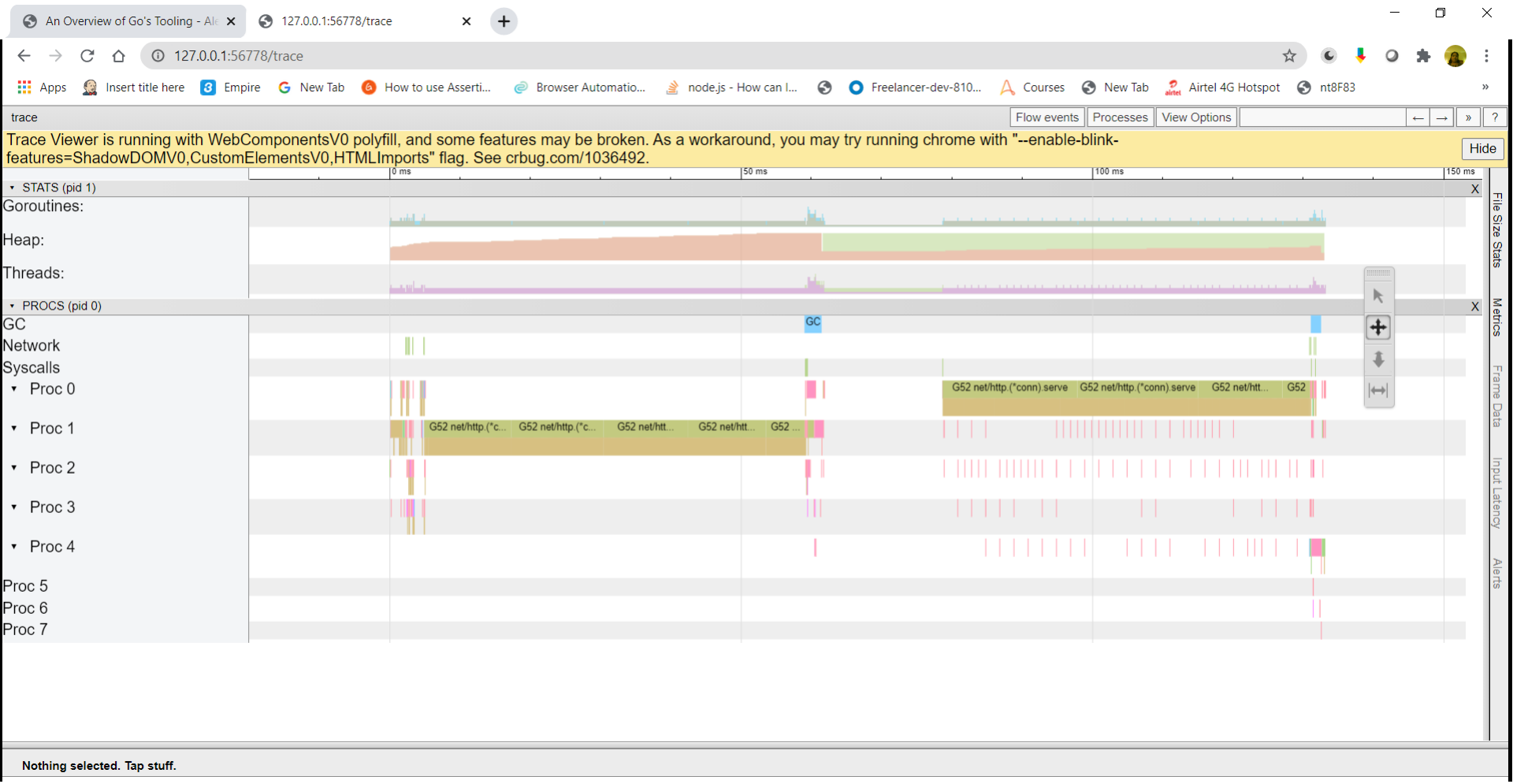
- $ go test -run=^$ -bench=^BenchmarkFoo$ -memprofile=/tmp/memprofile.out .
- $ go test -run=^$ -bench=^BenchmarkFoo$ -blockprofile=/tmp/blockprofile.out .
- $ go test -run=^$ -bench=^BenchmarkFoo$ -mutexprofile=/tmp/mutexprofile.out .
- go test -run=^$ -bench=^BenchmarkFoo$ -cpuprofile=cpuprofile.out .
- go tool pprof -http=:5500 cpuprofile.out

Tracing

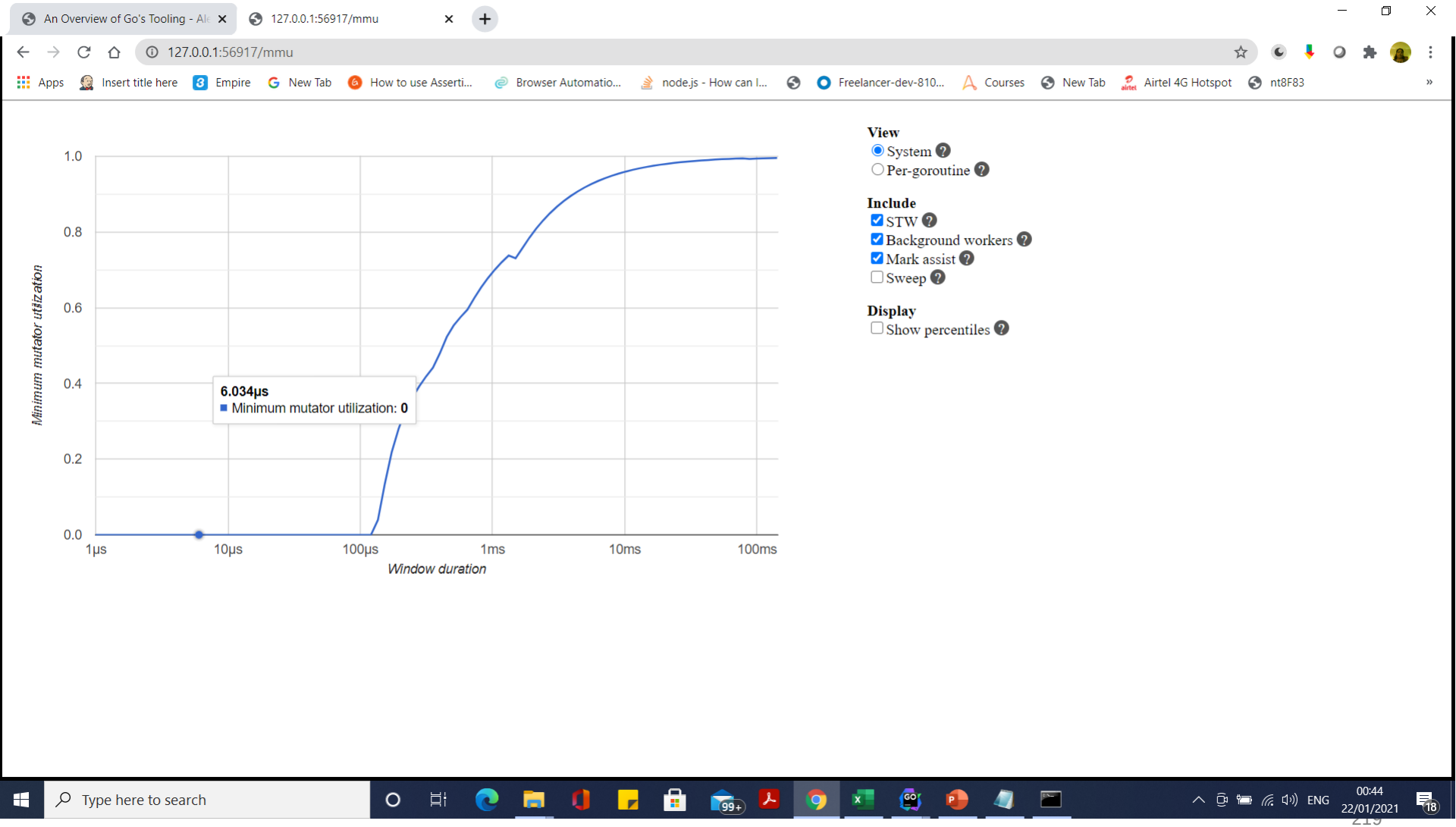
Administrator: Command Prompt - go tool trace trace.out

```
F:\go\src\ciscows\day5\dentalapp>go tool trace trace.out
2021/01/22 00:43:14 Parsing trace...
2021/01/22 00:43:14 Splitting trace...
2021/01/22 00:43:14 Opening browser. Trace viewer is listening on http://127.0.0.1:56872

F:\go\src\ciscows\day5\dentalapp>go test -run=^$ -bench=^BenchmarkFoo$ -trace=trace.out .
PASS
ok      _/F_/go/src/ciscows/day5/dentalapp      0.302s

F:\go\src\ciscows\day5\dentalapp>go tool trace trace.out
2021/01/22 00:43:42 Parsing trace...
2021/01/22 00:43:42 Splitting trace...
2021/01/22 00:43:42 Opening browser. Trace viewer is listening on http://127.0.0.1:56917
```

Tracing


An Overview of Go's Tooling - Al
127.0.0.1:56778/trace
127.0.0.1:56778/trace
Apps
Insert title here
Empire
New Tab
How to use Asserti...
Browser Automatio...
node.js - How can I...
Freelancer-dev-810...
Courses
New Tab
Airtel 4G Hotspot
nt8F83
trace
Flow events
Processes
View Options
Trace Viewer is running with WebComponentsV0 polyfill, and some features may be broken. As a workaround, you may try running chrome with "--enable-blink-features=ShadowDOMV0,CustomElementsV0,HTMLImports" flag. See crbug.com/1036492.
Hide
0 ms
50 ms
100 ms
150 ms
STATS (pid 1)
Goroutines:
Heap:
Threads:
PROCS (pid 0)
GC
Network
Syscalls
Proc 0
Proc 1
Proc 2
Proc 3
Proc 4
Proc 5
Proc 6
Proc 7
G52 net/http.(*conn).serve
G52 net/http.(*conn).serve
G52 net/htt...
G52
G52 net/http.(*c...
G52 net/http.(*c...
G52 net/htt...
G52 net/htt...
G52 ...
File Size Stats
Metrics
Frame Data
Input Latency
Alerts
Nothing selected. Tap stuff.

Tracing


An Overview of Go's Tooling - Al
127.0.0.1:56917/mmu
127.0.0.1:56917/mmu
Apps
Insert title here
Empire
New Tab
How to use Asserti...
Browser Automatio...
node.js - How can I...
Freelancer-dev-810...
Courses
New Tab
Airtel 4G Hotspot
nt8F83
Minimum mutator utilization
1.0
0.8
0.6
0.4
0.2
0.0
6.034µs
Minimum mutator utilization: 0
1µs
10µs
100µs
1ms
10ms
100ms
Window duration
View
System
Per-goroutine
Include
STW
Background workers
Mark assist
Sweep
Display
Show percentiles
Type here to search
ENG
00:44
22/01/2021

cgo

- Writing the C code
- Create a new folder in your $GOPATH/src. I've called it cgo-tutorial. Inside this folder, create a header file called greeter.h with the following content:

- #ifndef _GREETER_H
- #define _GREETER_H

- int greet(const char *name, int year, char *out);

- #endif

cgo

- Writing the C code
- Create a new folder in your $GOPATH/src. I've called it cgo-tutorial. Inside this folder, create a header file called greeter.h with the following content:

- #ifndef _GREETER_H
- #define _GREETER_H

- int greet(const char *name, int year, char *out);

- #endif

cgo


```
#include "greeter.h"
#include <stdio.h>


int greet(const char *name, int year, char *out) {
    int n;

    n = sprintf(out, "Greetings, %s from %d! We come in peace :)", name, year);


    return n;
}
```

cgo

- Now we can run gcc -c greeter.c to make sure that our library actually compiles.
- Writing the go code
- Create a file called main.go, and add the following to the top:
- package main
- / #cgo CFLAGS: -g -Wall
- // #include <stdlib.h>
- // #include "greeter.h"
- import "C"
- import (
- "fmt"
- "unsafe"
-)

cgo


```
func main() {
        name := C.CString("Gopher")
        defer C.free(unsafe.Pointer(name))

        year := C.int(2018)
}
```

Questions

Module Summary

- In this module we discussed
  - Overview of Maven
  - Maven archetypes
  - Maven life cycle phases
  - The pom.xml file
  - Creation of Java projects using Maven
  - Creation of war files