Golang

Parameswari Ettiappan

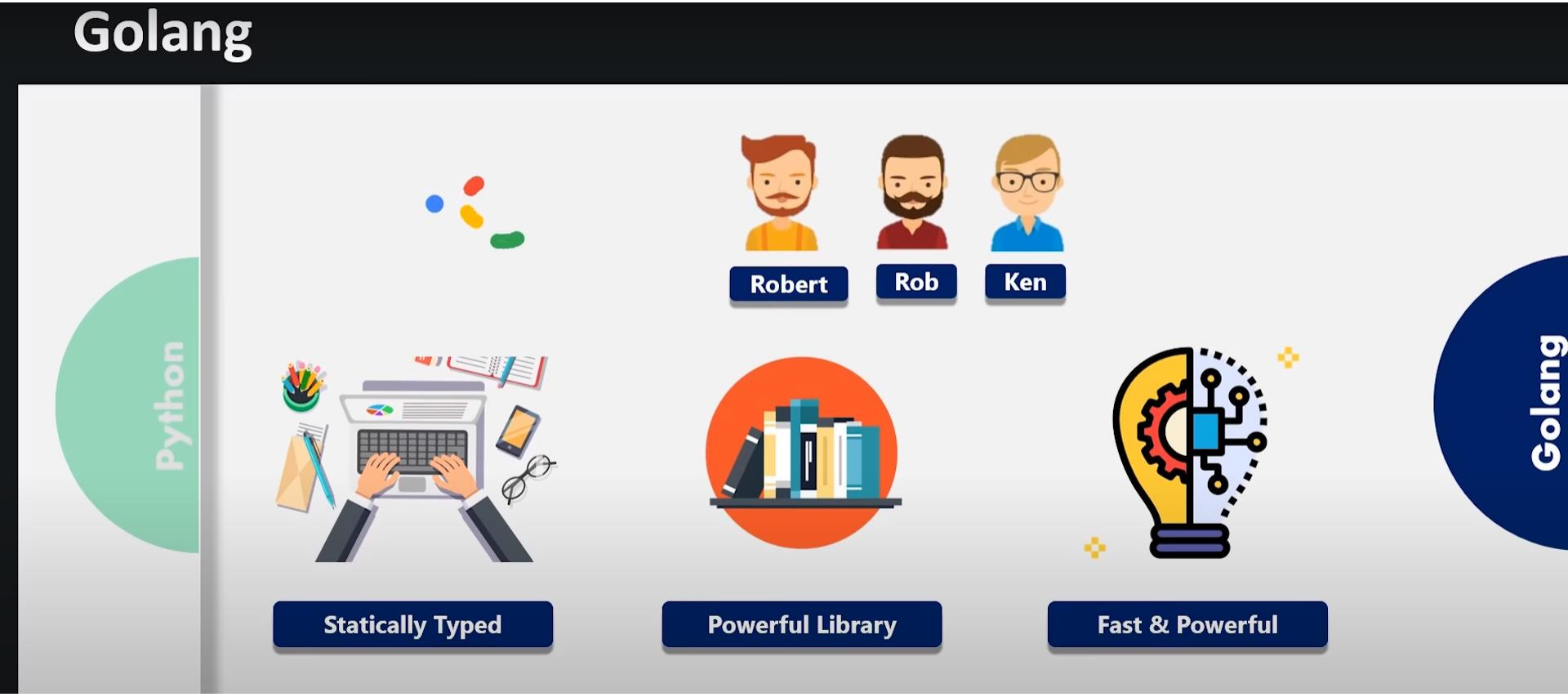
Goals

- 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)

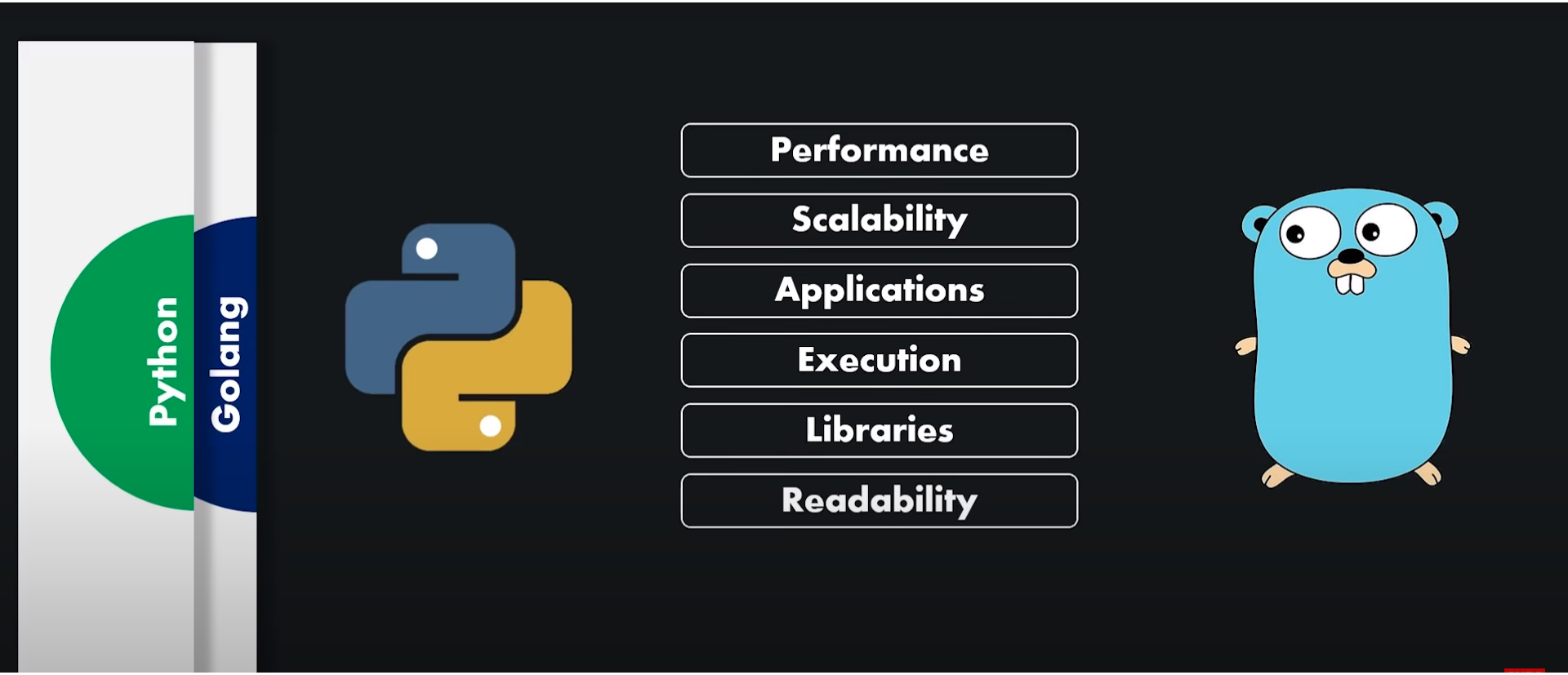
Does Golang have a future?

- 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

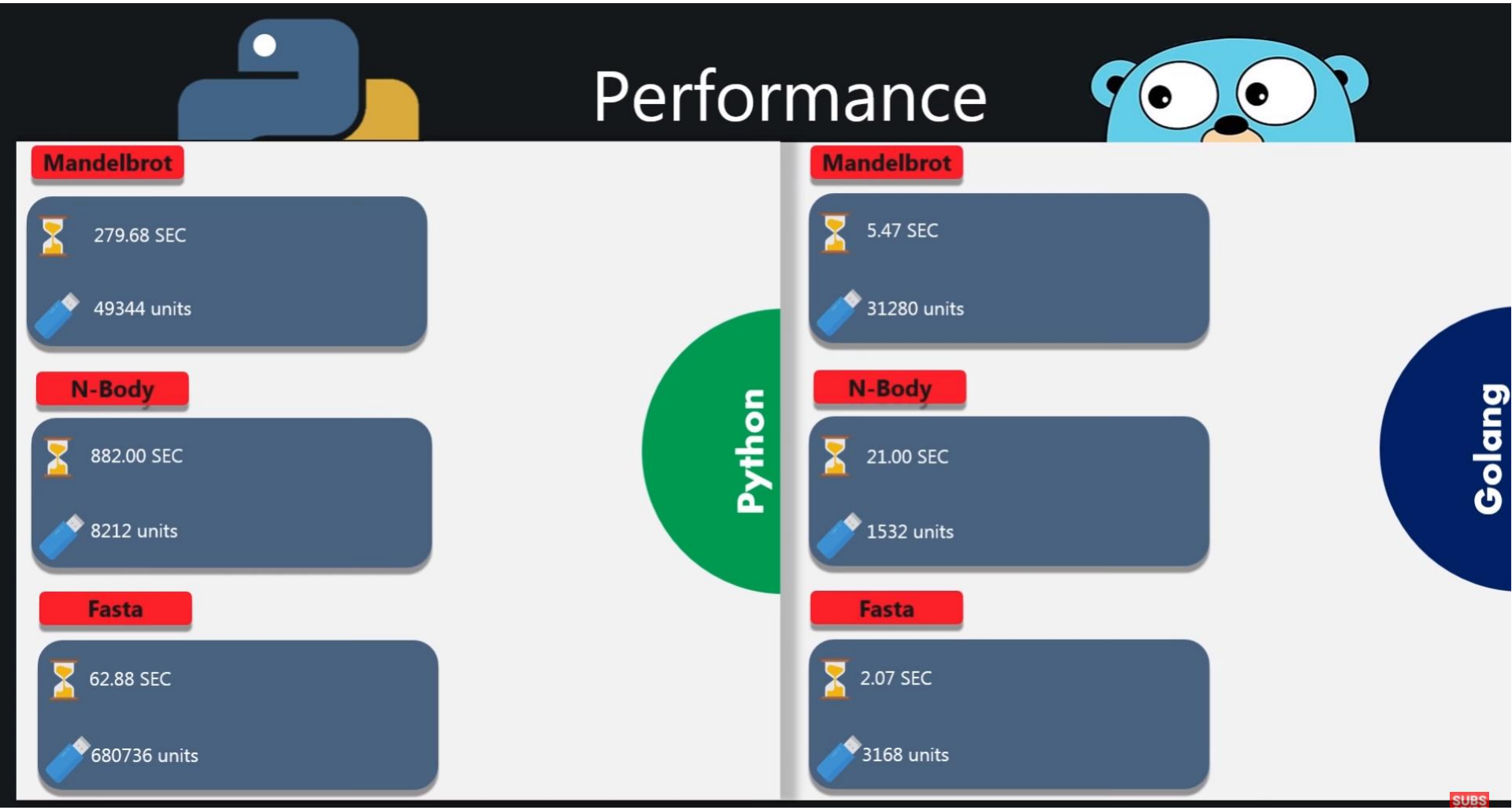
Is Golang better than Python and C++?

- 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++.

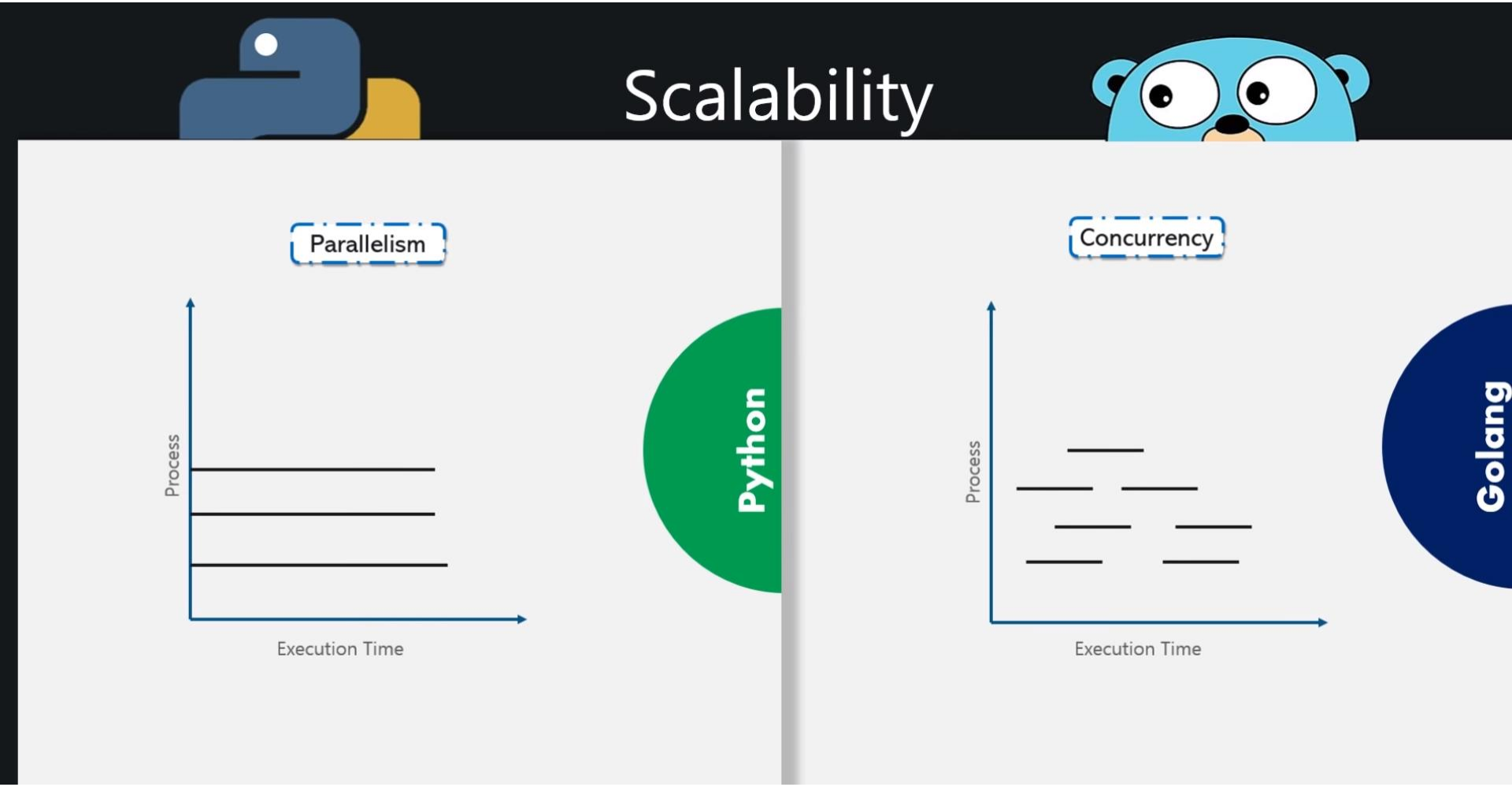
Golang vs Python



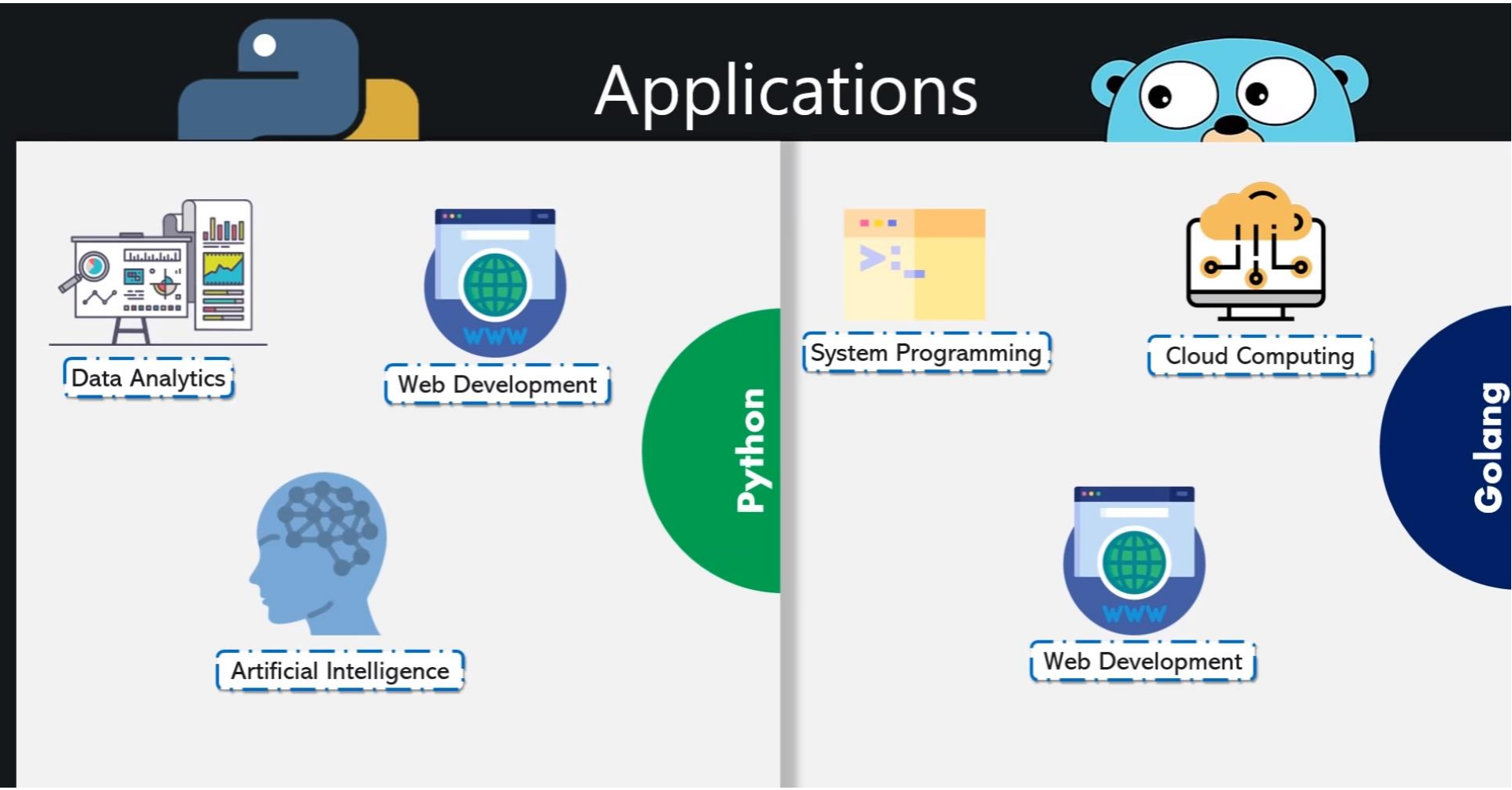
Golang vs Python



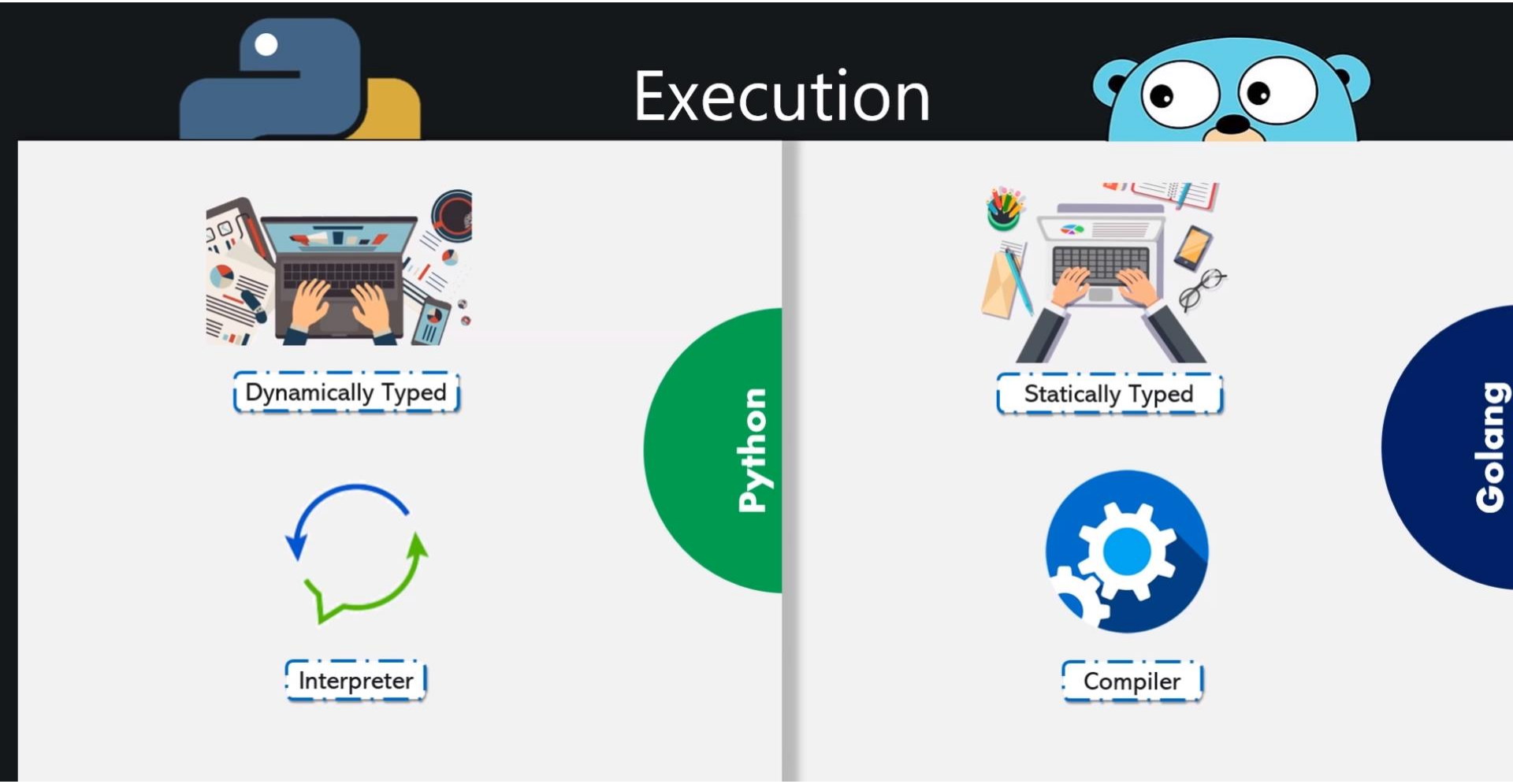
Golang vs Python



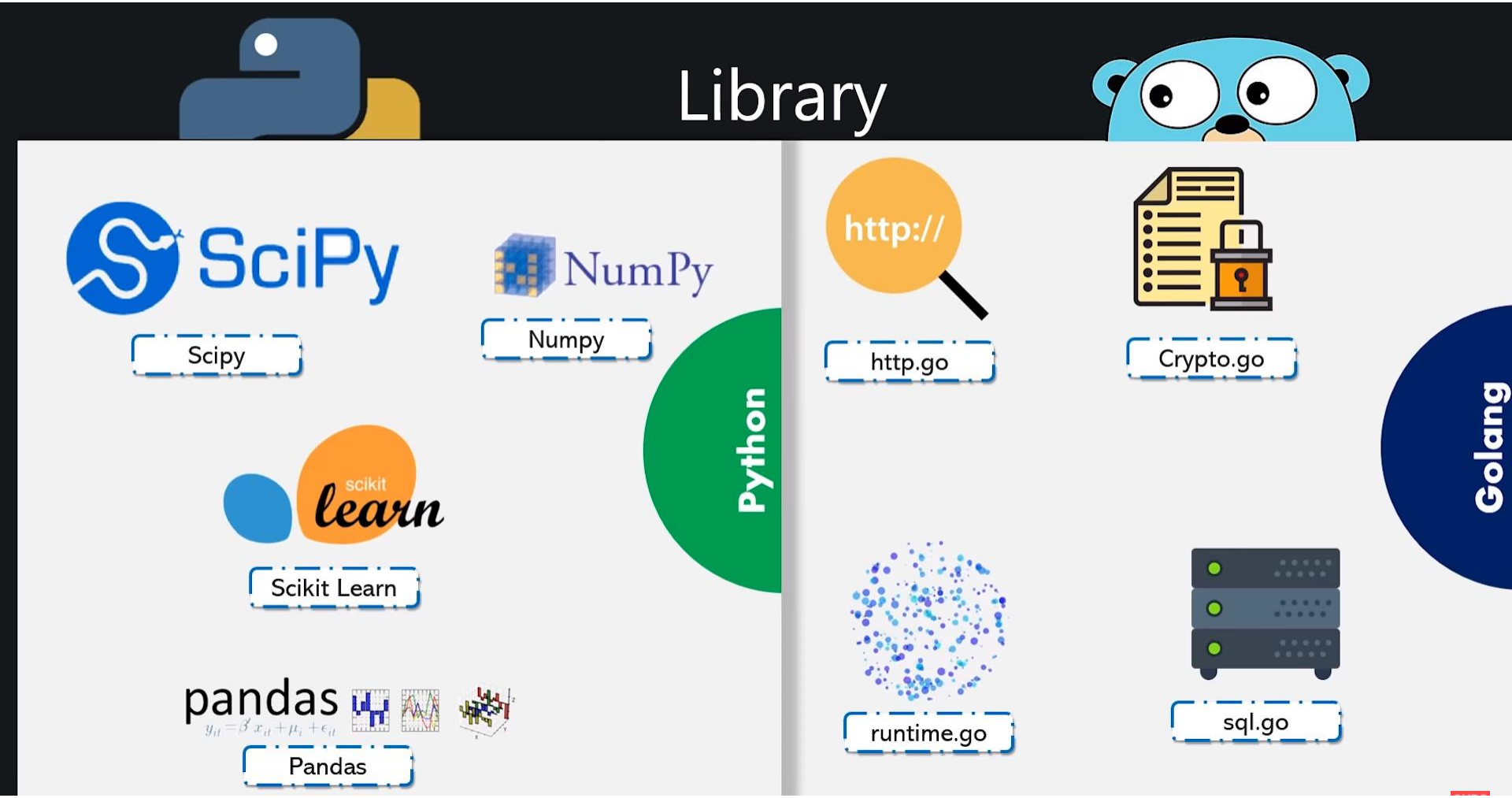
Golang vs Python



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?

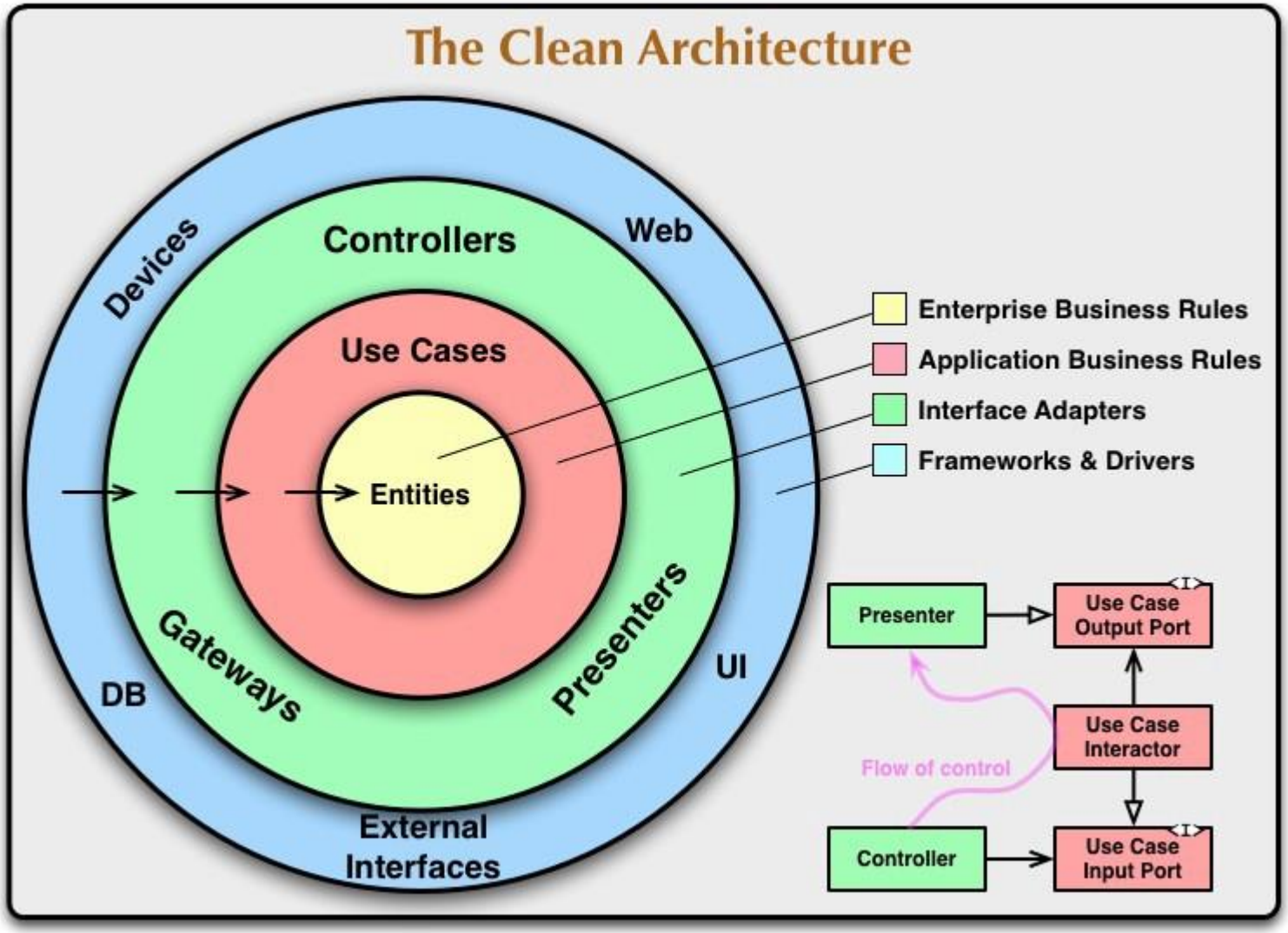
- Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
- Independent of Database. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your 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

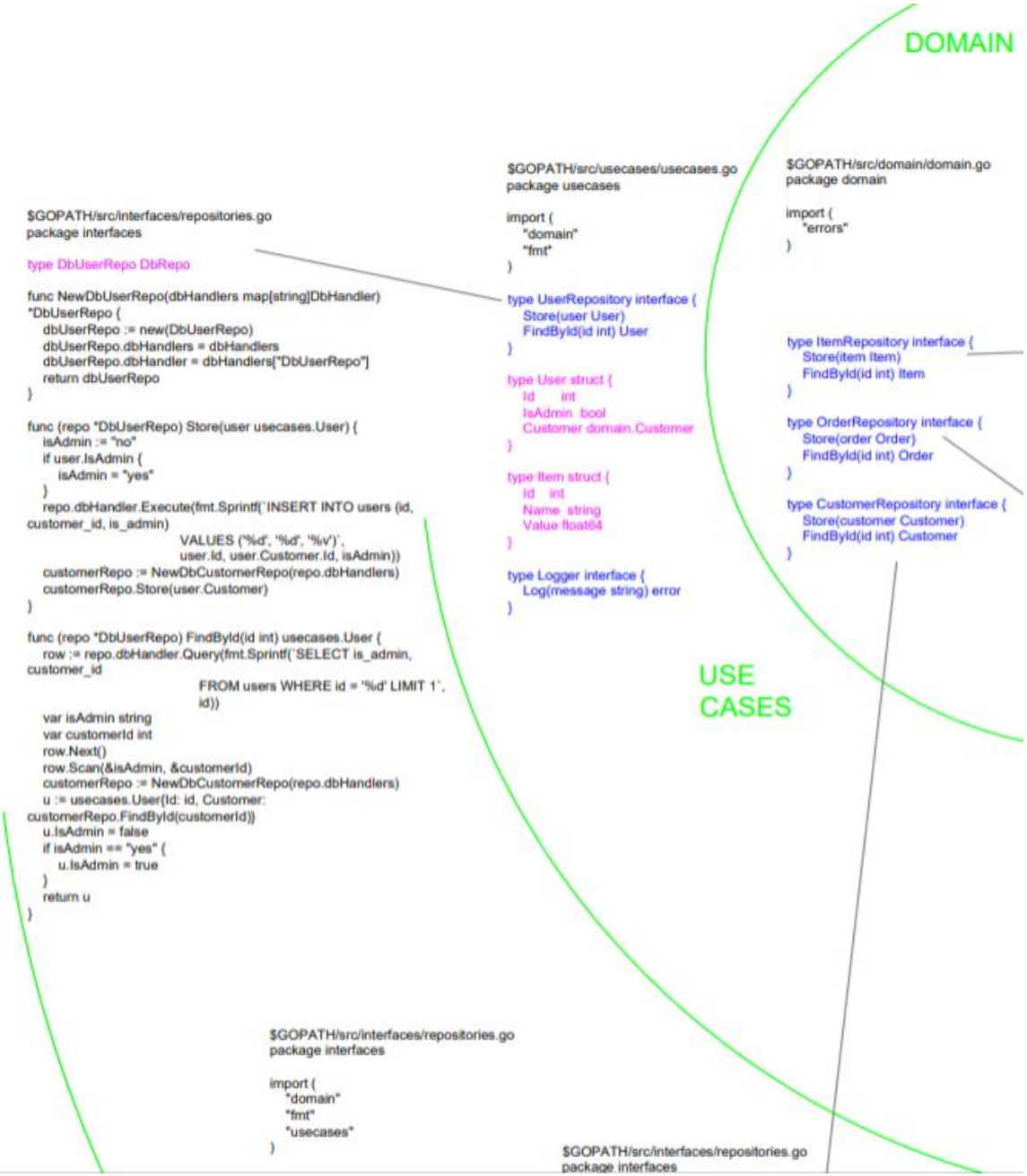
- The top directory contains three directories:
  - app: application package root directory
  - cmd: main package directory
  - vendor: several vendor packages directory

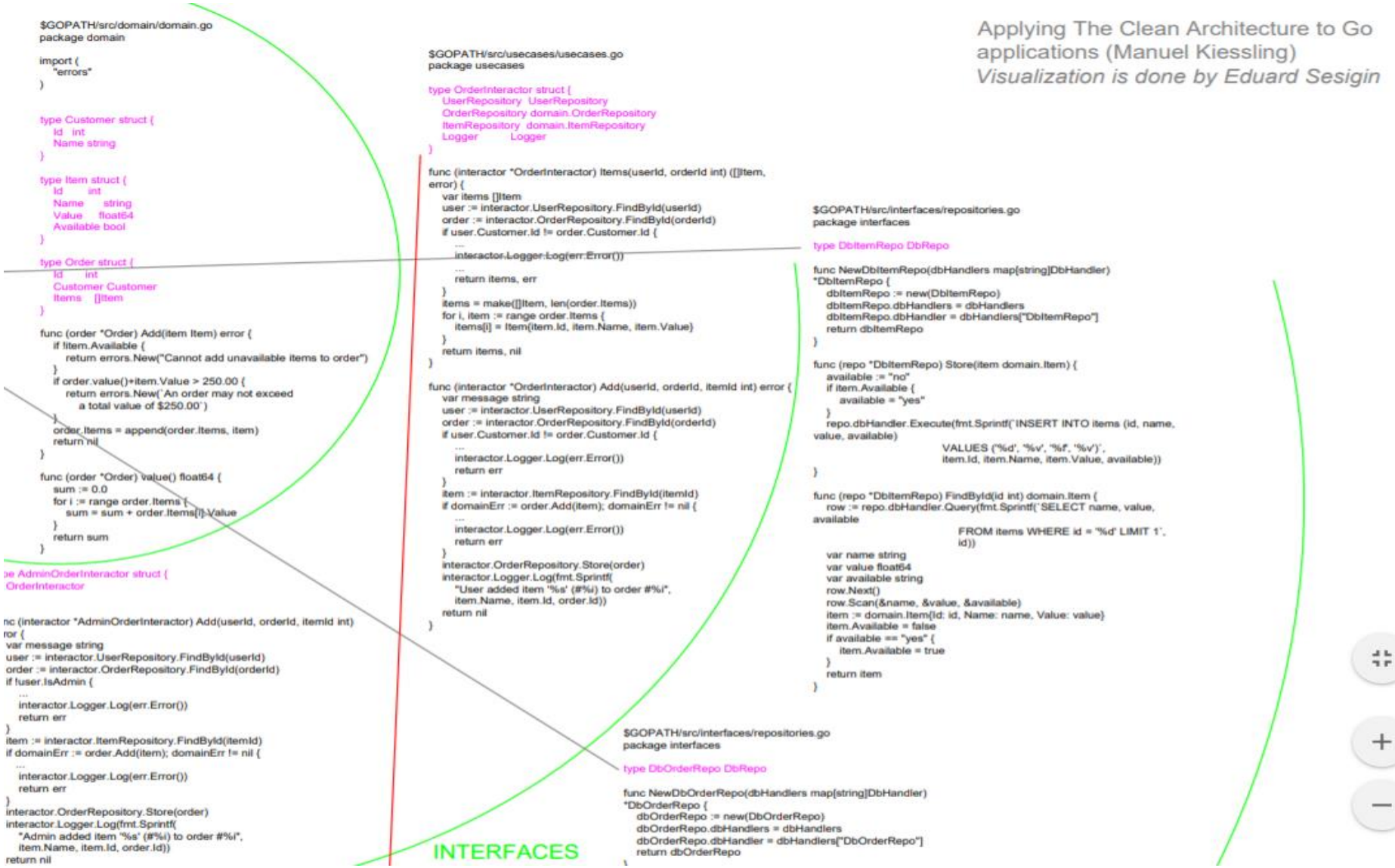
```
% tree
.
├── Makefile
├── README.md
├── app
│   ├── domain
│   │   ├── model
│   │   ├── repository
│   │   └── service
│   ├── interface
│   │   ├── persistence
│   │   └── rpc
│   ├── registry
│   └── usecase
├── cmd
│   └── 8am
│       └── main.go
└── vendor
    ├── vendor packages
    |...
```

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

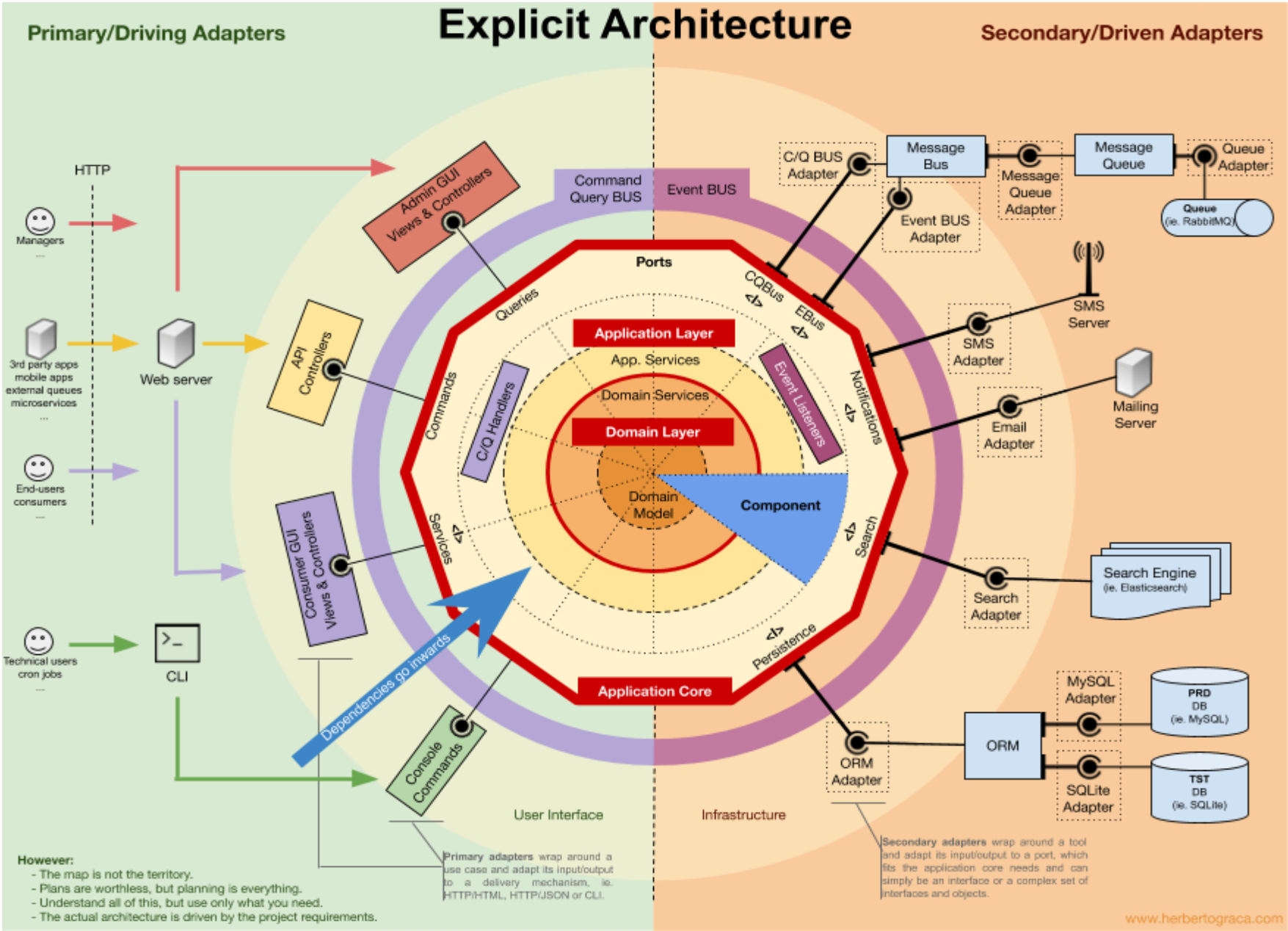
```
% tree
.
├── Makefile
├── README.md
├── app
│   ├── domain
│   │   ├── model
│   │   ├── repository
│   │   └── service
│   ├── interface
│   │   ├── persistence
│   │   └── rpc
│   ├── registry
│   └── usecase
├── cmd
│   └── 8am
│       └── main.go
└── vendor
    ├── vendor packages
    |...
```


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
   ```

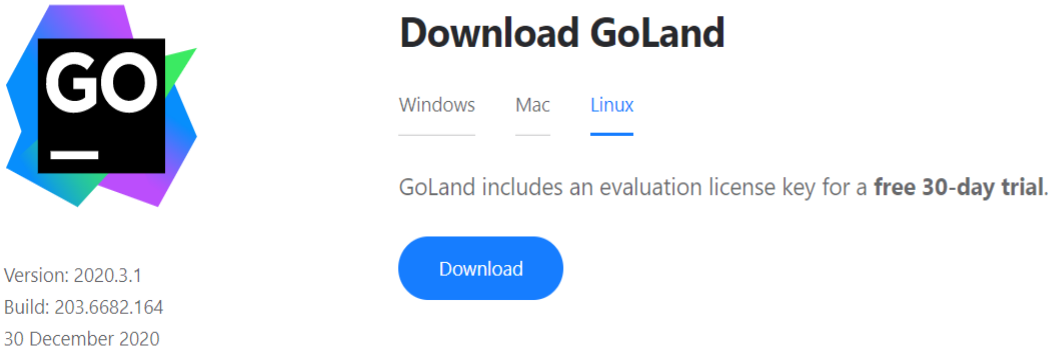
   Note: Changes made to a profile file may not apply until the next time you log into your computer. To apply the changes immediately, just run the shell commands directly or execute them from the profile using a command such as `source $HOME/.profile`.

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

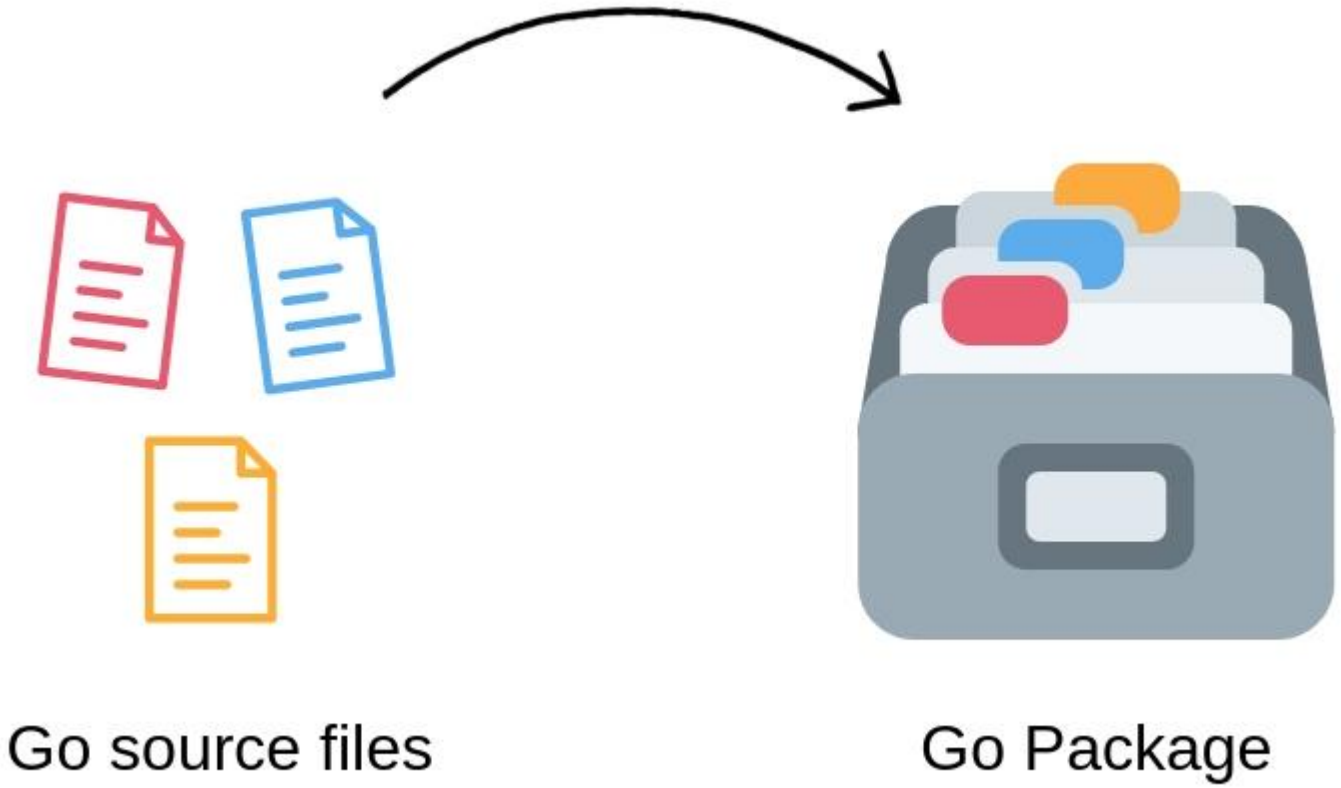
Golang Structure

- 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
*/
```

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)
}
```

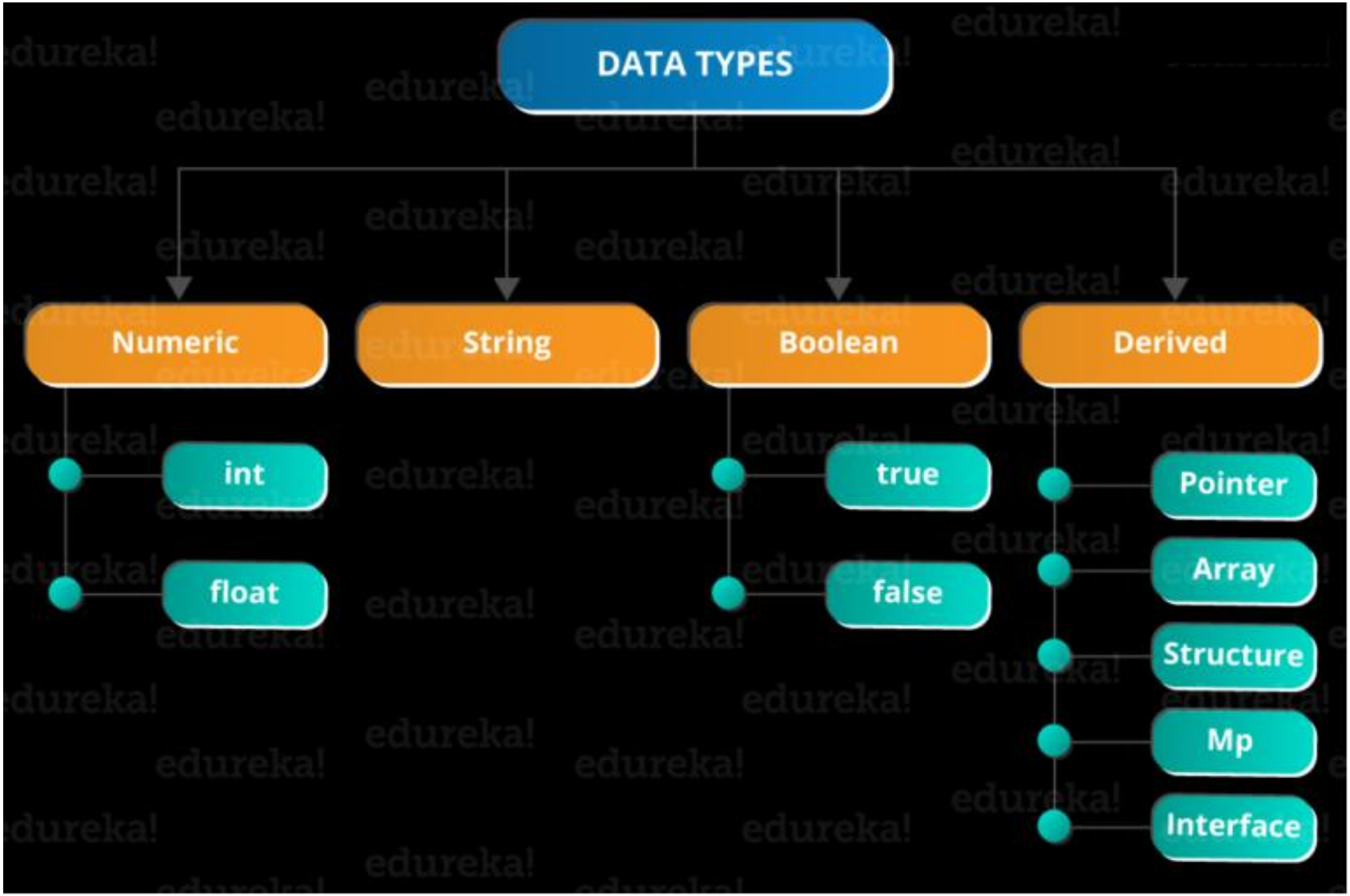
Type conversion


```
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
```

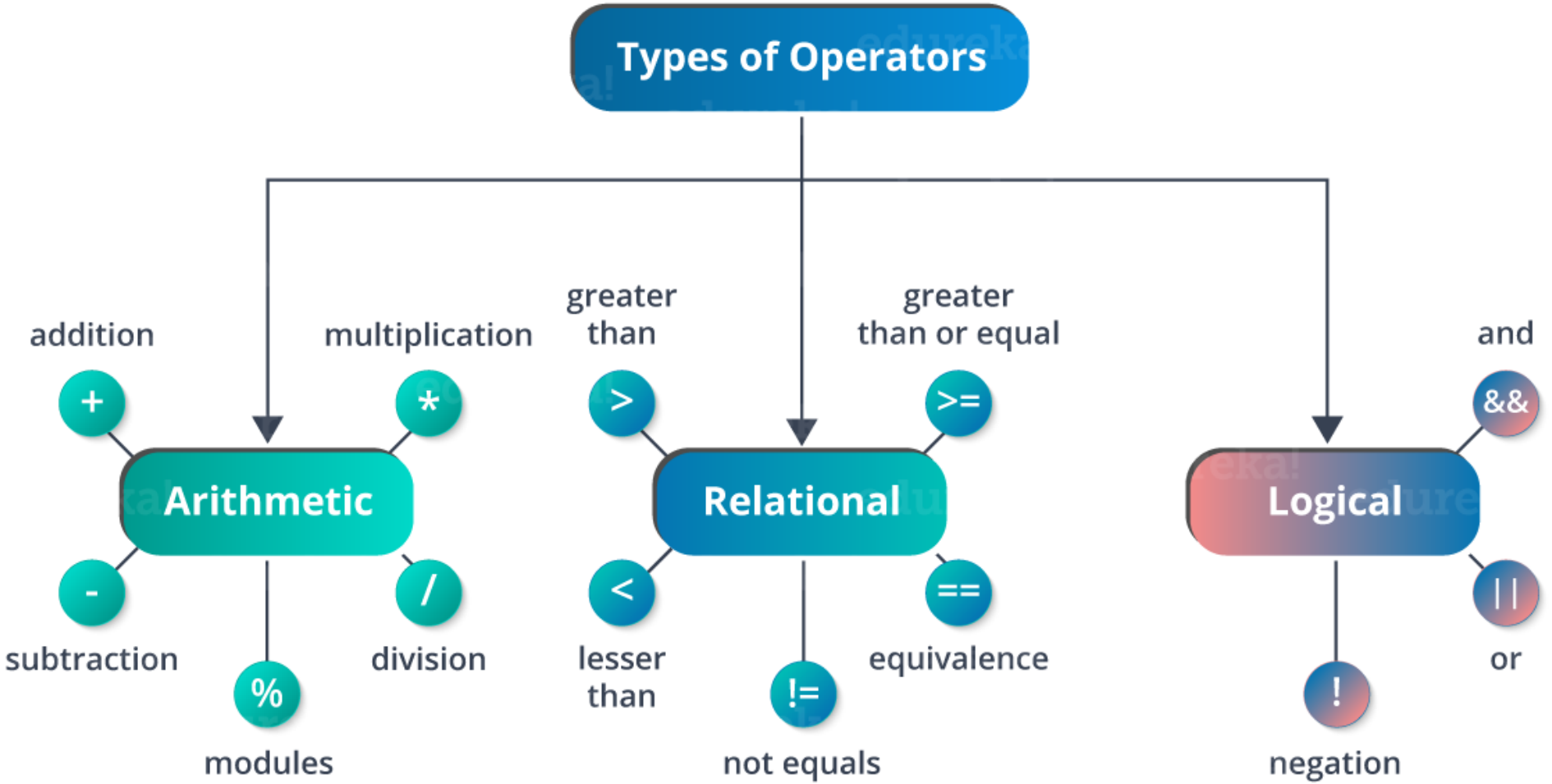
Data Types


```
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"}*

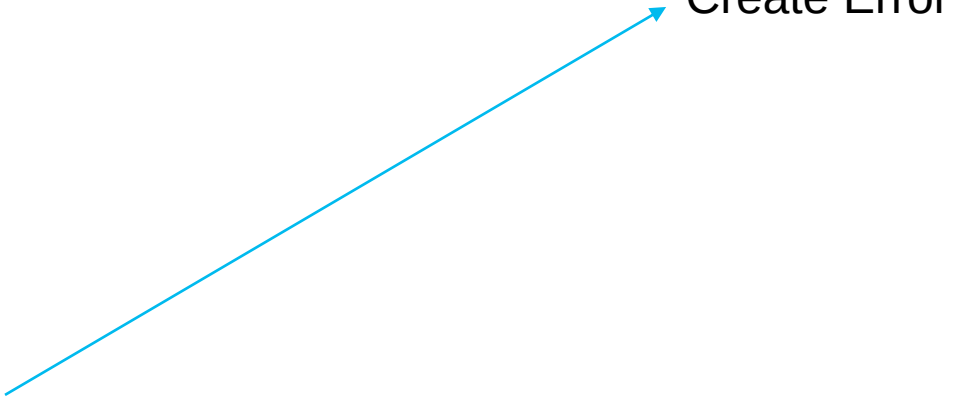
- `coral := [4]string{"blue coral", "staghorn coral", "pillar coral", "elkhorn coral”}`
- `fmt.Println(coral)`

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)
        }
```

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

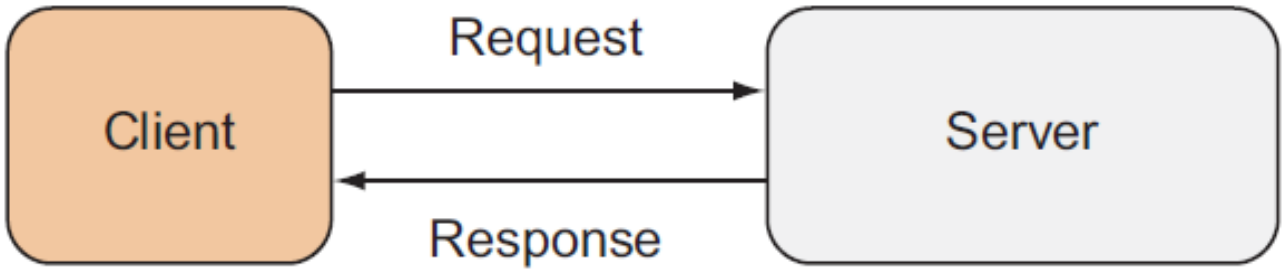
- Your program may have resources that it must clean up properly, even while a panic is being processed by the runtime.
- Go allows you to defer the execution of a function call until its calling function has completed execution.
- Deferred functions run even in the presence of a panic, and are used 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().

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

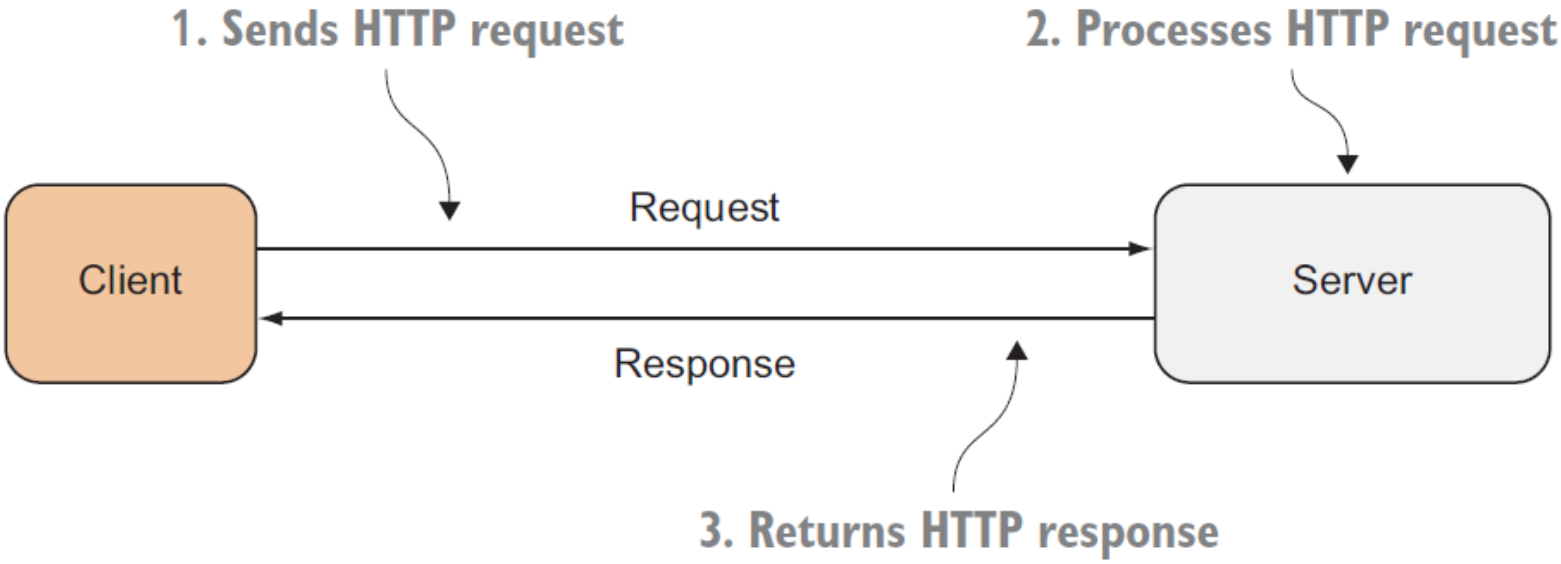
Parts of Web

- 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.

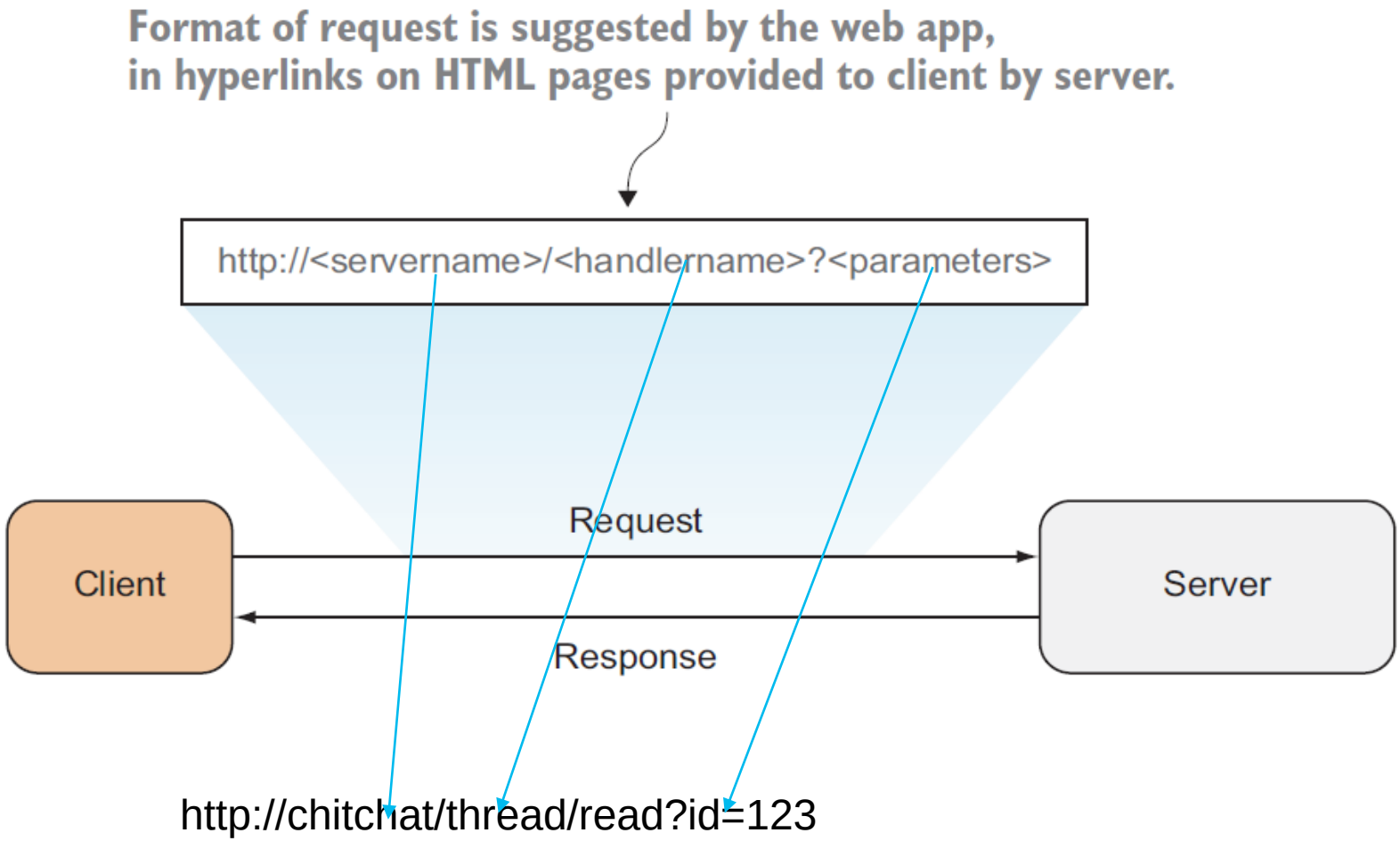
Sample App


```
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)
}
```

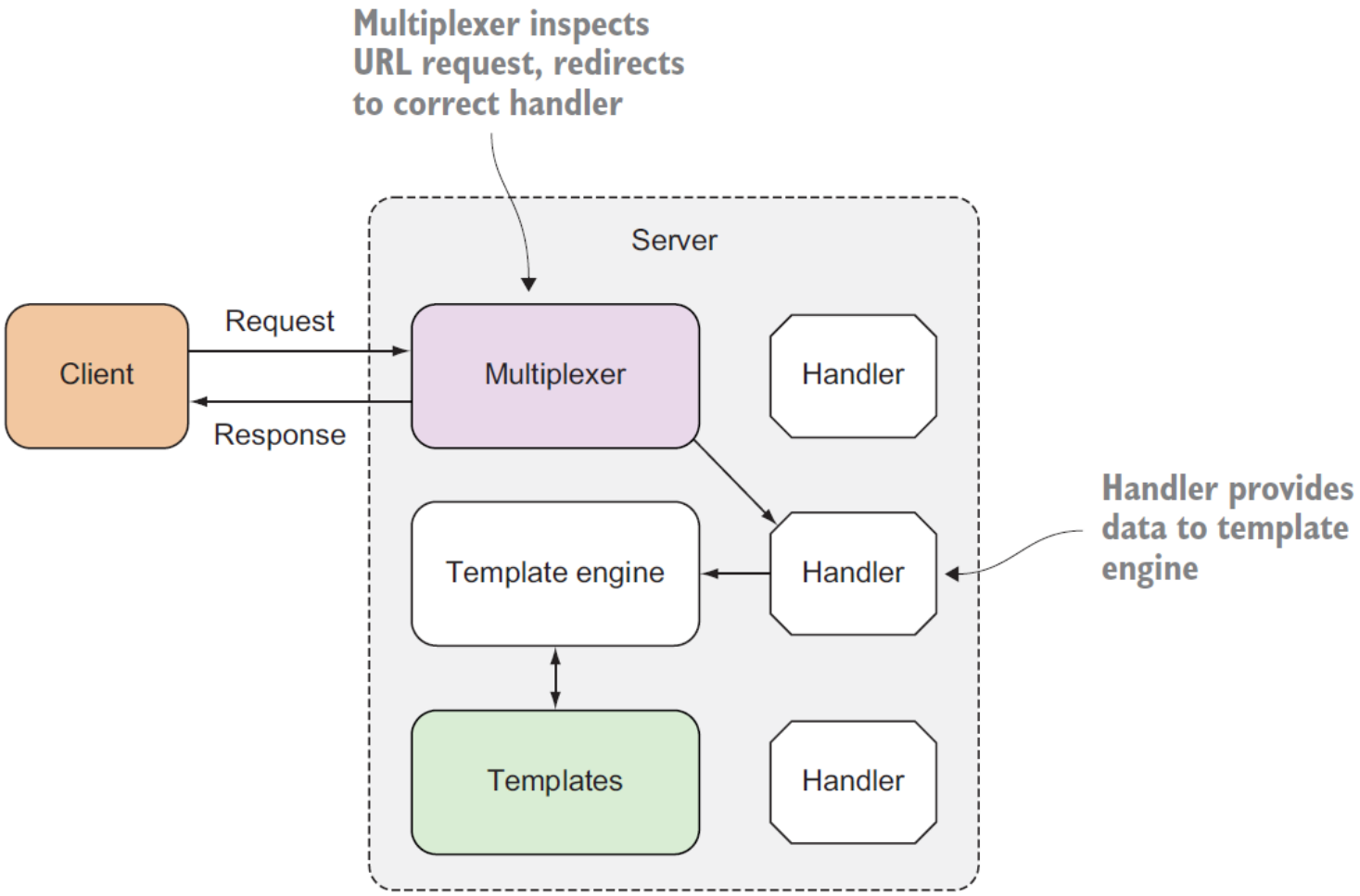
Application design



Application design



Application design



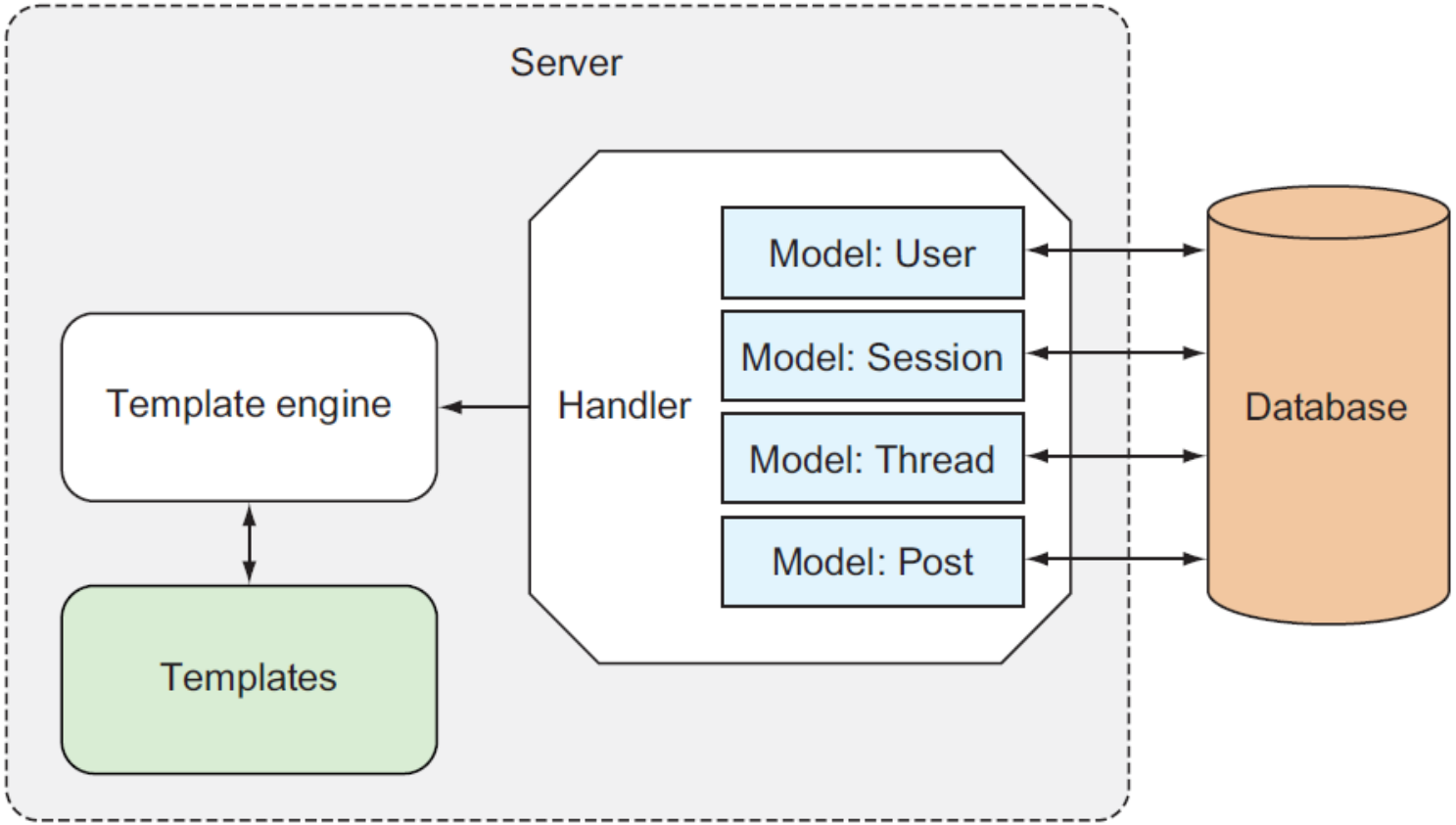
Data model

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)
  }
}
```

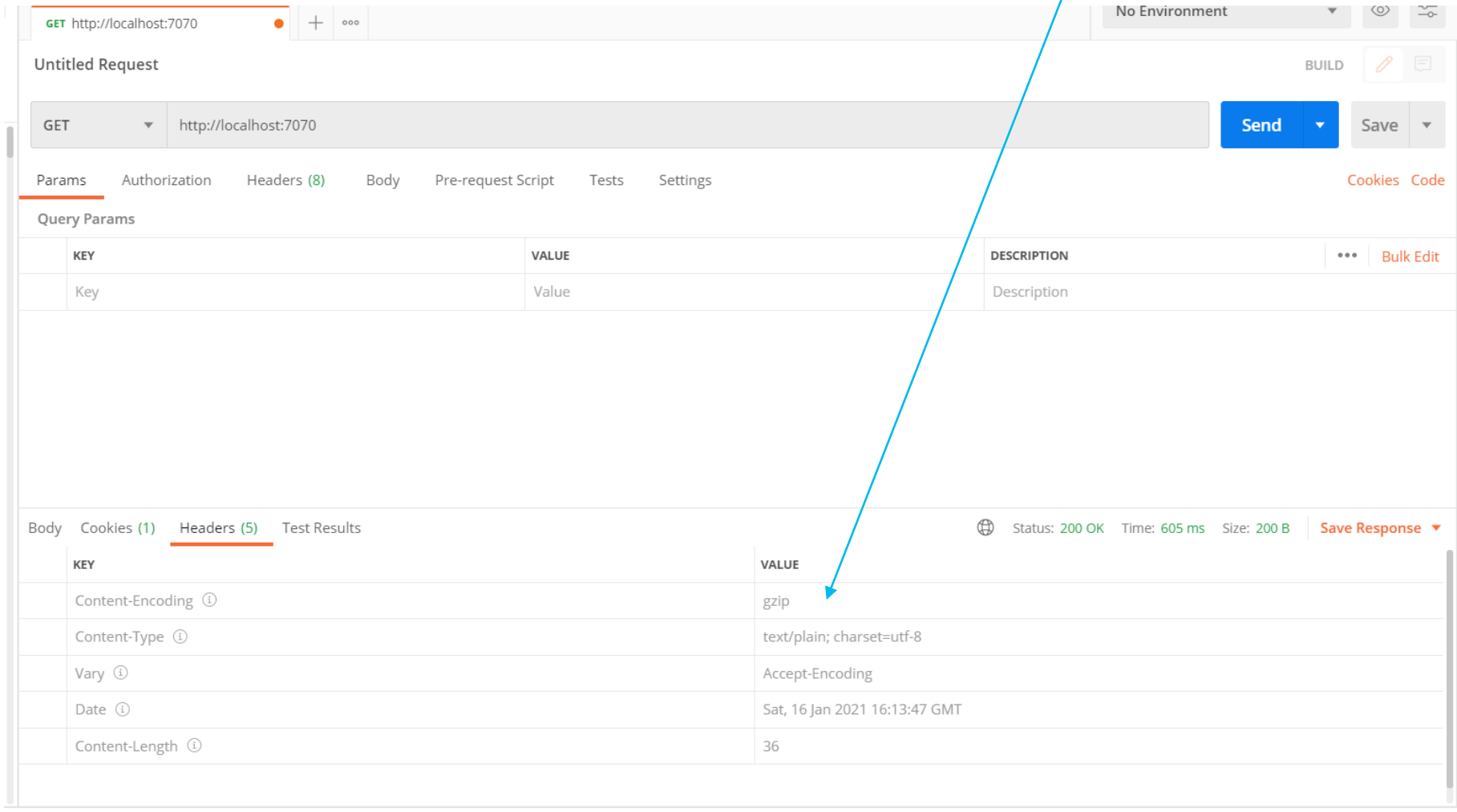
Optimizing HTTP server responses with GZIP compression

- 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
```

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