2 days Golang Programming Workshop
CloudNative Week

Wojciech Barczynski
(wbarczynski.pro@gmail.com)

Contents

1 Prerequiments

1.1 Audience

We design the workshop with the following assumptions about the audience:

- Have 1-year experience in other programming language.
- Feel good with Command Line Interface.

1.2 Your workstation

- Linux or OSX recommended.
- Basic:
  - Golang
  - a configured IDE or editor
  - Git
- Package manager exercise:
  - godep
- SQL and noSQL exercise (recommended with docker):
  - Postgres
  - MongoDB
- Nice to have:
  - Docker

Check Go Wiki to see how to configure your favorite editor to write golang programs.

1.3 Verify the setup

Let's run a hello world to check whether you can run go applications on your workstation.

Notice: No copy&paste, please.

1. Let's check the (in)famous GOPATH:

```
$ printenv | grep GOPATH
GOPATH=/Users/wb/workspace2/goprojects
```

2. Create a simple go program

```
$ mkdir -p $GOPATH/src/workshop-check
```

```
$ cd $GOPATH/src/workshop-check
$ touch main.go
```

```
$ cd $GOPATH/src/workshop-check
$ vim main.go
```

The main.go should have the following text:

```go
package main

import "fmt"

func main() {
    // 1 tabulator
    fmt.Println("Hello! YOUR_NAME")
}
```

Now, let's run it:

```
# 1. enforce formatting:
$ gofmt -w .
```

```
# 2. run
$ go run main.go
```

```
# 3. build and run:
$ go build .
$ ls
main.go  workshop-check
```

```
$ ./main.go
```

Imagine for a second, we want to have it as a tool in our PATH:

```
# install
$ go install

# magic?
$ ls $GOPATH/bin | grep workshop

$ export PATH=$PATH:$GOPATH/bin
$ cd

# run it
$ workshop-check
```

1.4 Golang Playground

Open the browser and run our program on golang playground: https://play.golang.org/. Notice: you can generate a link to your code sample.

2 Basics

Notice: No copy&paste!

2.1 Variable definition

0. Create new directory in your `$GOPATH`: `hello-worlder`. Copy the `main.go` from our `workspace-check` project.

1. Please extract your name as a variable, use the following definitions and mark the incorrect ones:

```
// 1
var myName string = "Natalia"
var myName = "Natalia"

// 2
myName := "Natalia"
```

```
// 3
var myName
myName = "Natalia"

// 4
var myName string
myName = "Natalia"

// 5
var myName string
myName := "Natalia"
```

Notice: in Golang, we use camelCase for variable names.

2. Mark the myName as `const`.

3. Declare a variable for your home country and city, use the following construct:

```
var (
  x = 10
  y = 20
)
```

2.2 Integers and Floats

No big surprise here, numbers:

- `int`, `int8, ..., int64`, byte → int8, rune → int32
- `unint`, `uint8, ..., uint64`
- `float64`, `float32, float64`
- `complex`

Golang does not support automatic conversion between types. Let's experience it.

1. Declare a variable `devExpDays` and `msg`:

```
package main

import (
  "fmt"
  "reflect"
  "strconv"
)

func main() {
  name := "Natalia"
  devExpDays := 365
  msg := name + " has " + devExpDays + " exp as developer"
  fmt.Println(msg)
}
```

Run it. What error message did you see?

2. To make it running, we need to use `strconv.Itoa`. Add the following import and call the function:

```
import (
  "strconv"
  "fmt"
)
```

3. Now let's go back from string to integer:

```
// imagine, we got it from the user:
devExpYears := "2"
devExpDays := 365 * devExpYears
```

to convert `devExpYears` use the following code:

```
// the famous error-return
days, err := strconv.Atoi("12020")
if err != nil {
  fmt.Printf("Cannot convert %v", err)
  return
}
```

You have more functions to convert from basic types to string and back, check Package strconv documentation.
Notice: `import (_ "strconv")`.

2.3 Boolean

Just to note: `true` and `false`, standard logical operators: `&&`, `!`, and `||`.

2.4 Math

Nothing dramatic here. For more advance mathematical functions, you should check the Package math:

```
import (
  "math"
)
```

2.5 Slices and hidden arrays

In golang, we use `slices`, seldom we use arrays.

1. If you want to defined an array, you specify the length explicitly:

```
arr1 := [...]string{"pa", "rr", "ot"}
arr2 := [3]string{"pa", "rr", "ot"}

fmt.Print(arr1)
fmt.Printf("%v", arr1)
```

Slice, an interface of the array, on the other hand we create with:

```
arr1 := [...]string{"pa", "rr", "ot"}

slice1 := []string{"pa", "na", "ma"}
slice2 := arr1[:]
```

2. What is the output?

```
  var three [3]int
  two := [2]int{10, 20}

  three = two

  fmt.Println(three)
  fmt.Println(two)
```

3. Let's define our hello world messages and add one more:

```
helloWorld := []string{"dzień dobry", "Hallo", "guten Tag"}
fmt.Printf("A: len: %d cap: %d \n", len(helloWorld),
  cap(helloWorld))

czechia := "Ahoj"
helloWorld = append(helloWorld, czechia)

fmt.Printf("B: len: %d cap: %d \n", len(helloWorld),
  cap(helloWorld))

fmt.Printf("%v\n", helloWorld)
```

Note down the len and cap in `A` and `B`:

4. Slices of slices. How would you write a one liner to print out:

- `["dzień dobry"]`:
- middle hallo messages:
- all except the last one:
- just 1rst element:

- just 15th element:

Hint: use `slice[x]`, `slice[x:]`, `slice[:x]`, and `slice[x:y]`

5. Watch out, the slices might bite your head off. Note, slice has *capacity*, *length*, and **pointer to the underlaying array** (see https://golang.org/pkg/reflect/#SliceHeader):

```
package main

import "fmt"

func main() {
  helloWorld := []string{"dzień dobry", "Ahoj", "Goodmorning"}
  eastEuropeHello := helloWorld[0:2]
  fmt.Printf("len: %d cap: %d \n", len(eastEuropeHello),
    cap(eastEuropeHello))

  eastEuropeHello[0] = "Dobry Wieczor"
  fmt.Printf("%v\n", helloWorld)
  fmt.Printf("%v\n", eastEuropeHello)
}
```

What is the result?

Replace `eastEuropeHello[0] = "Dobry Wieczor"` by:

```
 eastEuropeHello = append(eastEuropeHello, "Dobry Wieczor")
 eastEuropeHello = append(eastEuropeHello, "Dobry Wieczor")
```

What is the result? What has happend?

It should be not a suprise that:

```
  a := []int{1,3,5,7}
  b := []int{2, 4, 6}

  a = b
```

```
  a[0] = 99

  // prints the same
  fmt.Printf("%+v\n", a)
  fmt.Printf("%+v\n", b)
```

6. Let's fix that with the following code snipped:

```
newSlice := make([]string, 2)
copy(newSlice, slice)
```

7. We can also use make to create a slice with desired length and capacity:

```
  msgs := make([]string, 2, 20)
  msgs[0] = "Ahoj"
  msgs[1] = "Goodmorning"

  for idx := range msgs {
    fmt.Printf("%s\n", msgs[idx])
  }

  for idx, v := range msgs {
    fmt.Printf("%d, %s\n", idx, v)
  }

  for _, v := range msgs {
    fmt.Printf("%s\n", v)
  }
```

8. Note, when a function returns no result, use a *nil slice*:
`var nilSlice []string = nil`.

2.6 Control structure: Loops

In go there is only one loop keyword.

```
  for i := 0; i < 10; i++ {
    fmt.Println(i)
  }

  for i < 10 {
    fmt.Println(i)
    i++
  }
  // also map
  for index, value := range someSlice {
    fmt.Println(index, value)
  }
```

2.7 String

String is a read-only slice of bytes. Go source code is always in UTF-8.

1. Run the following code, note down the results:

```
const adress := "ul. Przeskok 2"

fmt.Printf("len: %s\n", len(adress))

fmt.Printf("1: %s\n", adress[0:3])
fmt.Printf("2: %c\n", adress[2])
fmt.Printf("3: %s\n", adress[2:])
fmt.Printf("4: %s\n", adress[5:])

fmt.Printf("5: %s\n", adress[16:])
fmt.Printf("6: %s\n", adress[:16])
```

2. Use the following example to build your own program printing out your 3 favorite emoicons:

```
const milk = "우유"

for index, runeValue := range milk {
  fmt.Printf("%c (%U) starts at byte position %d\n",
```

```
    runeValue, runeValue, index)
}
```

3. What does now happen?:

```
const milk = "우유"

  for i := 0; i < len(milk); i++ {
    fmt.Printf("byte: %x at the index %d\n",
      milk[i], i)
  }
```

Notice: Important packages are `Package strings` and `Package unicode/utf8`.

2.8 Maps

Build a program that displays a hello-world message for different languages.
1. Define the map:

```
  helloMsgs := map[string]string{}
  helloMsgs["pl"] = "Dzień Dobry"
  helloMsgs["en"] = "Good morning"
```

2. Read input from the user:

```
  helloMsgs := map[string]string{}
  helloMsgs["pl"] = "Dzień Dobry"
  helloMsgs["en"] = "Good morning"

  var lang string
  // read a single world in ASCII
  // skip error handling
  fmt.Scan(&lang)
```

2. Print the hello message:

```
if val, ok := helloMsgs[lang]; ok {
  // found
} else {
  // not found
}
```

3. Notice, golang has a cool feature:

```
  helloMsgs := map[string]string{
    "pl": "Dzień Dobry",
    "en": "Good morning",
  }
```

We want to have different greetings depending on the time of day:

```
  helloMsgs := map[string]map[string]string{
    "pl": {"morning": "Dzień Dobry"},
    "en": {"morning": "Good morning"},
  }
```

Now, users got bored with the same greeting:

```
  helloMsgs := map[string]map[string][]string{
    "pl": {"morning": []string{
      "Dzień Dobry",
      "Piękny poranek",
    }},
    "en": {"morning": []string{
      "Good morning",
      "Morning",
    }},
  }
```

Let's make it more readable:

```
  type daytimeGreetings map[string][]string
  type g map[string]daytimeGreetings
```

```
  helloMsgs := map[string]daytimeGreetings{
    "pl": {"morning": []string{
      "Dzień Dobry",
    }},
    "en": {"morning": []string{
      "Good morning",
    }},
  }
```

You can use this declarative style to define even the most complex JSON structures.

4. Write a program that randomize the messages, use *math/rand* and *time* packages to initialize random seed.

5. [Homework] Use time information to find out which part of the day we have.

2.9 User defined type

A short exercise to show you the types you can define over the basic and composite types:

2.10 Type Definitions

```
package main

import "fmt"

type myInt int

func display(i int) {
  fmt.Printf("%d", i)
}

func main() {
  var i myInt = 12
  i = i + 12
  // how to fix it?
```

```
  display(i)
}
```

2.11 Type Alias Declarations

```
package main

import "fmt"

type myInt = int

func display(i int) {
  fmt.Printf("%d", i)
}

func main() {
  var i myInt = 12
  i = i + 12
  display(i)
```

2.12 fmt.Printf

Check the https://gobyexample.com/string-formatting.

2.13 Functions

Let's split our program into nice functions:

1. Let's move the logic for displaying the hello message to a separate function:

```
  helloMsgs := map[string]string{
    "pl": "Dzień Dobry",
    "en": "Good morning",
  }
```

The function should take as arguments the map with messages and language.

2. Let's add return `true` if we have a message for given language and `false` if not. Notice:

```
func displayHello() (found bool) {
  // notice empty return
  return
}
```

2. Let's follow the golang way and return an error:

```
func displayHello() (err error) {
  err = fmt.Errorf("Unsupported language")
  return
}

func main() {
  err := displayHello()
  if err != nil {
    fmt.Printf("Not found!!! %v", err)
    return
  }

}
```

3. Functions are first class citizens in Golang and we often use them as arguments:

```
type letterDecorator func(string) string

//func printLetters(msg []string, decorator func(string) string)
func printLetters(msg []string, decorator letterDecorator) {
  for _, l := range msg {
    d := decorator(l)
    fmt.Println(d)
  }
}

func main() {
  alphabet := []string{"a", "b", "c", "d", "d"}
```

```
  printLetters(alphabet, func(s string) string {
    return strings.ToUpper(s)
  })

  printLetters(alphabet, func(s string) string {
    return "::" + strings.ToLower(s) + "::"
  })
}
```

4. Notice: we have support for variadic parameters in functions:
`func printSymbols(msg ...string)`:

```
printSymbols()
printSymbols("a", "z")
printSymbols(alphabet...)
```

5. We can move the execution of a function to the end of the scope with `defer`:

```
package main

import "fmt"

func main() {
  defer fmt.Println("booom!")

  for i := 0; i < 10; i++ {
     fmt.Println("tick...")
  }
}
```

2.14 Control structure: if and switch

2.14.1 switch

The *switch* can work on any data type, you do not need to switch on a value, see: https://github.com/golang/go/wiki/Switch.
The very common case for *switch* is to check types:

```
func do(v interface{}) string {
  switch u := v.(type) {
  case int:
    return strconv.Itoa(u*2) // u has type int
  case string:
    mid := len(u) / 2 // split - u has type string
    return u[mid:] + u[:mid] // join
  }
  return "unknown"
}
```

2.14.2 if

The *If* and *else* works as in other programming languages, except that you can put a language expression:

```
if err := dec.Decode(&val); err != nil {
  // handling error
}
// happy path
```

Example from the Package net:

```
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
  // here nerr is an instance of net.Error
  // and the error is Temporary
}
```

Let's build an app for writing and reading a file:

```
package main

import (
  "io/ioutil"
  "fmt"
)

func main() {
  // change to /dat1
  fPath := "/tmp/dat1"
  inD := []byte("hello\nWorld\n")
  if err := ioutil.WriteFile(fPath, inD, 0644); err != nil {
    panic(err) // failfast
  }
  fmt.Println("Write was successful!")

  outData, err := ioutil.ReadFile(fPath)
  if err != err {
     panic(err) // failfast
  }
```

```
  fmt.Print(string(outData))
}
```

If it works, change `fPath` to `fPath := "/tmp/dat1"`. What does happen?

2.15 Pointers

Remember:

- Because of the Golang design, your code will work usually faster if you pass small data types by value.
- Do not be overzealous with the pointers.
- Maps, slices, and pointers are reference types.

Write the following program:

```
package main

import "fmt"

func tryAnswerEverything(i int) {
  i = 45
}

func answerEverything(i *int) {
  *i = 42
}

func main() {
  i := 33

  fmt.Println(i)
  tryAnswerEverything(i)
  fmt.Println(i)

  answerEverything(&i)
  fmt.Println(i)
}
```

Notice, we can return in functions pointers to local variables: `func Answer() *int`.

2.16 Structures

In Golang, we do not have *classes*, instead *struct* with *methods*. Our language does not support inheritance, favors composition instead.

2.17 Structures and Methods

1. Write a program to manage employees:

```
type Employee struct {
    FirstName   string
    LastName    string
    leavesTotal int
    LeavesTaken int
}

func (empl *Employee) TakeHolidays(days int) error {
  // write an implementation
}

func (empl *Employee) limitExceeded(days int) bool {
  // write an implementation
}

func main() {
  empl := new(Employee)
  empl.FirstName = "Laste"
  empl.LastName = "BB"
  empl.leavesTotal = 26
  fmt.Println(empl.FirstName)
  // ...
}
```

Question? How to init the Employee outside our package?

```
// hide the implementation
type employee struct {
}
```

```
// provide a factory method New or NewEmployee
func NewEmployee(firstName string, lastName string,
    leaveDays int) *employee {

  return &employee{FirstName: firstName, LastName: lastName,
    totalLeaves : leaveDays}
}

// called: employee.New() where employee is a package name
```

2. Refactor our application and move implementation of the `employee` to a separate package. To do it, create a directory `employee` and create inside `employee.go`:

```
package employee

type Employee struct {
  FirstName   string
  LastName    string
  leavesTotal int
}

func New(firstName string, lastName string,
    leaveDays int) *Employee {

  return &Employee{FirstName: firstName, LastName: lastName,
    leavesTotal: 30}
}
```

Create an instance of Employee in `main.go`.

2.18 Pointer receiver vs value receiver

What is the difference?

```
func (empl *Employee) TakeHollidays(taken int) {
    empl.leavesTaken = empl.leavesTaken + taken
}
```

and:

```
func (empl Employee) TakeHollidays(taken int) {
    empl.leavesTaken = empl.leavesTaken + taken
}
```

Write a program to find out.

2.19 Structures and Interfaces

Your structure has to implement the functions from the interface:

```
package main

import "workshop-app/postgres"

type DataStore interface {
  GetEmployee(id int) string
}

type App struct {
  ds DataStore
}

func main() {
  app1 := App{ds: &postgres.PsqlStore{}}
  fmt.Print(app1.ds.GetEmployee(12))
}
```

We might have two configurable stores, one psql:

```
package postgres

type PsqlStore struct {
}

func (ps *PsqlStore) GetEmployee(id int) string {
  return "psql"
}
```

and one, mongodb:

```
package mongo

type MongoStore struct {
}

func (ps *MongoStore) GetEmployee(id int) string {
  return "mongo"
}
```

Run the app and verify that the mongoStore works as well. When it works, break it, for e.g., change the type of args in `MongoStore`.

2.20 Composition instead of Inheritance

```
type person struct {
  firstName string
}

func (p person) name() string {
  return p.firstName
}

type employee struct {
  EmployeeID string
  person
}

func main() {
  empl := employee{}
  empl.firstName = "Wojtek"
  empl.person.firstName = "Krzyś"
  fmt.Println(empl.firstName)
}
```

Notice: For polymorphism, you need to use *interface*.

2.21 Interfaces vs Functions

If you get high on interface, do not forget about the functions, an example from net/http:

```
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

we could use it to customize our employees leave policies:

```
package employee

type LeaveHandler func(taken int, total int) bool

func NewLeaveHandler(f func(int, int) bool) LeaveHandler {
  return f
}

func (f LeaveHandler) CanTakeHollidays(empl Employee) bool {
  r := f(empl.totalLeaves, empl.leavesTaken)
  return r
}
```

2.22 Packages, go get, godep, go mod

The traditional way to get your packages was `go get`. At this moment, we have two approaches `v.go` (new, favored by Google) and `godep` (old, community driven). Let's see how they work.

1. Create new project *workshop-gomod*, with main.go:

```
package main

import (
  "fmt"
  "net/http"
```

```
  "github.com/gorilla/mux"
)

func HelloHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, "Hello World")
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", HelloHandler)
    http.Handle("/", r)
}
```

2. Use *gomod* to capture the deps:

```
$ GO111MODULE=on go mod init
$ GO111MODULE=on go get github.com/gorilla/mux
$ ls

go.mod  go.sum  main.go

$ GO111MODULE=on go mod vendor
$ ls

$ ls vendor

$ GO111MODULE=on go list -m all
```

Notice: the debate is ongoing whether we should or must not commit `vendor` to your repo. You should, at least, push `go.mod` and `go.sum` to your repo.

3. Create new project *workshop-godep*, with main.go:

```
package main

import (
  "fmt"
```

```
  "net/http"

  "github.com/gorilla/mux"
)

func HelloHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, "Hello World")
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", HelloHandler)
    http.Handle("/", r)
}
```

4. Use *godep* to manage your dependencies:

```
$ dep
$ dep init -v
$ ls

vendor  Gopkg.lock  Gopkg.toml  main.go
```

You will have less surprises with `dep` at this moment. `v.go` will be the default soon, consult https://github.com/golang/go/wiki/Modules.

2.23 Linters

Linters are a part of the language:

- gofmt
- goimport - gofmt + sorting imports
- govet - now executed with tests

To apply the fixes, use `-w` flag:

```
$ gofmt -w .
$ goimports -w *.go && goimports -w */*.go
```

There are many linters out there, check, for e.g., awesome-go-linter page. You should call the linters as part of your CI/CD pipeline:

- linter
- test
- integration-test

2.24 Errors

Let's define our own error and see how they work. We will use the standard libraries for handling errors.
Notice `error` interface is:

```
type error interface {
  Error() string
}
```

2.24.1 Sentimental Errors

An example of such errors are: `sql.ErrNoRows` and `io.EOF`, they are declared as:

```
package sql

var errNoRows = errors.New("sql: no Rows available")
```

The advantage? It is simple to handle:

```
err := db.QueryRow("SELECT * FROM users WHERE id = ?", userID)
if err == sql.ErrNoRows {
  // an error we know
}  else if err != nil {
  // another eeror
}
```

Disadvantage? Not too much info, they become a part of your package API.

Please implement, an error for our Employee application using *Package errors*[1].

[1] https://godoc.org/github.com/pkg/errors

2.24.2 Custom Error Types

Use the following example of a custom error type to change the errors implementation in the `Employee` application:

```
type ParsingError struct {
    IncorrectValue string
}

func (e *ParsingError) Error() string {
    return fmt.Sprintf("Cannot parse %v: internal error",
      e.IncorrectValue)
}
```

if we use our custom error code, our code will get more complicated:

```
if err := Foo(); err != nil {
    switch e := err.(type) {
    case *ParsingError:
        //
    default:
        log.Println(e)
    }
}
```

2.24.3 Opaque errors

The idea:

- return your own errors
- provide functions to determine what has happended

An example for Dave Cheney blog [2]:

```
type temporary interface {
        Temporary() bool
```

[2] https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully

```
}

// IsTemporary returns true if err is temporary.
func IsTemporary(err error) bool {
        te, ok := err.(temporary)
        return ok && te.Temporary()
}
```

2.24.4 Wrapping/Annotating and Unwrapping errors

With help of the package errors[3], we can provide support for the stacktraces and handling error causes.
Let's see how to implement, a stacktrace support for our application:

```
package main

import (
  "fmt"
  "github.com/pkg/errors"
)

type stackTracer interface {
  StackTrace() errors.StackTrace
}

var errProcess = errors.New("boom")

func processData() error {
  return errProcess
}

func main() {
  err := processData()
  wrappedErr := errors.Wrap(err, "processing failed")
  fmt.Printf("%v", wrappedErr)

  if err, ok := wrappedErr.(stackTracer); ok {
    fmt.Printf("%+v", err.StackTrace())
```

[3]https://godoc.org/github.com/pkg/errors

```
  }
}
```

Unwrapping:

```
package main

import (
  "fmt"
  "net"

  "github.com/pkg/errors"
)

type myErrProcess error

var errProcess myErrProcess = errors.New("boom")

func processData() error {
  return errProcess
}

func main() {
  errData := processData()
  wrappedErr := errors.Wrap(errData, "processing failed")

  _, ok := errors.Cause(wrappedErr).(net.Error)
  if ok {
    fmt.Printf("net.Error")
  }

  errP, ok := errors.Cause(wrappedErr).(myErrProcess)
  if ok {
    fmt.Printf(errP.Error())
  }
}
```

2.24.5 Defer, Panic, and Recover

There is a way to recover from the panic, when we use `defer`, `panic`, and `recover`. We are not going to cover it in the introduction course.
If you cannot wait, check a defer-panic-and-recover blog post on golang.org and the golang wiki PanicAndRecover article.

2.24.6 Error best practices

- Handle errors once
- Error, keep on the left
- Notice: standard errors do not come with stacktraces
- Check a good article from 8thlight [4]

[4] https://8thlight.com/blog/kyle-krull/2018/08/13/exploring-error-handling-patterns-in-go.html

2.25 Tests

Create a simple test in a file – `main_test.go`. To run tests: `go test .`.

```
package main

import (
  "fmt"
)

func add(a int, b int) int {
  return a + b
}

func main() {
  fmt.Println(add(10,20))
}

func TestAdd(t *testing.T) {
  if add(10,25) != 20 {
    t.Fatal("Boom!")
  }
}
```

2.25.1 Table-driven tests

1. Create a project `workshop-test`:
main.go:

```
package main

import (
  "errors"
  "fmt"
)

var errUnknownOperation = errors.New("Unknown operation")

func Calculate(op string, a int, b int) (int, error) {
  switch op {
  case "+":
    return a + b, nil
  }
  return 0, errUnknownOperation
}

func main() {
  r, _ := Calculate("+", 1, 2)
  fmt.Println(r)
}
```

main_test.go:

```
package main

import (
  "testing"
)

func TestCalculate(t *testing.T) {
  testCases := map[string]struct {
    op       string
    a        int
    b        int
```

```
    expected int
  }{
    "simple add": {"+", 1, 3, 4},
  }

  for name, v := range testCases {
    t.Logf("test: %s", name)
    r, err := Calculate(v.op, v.a, v.b)
    if err != nil {
      t.Fatalf("%v", err)
    }
    if r != v.expected {
      t.Fatalf("Failed!")
    }
  }
}
```

Run the tests:

```
$ go test .
```

Notice: you can add `-race` to turn on the race detector.

2. Add, first the test, support for division.

2.25.2 Test with real X

Golang developers prefer to work against real databases, file systems, etc.

2.25.3 Tests short and long

```
if testing.Short() {
  t.Skip("skipping test in short mode.")
}
```

2.25.4 Integration tests

The best practice is to use build tags to distinguish integration tests:

```go
// +build integration

package service_test

func TestSomething(t *testing.T) {
  if service.IsMeaningful() != 42 {
    t.Errorf("oh no!")
  }
}
```

To run:

```
$ go test --tags integration ./...
```

2.25.5 Look ahead

There is much more:

- If your functions accept interfaces, return structs, they are easier to test.
- Check the brilliant blog on Go for Industrial Programming and the corresponding video.
- If you like the BDD style, look into ginko and https://github.com/onsi/gomega.

3 Your basic web app

Knowing the basics of Golang, let's build a web application.

3.1 Simplest

Writing a web server in Golang, thanks to very solid standard library, is faily simple:

```
package main

import (
  "io"
  "log"
  "net/http"
)

func main() {
  hello := func(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "Hello World!")
  }

  // Run http server on port 8080
  err := http.ListenAndServe(":8080", http.HandlerFunc(hello))

  // Log and die, in case something go wrong
  log.Fatal(err)
}
```

3.2 Multiplexed

To multiplex, we need to create a Multiplexer:

```
package main

import (
    "io"
    "log"
    "net/http"
)
```

```
func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("/hello", func(w http.ResponseWriter,
      r *http.Request) {
    io.WriteString(w, "Hello")
  })

  mux.HandleFunc("/world", func(w http.ResponseWriter,
      r *http.Request) {
    io.WriteString(w, "World")
  })

  log.Fatal(http.ListenAndServe(":8080", mux))
}
```

3.3 Handler as a struct

To customize handler, we can create a `struct`

```
type MyHandler struct {
  Greeting string
}

func (h *MyHandler) ServeHTTP(w http.ResponseWriter,
    r *http.Request) {
  fmt.Fprintf(w, "%s, %s!", h.Greeting, r.RemoteAddr)
}

func main() {
  log.Fatal(http.ListenAndServe(":8080", &MyHandler{
    Greeting: "Hello World!",
  }))
}
```

3.3.1 Sharing data structures among handlers

The following example shows how to share data among handlers, e.g., database connection details, configs:

```go
package main

import (
  "fmt"
  "log"
  "net/http"
)

type App struct {
  ServiceName string
  // Datasource
  // logging config
}

func (app *App) HelloWorld(w http.ResponseWriter,
    r *http.Request) {
  w.WriteHeader(http.StatusOK)
  fmt.Fprintf(w, "Hello World from " + app.ServiceName)
}

func main() {
  app := App{ServiceName: "MyApp"}
  mux := http.NewServeMux()
  mux.HandleFunc("/", app.HelloWorld)

  log.Fatal(http.ListenAndServe(":8080", mux))
}
```

3.3.2 Reading body

Extend the previous example to read the data passed with http body:

```go
func (h *Handler) ServeHTTP(w http.ResponseWriter,
      r *http.Request) {
  var data bytes.Buffer // []byte with IO

  // body, err := ioutil.ReadAll(r.Body)
  n, err := data.ReadFrom(r.Body) // read body to the buffer
  if err != nil {
```

```
    panic(err)
  }

  log.Printf("Got %d bytes from %s: %s\n", n, r.RemoteAddr,
    data.String())
}
```

Test it:

```
$ curl -d '{"name": "natalia"}' 127.0.0.1:8080
```

3.3.3 Parse URL

We have also support for parsing URL in net/url Package:

```
// "lang=pl"
q := r.URL.Query()
lang := q.Get("lang")
```

3.3.4 Write multilingual hello-world app

Multilingual hello-world app supports

- US1: *lang* on path `/` as a GET parameter to specify language *user* as a GET parameter to specify username for the greetings.
- US2: *lang* and *user* in body: `user:Wojtek,lang:pl`.
- if *lang* is missing, return 400.
- if *user* is missing, return 404.

3.3.5 Testing handlers

Create tests to cover the edge cases:

```
func TestHandlers(t *testing.T) {
  // Your handler to test
  handler := func(w http.ResponseWriter, r *http.Request) {
    http.Error(w, "Uh huh", http.StatusBadRequest)
  }
```

```
  // Create a request
  r, err := http.NewRequest("GET",
    "http://test.com?lang=pl&user=wojtek", nil)

  // Handle request and store result in w
  w := httptest.NewRecorder()
  handler(w, r)

  // Check out
  if w.Code != http.StatusOK {
    t.Fatal(w.Code, w.Body.String())
  }
}
```

4 Working with JSON

Let's change the input in body for our service to:

```
{
  "name": "Natalia",
  "lang": "en"
}
```

To learn how to use marshalling and unmarshalling, let's write a simple program that uses encoding/json package:

```
package main

import (
  "encoding/json"
  "fmt"
)

type Employee struct {
  FistName    string `json:"name"`
  LastName    string
  Internal    string `json:"-"`
  Mandatory   int    `json:"mandatory"`
```

```
  Zero        int    `json:"zero,omitempty"`
  iDoNotSeeIt int    `json:"notSeen"`
}

func main() {
  input := `{
    "name": "natalia",
    "lastName": "Buss"
  }`

  var empl Employee
  err := json.Unmarshal([]byte(input), &empl)
  if err != nil {
    // ...
    return
  }
  fmt.Println(empl.FistName)
  fmt.Println(empl.LastName)

  empl.Mandatory = 0
  empl.Zero = 0

  out, _ := json.Marshal(empl)
  fmt.Println(string(out))
}
```

Notice: you can build your custom Marshaller/Unmarsheller. `json` supports all data types.

Find out what `json.RawMessage` is? What is a use case for it?

5 Support POST and GET with gorilla/mux

If you want to build more complex web server, you should check gorilla/-mux:

```
package main

import (
```

```
  "fmt"
  "log"
  "net/http"
  "github.com/gorilla/mux"
)

func HelloGetHandler(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusOK)
  fmt.Fprintf(w, "GET")
}

func AddMsgHandler(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusOK)
  fmt.Fprintf(w, "Post")
}

func main() {
  r := mux.NewRouter()
  r.HandleFunc("/", HelloGetHandler).Methods("GET")
  r.HandleFunc("/", AddMsgHandler).Methods("POST")

  log.Fatal(http.ListenAndServe(":8080", r))
}
```

Refactor your application to use `gorilla/mux`.

6 Web app with memory storage

Build the following application, so we can add, display, and remove hello messages:

- `/hello_msg`, `POST` - add new hello message
- `/say_hello?user=natalia&lang=en`, `GET` - say hello
- `/hello_msg`, `GET` - list all messages
- `/hello_msg/{id}`, `DELETE` - remove hello message

Start as a simple array, later we can build it as a map.

7 ReadTimeout and WriteTimeout for http.Server

Remember that all IO operations should be cancel-able or timeout-able:

```
srv := &http.Server{
  Addr: "8080",
  Handler: h,
  ReadTimeout: 2s,
  WriteTimeout:2s,
  MaxHeaderBytes: 1 << 20,
}

srv.ListenAndServe()
```

8 Calling remote APIs

```
package main

import (
  "log"
  "net/http"
  "time"
)

func main() {
  c := &http.Client{
    Timeout: 2 * time.Second,
  }

  log.Println("Fetching...")
  resp, err := c.Get(
    "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json")

  if err != nil {
    log.Fatal(err)
  }
  defer resp.Body.Close()
}
```

Your task is to parse the output. While looking for the best way to parse it, use your writing-tests skills, so you do not DDOS mdn.github.io.

9 Build Hero API Client

Refactor the previous application and extract fetching list of herous to a `HeroClient`:

```
type HeroClient struct {
  Client *http.Client
}

func (c *HeroClient) GetThem() (string, error) {
  // your code
}
```

```
func main() {
  c := &http.Client{
    Timeout: 2 * time.Second,
  }

  hc := HeroClient{Client: c}

  // your code to read and display
  // superheroes JSON
}
```

10 Testing Calling remote APIs

You can also test whether your calls have proper format by using `httptest`:

```
package main

import (
  "fmt"
  "net/http"
  "net/http/httptest"
  "testing"

  "gotest.tools/assert"
)

func TestHeroClientAPI(t *testing.T) {

  server := httptest.NewServer(
    http.HandlerFunc(
      func(rw http.ResponseWriter, req *http.Request) {
        // Send response to be tested
        assert.Equal(t, req.URL.String(), "/some/path")
        rw.Write([]byte(`OK`))
      }),
  )

  // Close the server when test finishes
```

```
  defer server.Close()
  // Use Client & URL from our local test server
  api := HeroClient{server.Client()}
  r, err := api.GetThem()
  assert.NilError(t, err)
  fmt.Println(r)
}
```

Please refactor your code from previous exercise, add GET argument, and write the test.

11 Working with files

Based on https://gobyexample.com/writing-files and https://gobyexample.com/reading-files:

1. read /etc/passwd and find a line numer with your user
2. transform passwd to json (name, pid, gid, and path) and write to `${HOME}/passwd.json`

12 Parsing CLI args

Using an example from https://gobyexample.com/command-line-flags:

```
package main

import "flag"
import "fmt"

func main() {

    wordPtr := flag.String("word", "foo", "a string")

    numbPtr := flag.Int("numb", 42, "an int")
    boolPtr := flag.Bool("fork", false, "a bool")

    var svar string
    flag.StringVar(&svar, "svar", "bar", "a string var")
```

```
    flag.Parse()

    fmt.Println("word:", *wordPtr)
    fmt.Println("numb:", *numbPtr)
    fmt.Println("fork:", *boolPtr)
    fmt.Println("svar:", svar)
    fmt.Println("tail:", flag.Args())
}
```

build a program that prints all files or directories in a given *path*. The program should let us to specify regex for the file or directories names.

How would you test the CLI app?

13 Go Concurrency

Let's now take a look on the built-in concurrency:

- green threads (go routines)
- can run hundreds of thousands routines
- low overhead (dynamic stack)
- channels for communication
- scalable model

13.1 Goroutines and Channels

```
package main

import (
  "fmt"
  "time"
)

func say(s string) {
  for i := 0; i < 5; i++ {
    time.Sleep(100 * time.Millisecond)
    fmt.Println(s)
  }
}
```

```
func main() {
  go say("world")
  say("hello")
}
```

```
package main

import "fmt"

func sum(s []int, c chan int) {
  sum := 0
  for _, v := range s {
    sum += v
  }
  c <- sum // send sum to c
}

func main() {
  s := []int{7, 2, 8, -9, 4, 0}

  c := make(chan int)
  go sum(s[:len(s)/2], c)
  go sum(s[len(s)/2:], c)
  x, y := <-c, <-c // receive from c

  fmt.Println(x, y, x+y)
}
```

Based on the above examples, write a program that counts files in 2 directories. The first implementation should be sequential, the second - parallel.

13.2 sync.Mutex

How would you implement a global counter with sync.Mutex?

```
type SafeCounter struct {
  numberOfFiles  int
  mux sync.Mutex
}
```

```
//c.mux.Lock()
//c.mux.Unlock()
```

What are drawbacks of this solution?
Implement one counter with mutex and one with channel.

13.3 Further read

For more complex use cases:

- https://blog.golang.org/pipelines
- https://blog.golang.org/context

We will cover the Golang concurrency in the follow-up training.

14 Database Access

14.1 Postgres

Package `database/sql` provides generic interface for SQL databases. In our exe

1. Prepare the project

```
# anywhere
$ mkdir workshop-db
$ go mod init github.com/wojciech12/workshop-db
$ go get github.com/lib/pq
$ go get github.com/jmoiron/sqlx
```

2. Run psql:

```
# user: postgres
$ docker run --rm \
  --name workshop-psql \
  -e POSTGRES_DB=hello_world \
  -e POSTGRES_PASSWORD=nomoresecret \
  -d \
  -p 5432:5432 \
  postgres
```

Notice:

```
$ psql hello_world postgres -h 127.0.0.1 -p 5432
```

3. Connect to db:

```
package main

import (
  "database/sql"
  "fmt"
  "net/url"

  _ "github.com/lib/pq"
)

var driverName = "postgres"

func New(connectionInfo string) (*sql.DB, error) {
  db, err := sql.Open(driverName, connectionInfo)
  if err != nil {
    msg := fmt.Sprintf("cannot open db (%s) connection: %v",
      driverName, err)
    println(msg)
    return nil, err
  }
  return db, nil
}

func main() {
  user := url.PathEscape("postgres")
  password := url.PathEscape("nomoresecret")
  host := "127.0.0.1"
  port := "5432"
  dbName := "hello_world"
  sslMode := "disable"

  connInfo := fmt.Sprintf(
    "postgres://%s:%s@%s:%s/%s?sslmode=%s",
    user, password, host, port, dbName, sslMode)
```

```
  sql, err := New(connInfo)
  if err != nil {
    panic(err)
  }
  err = sql.Ping()
  if err != nil {
    panic(err)
  }
  defer sql.Close()
}
```

4. Let's create tables using the following definition:

```
CREATE TABLE users (
  id BIGSERIAL PRIMARY KEY,
  first_name TEXT,
  last_name TEXT);
```

5. Create table in Golang:

```
func createTableIfNotExist(sql *sql.DB) {
  _, err := sql.Exec(`CREATE TABLE users (
    id  BIGSERIAL PRIMARY KEY,
    first_name TEXT,
    last_name TEXT)`)
  fmt.Printf("%v\n", err)
}
```

6. Add lines:

```
func insertData(sql *sql.DB, firstName string,
    lastName string) error {
  iq := `INSERT INTO users (first_name, last_name)
         VALUES ($1,$2) RETURNING id;`
  stmt, err := sql.Prepare(iq)
```

```
  if err != nil {
    return err
  }
  defer stmt.Close()
  _, err = stmt.Exec(firstName, lastName)
  if err != nil {
    return err
  }
  return nil
}
```

6. Read lines:

```
func readData(sql *sql.DB) error {
  s := `SELECT id, first_name, last_name FROM users`
  rows, err := sql.Query(s)
  if err != nil {
    return err
  }
  defer rows.Close()

  type person struct {
    ID         int
    FirstName  string
    SecondName string
  }

  var p person
  for rows.Next() {
    if err := rows.Scan(
      &p.ID,
      &p.FirstName,
      &p.SecondName); err != nil {
      return err
    }
    fmt.Printf("%d %s %s", p.ID, p.FirstName, p.SecondName)
  }
  return nil
```

7. With sqlx[5], you can have more declarative code for working with your database:

```
dbx := sqlx.NewDb(sql, driverName)
```

```
func insertData2(sql *sqlx.DB, firstName string,
  lastName string) error {
  type input struct {
    FirstName string `db:"first_name"`
    LastName  string `db:"last_name"`
  }
  type output struct {
    ID int64 `db:"id"`
  }

  var out output
  var in input

  in.FirstName = firstName
  in.LastName = lastName

  sqlQuery := `INSERT INTO users ( first_name,
            last_name
          ) VALUES (
        :first_name,
        :last_name) RETURNING id`

  stmt, err := sql.PrepareNamed(sqlQuery)
  if err != nil {
    return err
  }
  err = stmt.Get(&out, in)
  if err != nil {
    return err
  }
  fmt.Println(out.ID)
  return nil
}
```

[5]https://github.com/jmoiron/sqlx

Notice: for select queries, you use `Queryx` and `err := rows.StructScan(&out)`.

8. Add support for the database in your web app.

14.2 Migrations

Presentation of golang-migrate/migrate[6].

14.3 Mongodb

A homework, prepare an application that uses mongodb as its database:

```
$ docker run  -p 27017:27017 \
  --name da-mongo \
   -d \
   mongo

# anywhere
$ mkdir workshop-mgo
$ go get github.com/globalsign/mgo
```

15 Logging

Most popular: `log`, `sirupsen/logrus`, and, for those who want to save every CPU cycle - `uber-go/zap`.

16 What is more in stdlib

With simple 25 reserve words and powerful standard library - `golang.org/pkg/`

- production ready http.Server
- cryptography (TLS, AES, RSA, HMAC, SHA, MD5)
- compression (gzip, zlib, lzw, bzip2, flate)
- filesystem modifications, subprocesses, system calls
- IO readers, writers, seekers, pipes
- SQL interface (third-party drivers needed)
- time, date
- http reverse proxy
- and more

[6] https://github.com/golang-migrate/migrate

17 Tools

17.1 goreleaser

A very sharp tool that greatly simplifies your CI/CD pipeline for Golang apps.

```
project_name: myapp
release:
  github:
    owner: YOUR_USER_OR_ORG
    name: myapp
  name_template: '{{.Tag}}'
builds:
- env:
  - CGO_ENABLED=0
  goos:
  - linux
  goarch:
  - amd64
  main: .
  ldflags: -s -w -X main.version={{.Version}} -X main.commit={{.Commit}} \
    -X main.date={{.Date}}
  binary: myapp
archive:
  format: tar.gz
  name_template: '{{ .ProjectName }}_{{ .Version }}_{{ .Os }}_{{ .Arch }}{{ if .Arm
    }}v{{ .Arm }}{{ end }}'
snapshot:
  name_template: snapshot-{{.ShortCommit}}
checksum:
  name_template: '{{ .ProjectName }}_{{ .Version }}_checksums.txt'
dist: dist
dockers:
  - image: YOUR_USER_OR_ORG/myapp
```

17.2 Docker

- Compile on your machine:
  `GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build ./...`
  and put just binary inside the Docker

- An alternative is to use multi-stage Docker builds
- Final image `alpine` or `ubuntu`

17.3 Benchmarks

Simple benchmarks with Go:

```
// fib.go
func Fib(n int) int {
        if n < 2 {
                return n
        }
        return Fib(n-1) + Fib(n-2)
}

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

```
$ go test -bench=.
```

17.4 Debugging

Debugging with `delve` and from vscode.

17.5 Docs

18 Outlook

What could be the next steps in learning Golang:

1. Go Concurrency:
    - Patterns: https://blog.golang.org/pipelines
    - Context: https://blog.golang.org/contexts

2. Graceful Shutdown, an example on gorilla/mux github
3. Running your app on Kuberentes and CloudNative
4. Serverless in Golang on AWS and GCP
5. Observability with Prometheus Stack, Opentracing, and EFK
6. GPRC ?
7. Golang Dev for K8S

Contact: `wbarczynski.pro@gmail.com`

19 References

- https://github.com/golang/go/wiki/CodeReviewComments
- https://golang.com/doc/effective_go.html