Golang Programming Workshop
Basics 2

Wojciech Barczynski

Contents

1 Golang project layout

- No rigid project structure;
- The best practice is to keep things simple;
- As project grows, you can move to more sophisticated directory structure.

Iteration 1:

```
|- _your_package_1
|- _your_package_2
|- _your_package_3
|- vendor/
|- main.go
|- go.mod
|- go.sum
\- README.md
```

Iteration 2:

```
|- cmd/
|  \- _your_app
|       \- main.go
|- _your_package_1
|- _your_package_2
|- vendor/
|- go.mod
|- go.sum
\- README.md
```

Iteration 3:

```
|- cmd/
|  \- _your_app
|       \- main.go
|- _your_package_1
|- _your_package_2
|- vendor/ # go mod download
|- tools/  # tools
|- go.mod
|- go.sum
\- README.md
```

Iteration 4 - `pkg` might be a good idea if you have other technologies in your repository:

```
|- cmd/
|  \- _your_app
|       \- main.go
|- _your_package_1
|- _your_package_2
|- pkg/
|- vendor/ # go mod download
|- tools/  # tools
|- go.mod
|- go.sum
\- README.md
```

Good source of inspiration:

- peter.bourgon.org/go-in-production/ (video);
- github.com/traefik/traefik.

2 Naming conventions in Golang

- camelCase: variables, function names, and files (go.dev/doc/effective_go#mixed-caps);
- Acronyms should be all capitals, as in ServeHTTP and IDProcessor;
- package names:
  - go.dev/doc/effective_go#names,
  - go.dev/blog/package-names,
  - following the business domain.

All the best practises for the naming variables in other languages apply to Golang:

- Consistent (easy to guess),

- Short (easy to type),
- Accurate.
- The greater the distance between a name's declaration and its uses, the longer the name should be.

All the best practises for naming functions in other languages apply to Golang as well:

- one function one thing,
- good name,
- not to long.

Notice: common mistake is to make Golang look like your favorite programming language.

3 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
}
```

```
package main

import "fmt"

func Divide(a, b int) (int, error) {
    if b == 0 {
        return 0, fmt.Errorf("can't divide '%d' by zero", a)
    }
    return a / b, nil
}
```

3.1 Convention

- No `try/catch` in Golang,
- Idiomatic way:

```
if err != nil {
        // often wrapped
        // not shown below
        return err
}

// or return
if err != nil {
        // logging the error
        return
}
```

- Returned as the last value:

```
func Divide(a, b int) (int, error) {
}
```

- error messages are usually written in lower-case and don't end in punctuation.

3.2 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)

// the old way:
if err == sql.ErrNoRows {
  // an error we know
}  else if err != nil {
  // another error
}
```

or

```
err := db.QueryRow("SELECT * FROM users WHERE id = ?", userID)

if err != nil {
        switch {
        // preferable
        case errors.Is(err, sql.ErrNoRows):
                // ...
        default:
                // ...
        }
}
```

Advantage? Simple. Disadvantage?

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

Please change the implementation of the Employee Mgmt application using *Package errors*[1]:

- return an error when the employee cannot take hollidays.

You will find the application in the github repo:
02_basics/examples/employee_mgmt.

3.3 Custom Error Types

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

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

```go
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:

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

```go
// new way
err := Foo()

if errors.As(&ParsingError{}) {
        // handling the error
}
```

Advantages? Disadvantages?

Notice that you can customzie both `Is` and `As`.

3.4 Opaque errors

The idea:

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

An example for Dave Cheney blog [2]:

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

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

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

3.5 Wrapping and Unwrapping errors

You should use this approach to errors as your default along with the custom type errors. Notice `%w` (stands for wrapping) and `errors.Is` in the code below:

```go
package main

import (
        "errors"
        "fmt"
)

var errAlertOn = errors.New("loud alert is on")

func disarmLock() error {
        // we tried
        // and failed :/
        return fmt.Errorf("cut wrong wire: %w", errAlertOn)
}

func breakSafe() error {
        err := disarmLock()
        return fmt.Errorf("failed to open the safe: %w", err)
}

func main() {
        err := breakSafe()

        if errors.Is(err, errAlertOn) {
                fmt.Printf("%+v", err)
        }
}
```

From go.dev/blog/go1.13-errors: *when adding additional context to an error, either with fmt.Errorf or by implementing a custom type, you need to decide whether the new error should wrap the original. There is no single answer to this question; it depends on the context in which the new error is created. Wrap an error to expose it to callers. Do not wrap an error when doing so would expose implementation details*

Your task: implement this error handling in your Employee managemetn application.

3.6 Stack trackes

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:

```go
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())
  }
}
```

Unwrapping:

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

```
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())
  }
}
```

4 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 panic and recover article.

5 Error best practices

- Handle errors once,
- Error, keep on the left,
- Notice: standard errors do not come with stacktraces,
- Read
  - An article from 8thlight [4],
  - Blog post on go erros - go.dev/blog/go1.13-errors.

Libraries:

- github.com/hashicorp/go-multierror
- github.com/go-errors/errors
- github.com/pkg/errors

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

6 Tests

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

```go
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!")
  }
}
```

6.1 Table-driven tests

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

```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`:

```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.
Notice: `go clean -testcache` to clean the cache.

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

6.2 Test with real X

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

6.3 Tests short and long

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

6.4 Integration tests

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

```
// +build integration

package service_test

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

To run:

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

6.4.1 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.

7 Your basic web app

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

7.1 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)
}
```

7.2 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))
}
```

7.3 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!",
  }))
}
```

7.3.1 Sharing data structures among handlers

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

```
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))
}
```

7.3.2 Reading body

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

```
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
```

7.3.3 Parse URL

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

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

7.3.4 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.

7.3.5 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())
  }
}
```

8 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?

9 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`.

10 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.

11 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()
```

12 Calling remote APIs

```go
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.

13 Build Hero API Client

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

```go
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
}
```

14 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.

15 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`

16 Parsing CLI args

For this exercise, we will use an example from 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?

The most popular Golang library for CLI application is github.com/spf13/cobra.

17 References

- https://github.com/golang/go/wiki/CodeReviewComments
- https://golang.com/doc/effective_go.html
- http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang