0.5 day Golang Programming Workshop

REST service

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.
- Knows how to work with go mod.

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

Notice: No copy&paste, please.

2 Your basic web app

Our first webapp. btw. What is a 12 factor app?[1]

2.1 Simplest

Writing a web server in Golang, thanks to a very solid standard library, is fairly 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)
}
```

2.2 Multiplexed

To multiplex, we need to create a Multiplexer:

```
package main

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

[1] https://12factor.net/

```
)

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))
}
```

2.3 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!",
  }))
}
```

2.4 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))
}
```

2.5 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
```

2.6 Parse URL

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

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

2.7 Simple hello-world

Let's make create a hello-world app with fixed dictionary of hello-world in different languages:

- `pl`: dzień dobry, dobry wieczor
- `en`: hi, welcome
- `de`: guten tag

Spec:

- Get hello-world, when *lang* and *user* come as a GET param
- if *lang* is missing, return 400.
- if *user* is missing, return 404.

2.8 Testing handlers

Create tests to cover the edge cases, use the following code for the start:

```
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())
  }
}
```

3 Prerequisites

3.1 Environment variables

Our application should get the configuration throught environments variables:

```
package main

import (
  "fmt"
  "os"
)

func main() {
  envValue, found := os.LookupEnv("LISTEN_PORT")
```

```
  if ! found {
    envValue = "8080"
  }
  fmt.Printf(envValue)
}
```

Check also libraries, e.g.,
github.com/jessevdk/go-flags.

3.2 Working with JSON

Let's give a user to add new hello messages, e.g.:

```
{
  "value": "Hi",
  "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?

3.3 REST, JSON, and composition

You can use composition to reuse common structures:

```
type Meta struct {
  MasterdataId string `json:"mdId"`
}

func GetName(data bytes.Buffer) {
  // private type
  type person struct {
    Name string `json:name`
    Meta
  }
```

```
  var p person
  err = json.Unmarshal(data.Bytes(), &p)
}
```

4 Level up web app

4.1 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`.

4.2 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.

4.3 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()
```

5 Calling remote web API

```
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*.

5.1 Build Hero API Client

Refactor the previous application and extract fetching list of heros 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
}
```

5.2 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.

6 Database Access

What a REST service is without a database, let's do it.

6.1 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[2], you can have more declarative code for working with your database:

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

```
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
}
```

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

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

6.2 Migrations

Presentation of golang-migrate/migrate[3].

6.3 Testing your database integration

In the Golang community, we test against real databases if we can. 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 Mongodb

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

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

Let's setup our project:

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

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

7 Best practises

1. Dependencies Injection, without the magic:

```
func main() {
    cfg := GetConfig()
    db, err := ConnectDatabase(cfg.URN)
    if err != nil {
        panic(err)
    }
    repo := NewProductRepository(db)
    service := NewProductService(cfg.AccessToken, repo)
    server := NewServer(cfg.ListenAddr, service)
    server.Run()
}
```

2. Dependencies direction from supporting pkgs to business logic pkgs.

3. `Context`

8 Observability

8.1 Monitoring with Prometheus

See https://github.com/wojciech12/talk_monitoring_with_prometheus

8.2 Logging with Logrus

Example for a talk on logging[4]

```
package main

import (
  "fmt"
  "net/http"

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

[4] https://github.com/wojciech12/talk_observability_logging

```
   log "github.com/sirupsen/logrus"
)

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

  log.WithFields(log.Fields{
      "method": r.Method,
      "handler": "hello",
  }).Info("hello!")
}

func WorldHandler(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusOK)
  fmt.Fprintf(w, "World!")

  log.WithFields(log.Fields{
      "method": r.Method,
      "handler": "world",
   }).Info("world!")
}

func ErrorHandler(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusOK)
  fmt.Fprintf(w, "Bye!")

  log.WithFields(log.Fields{
    "method": r.Method,
    "handler": "error",
  }).Error("What does 'bye' mean?!")
}

func main() {

  log.SetFormatter(&log.JSONFormatter{})

  r := mux.NewRouter()
  r.HandleFunc("/hello", HelloHandler)
  r.HandleFunc("/world", WorldHandler)
```

```
  r.HandleFunc("/error", ErrorHandler)
  http.ListenAndServe(":8080", r)
}
```

See also https://martinfowler.com/articles/domain-oriented-observability.html for a discussion on how and what to monitor.

9 Tools

9.1 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
```

9.2 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`

9.3 Performance tests

My favorite tool:

- wrk[5]
- wrk2[6] with `lua` scripting

[5] https://github.com/wg/wrk
[6] https://github.com/giltene/wrk2