Inteligentne Aplikacje Internetowe

Laboratorium 2

Opracowanie prostej aplikacji internetowej umożliwiającej ocenę obiektów oraz zapoznanie się z modułami i paczkami

Zadanie 1

Celem zadania jest przetestowanie prostego serwera http

Jednym z najprostszych sposobów uruchomienia serwera jest wywołanie funkcji **http.ListenAndServe** - pierwszym parametrem jest adres (można go pominąć) i port, drugi jest opcjonalny i będzie omówiony w kolejnych laboratoriach:

```
http.ListenAndServe(":8080", nil)
```

W momencie uruchomienia powyższej funkcji kod programu **zatrzymuje się** na tym poleceniu i rozpoczyna obsługę zapytań serwera. Zapytania serwera należy wcześniej ustawić, jednym z najprostszych sposobów jest obsługa wzorców (*ang.* pattern) i w momencie ich zgodności wywołanie konkretnej funkcji programu – służy do tego metoda **http.HandleFunc**. Przedstawia to poniższy przykład:

```
package main

import (
    "fmt"
    "net/http"
)

func indexFunc(w http.ResponseWriter, r *http.Request) {
    // zwrócenie strony głównej
    fmt.Fprintf(w, "<html><body>STRONA GŁÓWNA</body></html>")
}

func itemFunc(w http.ResponseWriter, r *http.Request) {
    // zwrócenie strony /item/* (* - oznacza dowolny ciąg znaków)
    fmt.Fprintf(w, "<html><body>STRONA ITEM<br>ADRES: ")
    fmt.Fprintf(w, r.RequestURI)
    fmt.Fprintf(w, "<br>METODA: ")
    fmt.Fprintf(w, r.Method)
    fmt.Fprintf(w, "</body></html>")
}

func main() {
    http.HandleFunc("/", indexFunc)
    http.HandleFunc("/item/", itemFunc)
    http.ListenAndServe("localhost:8080", nil)
}
```

W ramach pierwszej części zadania należy uruchomić powyższy przykład i przetestować adresy: localhost:8080/ localhost:8080/item/ localhost:8080/items localhost:8080/item/15
Wymuszenie zatrzymania serwera można uzyskać przez skrót w terminalu Ctrl + C

W powyższym przykładzie zawartość stron jest generowana "ręcznie". Istnieje jednak wiele innych możliwości przesyłania treści stron internetowych oraz plików, np.:

- Zwracanie statycznych plików:

```
http.ServeFile(w, r, "pages/strona.html")
```

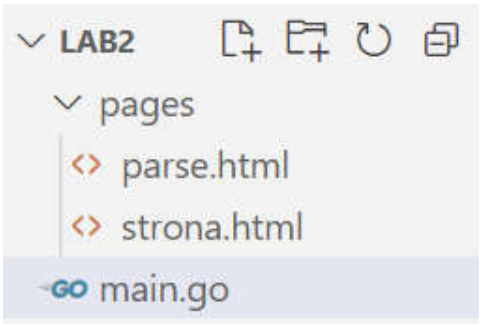
- Zwracanie parsowanych plików .html (w takich plikach można umieszczać treść dynamicznie), czyli wykorzystanie tzw. szablonów (*ang.* template):

```
tmpl, _ := template.ParseFiles("pages/parse.html")
tmpl.Execute(w, [data])
```

W powyższym kodzie [data] oznacza opcjonalną strukturę danych przekazywanych do strony. Znak _ oznacza że drugi parameter zwracany przez ParseFiles nie będzie wykorzystywany. Podmianę danych umożliwia zapis na stronie .html pomiędzy nawiasami {{ oraz }}. Dostęp do pól struktury odbywa się przez zapis: .NazwaPola (pola struktury muszą być publiczne – pisane z dużych liter – więcej o dostępie do pól w kolejnych laboratoriach).

Do przetestowania powyższych funkcjonalności należy dodać do folderu roboczego folder **pages** oraz umieścić w nim pliki: strona.html (o dowolnej treści HTML) i parse.html o następującej zawartości:

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
        <table>
            <tr>
                <th>Imię</th>
                <th>Indeks</th>
            </tr>
            <tr>
                <td>{{ .Name }}</td>
                <td>{{ .Index }}</td>
            </tr>
        </table>
    </body>
</html>
```



Następnie należy dodać w funkcji main obsługę dwóch kolejnych wzorców (oczywiście przed rozpoczęciem startu serwera):

```
    http.HandleFunc("/strona/", stronaFunc)
    http.HandleFunc("/parse/", parseFunc)
```

Do kodu programu należy również dodać strukturę student (będzie zawierała pola .Name oraz .Index, które będą przekazywane do strony parse.html) oraz funkcje stronaFunc i parseFunc. Implementację tych elementów przedstawia następujący kod źródłowy:

```
func stronaFunc(w http.ResponseWriter, r *http.Request) {
    // zwrócenie statycznej strony strona.html
    http.ServeFile(w, r, "pages/strona.html")
}

type student struct {
    Name  string
    Index int
}

func parseFunc(w http.ResponseWriter, r *http.Request) {
    // zwrócenie strony o dynamicznej zawartości
    tmpl, _ := template.ParseFiles("pages/parse.html")
    tmpl.Execute(w, &student{"Jan", 12345})
}
```

W celu realizacji drugiej części zadania należy przetestować adresy w przeglądarce:
localhost:8080/strona/ localhost:8080/parse/

Następnie należy zmienić pola struktury student na małą literę (name, index) oraz właściwości w pliku parse.html na {{ .name }} oraz {{ .index }} i sprawdzić czy strona działa poprawnie.

Zadanie 2

Celem zadania jest stworzenie prostego serwera zapisującego odsłony produktów do pliku. Na podstawie takich odsłon możliwe jest późniejsze ocenienie popularności produktów.

1. Utworzyć nowy folder roboczy (np. LAB2-Z2).
2. Utworzyć plik **main.go** z funkcją main oraz folder **pages**
3. W pliku main.go utworzyć strukturę ***item*** zawierającą następujące pola:
   Name (string), Price (float)
4. Utworzyć serwer http obsługujący stronę /item/ przez funkcję itemFunc.
5. Funkcja itemFunc powinna działać w sposób dynamiczny:
   - jeżeli podano adres /item/ powinna wyświetlać statyczną stronę .html z trzema linkami do produktów: /item/0/, /item/1/, /item/2/
   - jeżeli podano adres /item/nr/ powinna wyświetlać stronę z danymi produktu o wybranym identyfikatorze. Dane produktów powinny być odczytywane z tablicy globalnej podanej dalej jako przykład. Realizację tej części zadania należy zacząć od dodania następującego kodu:

```
var items = [3]item{
    {"OnePlus 10 Pro", 4500}, {"Asus X513EA", 2000},
    {"Samsung UE55TU7022K", 1900},
}

func getItemID(url string) int {
    fields := strings.Split(url, "/")
    if len(fields) < 2 || fields[2] == "" { return -1 } // - /item/
    id, err := strconv.Atoi(fields[2])
```

```
    if err != nil { return -2 } // - /item/not_a_number
    return id // /item/nr/
}

func itemFunc(w http.ResponseWriter, r *http.Request) {
    id := getItemID(r.RequestURI) // pobranie ID przedmiotu z adresu strony
    // zadanie (miejsce poniżej na własny kod)
    // gdy id < 0 - wyświetlić stronę z listą linków /item/0/, /item/1/, /item/2/
    // gdy id >= 0 - wyświetić stronę z informacjami o przedmiocie
    // korzystanie z tablicy globalnej: - &items[id]
}
```

6. Po implementacji kodu sprawdzić działanie programu dla następujących adresów:
   localhost:8080/item/ localhost:8080/item/0 localhost:8080/item/1
   localhost:8080/item/2 localhost:8080/item/3 localhost:8080/item/zet
7. Zmodyfikować kod funkcji itemFunc tak, aby przy wyświetlaniu informacji o konkretnym produkcie program zapisywał do pliku logs.txt identyfikator przedmiotu oraz bieżącą datę. Dopis danych do plików może wyglądać następująco:

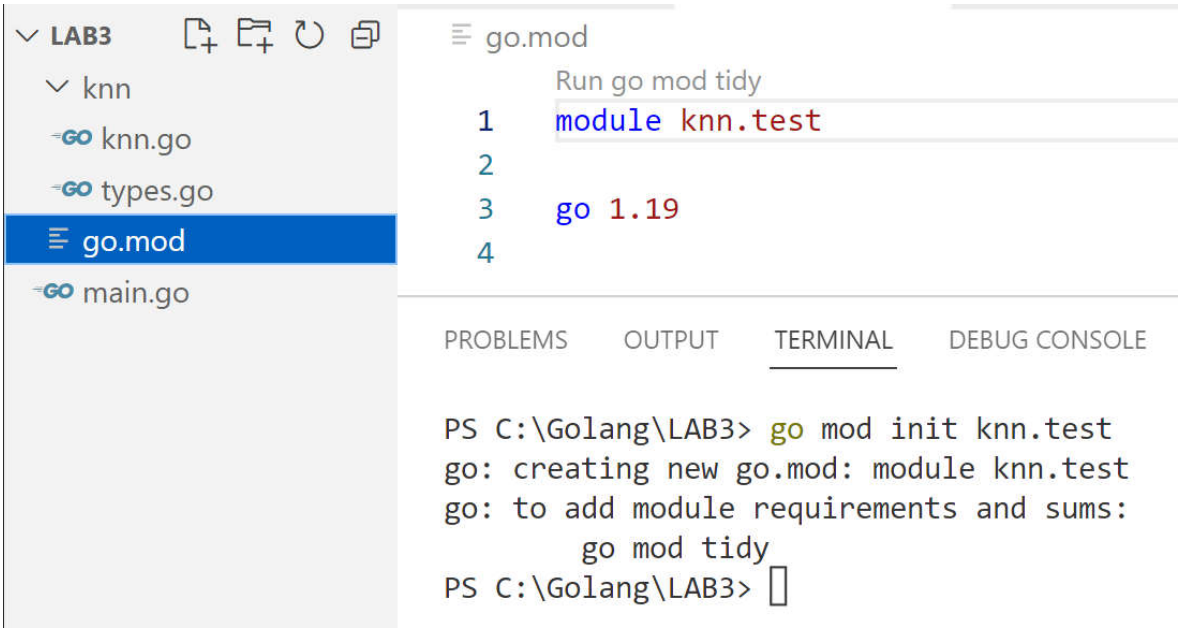
```
file, _ := os.OpenFile("logs.txt", os.O_APPEND|os.O_CREATE, 0600)
file.WriteString(fmt.Sprint(id) + "\t" + time.Now().String() + "\n")
file.Close()
```

8. Wynik zaprezentować prowadzącemu w celu zaliczenia laboratorium.
9. **Zadanie na plus**
   Rozszerzyć działanie programu tak, aby w przypadku gdy przedmiotu nie ma w tablicy items wyświetlała się strona z błędem – nie znaleziono przedmiotu. Obsługę błędu dodać również w przypadku gdy identyfikator produktu jest równy -2 (podanie ciągu znaków w adresie zamiast numeru produktu).
10. **Zadanie na dwa plusy**
    Zamiast przedmiotów (item) wyświetlać na stronie informacje o samochodach (samochody wczytać tak jak w poprzednim laboratorium z pliku **cars.txt**, ale wynik funkcji wczytującej umieścić w zmiennej globalnej **cars**). Znaleźć samodzielnie sposób na wyświetlenie strony z wszystkimi linkami do podstron z samochodami.

Moduły i paczki

Programy w języku Go można dzielić na mniejsze paczki (nazywane też pakietami). Przyjęło się aby dla każdej paczki tworzyć osobny folder, a **paczki nazywać tak samo jak folder**. W każdym folderze może być wiele plików tworzących razem jedną paczkę, a deklarowane zmienne, funkcje i struktury z jednego pliku są widoczne przez kompilator w innym pliku z tego samego folderu (czasem wymaga to kompilacji projektu lub folderu – można to wykonać przez go build . przechodząc do folderu).

Zbiór paczek (pakietów) jest nazywany modułem. Aby moduł działał prawidłowo i była możliwość poprawnego korzystania z jednych paczek w innych paczkach należy taki moduł stworzyć. Służy do tego polecenie w terminalu: **go mod init nazwa_modułu.** Pracę z paczkami i modułami przedstawia przykład:



Na powyższym zrzucie ekranu widać **folder knn** (paczka knn) i dwa pliki **knn.go** oraz **types.go** (w każdym z tych plików musi być nagłówek package knn – zgodnie z nazwą paczki/folderu). Ponadto w folderze pojawił się plik **go.mod** (został on wygenerowany automatycznie po wprowadzeniu komendy **go mod init knn.test**). Od tego momentu tworzony projekt stanowi moduł o nazwie **knn.test**. Aby w paczkach lub pliku main.go korzystać z innych paczek należy użyć import i podać nazwę modułu oraz nazwę paczki. Pokazuje to przykład poniżej:

Plik **knn.go** (paczka knn):

```
package knn

import (
    "fmt"
)

func TestFunkcjiPaczkiKNN() {
    fmt.Println("TEST")
}
```

Plik **main.go** (plik główny – nie umieszczony w żadnym folderze):

```
package main

import (
    "knn.test/knn"
)

func main() {
    knn.TestFunkcjiPaczkiKNN()
}
```

Uwaga, import paczki z folderu należy w tym wypadku często napisać ręcznie.

Uwaga, przy wykorzystaniu funkcji, struktur i danych z innych paczek należy ich nazwy poprzedzić nazwą paczki (stąd knn.TestFunkcjiPaczkiKNN())

Uwaga, jedynie funkcje, struktury i dane pisane z dużej litery są dostępne dla innych paczek. Stworzenie funkcji o nazwie np. testFunkcji() nie pozwoli jej użyć w innych paczkach (funkcja będzie jednak dostępna w ramach paczki – można ją traktować jako funkcję prywatną)

Wynik działania:

```
PS C:\Golang\LAB3> go run main.go
TEST
```

Zadanie 3

Utworzyć moduł o dowolnej wybranej przez siebie nazwie. W ramach modułu utworzyć dwa foldery:

- mat (w środku plik func.go z zawartością poniżej)
- test (w środku plik test.go z dowolną napisaną przez siebie funkcją)

W ramach zaliczenia zadania stworzyć plik main.go i wykorzystać w nim funkcje z obu paczek. Samodzielnie dodać słowa kluczowe import i nazwy paczek (kolejne paczki podaje się w nowej linii).

Zawartość pliku **func.go** z folderu **mat**:

```go
package mat

func MinMax(tab []int) (int, int) {
    if len(tab) < 1 {
        return 0, 0
    }
    min, max := tab[0], tab[0]
    for i := 1; i < len(tab); i++ {
        if tab[i] < min {
            min = tab[i]
        }
        if tab[i] > max {
            max = tab[i]
        }
    }
    return min, max
}
```