Tworzenie Aplikacji Internetowych
Laboratorium 12

Technologie po stronie serwera – Język Go – GET, POST i wykorzystanie bibliotek

Dane pomiędzy stronami html można przesyłać za pomocą znanych już metod GET oraz POST.

GET

W przypadku języka Go, dane przesyłane za pomocą GET można wyciąć ręcznie z adresu URL strony żądania. Np. posiadając wzorzec /strona/ i otwierając w przeglądarce stronę /strona/artykul/15 można dostać się do słowa np. określającego typ treści (artykuł) oraz jej numeru (15) i wyświetlić użytkownikowi odpowiednie dane. Wycięcie tych wartości w sposób ręczny wygląda następująco:

```
func stronaFunc(w http.ResponseWriter, r *http.Request) {
    dane := strings.Split(strings.Trim(r.URL.Path, "/"), "/")
    for i := 0; i < len(dane); i++ {
        fmt.Println(i, dane[i])
    }
}

func main() {
    http.HandleFunc("/strona/", stronaFunc)
    http.ListenAndServe(":8080", nil)
}
```

Powyższy kod dla wywołania http://localhost:8080/strona/artykul/15 wyświetli:

0 strona
1 artykul
2 15

Obsługa treści strony w taki sposób (np. wyświetlanie różnych artykułów zależnie od numeru podanego przez parametr strony) może być uciążliwa – należałoby sprawdzać liczbę parametrów, rzutować je na konkretne typy, a dopiero później przetwarzać (dbając również o odpowiednią obsługę błędów). Tą czynność można jednak uprościć korzystając z różnych bibliotek (dalsza część instrukcji)

POST

W przypadku danych przesyłane przez formularze i metodę POST odczyt danych jest prostszy. Dla przykładowego formularza:

```
    <form method="post" action="/strona/">
        <input name="artykul">
        <input type="submit" value="Wyślij">
    </form>
```

Dane możemy odczytać następująco:

```
func stronaFunc(w http.ResponseWriter, r *http.Request) {
    ids := r.PostFormValue("artykul")
    fmt.Println(ids)
}
```

Biblioteka - Gorilla/mux

Biblioteka Gorilla/mux upraszcza trasowanie stron serwera według konkretnych wzorców, metod, schematów itp.. Najprostszą funkcjonalność przedstawia przykład poniżej. Tworząc HandleFunc dla /item/{id}, możemy pobrać wartość pola {id} z adresu GET w bardzo prosty sposób:

```
package main

import (
    "github.com/gorilla/mux"
)

func itemFunc(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)                // get variables from router as map
    id, _ := strconv.Atoi(vars["id"]) // get id value from map
    fmt.Println("Id: ", id)            // print id value
    tmpl, _ := template.ParseFiles("pages/page.html")
    tmpl.Execute(w, nil)
}

func main() {
    router := mux.NewRouter()                    // new router
    router.HandleFunc("/item/{id}", itemFunc)    // routing /item/(*)/
    http.ListenAndServe(":8080", router)         // start serwer with routing
}
```

Przykładowe wzorce:
/item/{klucz} - /item/(*) - pobranie vars["klucz"]
/product/{category}/{id:[0-9]+} - /product/(*)/(*numer) - vars["category"] oraz vars["id"]

Inne możliwości specjalizacji routingu:
.Host("host") // obsługuje tylko żądania z danego hosta (nazwa akceptuje wyrażenia regularne)
.PathPrefix("prefix") // obsługuje tylko żądania z danym prefiksem w adresie
.Methods("GET") // obsługuje tylko żądania GET
.Methods("POST") // obsługuje tylko żądania POST (podobnie działa to dla PUT itp.)
.Methods("GET", "POST") // obsługuje żądania GET i POST
.Schemes("https") // tylko połączenia szyfrowane
.Headers("…", "…") // tylko żądania konkretnego typu

Istnieje możliwość łączenia powyższych ustawień, przykład:

```
router.HandleFunc("/item/{id}", aptFunc).Methods("GET")
```

Pozostałą część funkcjoanlności związaną z wyświetlaniem stron można wykonywać w poznany wcześniej sposób za pomocą wzorców i przekazywania do nich struktur.

Zadanie 1

1. Utworzyć nowy projekt w języku - wybrać katalog, utworzyć plik main.go oraz moduł: go mod init lab12
2. Pobrać bibliotekę gorilla/mux
   go get -u github.com/gorilla/mux

3. Podejrzeć plik go.mod oraz go.sum (powinny zawierać informacje o pobranej bibliotece)
4. Dodać następujący routing:

```
r := mux.NewRouter()
r.HandleFunc("/page/{mode}/{id:[0-9]+}", pageFunc)
http.ListenAndServe(":8080", r)
```

5. Obsłużyć go następującą funkcją:

```
func pageFunc(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    page := vars["mode"]
    id := vars["id"]
    fmt.Fprintf(w, "<html><body>PG: %v<br>ID: %v</body></html>", page, id)
}
```

6. Przetestować rozwiązanie dla różnych adresów stron internetowych
7. Dodać analogicznie obsługę strony /losuj/[a]/[b]/, gdzie [a] może być dowolną liczbą, oraz [b] może być dowolną liczbą. Strona powinna wyświetlać [a] liczb losowych z zakresu od [0] do [b]. Należy pamiętać o przekonwertowaniu parametrów string na int, a następnie odpowiednie dane przekazać do strony html (ręcznie, lub za pomocą template).
8. Zaprezentować wynik prowadzącemu
9. **(na plus)** dodać obsługę błędów (wyświetlać na stronie odpowiednie komunikaty gdy a jest mniejsze od 0 lub b jest mniejsze od 2).

Inne warte uwagi biblioteki:

Biblioteka – a-h/templ

Powyższa biblioteka umożliwia bardziej przyjemne tworzenie szalbonów stron HTML i podmieniania w ich treści, jej instalacja i wykorzystanie wykracza jednak poza zakres laboratorium.

Biblioteka - patrikeh/go-deep

Biblioteka ta umożliwia tworzenie i uczenie różnych systemów AI (jest dosyć prostą implementacją i posiada niewielką funkcjonalność, natomiast jest warta uwagi).

Zadanie 2

1. Zainstalować w nowym (lub bieżącym) projekcie bibliotekę go-deep:
   go get -u github.com/patrikeh/go-deep
2. Przetestować uczenie i działanie prostej sieci neuronowej (kod na następnej stronie)
3. Utworzyć serwer HTTP, który będzie:
   - Posiadał zmienną globalną zawierającą obiekt nauczonej sieci neuronowej (zmienna n z przykładu z następnej strony)
   - Zwracał dane JSON z wynikiem predykcji sieci dla następującego adresu: /predykcja/[input1]/[input2]
   - Posiadał obsługę adresu /resetuj/, przy której sieć neuronowa będzie uczona od nowa, a wynik błędu będzie wyświetlany na stronie
4. Wynik zaprezentować w celu zaliczenia laboratorium
5. **(na plus)** dodać obsługę /konfiguracja/ z formularzem, za pomocą którego użytkownik będzie mógł podać nowe parametry konfiguracji sieci (dotyczące Layout oraz Bias)

```
    // inicjalizacja ziarna losowania
    rand.Seed(time.Now().UTC().UnixNano())
    // dane uczące
    var data = training.Examples{
        {Input: []float64{2.7810836, 2.550537003}, Response: []float64{0}},
        {Input: []float64{1.465489372, 2.362125076}, Response: []float64{0}},
        {Input: []float64{3.396561688, 4.400293529}, Response: []float64{0}},
        {Input: []float64{1.38807019, 1.850220317}, Response: []float64{0}},
        {Input: []float64{7.627531214, 2.759262235}, Response: []float64{1}},
        {Input: []float64{5.332441248, 2.088626775}, Response: []float64{1}},
        {Input: []float64{6.922596716, 1.77106367}, Response: []float64{1}},
        {Input: []float64{8.675418651, -0.242068655}, Response: []float64{1}},
    }
    // utworzenie sieci neuronowej (2 wejścia, 2-2-1 neuronów)
    n := deep.NewNeural(&deep.Config{
        Inputs:     2,
        Layout:     []int{2, 2, 1},
        Activation: deep.ActivationSigmoid,
        Mode:       deep.ModeBinary,
        Weight:     deep.NewNormal(1.0, 0.0),
        Bias:       true,
    })
    // optymalizacja sieci neuronowej
    optimizer := training.NewSGD(0.05, 0.1, 1e-6, true)
    trainer := training.NewTrainer(optimizer, 50)
    training, heldout := data.Split(0.5)
    trainer.Train(n, training, heldout, 1000)
    // wyświetlenie predykcji dla danych
    blad := 0.0
    for i := 0; i < len(data); i++ {
        predykcja := n.Predict(data[i].Input)
        prawidlowo := data[i].Response
        blad += math.Abs(predykcja[0] - prawidlowo[0])
        fmt.Println("PREDYKCJA:", predykcja, ", PRAWIDŁOWO:", prawidlowo)
        fmt.Println(", BLAD:", blad)
    }
    fmt.Println("BŁĄD (SUMA):", blad)
```