Solution Review: Web Application for Statistics

This lesson discusses the solution to the challenge given in the previous lesson.

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
        "fmt"
        "log"
        "net/http"
        "sort"
        "strconv"
        "strings"
)

type statistics struct {
        numbers []float64
        mean    float64
        median  float64
}

const form = `<html><body><form action="/" method="POST">
<h1>Statistics</h1>
<h5>Compute base statistics for a given list of numbers</h5>
<label for="numbers">Numbers (comma or space-separated):</label><br>
<input type="text" name="numbers" size="30"><br />
<input type="submit" value="Calculate">
</form></html></body>`

const error = `<p class="error">%s</p>`

var pageTop = ""
var pageBottom = ""

func main() { // Define a root handler for requests to function homePage, and start the webse
        http.HandleFunc("/", homePage)
        if err := http.ListenAndServe(":3000", nil); err != nil {
                log.Fatal("failed to start server", err)
        }
}

func homePage(writer http.ResponseWriter, request *http.Request) { // Write an HTML header, p
        writer.Header().Set("Content-Type", "text/html")
        writer.Header().Set("Content-Type", "text/html")
        err := request.ParseForm() // Must be called before writing response
        fmt.Fprint(writer, pageTop, form)
        if err != nil {
                fmt.Fprintf(writer, error, err)
        } else {
                if numbers, message, ok := processRequest(request); ok {
                        stats := getStats(numbers)
                        fmt.Fprint(writer, formatStats(stats))
                } else if message != "" {
                        fmt.Fprintf(writer, error, message)
                }
        }
        fmt.Fprint(writer, pageBottom)
}

func processRequest(request *http.Request) ([]float64, string, bool) { // Capture the numbers
        var numbers []float64
        if slice, found := request.Form["numbers"]; found && len(slice) > 0 {
                text := strings.Replace(slice[0], ",", " ", -1)
                for _, field := range strings.Fields(text) {
                        if x, err := strconv.ParseFloat(field, 64); err != nil {
                                return numbers, "'" + field + "' is invalid", false
                        } else {
                                numbers = append(numbers, x)
                        }
                }
        }
        if len(numbers) == 0 {
                return numbers, "", false // no data first time form is shown
        }
        return numbers, "", true
}

func getStats(numbers []float64) (stats statistics) { // sort the values to get mean and medi
        stats.numbers = numbers
        sort.Float64s(stats.numbers)
        stats.mean = sum(numbers) / float64(len(numbers))
        stats.median = median(numbers)
        return
}

func sum(numbers []float64) (total float64) { // seperate function to calculate the sum for m
        for _, x := range numbers {
                total += x
        }
        return
}

func median(numbers []float64) float64 { // seperate function to calculate the median
        middle := len(numbers) / 2
        result := numbers[middle]
        if len(numbers)%2 == 0 {
                result = (result + numbers[middle-1]) / 2
        }
        return result
}

func formatStats(stats statistics) string {
        return fmt.Sprintf(`<table border="1">
<tr><th colspan="2">Results</th></tr>
<tr><td>Numbers</td><td>%v</td></tr>
<tr><td>Count</td><td>%d</td></tr>
<tr><td>Mean</td><td>%f</td></tr>
<tr><td>Median</td><td>%f</td></tr>

</table>`, stats.numbers, len(stats.numbers), stats.mean, stats.median)
}
```

In the code above, we define a struct `statistics` at **line 11** to contain all the input data: `numbers` of type `[]float64` and the calculation results on them: `mean` and `median`.

As we have seen before, the HTML string for the web page is contained in a constant form, defined from **line 17** to **line 23**. We also define a constant for error reporting at **line 25**. The `main()` routine is very succinct, defining a root handler for requests to a function `homePage`, and starting the web server combined with error-handling from **line 32** to **line 34**.

Now, look at the header of the `homePage()` function at **line 37**. It starts by writing an HTML header at **line 38** and then calls the `ParseForm method` at **line 39**. Then, we use `Fprint` at **line 40** to write the form HTML to the `ResponseWriter` writer. We check for an error at **line 41**. If there is one, we print it out at **line 42**. If not, we call the `processRequest` function at **line 44**. This takes the request value as a parameter, and returns the input numbers.

Now, look at the header of the `processRequest()` function at **line 54**. At **line 56**, we see that the result of `request.Form["numbers"]` is captured in a slice. **Line 57** replaces all commas in the input by spaces. Then, we iterate over all the fields at **line 58**. We convert each field to a float at **line 59**. If there is an error, we return an error string as the 2nd return value, and false as the 3rd at **line 60**. If no error, we append the float to `numbers` at **line 62**. **Line 66** checks whether there are input numbers; if not, this is the first time the form is shown. If everything goes ok, we return the `numbers` slice together with an empty error string and true at **line 69**.

Coming back to **line 44**, we call the function `getStats` on the `numbers` at **line 45**, if everything is ok. The results from `getStats` are stored in struct value `stats`, which are sent to the web server output with `Fprint` at **line 46**, after being transformed by `formatStats`. **Line 47** handles the case of a possible error and **line 51** prints a `pageBottom` output.

Now, look at the header of the `getStats()` function at **line 72**. It first sorts the `numbers` at **line 74** using the `sort` package. Then, it calculates the *mean* and *median* values at **line** 75 and **line** 76, respectively, and stores them in the `stats` struct, which is returned at **line** 77.

We have separate functions for calculating the *sum* and *median* of values. To calculate the sum a function `sum` is implemented from **line 80** to **line 85**. To calculate the median a function `median` is implemented from **line 87** to **line 94**. They use simple mathematics to calculate these results.

Now, look at the header of `formatStats` function at **line 96**. It returns an HTML

markup and uses `Sprintf` to embed the input values and the calculated values. More specifically, it substitutes in the values of `stats.numbers`, `len(stats.numbers)`, `stats.mean`, `stats.median` in respectively **%v**, **%d**, **%f** and **%f** (from **line 98** to **line 102**).

That is it for the solution. In the next lesson, we'll discuss how to make an application robust over the web.