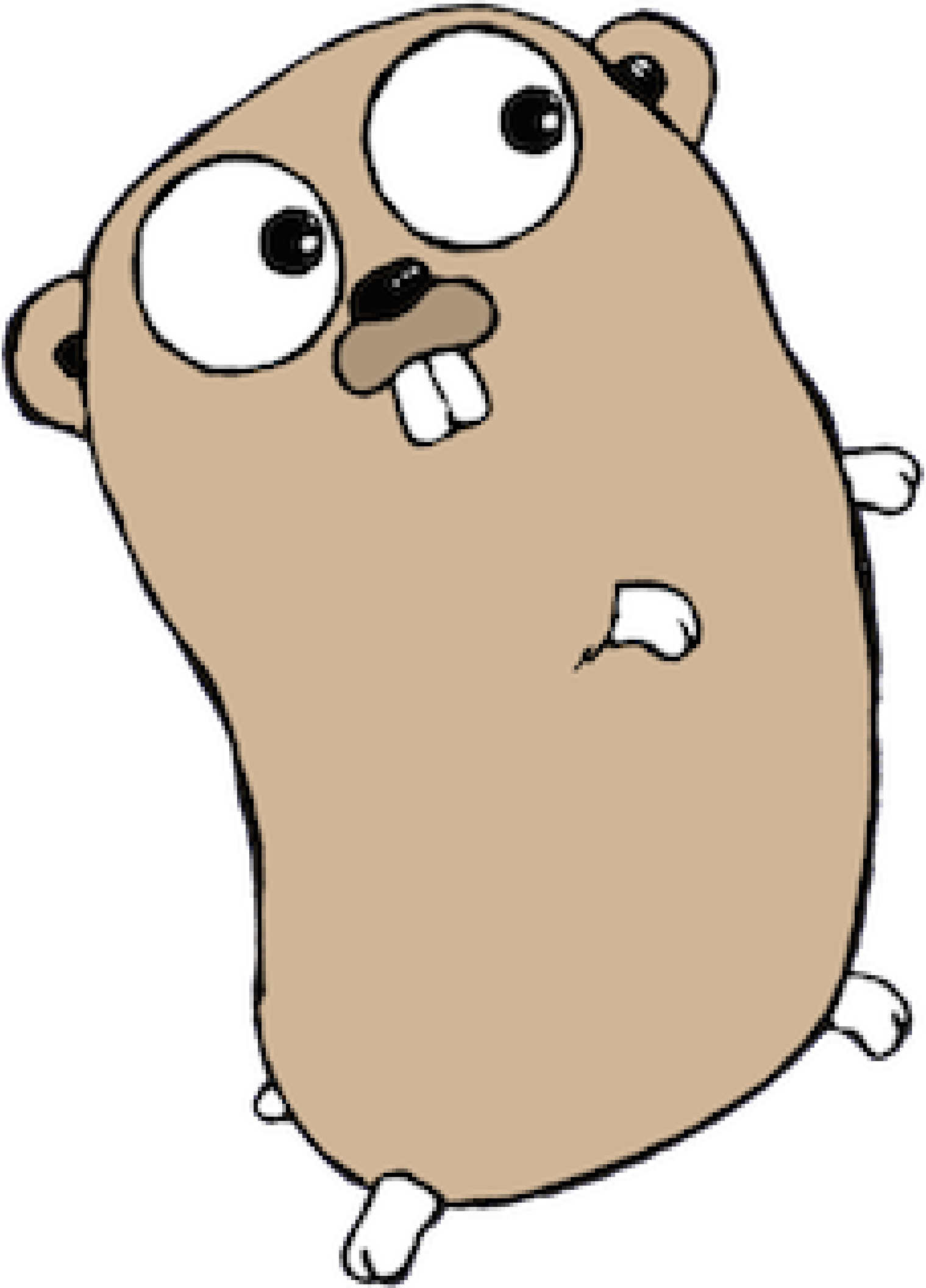

How do I use the template package and handle forms?

Satish Talim

This book is for sale at http://leanpub.com/howtousethetemplatepackageandhandleforms

This version was published on 2015-04-05

Also By Satish Talim

How Do I Write And Deploy Simple Web Apps With Go?

Building a package in Go

How do I use Sourcegraph with Go?

How do I use Sourcegraph with Ruby?

How to Deploy a Go Web App to the Google App Engine 101

How to Deploy a Go Web App to Heroku 101

New to Go? Want to know how to use the template package and handle forms? This eBook quickly guides you to do exactly that.

Contents

Package template

text/template

Usage: `import "text/template"`

Most server-side languages have a mechanism for taking predominantly static pages and inserting a dynamically generated component, such as a list of items. Typical examples are scripts in Java Server Pages, PHP scripting and many others. Go has adopted a relatively simple scripting language in the template[1] package.

The package is designed to take text as input and output different text, based on transforming the original text using the values of an object.

To generate HTML output, we shall soon use package `html/template`, which has the same interface as this package but automatically secures HTML output against certain attacks.

The original source is called a `template` and will consist of text that is transmitted unchanged, and embedded commands which can act on and change text. The commands are delimited by `{{ ... }}`, similar to the JSP commands `<%= ... =%>` and PHPs `<?php ... ?>`.

A template is applied to a Go object. Fields from that Go object can be inserted into the template, and you can "dig" into the object to find sub-fields, etc. The current object is represented as `.`, so that to insert the value of the current object as a string, you use `{{.}}`. The package uses the `fmt` package by default to work out the string used as inserted values.

To insert the value of a field of the current object, you use the field name prefixed by `.`. For example, if the object is of type:

```
type Student struct {
        Name string
}
```

then you insert the values of `Name` by `The name is {{.Name}}`.

Thus, templates are a way to merge generic text with more specific text i.e. retain the content that is common in the template and then substitute the specific content as required.

The syntax of such definitions is to surround each template declaration with a "define" and "end" action.

The define action names the template being created by providing a string constant. Here is a simple example:

[1] http://golang.org/pkg/text/template

```
{{define "T1"}}ONE{{end}}
{{define "T2"}}TWO{{end}}
{{define "T3"}}{{template "T1"}} {{template "T2"}}{{end}}
{{template "T3"}}
```

This defines two templates, T1 and T2, and a third T3 that invokes the other two when it is executed. Finally it invokes T3. If executed this template will produce the text

```
ONE TWO
```

In Go, we use the `template` package and methods like `Parse`, `ParseFile`, `Execute` to load a template from a string or file and then perform the merge. The content to merge is within a defined type and that has exported fields, i.e. fields within the `struct` that are used within the template have to start with a capital letter.

Let us look at a simple example.

Make a new folder and cd to it as follows:

```
$ mkdir $GOPATH/src/github.com/SatishTalim/texttmpl
$ cd $GOPATH/src/github.com/SatishTalim/texttmpl
```

In this folder write the program `stud_struct.go` as follows:

Program stud_struct.go

```go
package main

import (
        "log"
        "os"
        "text/template"
)

type Student struct {
        //exported field since it begins
        //with a capital letter
        Name string
}

func main() {
        //define an instance
        s := Student{"Satish"}

        //create a new template with some name
        tmpl := template.New("test")

        //parse some content and generate a template
        tmpl, err := tmpl.Parse("Hello {{.Name}}!")
        if err != nil {
                log.Fatal("Parse: ", err)
                return
        }

        //merge template 'tmpl' with content of 's'
        err1 := tmpl.Execute(os.Stdout, s)
        if err1 != nil {
                log.Fatal("Execute: ", err1)
                return
        }
}
```

You can now run the program by typing:

```
$ go run stud_struct.go
```

The output is:

```
Hello Satish!
```

Note:

- `New` allocates a new template with the given name.
- `Parse` parses a string into a template.
- To include the content of a field within a template, enclose it within curly braces and add a dot at the beginning. E.g. if `Name` is a field within a `struct` and its value needs to be substituted while merging, then include the text `{{.Name}}` in the template. Do remember that the field name has to be present and it should also be exported (i.e. it should begin with a capital letter in the type definition), or there could be errors. All text outside `{{.Name}}` is copied to the output unchanged.
- We have used the predefined variable `os.Stdout` which refers to the standard output to print out the merged data - `os.Stdout` implements `io.Writer`.
- `Execute` applies a parsed template to the specified data object, and writes the output to `os.Stdout`.

Pipelines

A pipeline may be "chained" by separating a sequence of commands with pipeline characters `'|'`. In a chained pipeline, the result of each command is passed as the last argument of the following command. The output of the final command in the pipeline is the value of the pipeline.

The output of a command will be either one value or two values, the second of which has type `error`. If that second value is present and evaluates to non-nil, execution terminates and the error is returned to the caller of `Execute`.

Let us look at an example.

Make a new folder and cd to it as follows:

```
$ mkdir $GOPATH/src/github.com/SatishTalim/person
$ cd $GOPATH/src/github.com/SatishTalim/person
```

In this folder write the program `person.go` as follows:

Program person.go

```
package main

import (
        "log"
        "os"
        "text/template"
)

type Person struct {
        Name   string
        Emails []string
}

const tmpl = `The name is {{.Name}}.
{{range .Emails}}
    His email id is {{.}}
{{end}}
`

func main() {
        person := Person{
                Name:   "Satish",
                Emails: []string{"satish@rubylearning.org", "satishtalim@gmail.com"},
        }

        t := template.New("Person template")

        t, err := t.Parse(tmpl)
        if err != nil {
                log.Fatal("Parse: ", err)
                return
        }

        err = t.Execute(os.Stdout, person)
        if err != nil {
                log.Fatal("Execute: ", err)
                return
        }
}
```

You can now run the program by typing:

```
$ go run person.go
```

The output is:

```
The name is Satish.

    His email id is satish@rubylearning.org

    His email id is satishtalim@gmail.com
```

In the above program we have `{{range .Emails}}`. With `range` the current object `.` is set to the successive elements of the array or slice `Emails`.

Variables

The template package allows you to define and use variables. In the above example, how would we print each person's email address prefixed by their name? Let's modify the above program.

In the code snippet:

```
{{range .Emails}}
    {{.}}
{{end}}
```

We cannot access the `Name` field as `.` is now traversing the array elements and the `Name` is outside of this scope. The solution is to save the value of the `Name` field in a variable that can be accessed anywhere in its scope. Variables in templates are prefixed by `$`. So we write:

```
{{$name := .Name}}
{{range .Emails}}
    Name is {{$name}}, email is {{.}}
{{end}}
```

The modified program, named `new_person.go` is:

Program new_person.go

```
package main

import (
        "log"
        "os"
        "text/template"
)

type Person struct {
        Name   string
        Emails []string
}

const tmpl = `{{$name := .Name}}
{{range .Emails}}
    Name is {{$name}}, email is {{.}}
{{end}}
`

func main() {
        person := Person{
                Name:   "Satish",
                Emails: []string{"satish@rubylearning.org", "satishtalim@gmail.com"},
        }

        t := template.New("Person template")

        t, err := t.Parse(tmpl)
        if err != nil {
                log.Fatal("Parse: ", err)
                return
        }

        err = t.Execute(os.Stdout, person)
        if err != nil {
                log.Fatal("Execute: ", err)
                return
        }
}
```

You can now run the program by typing:

```
$ go run new_person.go
```

The output is:

```
    Name is Satish, email is satish@rubylearning.org

    Name is Satish, email is satishtalim@gmail.com
```

The Go `template` package is useful for certain kinds of text transformations involving inserting values of objects. It does not have the power of, say, regular expressions, but is faster and in many cases will be easier to use than regular expressions.

A Static Site with Go

Let's build a web app in Go to display a static site, which we call "Dosa Diner".

Dosa is a fermented crepe or pancake made from rice batter and black lentils. This staple dish is widely popular in all southern Indian states Karnataka, Andhra Pradesh, Tamil Nadu and Kerala, as well as being popular in other countries like Sri Lanka, Malaysia and Singapore.

Start by creating the folders to hold the project:

DosaSite folder structure

```
c:\go_projects
\---go
    \---src
        \---github.com
            \---SatishTalim
                \---dosasite
                    |   dosasite.go
                    |
                    \---public
                        |   index.html
                        |
                        +---images
                        |       dosa.jpg
                        |
                        \---stylesheets
                                dosasite.css
```

We will write our Go code in the file `dosasite.go` and some sample HTML and CSS files in a `public` folder.

File index.html

```
<!DOCTYPE html>
<html>

  <head>
    <title>Dosa Diner</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="stylesheets/dosasite.css">
  </head>

  <body>
  <h1><img src="images/dosa.jpg" alt="Dosa Diner" />Dosa Diner</h1>

  <h2>The Restaurant</h2>
  <p>The Dosa Diner offers casual lunch and dinner fare in a hip atmosphere.
     The menu changes regularly to highlight the freshest ingredients.</p>

  <h2>Catering Services</h2>
  <p>You have fun... we'll do the cooking. Dosa Diner Catering can handle events
     from snacks for bridge club to elegant corporate fundraisers.</p>

  <h2>Location and Hours</h2>
  <p>Deccan Corner in Pune, India;
  Monday through Thursday 11am to 9pm, Friday and Saturday, 11am to midnight.</p>
  </body>

</html>
```

File dosasite.css

```
body {
  background-color: #C2A7F2;
  font-family: sans-serif;
}
h1 {
  color: #2A1959;
  border-bottom: 2px solid #2A1959;
}
h2 {
  color: #474B94;
  font-size: 1.2em;
}
h2, p {
  margin-left: 120px;
}
```

Once the above files are created, the code we need to get up and running is quite compact:

Program dosasite.go

```
1  package main
2
3  import (
4          "fmt"
5          "log"
6          "net/http"
7          "os"
8  )
9
10 func main() {
11         fs := http.FileServer(http.Dir("public"))
12         http.Handle("/", fs)
13
14         fmt.Println("Listening...")
15         err := http.ListenAndServe(GetPort(), nil)
16         if err != nil {
17                 log.Fatal("ListenAndServe: ", err)
18                 return
19         }
20 }
21
22 // Get the Port from the environment so we can run on Heroku (more of this later)
23 func GetPort() string {
24         var port = os.Getenv("PORT")
25         // Set a default port if there is nothing in the environment
26         if port == "" {
27                 port = "4747"
28                 fmt.Println("INFO: No PORT environment variable detected, defaulting to " + port)
29         }
30         return ":" + port
31 }
```

- like all Go programs that need to be executed, our program has a package `main`.
- we import the `fmt`[2] and `net/http`[3] packages from the Go standard library.
- to work with some printing functions, we import the package `fmt`.
- for web related http functionality, we import the package `http`. Any functions within that we refer as `http.function_name`.
- we use the FileServer[4] function to create a handler that responds to HTTP requests with the contents of a given FileSystem[5].

[2] http://golang.org/pkg/fmt/
[3] http://golang.org/pkg/net/http/
[4] http://golang.org/pkg/net/http/#FileServer
[5] http://golang.org/pkg/net/http/#FileSystem

- we use the operating system's file system implementation by `http.Dir`
- we've used the `public` folder relative to our application, but you could use any other folder on your system (or indeed anything that implements the `FileSystem` interface).
- we use the Handle[6] function to register it as the handler for all requests, and launch the server listening on port 4747.

Now you can run program with the `go` tool:

```
$ cd $GOPATH/src/github.com/SatishTalim/dosasite
$ go run dosasite.go
```

Now open http://localhost:4747/index.html[7] in your browser. You should see the HTML page we have made.

DosaSite

html/template

Usage: `import "html/template"`

Package template html/template[8] implements data-driven templates for generating HTML output safe against code injection. It provides the same interface as package `text/template` and should be used instead of `text/template` whenever the output is HTML.

Modify dosasite.go to use templates

Previously we had written the program `dosasite.go` where in the program, *all* requests are being handled by our file server. Let's make a slight adjustment so that it only handles request paths that begin with the pattern `/public/` instead.

[6] http://golang.org/pkg/net/http/#Handle
[7] http://localhost:4747/index.html
[8] http://golang.org/pkg/html/template/

```
...
fs := http.FileServer(http.Dir("public"))
http.Handle("/public/", http.StripPrefix("/public/", fs))
...
```

The function StripPrefix[9] returns a handler that serves HTTP requests by removing the given prefix from the request URL's `Path` and invoking the handler `fs`. `StripPrefix` handles a request for a path that doesn't begin with prefix by replying with an HTTP 404 not found error.

Now you can run program with the `go` tool:

```
$ cd $GOPATH/src/github.com/SatishTalim/dosasite
$ go run dosasite.go
```

Now open http://localhost:4747/public/index.html[10] in your browser. You should see the HTML page we have made.

Next, create a `templates` folder as shown below, containing a `layout.html` file with shared markup, and an `indexnew.html` file with some page-specific content.

DosaSite modified folder structure

```
c:\go_projects
\---go
    \---src
        \---github.com
            \---SatishTalim
                \---dosasite
                    |   dosasite.go
                    |
                    \---public
                    |   |
                    |   +---images
                    |   |       dosa.jpg
                    |   |
                    |   \---stylesheets
                    |           dosasite.css
                    \---templates
                        indexnew.html
                        layout.html
```

[9] http://golang.org/pkg/net/http/#StripPrefix
[10] http://localhost:4747/public/index.html

Program layout.html

```
{{define "layout"}}
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>{{template "title"}}</title>
  <link rel="stylesheet" href="/public/stylesheets/dosasite.css">
</head>

<body>
  {{template "body"}}
</body>
</html>
{{end}}
```

Program indexnew.html

```
{{define "title"}}Dosa Diner{{end}}

{{define "body"}}
  <h1><img src="public/images/dosa.jpg" alt="Dosa Diner" />Dosa Diner</h1>

  <h2>The Restaurant</h2>
  <p>The Dosa Diner offers casual lunch and dinner fare in a hip atmosphere.
     The menu changes regularly to highlight the freshest ingredients.</p>

  <h2>Catering Services</h2>
  <p>You have fun... we'll do the cooking. Dosa Diner Catering can handle events
     from snacks for bridge club to elegant corporate fundraisers.</p>

  <h2>Location and Hours</h2>
  <p>Deccan Corner in Pune, India;
  Monday through Thursday 11am to 9pm, Friday and Saturday, 11am to midnight.</p>
{{end}}
```

Go templates, as discussed before, are essentially just named text blocks surrounded by `{{define}}` and `{{end}}` tags. Templates can be embedded into each other, as we do above where the `layout` template embeds both the `title` and `body` templates.

The modified `dosasite.go` program is:

Program dosasite.go

```
package main

import (
        "fmt"
        "html/template"
        "log"
        "net/http"
        "os"
        "path"
)

func main() {
        fs := http.FileServer(http.Dir("public"))
        http.Handle("/public/", http.StripPrefix("/public/", fs))

        http.HandleFunc("/", ServeTemplate)

        fmt.Println("Listening...")
        err := http.ListenAndServe(GetPort(), nil)
        if err != nil {
                log.Fatal("ListenAndServe: ", err)
                return
        }
}

// Get the Port from the environment so we can run on Heroku
func GetPort() string {
        var port = os.Getenv("PORT")
        // Set a default port if there is nothing in the environment
        if port == "" {
                port = "4747"
                fmt.Println("INFO: No PORT environment variable detected, defaulting to " + port)
        }
        return ":" + port
}

func ServeTemplate(w http.ResponseWriter, r *http.Request) {
        lp := path.Join("templates", "layout.html")
        fp := path.Join("templates", r.URL.Path)

        // Return a 404 if the template doesn't exist
        info, err := os.Stat(fp)
```

```
43          if err != nil {
44                  if os.IsNotExist(err) {
45                          http.NotFound(w, r)
46                          return
47                  }
48          }
49
50          // Return a 404 if the request is for a directory
51          if info.IsDir() {
52                  http.NotFound(w, r)
53                  return
54          }
55
56          templates, _ := template.ParseFiles(lp, fp)
57          if err != nil {
58                  fmt.Println(err)
59                  http.Error(w, "500 Internal Server Error", 500)
60                  return
61          }
62
63          templates.ExecuteTemplate(w, "layout", nil)
64  }
```

In the above program, we've added the `html/template` and `path` packages to the `import` statement.

We've then specified that all the requests not picked up by the static file server should be handled with a new `ServeTemplate` function.

In the `ServeTemplate` function, we build paths to the layout file and the template file corresponding with the request. Rather than manual concatenation we use Join[11], which has the advantage of cleaning the path to help prevent directory traversal attacks.

We then use the ParseFiles[12] function to bundle the requested template and layout into a template set. Finally, we use the ExecuteTemplate[13] function to render a named template in the set, in our case the layout template.

Our code also has some error handling:

- Send a 404 response if the requested template doesn't exist.
- Send a 404 response if the requested template path is a directory.
- Send and print a 500 response if the `template.ParseFiles` function throws an error.

Special thanks to Alex Edwards whose article[14] has been adapted for this topic.

[11] http://golang.org/pkg/path/#Join
[12] http://golang.org/pkg/text/template/#Template.ParseFiles
[13] http://golang.org/pkg/text/template/#Template.ExecuteTemplate
[14] http://www.alexedwards.net/blog/serving-static-sites-with-go

Handling Forms

If we want users to be able to post their own messages on a webpage, we need a way to process this information submitted by the user with a web form. The Go `http` package makes processing form data easy.

Let us see how this is done.

Web app stringupper

The directory structure for this app is as follows:

App folder structure

```
C:\go_projects>
\---go
    \---src
        \---github.com
            \---SatishTalim
                +---stringupper
                |   |   stringupper.go
                |   |
                |   \---css
                |           upper.css
```

Here are the contents of the file `upper.css`:

upper.css

```css
body {
  background-color: #C2A7F2;
  font-family: sans-serif;
}
h1 {
  color: #2A1959;
  border-bottom: 2px solid #2A1959;
}
h2 {
  color: #474B94;
  font-size: 1.2em;
}
h2 {
  margin-left: 120px;
}
```

Here is the code for the app `stringupper.go`:

stringupper.go

```
package main

import (
        "fmt"
        "html/template"
        "log"
        "net/http"
        "os"
        "strings"
)

func main() {
        // Add a handler to handle serving static files from a specified directory
        // The reason for using StripPrefix is that you can change the served
        // directory as you please, but keep the reference in HTML the same.
        http.Handle("/css/", http.StripPrefix("/css/", http.FileServer(http.Dir("css"))))

        http.HandleFunc("/", root)
        http.HandleFunc("/upper", upper)
        fmt.Println("listening...")
        err := http.ListenAndServe(GetPort(), nil)
        if err != nil {
                log.Fatal("ListenAndServe: ", err)
                return
        }
}

func root(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, rootForm)
}

const rootForm = `
  <!DOCTYPE html>
    <html>
      <head>
        <meta charset="utf-8">
        <link rel="stylesheet" href="css/upper.css">
        <title>String Upper</title>
      </head>
      <body>
        <h1>String Upper</h1>
        <p>The String Upper Service will accept a string from you and
```

```
43              return you the Uppercase version of the original string. Have fun!</p>
44            <form action="/upper" method="post" accept-charset="utf-8">
45              <input type="text" name="str" value="Type a string..." id="str">
46              <input type="submit" value=".. and change to uppercase!">
47            </form>
48          </body>
49        </html>
50  `
51  var upperTemplate = template.Must(template.New("upper").Parse(upperTemplateHTML))
52
53  func upper(w http.ResponseWriter, r *http.Request) {
54          strEntered := r.FormValue("str")
55          strUpper := strings.ToUpper(strEntered)
56          err := upperTemplate.Execute(w, strUpper)
57          if err != nil {
58                  http.Error(w, err.Error(), http.StatusInternalServerError)
59          }
60  }
61
62  const upperTemplateHTML = `
63  <!DOCTYPE html>
64    <html>
65      <head>
66        <meta charset="utf-8">
67        <link rel="stylesheet" href="css/upper.css">
68        <title>String Upper Results</title>
69      </head>
70      <body>
71        <h1>String Upper Results</h1>
72        <p>The Uppercase of the string that you had entered is:</p>
73        <pre>{{html .}}</pre>
74      </body>
75    </html>
76  `
77
78  // Get the Port from the environment so we can run on Heroku
79  func GetPort() string {
80          var port = os.Getenv("PORT")
81          // Set a default port if there is nothing in the environment
82          if port == "" {
83                  port = "4747"
84                  fmt.Println("INFO: No PORT environment variable detected, defaulting to " + port)
85          }
86          return ":" + port
87  }
```

This app has two handlers: the path `/` is mapped to `root`, which displays a web form for the user to enter some text as a string. The path `/upper` is mapped to `upper`, which displays to the user the text entered by him/her in uppercase.

The `upper` function gets the form data by calling `r.FormValue` and passes it to `upperTemplate.Execute` that writes the rendered template to the `http.ResponseWriter`. In the template code, the content is automatically filtered to escape HTML special characters. The automatic escaping is a property of the html/template[15] package, as distinct from text/template[16].

The line `{{html .}}` in `const upperTemplateHTML`, `html` is a predefined global function that returns the escaped HTML equivalent of the textual representation of its arguments, `.` in this case.

You can refer to Handling Web Forms[17] for more information.

You can now run the program by typing:

```
1 $ go run stringupper.go
```

Have fun!!

[15] http://golang.org/pkg/html/template/
[16] http://golang.org/pkg/text/template/
[17] https://developers.google.com/appengine/docs/go/gettingstarted/handlingforms