Templates

Introduction

Go’s `html/template` package provides a rich templating language for HTML templates. It is mostly used in web applications to display data in a structured way in a client’s browser. One great benefit of Go’s templating language is the automatic escaping of data. There is no need to worry about XSS attacks as Go parses the HTML template and escapes all inputs before displaying it to the browser.

First Template

Writing a template in Go is very simple. This example shows a TODO list, written as an unordered list (ul) in HTML. When rendering templates, the data passed in can be any kind of Go’s data structures. It may be a simple string or a number, it can even be nested data structure as in the example below. To access the data in a template the top most variable is access by `{{.}}`. The dot inside the curly braces is called the pipeline and the root element of the data.

```
data := TodoPageData{
    PageTitle: "My TODO list",
    Todos: []Todo{
        {Title: "Task 1", Done: false},
        {Title: "Task 2", Done: true},
        {Title: "Task 3", Done: true},
    },
}
```

```
<h1>{{.PageTitle}}</h1>
<ul>
    {{range .Todos}}
        {{if .Done}}
            <li class="done">{{.Title}}</li>
        {{else}}
            <li>{{.Title}}</li>
        {{end}}
    {{end}}
</ul>
```

Control Structures

The templating language contains a rich set of control structures to render your HTML. Here you will get an overview of the most commonly used ones. To get a detailed list of all possible structures visit: text/template

Control Structure	Definition
`{{/* a comment */}}`	Defines a comment
`{{.}}`	Renders the root element
`{{.Title}}`	Renders the “Title”-field in a nested element
`{{if .Done}} {{else}} {{end}}`	Defines an if-Statement
`{{range .Todos}} {{.}} {{end}}`	Loops over all “Todos” and renders each using `{{.}}`
`{{block "content" .}} {{end}}`	Defines a block with the name “content”

Parsing Templates from Files

Template can either be parsed from a string or a file on disk. As it is usually the case, that templates are pares from disk, this example shows how to do so. In this example there is a template file in the same directory as the Go program called `layout.html` .

```
tmpl, err := template.ParseFiles("layout.html")
// or
tmpl := template.Must(template.ParseFiles("layout.html"))
```

Execute a Template in a Request Handler

Once the template is parsed from disk it's ready to be used in the request handler. The `Execute` function accepts an `io.Writer` for writing out the template and an `interface{}` to pass data into the template. When the function is called on an `http.ResponseWriter` the Content-Type is header is automatically set in the HTTP response to `Content-Type: text/html; charset=utf-8` .

```
func(w http.ResponseWriter, r *http.Request) {
    tmpl.Execute(w, "data goes here")
}
```

The Code (for copy/paste)

This is the complete code that you can use to try out the things you've learned in this example.

```
package main

```

```
import (
    "html/template"
    "net/http"
)


type Todo struct {
    Title string
    Done  bool
}


type TodoPageData struct {
    PageTitle string
    Todos     []Todo
}


func main() {
    tmpl := template.Must(template.ParseFiles("layout.html"))
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        data := TodoPageData{
            PageTitle: "My TODO list",
            Todos: []Todo{
                {Title: "Task 1", Done: false},
                {Title: "Task 2", Done: true},
                {Title: "Task 3", Done: true},
            },
        }
        tmpl.Execute(w, data)
    })
    http.ListenAndServe(":80", nil)
}
```

```
<h1>{{.PageTitle}}</h1>
<ul>
    {{range .Todos}}
        {{if .Done}}
            <li class="done">{{.Title}}</li>
        {{else}}
            <li>{{.Title}}</li>
        {{end}}
    {{end}}
```

```
</ul>
```

Legal Disclosure Privacy Statement