

Templates

text templates

godoc.org/text/template

Bookmarks M G D Y T CNN digg PM Hawk J Other

GoDoc Home Index About Search

Go: text/template Index | Examples | Files | Directories

package template

```
import "text/template"

Package template implements data-driven templates for generating textual output.
```

To generate HTML output, see package html/template, which has the same interface as this package but automatically secures HTML output against certain attacks.

Templates are executed by applying them to a data structure. Annotations in the template refer to elements of the data structure (typically a field of a struct or a key in a map) to control execution and derive values to be displayed. Execution of the template walks the structure and sets the cursor, represented by a period '.' and called "dot", to the value at the current location in the structure as execution proceeds.

The input text for a template is UTF-8-encoded text in any format. "Actions"--data evaluations or control structures--are delimited by "{{" and "}}"; all text outside actions is copied to the output unchanged. Except for raw strings, actions may not span newlines, although comments can.

Once parsed, a template may be executed safely in parallel.

Here is a trivial example that prints "17 items are made of wool".

```
type Inventory struct {
    Material string
    Count     uint
}
sweaters := Inventory{"wool", 17}
tmpl, err := template.New("test").Parse("{{.Count}} items are made of {{.Material}}")
if err != nil { panic(err) }
err = tmpl.Execute(os.Stdout, sweaters)
if err != nil { panic(err) }
```

```
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 func main() {
10    tpl, err := template.ParseFiles("tpl.gohtml")
11    if err != nil {
12        log.Fatalln(err)
13    }
14    err = tpl.Execute(os.Stdout, nil)
15    if err != nil {
16        log.Fatalln(err)
17    }
18 }
19
```

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6 </head>
7 <body>
8     <h1>Hello</h1>
9 </body>
10 </html>
```

Terminal

```
+ 01 $ go run main.go
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <h1>Hello</h1>
</body>
</html>01 $
```

template actions

```
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 func main() {
10    tpl, err := template.ParseFiles("tpl.gohtml")
11    if err != nil {
12        log.Fatalln(err)
13    }
14    err = tpl.Execute(os.Stdout, nil)
15    if err != nil {
16        log.Fatalln(err)
17    }
18}
19
```

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6 </head>
7 <body>
8     <h1>{{5}}</h1>
9 </body>
10 </html>
```

Terminal

```
+ 02 $ go run main.go
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <h1>5</h1>
</body>
</html>02 $ 
```

```
main.go x tpl.gohtml x
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 func main() {
10    tpl, err := template.ParseFiles("tpl.gohtml")
11    if err != nil {
12        log.Fatalln(err)
13    }
14    err = tpl.Execute(os.Stdout, 10)
15    if err != nil {
16        log.Fatalln(err)
17    }
18 }
```

Think of the “.” like the “.” in Unix. Just like in Unix the “.” means the current directory, here the “.” means the current data the “.” is known as the pipeline

```
main.go x tpl.gohtml x
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6 </head>
7 <body>
8     <h1>{{.}}</h1>
9 </body>
10 </html>
```

Terminal

```
+ 03 $ go run main.go
<!doctype html>
X <html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <h1>10</h1>
</body>
</html>03 $
```

```
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 func main() {
10     tpl, err := template.ParseFiles("tpl.gohtml")
11     if err != nil {
12         log.Fatalln(err)
13     }
14     err = tpl.Execute(os.Stdout, 5*5)
15     if err != nil {
16         log.Fatalln(err)
17     }
18 }
```

Think of the “.” like the “.” in Unix. Just like in Unix the “.” means the current directory, here the “.” means the current data

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6 </head>
7 <body>
8     <h1>{{.}}</h1>
9 </body>
10 </html>
```

Terminal

```
+ 04 $ go run main.go
<!doctype html>
X <html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <h1>25</h1>
</body>
</html>04 $
```

```
main.go
package main
import (
    "log"
    "os"
    "text/template"
)
func main() {
    tpl, err := template.ParseFiles("tpl.gohtml")
    if err != nil {
        log.Fatalln(err)
    }
    err = tpl.Execute(os.Stdout, []int{1,2,3,4,5})
    if err != nil {
        log.Fatalln(err)
    }
}
```

```
tpl.gohtml
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <ul>
        {{ range . }}
        <li>{{.}}</li>
        {{ end }}
    </ul>
</body>
</html>
```

Think of the “.” like the “.” in Unix. Just like in Unix the “.” means the current directory, here the “.” means the current data

Go's templates are similar to
other templating



Logic-less templates.

Available in [Ruby](#), [JavaScript](#), [Python](#), [Erlang](#), [node.js](#), [PHP](#), [Perl](#), [Perl6](#), [Objective-C](#), [Java](#), [C#/.NET](#), [Android](#), [C++](#), [Go](#), [Lua](#), [ooc](#), [ActionScript](#), [ColdFusion](#), [Scala](#), [Clojure](#), [Fantom](#), [CoffeeScript](#), [D](#), [Haskell](#), [XQuery](#), [ASP](#), [Io](#), [Dart](#), [Haxe](#), [Delphi](#), [Racket](#), [Rust](#), [OCaml](#), [Swift](#), [Bash](#), [Julia](#), [R](#), [Crystal](#), and for [Common Lisp](#)

Works great with [TextMate](#), [Vim](#), [Emacs](#), [Coda](#), and [Atom](#)

The Manual: [mustache\(5\)](#) and [mustache\(1\)](#)

The screenshot shows the Handlebars.js homepage. At the top, there's a navigation bar with links for Apps, Bookmarks, and various social media icons. The main header features the word "handlebars" in a large, white, sans-serif font next to a stylized black mustache icon. To the left of the header, there's a graphic of an orange t-shirt with "SWAG" printed on it, accompanied by a blue "NEW" badge. Below the header, a text box states: "Handlebars provides the power necessary to let you build semantic template effectively with no frustration." Another text box below it says: "Handlebars is largely compatible with Mustache templates. In most cases it is to swap out Mustache with Handlebars and continue using your current template. Complete details can be found [here](#)." A large orange button labeled "Installation" is visible. At the bottom, a section titled "Getting Started" includes a snippet of Handlebars template code:

```
<div class="entry">
  <h1>{{title}}</h1>
  <div class="body">
    {{body}}
  </div>
</div>
```

The image shows a Go development environment with two code editors and a terminal window.

main.go

```
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 func main() {
10    tpl, err := template.ParseFiles("tpl.gohtml")
11    if err != nil {
12        log.Fatalln(err)
13    }
14    err = tpl.Execute(os.Stdout, []int{})
15    if err != nil {
16        log.Fatalln(err)
17    }
18 }
```

tpl.gohtml

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6 </head>
7 <body>
8 <ul>
9     {{ range . }}
10    <li>{{.}}</li>
11    {{ else }}
12    <li>NO ITEMS</li>
13    {{ end }}
14 </ul>
15 </body>
16 </html>
```

Terminal

```
+ 06 $ go run main.go
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
<ul>
<li>NO ITEMS</li>
</ul>
</body>
06 $
```

A red arrow points from the line `{{.}}` in the `tpl.gohtml` file to the terminal output. Another red arrow points from the line `NO ITEMS` in the `tpl.gohtml` file to the terminal output.

08/main.go x

```
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 type Page struct {
10    Title string
11    Body   string
12 }
13
14 func main() {
15     tpl, err := template.ParseFiles("tpl.gohtml")
16     if err != nil {
17         log.Fatalln(err)
18     }
19     err = tpl.Execute(os.Stdout, Page{
20         Title: "My Title 2",
21     })
22     if err != nil {
23         log.Fatalln(err)
24     }
25 }
```

08/main.go x tpl.gohtml x

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>{{.Title }}</title>
6 </head>
7 <body>
8
9 </body>
10 </html>
```

Terminal

```
+ 08 $ go run main.go
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Title 2</title>
</head>
<body>
</body>
08 $
```

```
main.go x tpl.gohtml x
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 type Page struct {
10    Title string
11    Body  string
12 }
13
14 func main() {
15    var err error
16
17    tpl := template.New("tpl.gohtml")
18    tpl = tpl.Funcs(template.FuncMap{
19        "mycustomfunc": func() string {
20            return "This should work"
21        },
22    })
23    _, err = tpl.ParseFiles("tpl.gohtml")
24    if err != nil {
25        log.Fatalln(err)
26    }
27    err = tpl.Execute(os.Stdout, Page{
28        Title: "My Title",
29    })
30    if err != nil {
31        log.Fatalln(err)
32    }
33 }
```

```
main.go x tpl.gohtml x
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>{{_.Title }}</title>
6 </head>
7 <body>
8 {{mycustomfunc}}
9 </body>
10 </html>
```

Terminal

```
+ 09_function $ go run main.go
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Title</title>
</head>
<body>
This should work
</body>
</html>09_function $
```

The image shows a Go development environment with two code editors and a web browser.

Code Editors:

- main.go:** Contains the main function and imports for log, os, and text/template packages. It defines a Page struct and uses template.New to create a template with a custom function mycustomfunc that returns "This should work". It then parses files, executes the template, and logs errors.
- tpl.gohtml:** Contains the template code with a single line: {{ .Title }}

Browser:

godoc.org/text/template

type Template

- o func Must(t *Template, err error) *Template
- o func New(name string) *Template ← Red arrow pointing here
- o func ParseFiles(filenames ...string) (*Template, error)
- o func ParseGlob(pattern string) (*Template, error)
- o func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Template, error)
- o func (t *Template) Clone() (*Template, error)
- o func (t *Template) DefinedTemplates() string
- o func (t *Template) Delims(left, right string) *Template
- o func (t *Template) Execute(wr io.Writer, data interface{}) (err error)
- o func (t *Template) ExecuteTemplate(wr io.Writer, name string, data interface{}) error
- o func (t *Template) Funcs(funcMap FuncMap) *Template
- o func (t *Template) Lookup(name string) *Template
- o func (t *Template) Name() string
- o func (t *Template) New(name string) *Template
- o func (t *Template) Option(opt ...string) *Template
- o func (t *Template) Parse(text string) (*Template, error)
- o func (t *Template) ParseFiles(filenames ...string) (*Template, error)
- o func (t *Template) ParseGlob(pattern string) (*Template, error)
- o func (t *Template) Templates() []*Template

The image shows a Go development environment with two code files and a browser displaying the Go documentation for the `text/template` package.

Code Files:

- main.go:** Contains the main application logic, including imports for `log` and `os`, a `Page` struct definition, and a `main` function that creates a template, parses it, executes it, and logs errors.
- tpl.gohtml:** Contains a Go template with a custom function named `mycustomfunc` that returns the string "This should work".

Browser Documentation:

The browser window displays the `godoc.org/text/template` documentation. The `Template` type is defined with the following methods:

- `func Must(t *Template, err error) *Template`
- `func New(name string) *Template`** (highlighted by a red arrow)
- `func ParseFiles(filenames ...string) (*Template, error)`
- `func ParseGlob(pattern string) (*Template, error)`
- `func Parse(text string) (*Template, error)`
- `func ParseTree(name string, tree *parse.Tree) (*Template, error)`
- `func Option(opt ...string) *Template`
- `func ParseGlob(pattern string) (*Template, error)`
- `func Templates() []*Template`

A callout box highlights the `New` method, which is annotated with the text: "New allocates a new, undefined template with the given name."

The image shows a Go development environment with two code files and a browser displaying the `text/template` package documentation.

main.go:

```
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 type Page struct {
10    Title string
11    Body  string
12 }
13
14 func main() {
15    var err error
16
17    tpl := template.New("tpl.gohtml")
18    tpl = tpl.Funcs(template.FuncMap{
19        "mycustomfunc": func() string {
20            return "This should work"
21        },
22    })
23    _, err = tpl.ParseFiles("tpl.gohtml")
24    if err != nil {
25        log.Fatalln(err)
26    }
27    err = tpl.Execute(os.Stdout, Page{
28        Title: "My Title",
29    })
30    if err != nil {
31        log.Fatalln(err)
32    }
33 }
```

tpl.gohtml:

```
<html>
<head>
<title>{{.Title}}</title>
</head>
<body>
{{.Body}}
</body>
</html>
```

godoc.org/text/template:

The browser shows the `Template` type and its methods. A red box highlights the `New` method, which is annotated with `func New(name string) *Template`. Another red box highlights the `Funcs` method, which is annotated with `func (*Template) Funcs(funcMap FuncMap) *Template`.

Annotations:

- func New:** `New allocates a new`
- func (*Template) Funcs:** `Funcs adds the elements of the argument map to the template's function map. It panics if a value in the map is not a function with appropriate return type. However, it is legal to overwrite elements of the map. The return value is the template, so calls can be chained.`

The image shows a Go development environment with two code files and a browser window.

main.go content:

```
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 type Page struct {
10    Title string
11    Body  string
12 }
13
14 func main() {
15    var err error
16
17    tpl := template.New("tpl.gohtml")
18    tpl = tpl.Funcs(template.FuncMap{
19        "mycustomfunc": func() string {
20            return "This should work"
21        },
22    })
23    _, err = tpl.ParseFiles("tpl.gohtml")
24    if err != nil {
25        log.Fatalln(err)
26    }
27    err = tpl.Execute(os.Stdout, Page{
28        Title: "My Title",
29    })
30    if err != nil {
31        log.Fatalln(err)
32    }
33 }
```

tpl.gohtml content:

```
<html>
<head>
<title>{{.Title}}</title>
</head>
<body>
{{.Body}}
</body>
</html>
```

godoc.org/text/template content:

type Template

- func Must(t *Template, err error) *Template
- func New(name string) *Template
- func ParseFiles(filenames ...string) (*Template, error)

type FuncMap

```
type FuncMap map[string]interface{}
```

FuncMap is the type of the map defining the mapping from names to functions. Each function must have either a single return value, or two return values of which the second has type error. In that case, if the second (error) return value evaluates to non-nil during execution, execution terminates and Execute returns that error.

func (*Template) Funcs(funcMap FuncMap) *Template

Funcs adds the elements of the argument map to the template's function map. It panics if a value in the map is not a function with appropriate return type. However, it is legal to overwrite elements of the map. The return value is the template, so calls can be chained.

```
func (t *Template) Funcs(pattern string) (*Template, error)
func (t *Template) Funcs(filenames ...string) (*Template, error)
func (t *Template) Funcs(files []string) (*Template, error)
func (t *Template) Funcs(templates() []*Template)
```

The image shows a Go development environment with two code files and a browser displaying the `text/template` package documentation.

main.go:

```
1 package main
2
3 import (
4     "log"
5     "os"
6     "text/template"
7 )
8
9 type Page struct {
10    Title string
11    Body  string
12 }
13
14 func main() {
15    var err error
16
17    tpl := template.New("tpl.gohtml")
18    tpl = tpl.Funcs(template.FuncMap{
19        "mycustomfunc": func() string {
20            return "This should work"
21        },
22    })
23    _, err = tpl.ParseFiles("tpl.gohtml")
24    if err != nil {
25        log.Fatalln(err)
26    }
27    err = tpl.Execute(os.Stdout, Page{
28        Title: "My Title",
29    })
30    if err != nil {
31        log.Fatalln(err)
32    }
33 }
```

tpl.gohtml:

```
<html>
<head>
<title>{{.Title}}</title>
</head>
<body>
{{.Body}}
</body>
</html>
```

godoc.org/text/template:

type Template

- func Must(t *Template, err error) *Template
- func New(name string) *Template
- func ParseFiles(filenames ...string) (*Template, error)

type FuncMap

```
type FuncMap map[string]interface{}
```

FuncMap is the type of the map defining the mapping from names to functions. Each function must have either one value, or two return values of which the second has type error. In that case, if the second (error) value is non-nil during execution, execution terminates and Execute returns that error.

func (*Template) ParseFiles

```
func (t *Template) ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles parses the named files and associates the resulting templates with t. If an error occurs, parsing stops and the returned template is nil; otherwise it is t. There must be at least one file. Since the templates created by ParseFiles are named by the base names of the argument files, t should usually have the name of one of the (base) names of the files. If it does not, depending on t's contents before calling ParseFiles, t.Execute may fail. In that case use t.ExecuteTemplate to execute a valid template.

It panics if a value in the map is not a function with exactly one argument. The return value is the template, so t.ExecuteTemplate can be used.

Funcs adds not a function with exactly one argument. The return value is the template, so t.ExecuteTemplate can be used.

The image shows a Go development environment with two code files and a browser displaying the `text/template` package documentation.

Code Files:

- main.go:** Contains the main application logic, including imports for `log`, `os`, and `text/template`. It defines a `Page` struct and implements the `main` function.
- tpl.gohtml:** Contains a template definition with a custom function `mycustomfunc` that returns the string "This should work".

Browser Documentation:

The browser window displays the `godoc.org/text/template` documentation. The `Template` type has three methods:

- `Must`: Returns a `*Template` object if no error occurs.
- `New`: Creates a new `*Template` object from a template name.
- `ParseFiles`: Parses multiple files and associates them with the template.

The `Execute` method is highlighted with a red box. Its documentation states:

`func (*Template) Execute(wr io.Writer, data interface{}) (err error)`

Execute applies a parsed template to the specified data object, and writes the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel.

The `ParseFiles` method is also highlighted with a red box. Its documentation states:

`func (*Template) ParseFiles(filenames ...string) (*Template, error)`

ParseFiles parses the named files and associates the resulting templates with t. If an error occurs, parsing stops and the returned template is nil; otherwise it is t. There must be at least one file. Since the templates created by ParseFiles are named by the base names of the argument files, t should usually have the name of one of the (base) names of the files. If it does not, depending on t's contents before calling ParseFiles, t.Execute may fail. In that case use t.ExecuteTemplate to execute a valid template.

The `Funcs` method is mentioned in the `ParseFiles` documentation: "Funcs adds not a function with the same name as the template, but a function whose value is the template, so t.ExecuteTemplate can be used."

The image shows a Go development environment with three panes:

- main.go** (Left Pane): The Go source code. A red arrow points from line 20 to the `Funcs` call.
- tpl.gohtml** (Top Right Pane): The template file. An arrow points from line 8 to the `uppercase` function call.
- Terminal** (Bottom Right Pane): The output of the command `go run main.go`. It shows the rendered HTML output.

```
main.go
1 package main
2
3 import (
4     "log"
5     "os"
6     "strings"
7     "text/template"
8 )
9
10 type Page struct {
11     Title string
12     Body   string
13 }
14
15 func main() {
16     var err error
17
18     tpl := template.New("tpl.gohtml")
19     tpl = tpl.Funcs(template.FuncMap{
20         "uppercase": func(str string) string {
21             return strings.ToUpper(str)
22         },
23     })
24     tpl, err = tpl.ParseFiles("tpl.gohtml")
25     if err != nil {
26         log.Fatalln(err)
27     }
28     err = tpl.Execute(os.Stdout, Page{
29         Title: "My Title 2",
30     })
31     if err != nil {
32         log.Fatalln(err)
33     }
34 }
```

```
tpl.gohtml
1 <!doctype html>
2 <html lang="en">
3     <head>
4         <meta charset="UTF-8">
5         <title>{{_.Title }}</title>
6     </head>
7     <body>
8         {{uppercase .Title}}
9     </body>
10    </html>
```

```
Terminal
+ 10_function $ go run main.go
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Title 2</title>
</head>
<body>
    MY TITLE 2
</body>
</html>10_function $
```

```
main.go x tpl.gohtml x
1 package main
2
3 import (
4     "log"
5     "os"
6     "strings"
7     "text/template"
8 )
9
10 type Page struct {
11     Title string
12     Body   string
13 }
14
15 func main() {
16     var err error
17
18     tpl := template.New("tpl.gohtml")
19     tpl = tpl.Funcs(template.FuncMap{
20         "uppercase": func(str string) string {
21             return strings.ToUpper(str)
22         },
23     })
24     tpl, err = tpl.ParseFiles("tpl.gohtml")
25     if err != nil {
26         log.Fatalln(err)
27     }
28     err = tpl.Execute(os.Stdout, Page{
29         Title: "My Title 2",
30         Body: `hello world <script>alert("Yow!");</script>`,
31     })
32     if err != nil {
33         log.Fatalln(err)
34     }
35 }
```

```
main.go x tpl.gohtml x
1 <!doctype html>
2 <html lang="en">
3     <head>
4         <meta charset="UTF-8">
5         <title>{{_.Title }}</title>
6     </head>
7     <body>
8     {{uppercase .Title}}
9     {{_.Body }} ←
10    </body>
11 </html>
```

```
Terminal
+ 11 $ go run main.go
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Title 2</title>
</head>
<body>
MY TITLE 2
hello world <script>alert("Yow!");</script>
</body>
</html>11 $
```

html templates

The image shows a Go development environment with three panes:

- main.go** (Left Pane):

```
1 package main
2
3 import (
4     "log"
5     "os"
6     "strings"
7     "html/template"
8 )
9
10 type Page struct {
11     Title string
12     Body   string
13 }
14
15 func main() {
16     var err error
17
18     tpl := template.New("tpl.gohtml")
19     tpl = tpl.Funcs(template.FuncMap{
20         "uppercase": func(str string) string {
21             return strings.ToUpper(str)
22         },
23     })
24     _, err = tpl.ParseFiles("tpl.gohtml")
25     if err != nil {
26         log.Fatalln(err)
27     }
28     err = tpl.Execute(os.Stdout, Page{
29         Title: "My Title 2",
30         Body: `hello world <script>alert("Yow!");</script>`})
31     if err != nil {
32         log.Fatalln(err)
33     }
34 }
```
- tpl.gohtml** (Top Right Pane):

```
1 <!doctype html>
2 <html lang="en">
3     <head>
4         <meta charset="UTF-8">
5         <title>{{_.Title }}</title>
6     </head>
7     <body>
8         {{uppercase .Title}}
9         {{_.Body }}
10    </body>
11 </html>
```
- Terminal** (Bottom Right Pane):

```
+ <!doctype html>n.go
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Title 2</title>
</head>
<body>
MY TITLE 2
hello world &lt;script&gt;alert(&#34;Yow!&#34;);&lt;/script&gt;
</body>
</html>01 $
```

A red arrow points from the `uppercase` function call in `main.go` to the `uppercase` template function definition in `tpl.gohtml`. Another red arrow points from the output in the Terminal pane to the rendered `MY TITLE 2` text.

The image shows a Go development environment with three panes:

- main.go**: The code defines a `Page` struct with `Title` and `Body` fields, where `Body` is of type `template.HTML`. It also contains logic to parse a template file named `tpl.gohtml`.
- tpl.gohtml**: The template file defines a page with a title and body, using a custom function `uppercase` to convert the title to uppercase.
- Terminal**: The output of running the program, showing the generated HTML with the converted title.

```
main.go
1 package main
2
3 import (
4     "html/template"
5     "log"
6     "os"
7     "strings"
8 )
9
10 type Page struct {
11     Title string
12     Body   template.HTML
13 }
14
15 func main() {
16     log.SetFlags(0)
17
18     var err error
19
20     tpl := template.New("tpl.gohtml")
21     tpl = tpl.Funcs(template.FuncMap{
22         "uppercase": func(str string) string {
23             return strings.ToUpper(str)
24         },
25     })
26     _, err = tpl.ParseFiles("tpl.gohtml")
27     if err != nil {
28         log.Fatalln(err)
29     }
30     err = tpl.Execute(os.Stdout, Page{
31         Title: "My Title 2",
32         Body: template.HTML(`hello world <script>alert("Yow!");</script>`),
33     })
34     if err != nil {
35         log.Fatalln(err)
36     }
37 }
```

```
tpl.gohtml
1 <!doctype html>
2 <html lang="en">
3     <head>
4         <meta charset="UTF-8">
5         <title>{{_.Title }}</title>
6     </head>
7     <body>
8         {{uppercase .Title}}
9         {{_.Body }}
10    </body>
11 </html>
```

```
Terminal
+ 02 $ go run main.go
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Title 2</title>
</head>
<body>
    MY TITLE 2
    hello world <script>alert("Yow!");</script>
</body>
</html>02 $
```

ParseFiles

ParseGlob

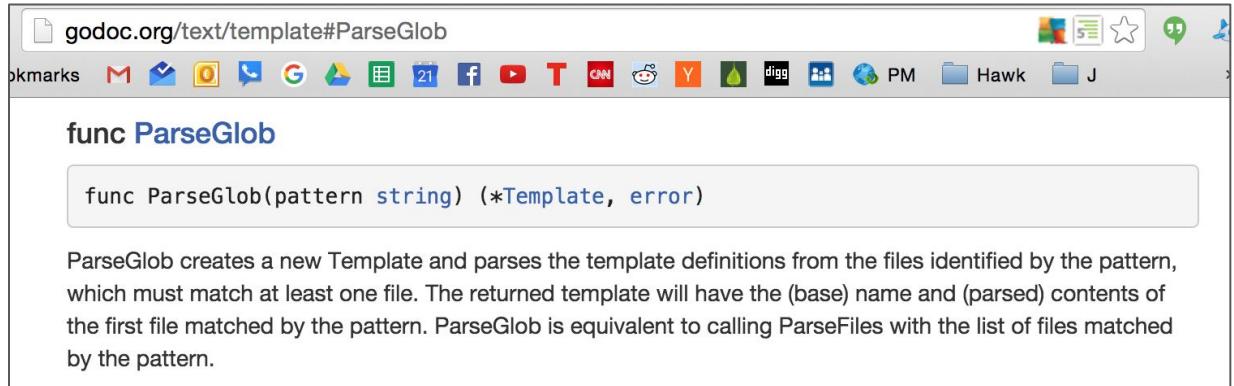
Execute

ExecuteTemplate

func ParseFiles

```
func ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles creates a new Template and parses the template definitions from the named files. The returned template's name will have the base name and parsed contents of the first file. There must be at least one file. If an error occurs, parsing stops and the returned *Template is nil.



The screenshot shows a web browser window displaying the godoc.org documentation for the `ParseGlob` function. The title bar reads "godoc.org/text/template#ParseGlob". The page header includes a "bookmarks" button and various social sharing icons (M, G+, Digg, etc.). The main content area shows the `func ParseGlob` signature and its description.

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob creates a new Template and parses the template definitions from the files identified by the pattern, which must match at least one file. The returned template will have the (base) name and (parsed) contents of the first file matched by the pattern. ParseGlob is equivalent to calling ParseFiles with the list of files matched by the pattern.

func (*Template) Execute

```
func (t *Template) Execute(wr io.Writer, data interface{}) (err error)
```

Execute applies a parsed template to the specified data object, and writes the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel.

func (*Template) ExecuteTemplate

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data interface{}) error
```

ExecuteTemplate applies the template associated with t that has the given name to the specified data object and writes the output to wr. If an error occurs executing the template or writing its output, execution stops, but partial results may already have been written to the output writer. A template may be executed safely in parallel.

The screenshot shows a Go development environment with the following components:

- Project:** A sidebar listing various Go packages and files, including 32_package-path/filepath, 33_package-time, 34_hash, 35_packagefilepath, 36_concurrency, 37_review-exercises, 38_JSON, 39_packages, 40_testing, 41_TCP, 42_HTTP, 43_HTTP-server, 44_MUX_routing, 45_serving-files, 46_review, and 47_templates. The 47_templates folder contains sub-folders 01_text-templates and 02_html-templates, which further contain files 01, 02, 03, and 04. The file main.go is currently selected.
- Terminal:** A terminal window showing the output of the command \$ go run main.go. The output consists of two sets of HTML documents. The first set, from tpl, has a title of "My Title 2" and a body of "hello world". The second set, from tpl2, has a title of "My Title 2" and a body of "hello world". Both sets include a footer with a separator line and a copyright notice.
- Code Editor:** The main editor area displays the main.go file. The code defines a `Page` struct and a `main()` function. It uses the `template` package to parse and execute HTML templates. Red arrows highlight several lines of code:
 - A red arrow points to the declaration of `tpl *template.Template`.
 - A red arrow points to the call to `tpl.ParseFiles("tpl.gohtml", "tpl2.gohtml")`.
 - A red arrow points to the call to `tpl.Execute(os.Stdout, Page{Title: "My Title 2", Body: "hello world",})`.
 - A red arrow points to the call to `tpl.ExecuteTemplate(os.Stdout, "tpl.gohtml", Page{Title: "My Title 2", Body: "hello world",})`.
 - A red arrow points to the call to `tpl.ExecuteTemplate(os.Stdout, "tpl2.gohtml", Page{Title: "My Title 2", Body: "hello world",})`.

Project

- 32_package-path-filename
- 33_package-time
- 34_hash
- 35_package-filename
- 36_concurrency
- 37_review-exercises
- 38_JSON
- 39_packages
- 40_testing
- 41_TCP
- 42_HTTP
- 43_HTTP-server
- 44_MUX_routing
- 45_serving-files
- 46_review
- 47_templates
 - 01_text-templates
 - 02_html-templates
 - 01
 - 02
 - 03
 - 04
 - 05
 - templates
- main.go
- x03_exercises
- 48_passing-data
- 98_tmp-draw-from
- uu_lynda
- vv99_trial
- ww100_whatevah
- xx_exercises-for-later
- xx_stringer
- .gitignore
- README.md
- temp.html

External Libraries

- Go SDK
- GOPATH <GolangTraining>

main.go

```
9
10 type Page struct {
11     Title string
12     Body  string
13 }
14
15 func main() {
16     var err error
17     var tpl *template.Template
18     tpl, err = tpl.ParseGlob("templates/*.gohtml")
19     if err != nil {
20         log.Fatalln(err)
21     }
22
23     err = tpl.Execute(os.Stdout, Page{
24         Title: "My Title 2",
25         Body:  "hello world",
26     })
27     if err != nil {
28         log.Fatalln(err)
29     }
30     fmt.Println("\n*****")
31
32     err = tpl.ExecuteTemplate(os.Stdout, "tpl.gohtml", Page{
33         Title: "My Title 2",
34         Body:  "hello world",
35     })
36     if err != nil {
37         log.Fatalln(err)
38     }
39     fmt.Println("\n*****")
40
41     err = tpl.ExecuteTemplate(os.Stdout, "tpl2.gohtml", Page{
42         Title: "My Title 2",
43         Body:  "hello world",
44     })
45     if err != nil {
46         log.Fatalln(err)
47     }
48 }
```

Terminal

```
+ 05 $ go run main.go
+ <!doctype html>
+ <html lang="en">
+ <head>
+     <meta charset="UTF-8">
+     <title>My Title 2</title>
+ </head>
+ <body>
+ <p>from tpl: My Title 2</p>
+ <p>from tpl: hello world</p>
+ </body>
+ </html>
+ *****
+ <!doctype html>
+ <html lang="en">
+ <head>
+     <meta charset="UTF-8">
+     <title>My Title 2</title>
+ </head>
+ <body>
+ <p>from tpl: My Title 2</p>
+ <p>from tpl: hello world</p>
+ </body>
+ </html>
+ *****
+ <!doctype html>
+ <html lang="en">
+ <head>
+     <meta charset="UTF-8">
+     <title>My Title 2</title>
+ </head>
+ <body>
+ <p>This is the title from tpl2: My Title 2</p>
+ <p>This is the body from tpl2: hello world</p>
+ </body>
+ </html>
+ 05 $
```

exercises

Hello World

- Create an http application that returns Hello <Your Name> using a template.

The image shows a code editor interface with two tabs: "main.go" and "hw.gohtml".

main.go:

```
1 package main
2
3 import (
4     "html/template"
5     "log"
6     "os"
7 )
8
9 func main() {
10    |
11    name := "Todd"
12
13    // parse template
14    tpl, err := template.ParseFiles("hw.gohtml")
15    if err != nil {
16        log.Fatalln(err)
17    }
18
19    // execute template
20    err = tpl.Execute(os.Stdout, name)
21    if err != nil {
22        log.Fatalln(err)
23    }
24 }
```

hw.gohtml:

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6 </head>
7 <body>
8 Hello {{.}}
9 </body>
10 </html>
```

Output Terminal:

```
01 $ go run main.go
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
Hello Todd
</body>
```

Hello URL

- Create an http application that returns the currently selected URL (use a variable in your template)

The image shows a Go development environment with two code files and their rendered output.

File 1: hw.gohtml

```
1 package main
2
3 import (
4     "html/template"
5     "log"
6     "net/http"
7 )
8
9 func main() {
10    // parse template
11    tpl, err := template.ParseFiles("hw.gohtml")
12    if err != nil {
13        log.Fatalln(err)
14    }
15
16    // function
17    http.HandleFunc("/", func(res http.ResponseWriter, req *http.Request) {
18        // execute template
19        err = tpl.Execute(res, req.RequestURI)
20        if err != nil {
21            log.Fatalln(err)
22        }
23    })
24
25    // create server
26    http.ListenAndServe(":9000", nil)
27 }
```

File 2: main.go

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6 </head>
7 <body>
8 <h1>FIRST TEMPLATE</h1>
9 {{.}}
10 </body>
11 </html>
```

Output:

FIRST TEMPLATE

/doggy

Parse CSV

- Parse a CSV file then, using templates, send the results as HTML to a web browser.

table.csv x main.go x

```
1 | date,Open,High,Low,Close,Volume,Adj Close
2 | 2015-07-09,523.119995,523.77002,520.349976,520.679993,1839400,520.679993
3 | 2015-07-08,521.049988,522.734009,516.109985,516.830017,1264600,516.830017
4 | 2015-07-07,523.130005,526.179993,515.179993,525.02002,1595700,525.02002
5 | 2015-07-06,519.50,525.25,519.00,522.859985,1276500,522.859985
6 | 2015-07-02,521.080017,524.650024,521.080017,523.400024,1234000,523.400024
```

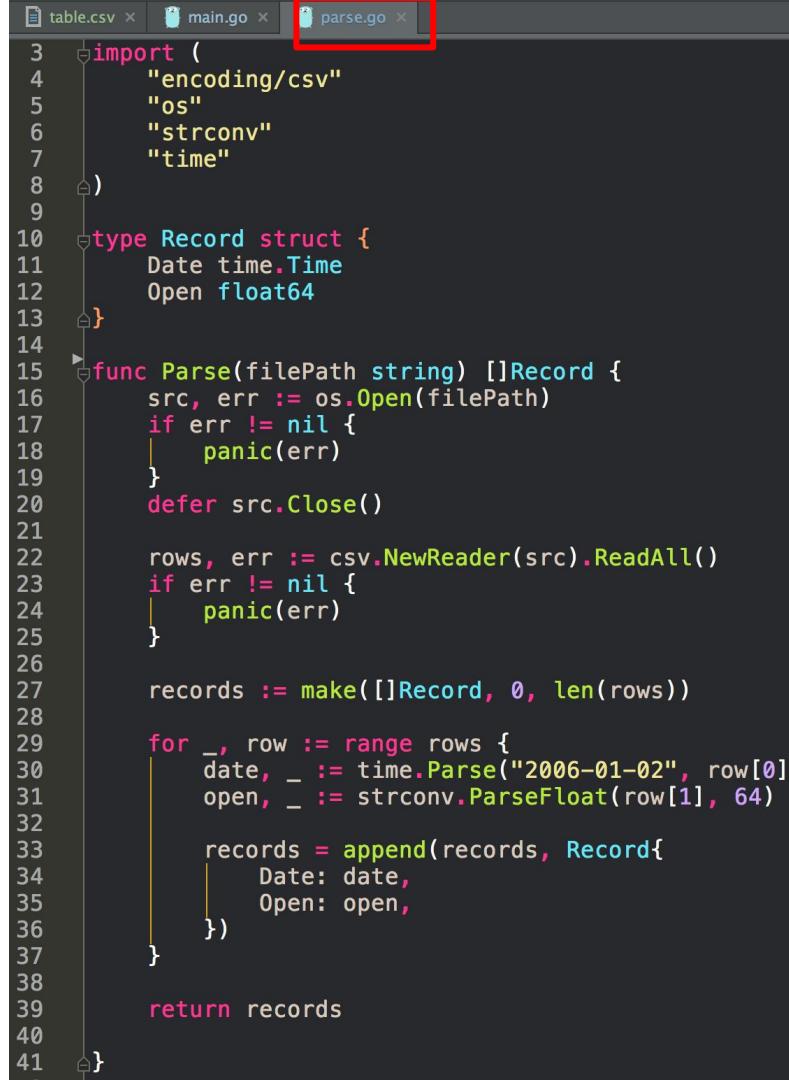
Take this,
and make it look like this

The screenshot shows a Google Chrome browser window with the title "Inbox (7) - toddmcle" and the URL "localhost:9000". The page content is a template for stock data, featuring a large heading "STOCK DATA TEMPLATE" and a bulleted list of time-series data points.

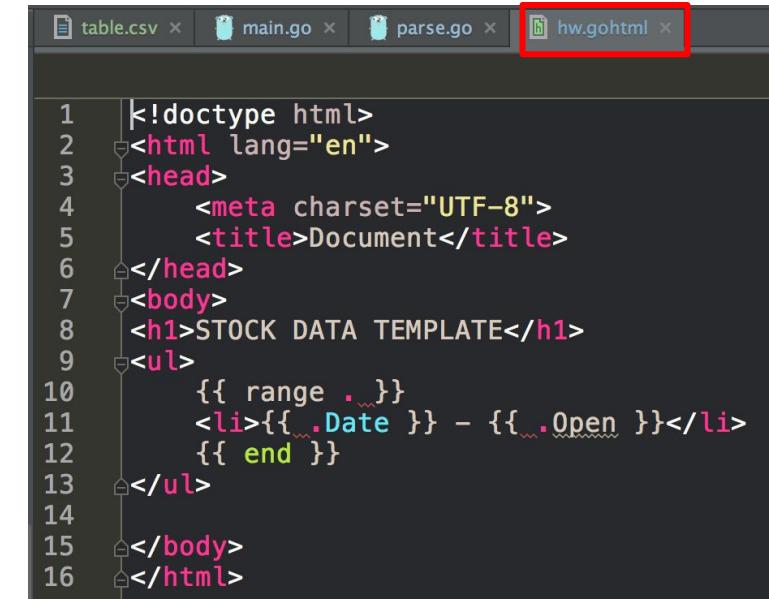
STOCK DATA TEMPLATE

- 0001-01-01 00:00:00 +0000 UTC - 0
- 2015-07-09 00:00:00 +0000 UTC - 523.119995
- 2015-07-08 00:00:00 +0000 UTC - 521.049988
- 2015-07-07 00:00:00 +0000 UTC - 523.130005
- 2015-07-06 00:00:00 +0000 UTC - 519.5
- 2015-07-02 00:00:00 +0000 UTC - 521.080017
- 2015-07-01 00:00:00 +0000 UTC - 524.72998
- 2015-06-30 00:00:00 +0000 UTC - 526.02002
- 2015-06-29 00:00:00 +0000 UTC - 525.01001
- 2015-06-26 00:00:00 +0000 UTC - 537.26001
- 2015-06-25 00:00:00 +0000 UTC - 538.869995

```
1 package main
2
3 import (
4     "html/template"
5     "log"
6     "net/http"
7     "github.com/goestoeleven/GolangTraining/47_templates/04_template_csv-parse/parse"
8 )
9
10 func main() {
11     // parse csv
12     records := parse.Parse("table.csv")
13
14     // parse template
15     tpl, err := template.ParseFiles("hw.gohtml")
16     if err != nil {
17         log.Fatalln(err)
18     }
19
20     // function
21     http.HandleFunc("/", func(res http.ResponseWriter, req *http.Request) {
22         // execute template
23         err = tpl.Execute(res, records)
24         if err != nil {
25             log.Fatalln(err)
26         }
27     })
28
29     // create server
30     http.ListenAndServe(":9000", nil)
31 }
```



```
table.csv x main.go x parse.go x
3 import (
4     "encoding/csv"
5     "os"
6     "strconv"
7     "time"
8 )
9
10 type Record struct {
11     Date time.Time
12     Open float64
13 }
14
15 func Parse(filePath string) []Record {
16     src, err := os.Open(filePath)
17     if err != nil {
18         panic(err)
19     }
20     defer src.Close()
21
22     rows, err := csv.NewReader(src).ReadAll()
23     if err != nil {
24         panic(err)
25     }
26
27     records := make([]Record, 0, len(rows))
28
29     for _, row := range rows {
30         date, _ := time.Parse("2006-01-02", row[0])
31         open, _ := strconv.ParseFloat(row[1], 64)
32
33         records = append(records, Record{
34             Date: date,
35             Open: open,
36         })
37     }
38
39     return records
40 }
41
```



```
table.csv x main.go x parse.go x hw.gohtml x
1 <!doctype html>
2 <html lang="en">
3     <head>
4         <meta charset="UTF-8">
5         <title>Document</title>
6     </head>
7     <body>
8         <h1>STOCK DATA TEMPLATE</h1>
9         <ul>
10            {{ range . }}
11            <li>{{_.Date }} - {{_.Open }}</li>
12            {{ end }}
13        </ul>
14
15    </body>
16 </html>
```