Go Templating

Lecture 05

Learning Outcomes

After completing this lesson, you should be able to

- Explain **what Go templating** is
- Explain the different **components of Go Templating**
- Explain and use **template actions** such as
  - **Conditional**, **loop**, **set**, **include**
- Explain **arguments**, **variables**, **pipelines**, **functions**
- Explain the **context-awareness** feature of Go Templating Engine

Templating

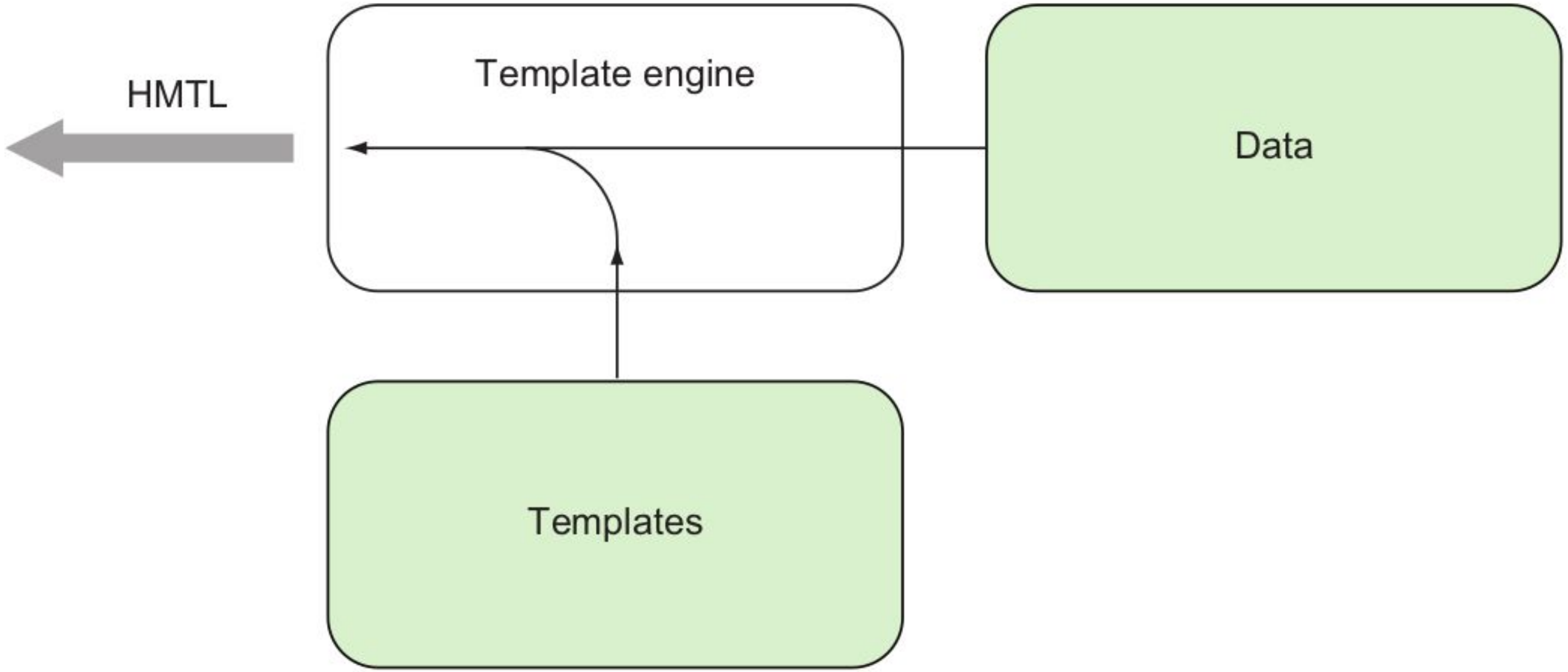
A web template is a predesigned HTML page that’s used repeatedly by a software program, called a template engine, to generate one or more HTML pages

Go’s standard library provides two template packages

`text/template`

`html/template`

Go Templating Components

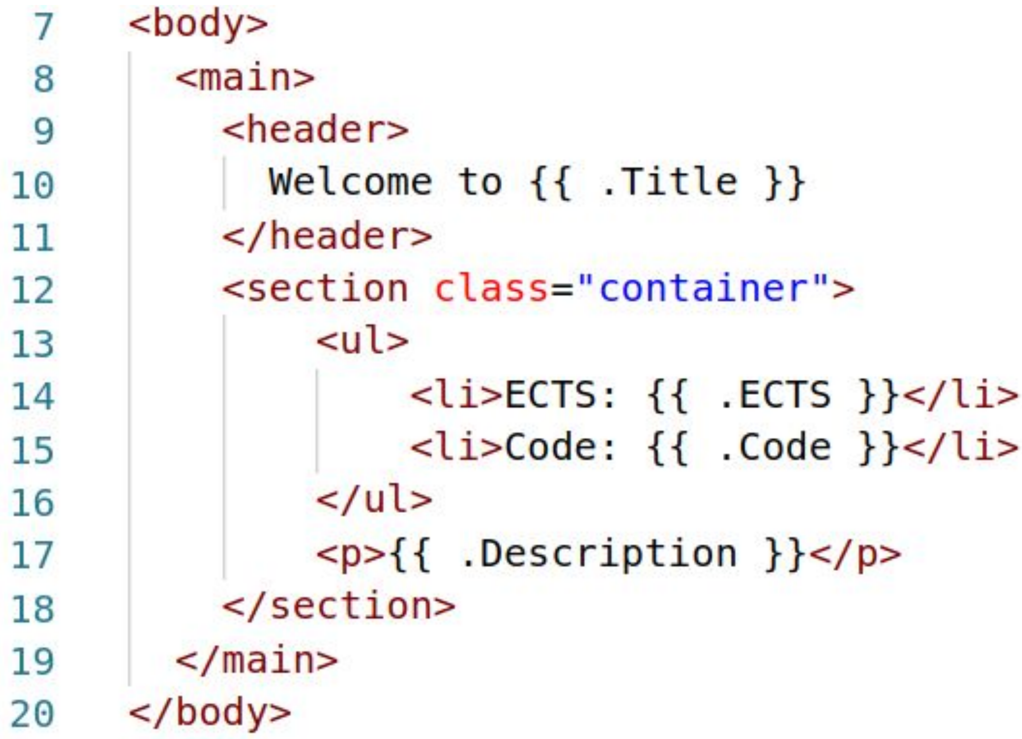


Go Templating Components

Template engines often combine data with templates to produce the final HTML

Go templates are **`text or html`** documents , with certain commands embedded in them, called **actions**

Actions are added between two double braces, `{{` and `}}`

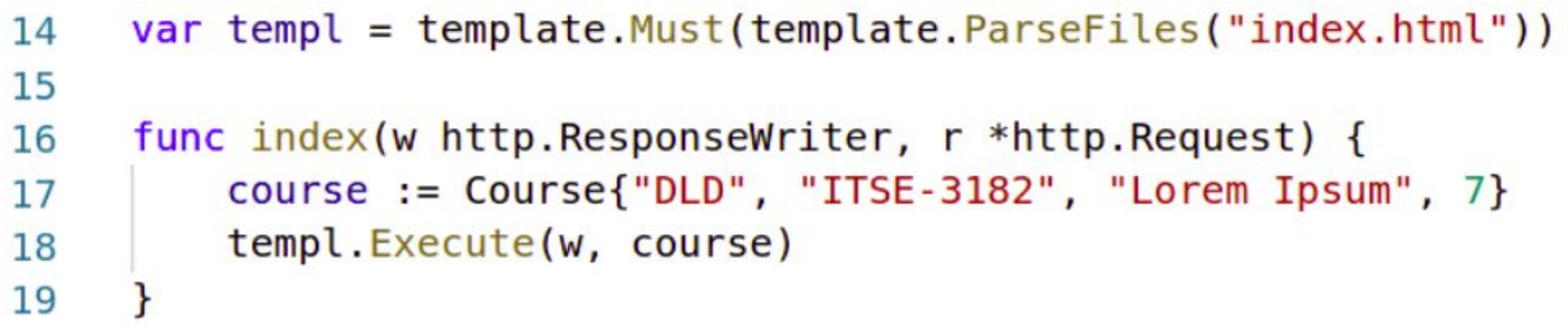


```html
<body>
  <main>
    <header>
      Welcome to {{ .Title }}
    </header>
    <section class="container">
      <ul>
        <li>ECTS: {{ .ECTS }}</li>
        <li>Code: {{ .Code }}</li>
      </ul>
      <p>{{ .Description }}</p>
    </section>
  </main>
</body>
```

Go Templating Components

The text is parsed and executed by the template engine to produce another piece of text

Handlers usually call template engines to **combine data with** the **templates** and return the resultant HTML to the client



```go
var templ = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
    course := Course{"DLD", "ITSE-3182", "Lorem Ipsum", 7}
    templ.Execute(w, course)
}
```

Go Templating Components

Go has a `text/template` standard library that's a generic template engine for any type of text format

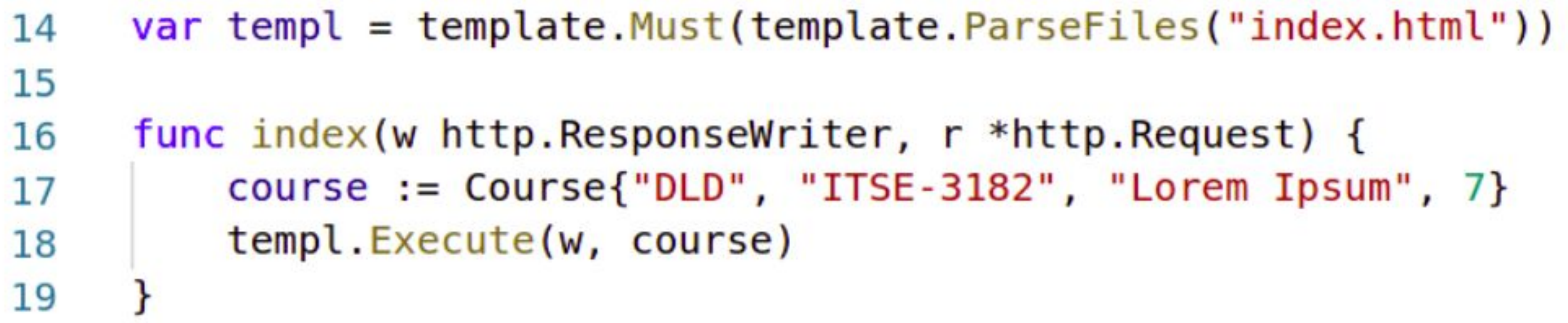
It also has an `html/template` library that's a specific template engine for **`HTML`**

> https://golang.org/pkg/html/template/
>
> https://golang.org/pkg/text/template/

Go Templating Components

Using the Go web template engine requires two steps:

1. **Parse the text-formatted template source**, which can be a string or from a template file, **to create a parsed template struct**
2. **Execute the parsed template**, passing a **`ResponseWriter`** and some **`data`** to it



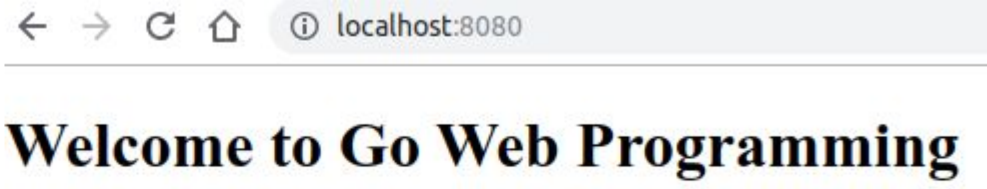
```go
var templ = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
	course := Course{"DLD", "ITSE-3182", "Lorem Ipsum", 7}
	templ.Execute(w, course)
}
```

Parsing templates

`ParseFiles` function/method can take in one or more filenames as parameters but it returns one template, regardless of the number of files it's passed

Another way to parse files is to use the `ParseGlob` function, which uses pattern matching instead of specific files

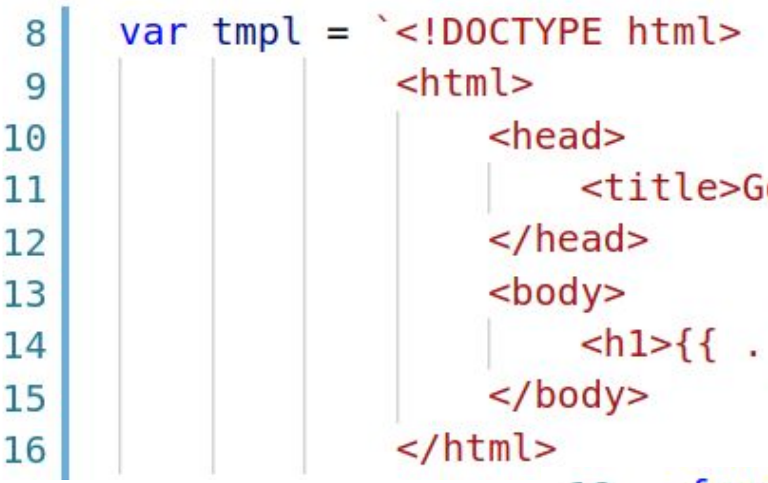
```
    t, _ := template.ParseGlob("*.html")
```

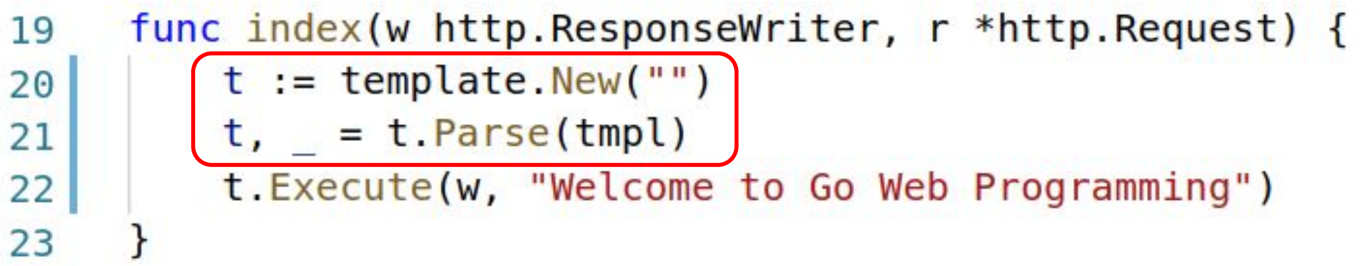

localhost:8080

Welcome to Go Web Programming

Parsing templates

You can also parse templates using **strings**


```
var tmpl = `<!DOCTYPE html>
            <html>
                <head>
                    <title>Go Web Programming</title>
                </head>
                <body>
                    <h1>{{ . }}</h1>
                </body>
            </html>
```


```
func index(w http.ResponseWriter, r *http.Request) {
    t := template.New("")
    t, _ = t.Parse(tmpl)
    t.Execute(w, "Welcome to Go Web Programming")
}
```

Exercise

Declare a go variable and put some HTMl to it and parse the it using Go Template

Parsing templates

The `Must` function wraps around a function that returns a pointer to a template and an error, and panics if the error is not a nil

`Must` is used to verify that a template is valid during parsing

```
    t := template.Must(template.ParseFiles("index.html"))
```

*In Go, **panicking** refers to a situation where the **normal flow of execution is stopped**, and if it's **within a function**, the **function returns to its caller**. The **process continues** up the stack **until it reaches** the **main program**, which **then crashes***

Executing templates

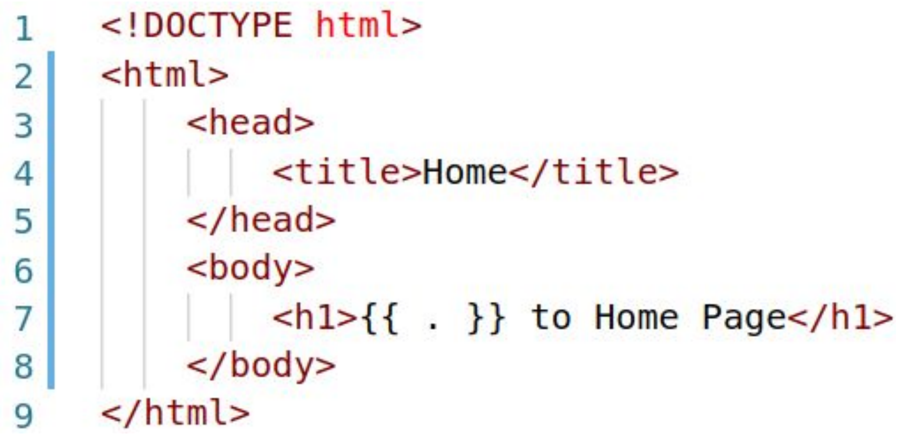
With a **single parsed template** file

```go
var templ = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
	course := Course{"DLD", "ITSE-3182", "Lorem Ipsum", 7}
	templ.Execute(w, course)
}
```

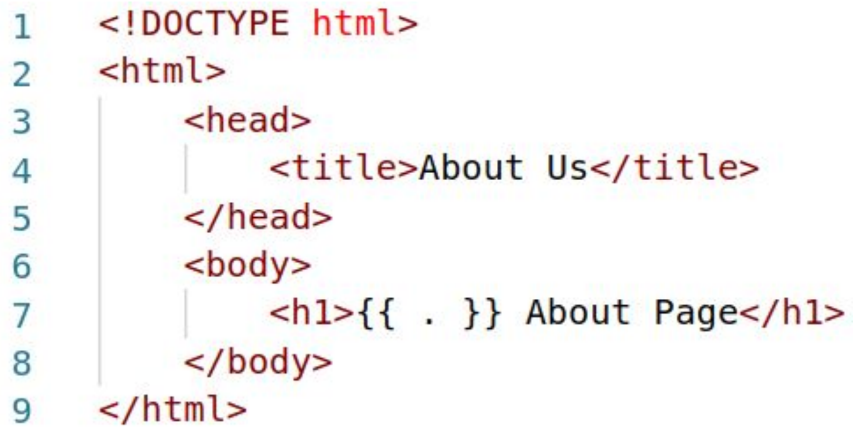
Executing templates

With **multiple parsed template** files

Assume you have two template files `index.html` and `about.html` with the following content



```html
<!DOCTYPE html>
<html>
    <head>
        <title>Home</title>
    </head>
    <body>
        <h1>{{ . }} to Home Page</h1>
    </body>
</html>
```

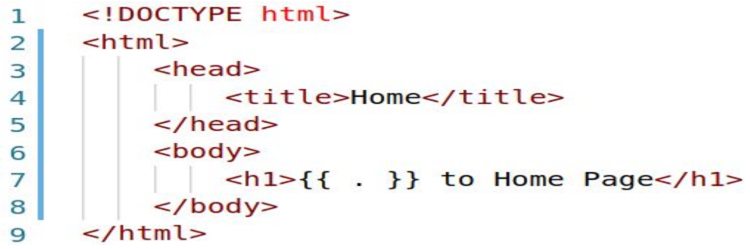


```html
<!DOCTYPE html>
<html>
    <head>
        <title>About Us</title>
    </head>
    <body>
        <h1>{{ . }} About Page</h1>
    </body>
</html>
```

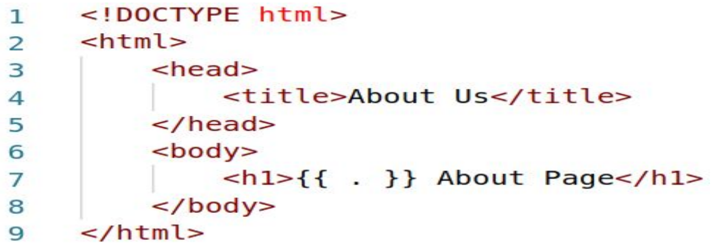
Executing templates

With **multiple parsed template** files

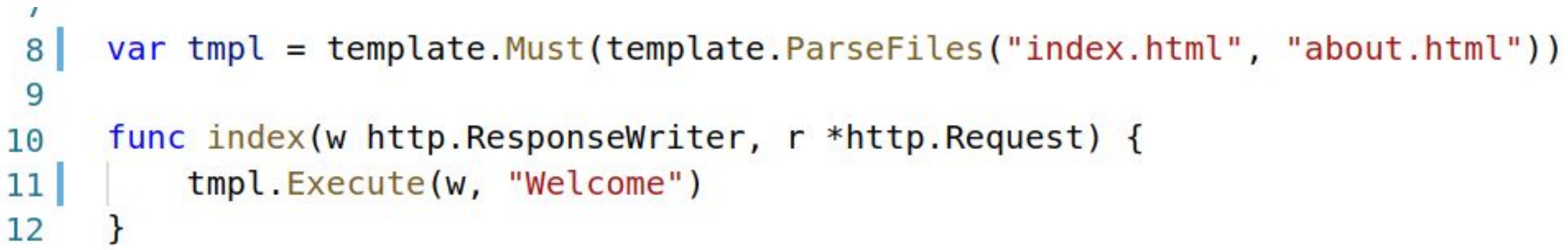
What would be the output after executing the following code?



```html
<!DOCTYPE html>
<html>
    <head>
        <title>Home</title>
    </head>
    <body>
        <h1>{{ . }} to Home Page</h1>
    </body>
</html>
```



```html
<!DOCTYPE html>
<html>
    <head>
        <title>About Us</title>
    </head>
    <body>
        <h1>{{ . }} About Page</h1>
    </body>
</html>
```



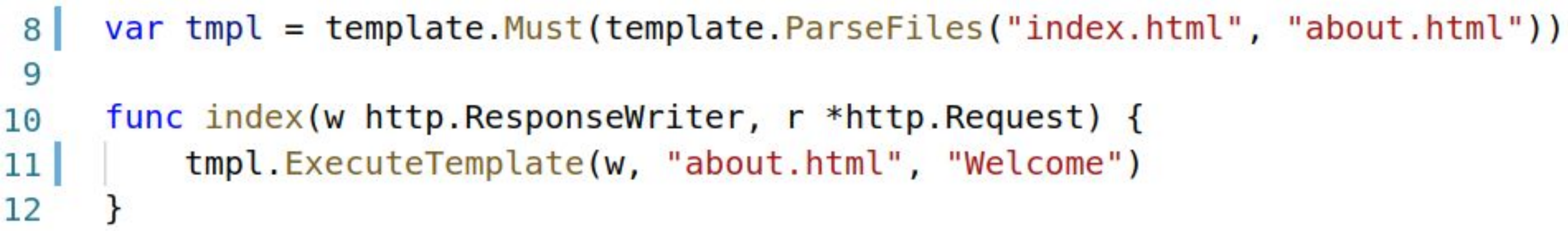
```go
var tmpl = template.Must(template.ParseFiles("index.html", "about.html"))

func index(w http.ResponseWriter, r *http.Request) {
    tmpl.Execute(w, "Welcome")
}
```

Executing templates

With **multiple parsed template** files

To execute a specific template from a set of template files, you should use the **`ExecuteTemplate`** function

For example you can use the following code to execute the **`about.html`** template



```go
var tmpl = template.Must(template.ParseFiles("index.html", "about.html"))

func index(w http.ResponseWriter, r *http.Request) {
    tmpl.ExecuteTemplate(w, "about.html", "Welcome")
}
```

Exercise

Write a handler that will selectively execute one of two parsed .html files

Actions

Actions are embedded commands in Go templates, placed between **double braces**, {{ and }}

Some of the common actions are:

Conditional actions

Iterator actions

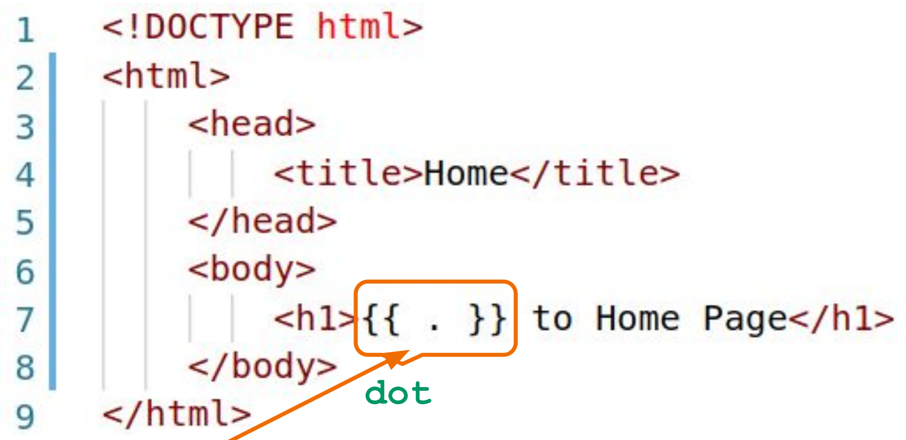
Set actions

Include actions

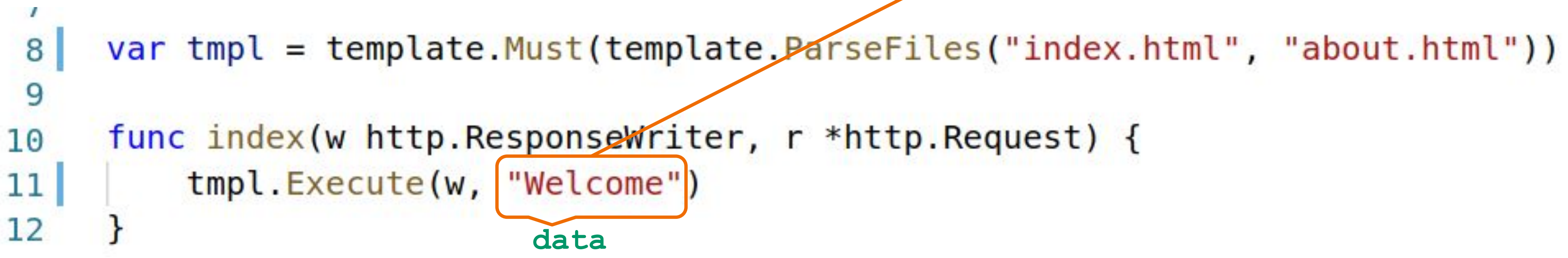
Actions

The **dot** `(.)` is an action

The dot is the evaluation of the data that's passed to the template as a second an argument to **`Execute`** or **`ExecuteTemplate`** methods



```html
<!DOCTYPE html>
<html>
    <head>
        <title>Home</title>
    </head>
    <body>
        <h1>{{ . }} to Home Page</h1>
    </body>
</html>
```



```go
var tmpl = template.Must(template.ParseFiles("index.html", "about.html"))

func index(w http.ResponseWriter, r *http.Request) {
    tmpl.Execute(w, "Welcome")
}
```

Conditional actions

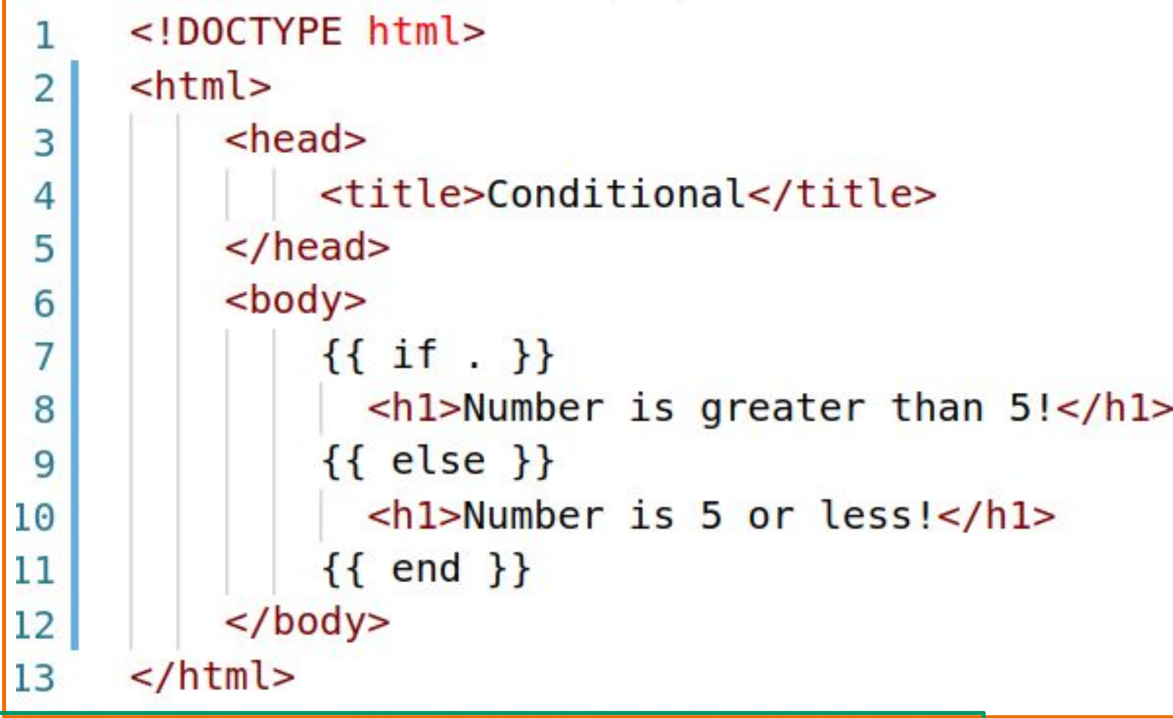
```
{{ if arg }}
   some content
{{ end }}
```

```
{{ if arg }}
   some content
{{ else }}
   other content
{{ end }}
```

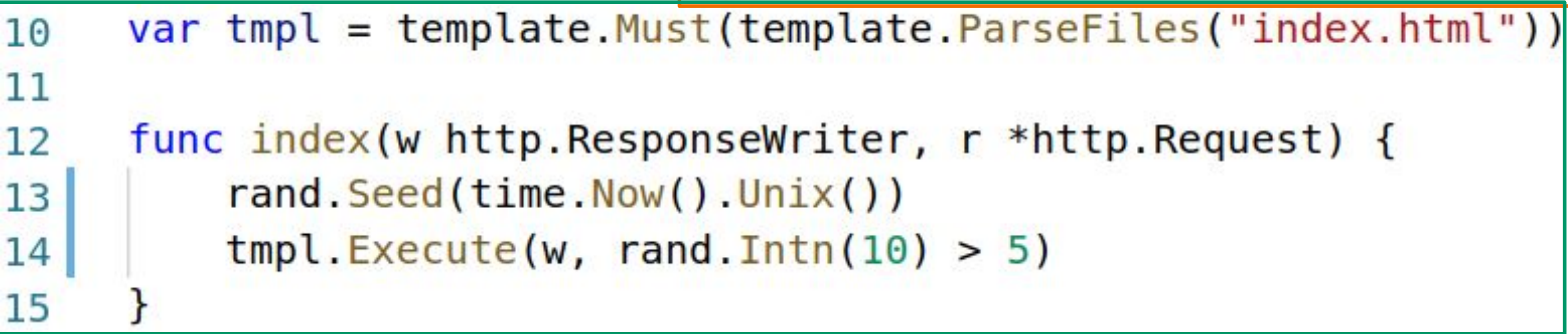
`arg` refers to argument values such as a **`literal`** value of **`string`**, **`number`** or **`boolean`**; a **`variable`**, a **`function`**/**`method`** or an **`expression`** that returns a value

Conditional actions

Example



```html
<!DOCTYPE html>
<html>
    <head>
        <title>Conditional</title>
    </head>
    <body>
        {{ if . }}
          <h1>Number is greater than 5!</h1>
        {{ else }}
          <h1>Number is 5 or less!</h1>
        {{ end }}
    </body>
</html>
```



```go
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
    rand.Seed(time.Now().Unix())
    tmpl.Execute(w, rand.Intn(10) > 5)
}
```

Iterator actions

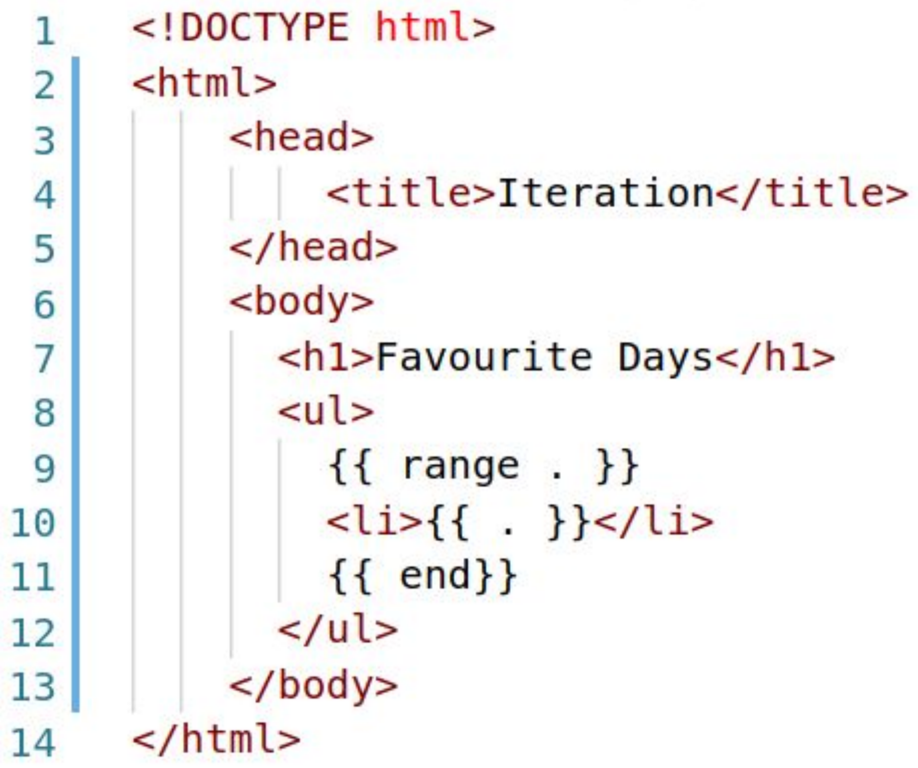
```
{{ range arg }}
   You can access arg elements here
{{ end }}
```

```
{{ range arg }}
   You can access arg elements here
{{ else }}
   This section takes effect if arg is nil
{{ end }}
```

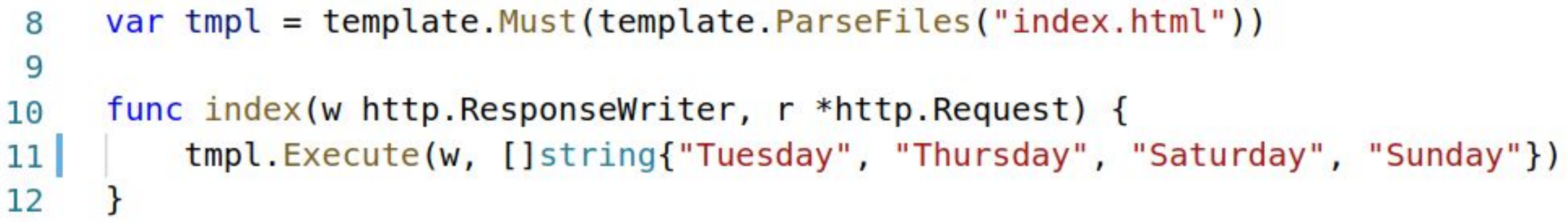
Iterator actions

Example 1:

The {{ . }} within the iterator loop is an element in the slice



```html
<!DOCTYPE html>
<html>
        <head>
                <title>Iteration</title>
        </head>
        <body>
            <h1>Favourite Days</h1>
            <ul>
                {{ range . }}
                <li>{{ . }}</li>
                {{ end}}
            </ul>
        </body>
</html>
```

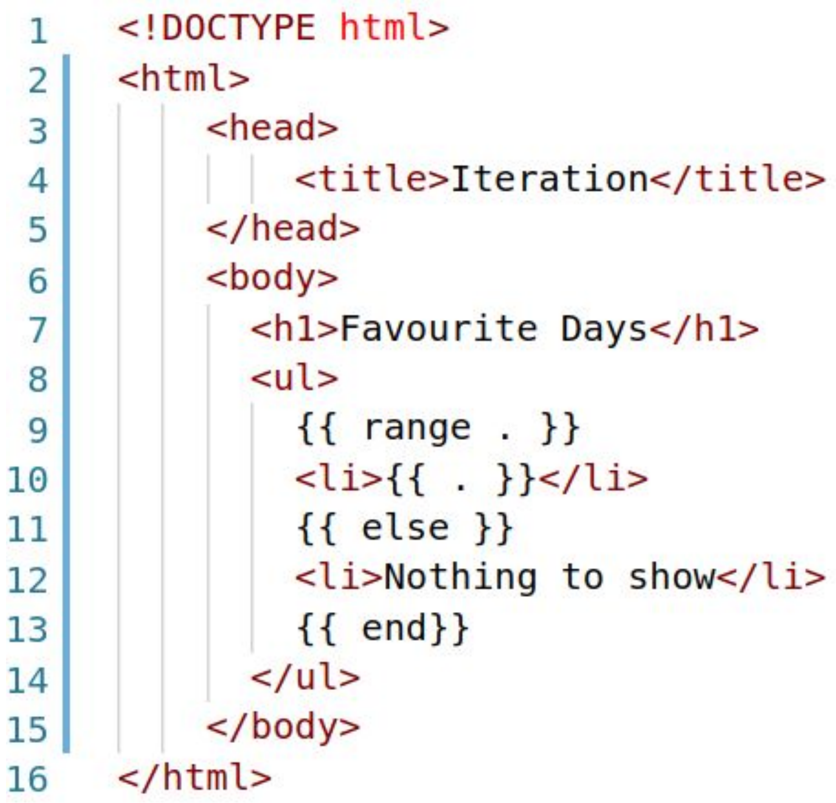


```go
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
    tmpl.Execute(w, []string{"Tuesday", "Thursday", "Saturday", "Sunday"})
}
```

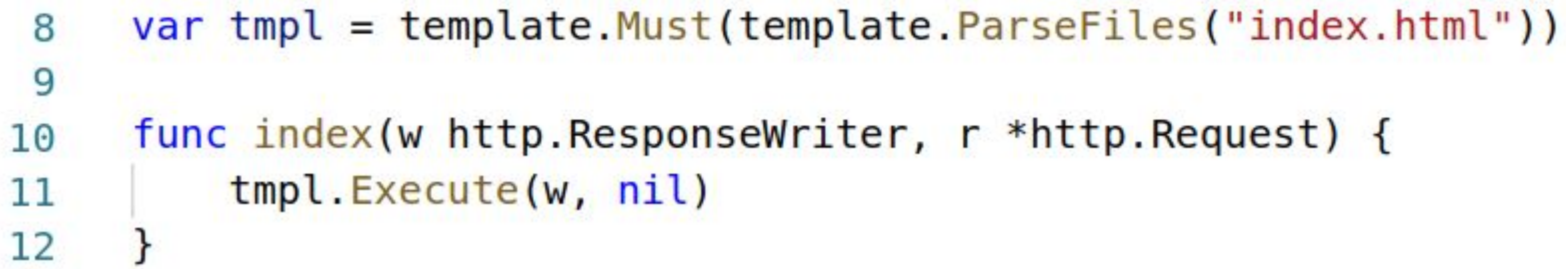
Iterator actions

Example 2:

The content of `{{ else }}` and `{{ end }}` block will be displayed if the dot `(.)` is `nil`



```html
<!DOCTYPE html>
<html>
        <head>
            <title>Iteration</title>
        </head>
        <body>
          <h1>Favourite Days</h1>
          <ul>
            {{ range . }}
            <li>{{ . }}</li>
            {{ else }}
            <li>Nothing to show</li>
            {{ end}}
          </ul>
        </body>
</html>
```



```go
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
	tmpl.Execute(w, nil)
}
```

Set actions

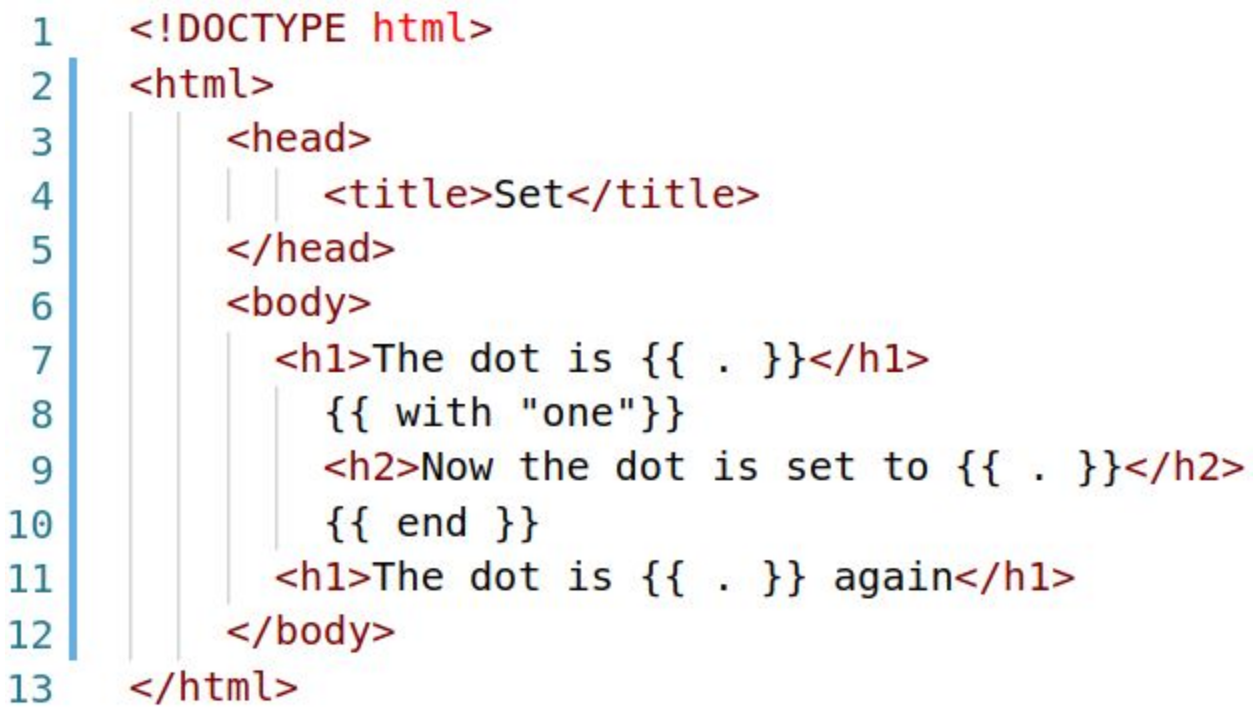
```
{{ with arg }}
  Dot is set to the arg
{{ end }}
```

```
{{ with arg }}
  Dot is set to arg
{{ else }}
  Fallback if arg is empty
{{ end }}
```

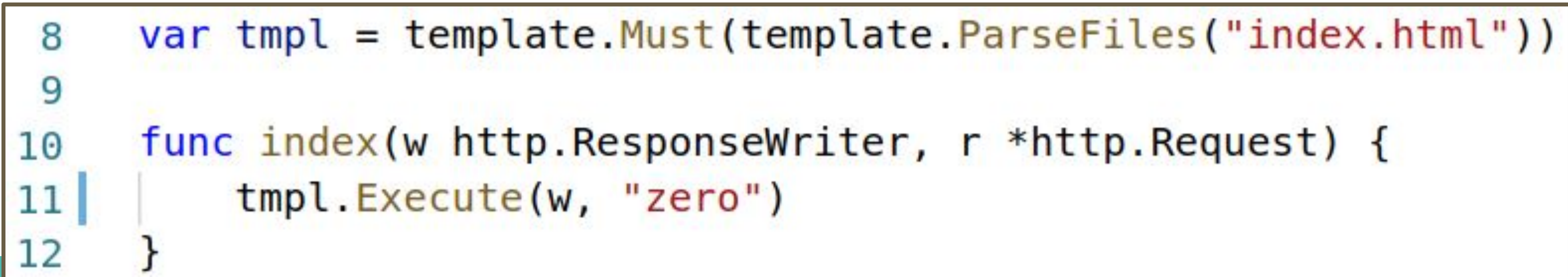
Set actions

Example 1:

Allow us to set the value of dot (.) within the enclosed section



```html
<!DOCTYPE html>
<html>
    <head>
        <title>Set</title>
    </head>
    <body>
      <h1>The dot is {{ . }}</h1>
        {{ with "one"}}
        <h2>Now the dot is set to {{ . }}</h2>
        {{ end }}
      <h1>The dot is {{ . }} again</h1>
    </body>
</html>
```

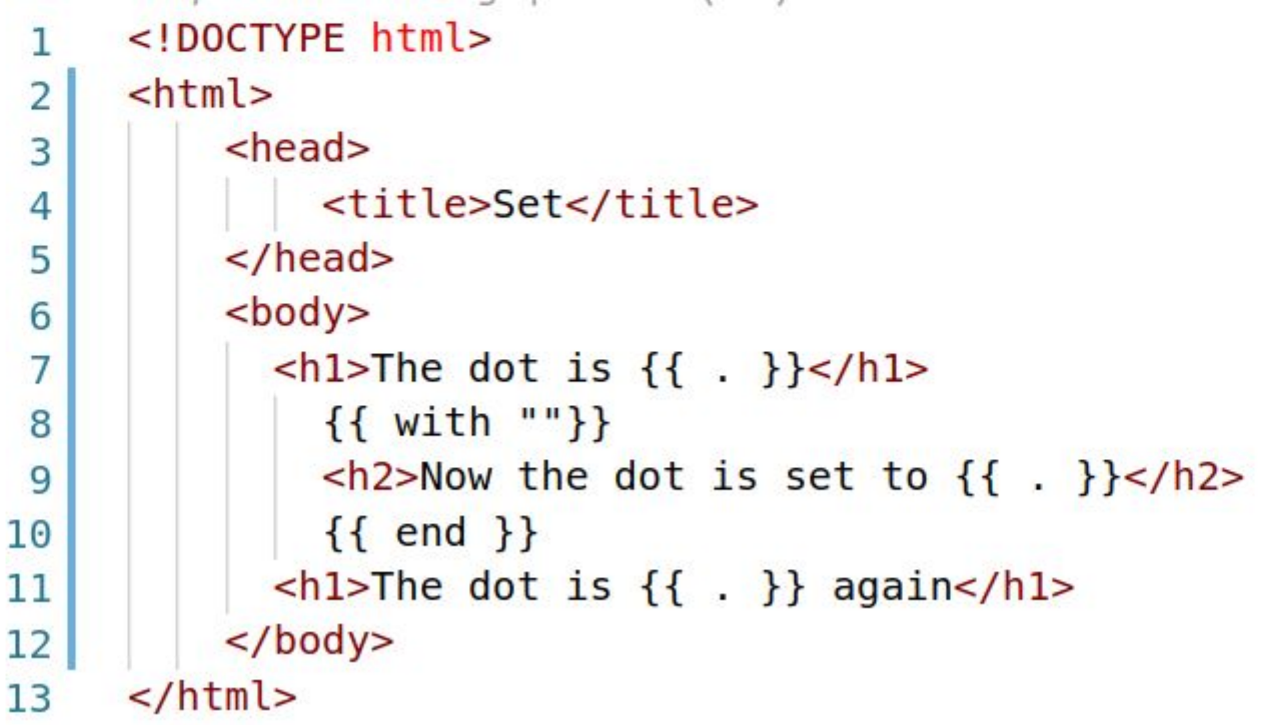


```go
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
    tmpl.Execute(w, "zero")
}
```

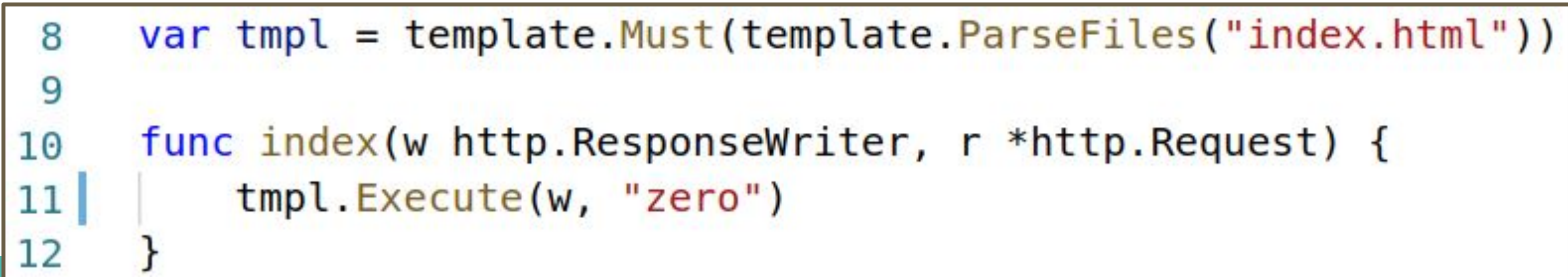
Set actions

Example 2:

What would be the output for the case shown on the right?



```
<!DOCTYPE html>
<html>
        <head>
            <title>Set</title>
        </head>
        <body>
          <h1>The dot is {{ . }}</h1>
            {{ with ""}}
            <h2>Now the dot is set to {{ . }}</h2>
            {{ end }}
          <h1>The dot is {{ . }} again</h1>
        </body>
</html>
```

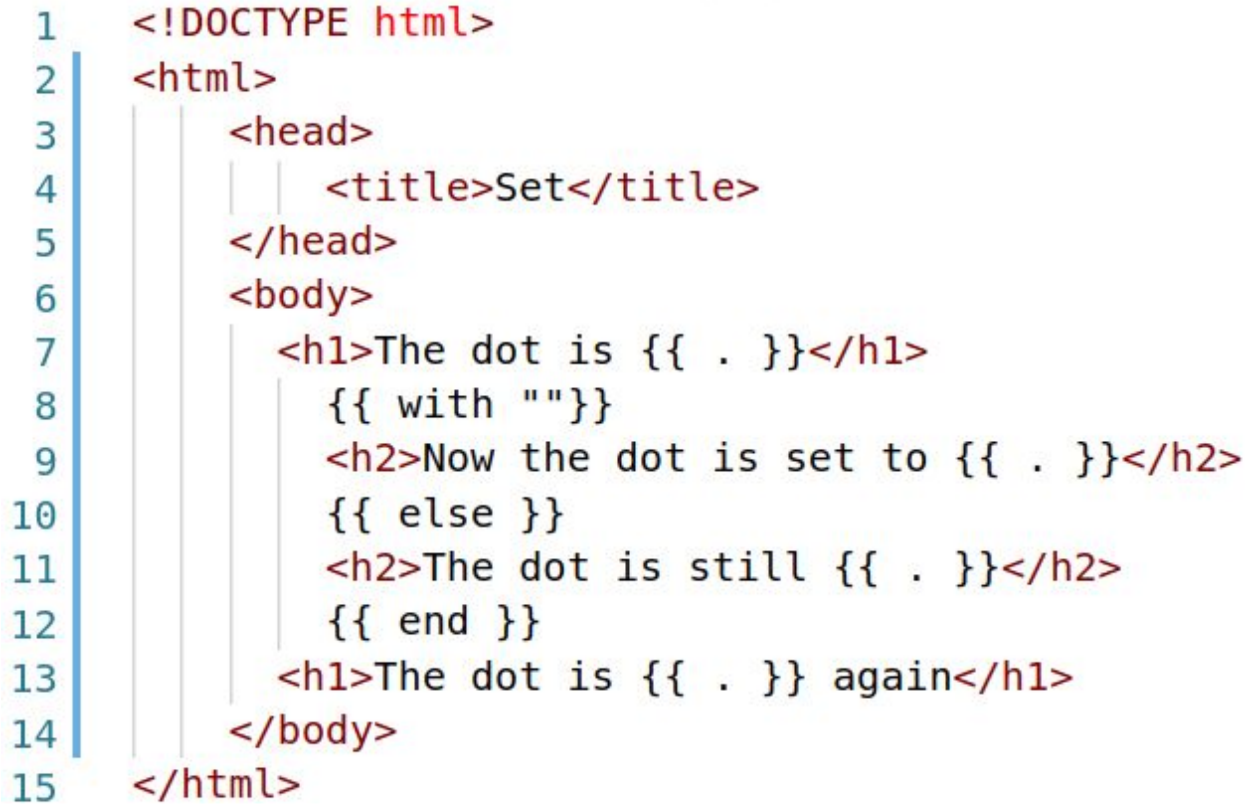


```
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
    tmpl.Execute(w, "zero")
}
```

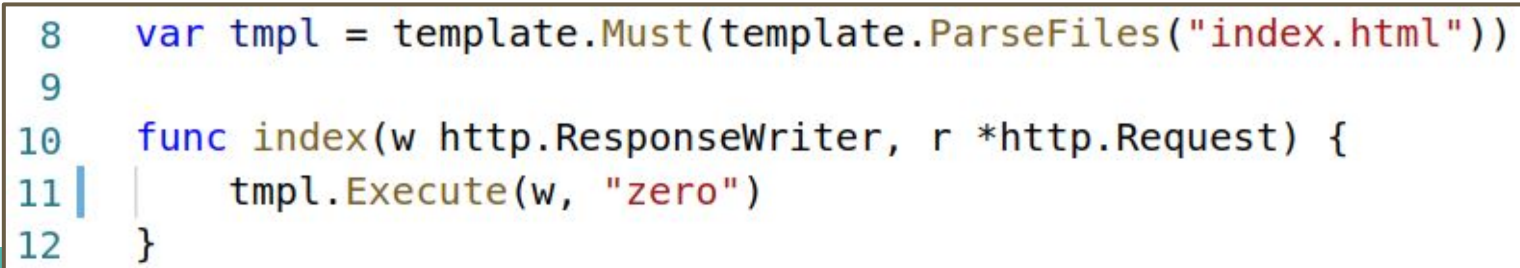
Set actions

Example 3:

What would be the output for the case shown on the right?



```html
<!DOCTYPE html>
<html>
    <head>
        <title>Set</title>
    </head>
    <body>
      <h1>The dot is {{ . }}</h1>
        {{ with ""}}
        <h2>Now the dot is set to {{ . }}</h2>
        {{ else }}
        <h2>The dot is still {{ . }}</h2>
        {{ end }}
      <h1>The dot is {{ . }} again</h1>
    </body>
</html>
```

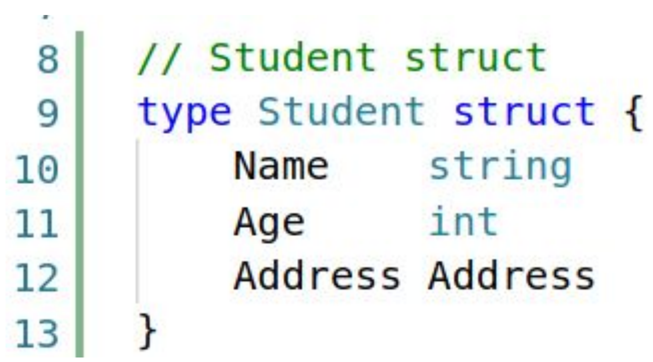


```go
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
    tmpl.Execute(w, "zero")
}
```

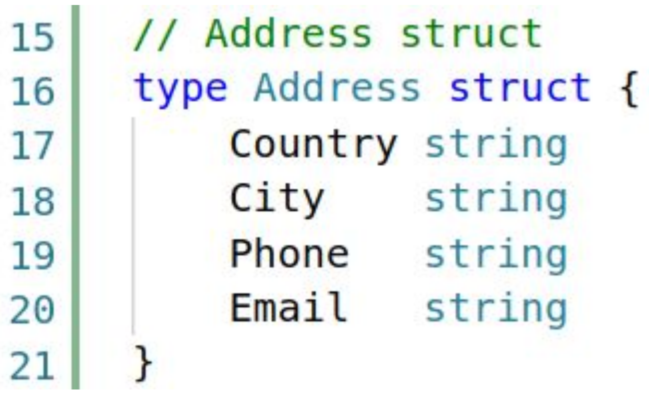
Set Action

Example 4:

Assume you have the following two structs



```go
// Student struct
type Student struct {
	Name    string
	Age     int
	Address Address
}
```

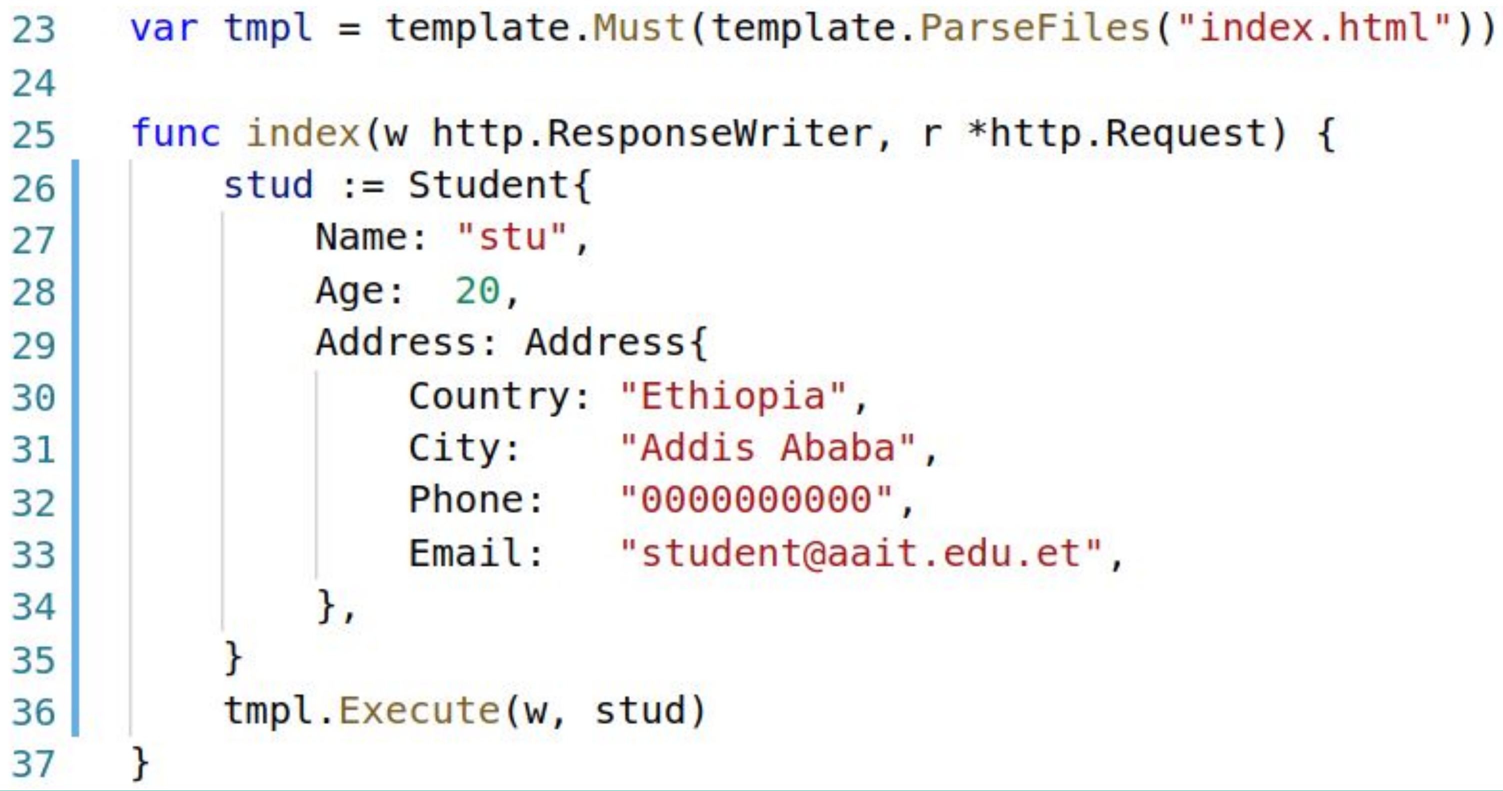


```go
// Address struct
type Address struct {
	Country string
	City    string
	Phone   string
	Email   string
}
```

Set Action

Example 4:

And assume you have the handler shown here

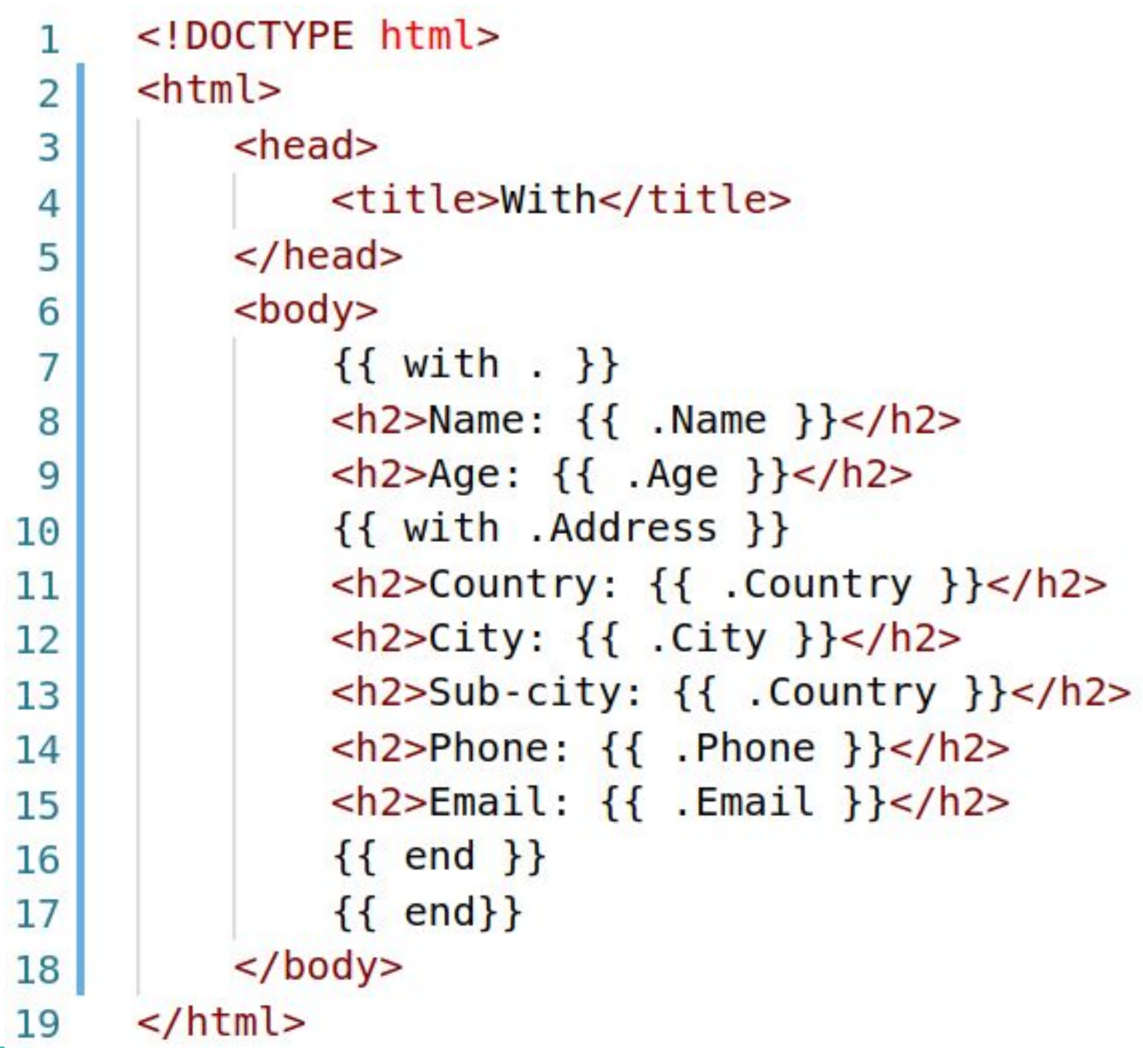


```go
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
	stud := Student{
		Name:  "stu",
		Age:   20,
		Address: Address{
			Country: "Ethiopia",
			City:    "Addis Ababa",
			Phone:   "0000000000",
			Email:   "student@aait.edu.et",
		},
	}
	tmpl.Execute(w, stud)
}
```

Set Action

Example 4:

You can access the nested Address struct elements using **`with`** action



```
<!DOCTYPE html>
<html>
    <head>
        <title>With</title>
    </head>
    <body>
        {{ with . }}
        <h2>Name: {{ .Name }}</h2>
        <h2>Age: {{ .Age }}</h2>
        {{ with .Address }}
        <h2>Country: {{ .Country }}</h2>
        <h2>City: {{ .City }}</h2>
        <h2>Sub-city: {{ .Country }}</h2>
        <h2>Phone: {{ .Phone }}</h2>
        <h2>Email: {{ .Email }}</h2>
        {{ end }}
        {{ end}}
    </body>
</html>
```

Include actions

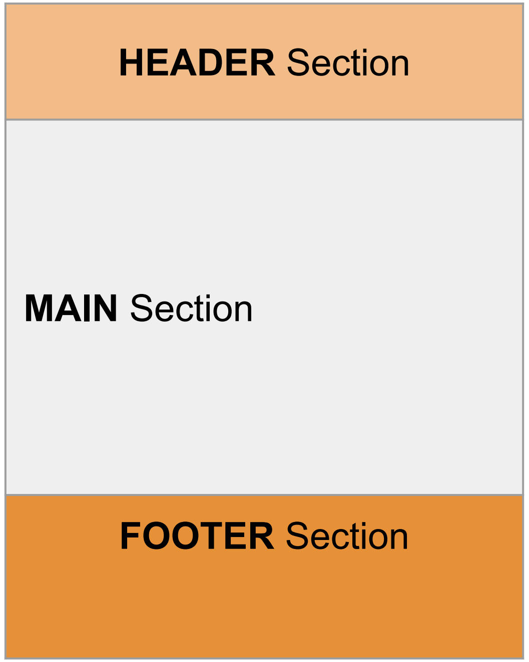
```
{{ template "name" }}
```

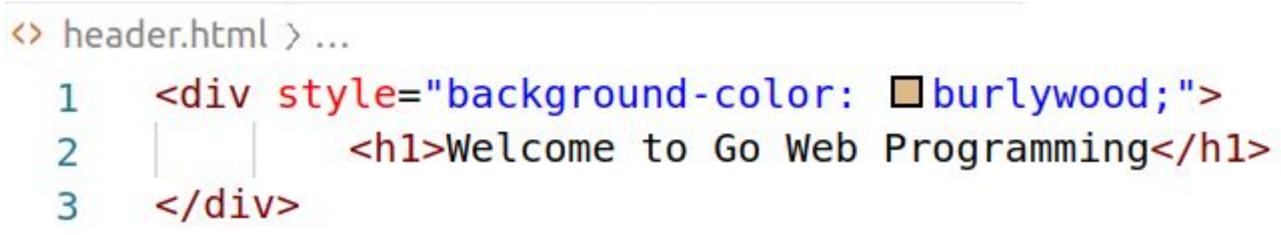
Allow us to **include/nest** a template inside another template

Name is the **name** of the template to be included

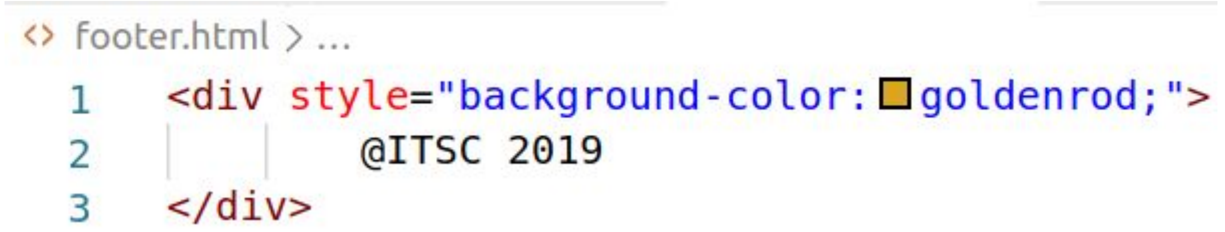
Include actions

Example Web Page Structure




header.html > ...

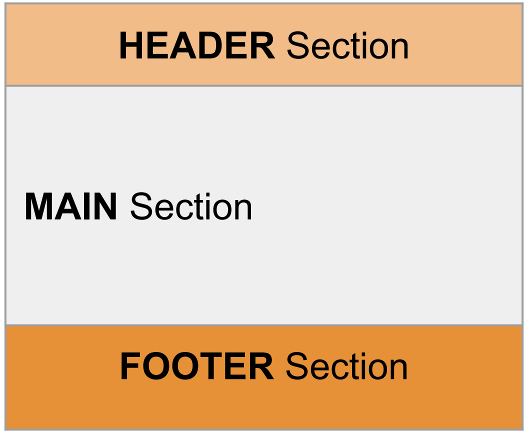
```
<div style="background-color: burlywood;">
        <h1>Welcome to Go Web Programming</h1>
</div>
```


footer.html > ...

```
<div style="background-color: goldenrod;">
        @ITSC 2019
</div>
```

Include actions

index.html

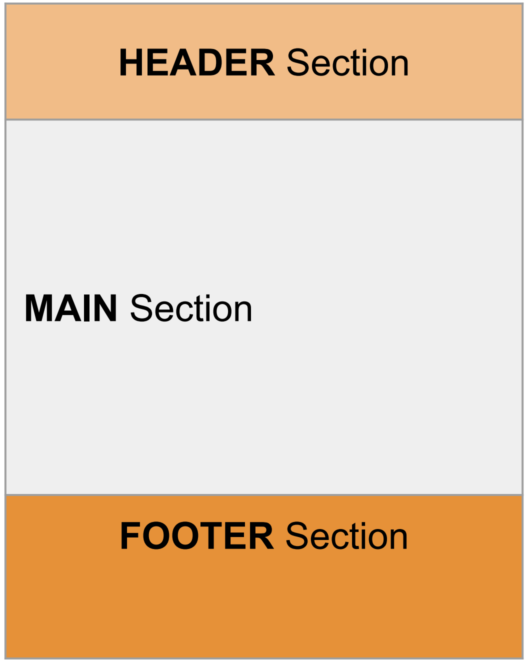


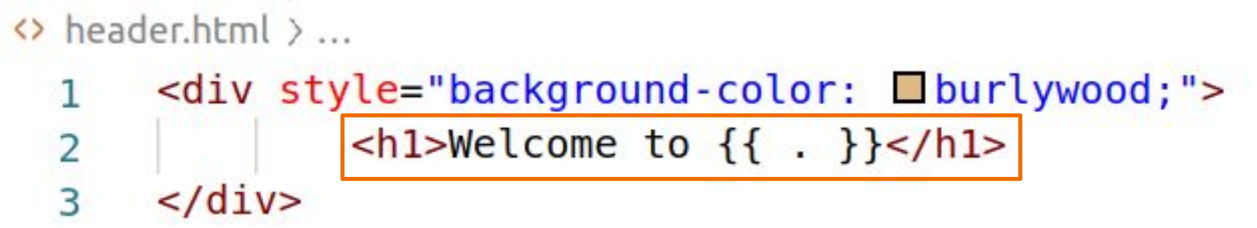
```html
<!DOCTYPE html>
<html>
    <head>
        <title>Include</title>
    </head>
    <body>
      {{ template "header.html" }}
      <hr>
      <div style="background-color: honeydew;">
        <h2>{{ . }}</h2>
      </div>
      <hr>
      {{ template "footer.html" }}
    </body>
</html>
```

```go
var tmpl = template.Must(template.ParseFiles("index.html", "header.html", "footer.html"))

func index(w http.ResponseWriter, r *http.Request) {
    tmpl.Execute(w, "Lorem ipsum")
}
```

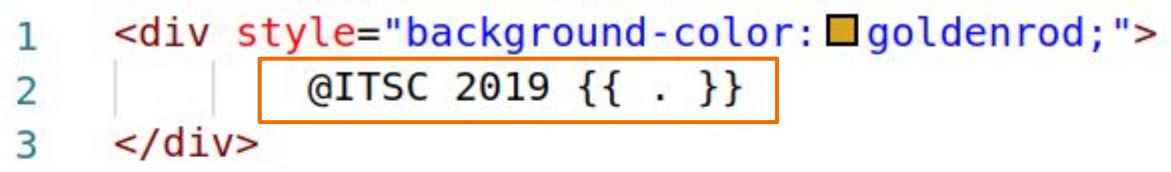
Include actions

What would be the output with the following modification to previous page




header.html › ...

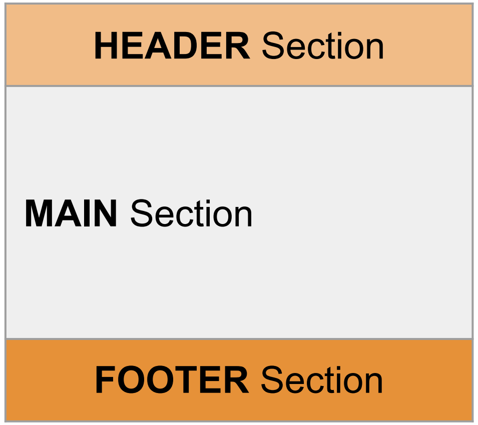
```
<div style="background-color: burlywood;">
        <h1>Welcome to {{ . }}</h1>
</div>
```

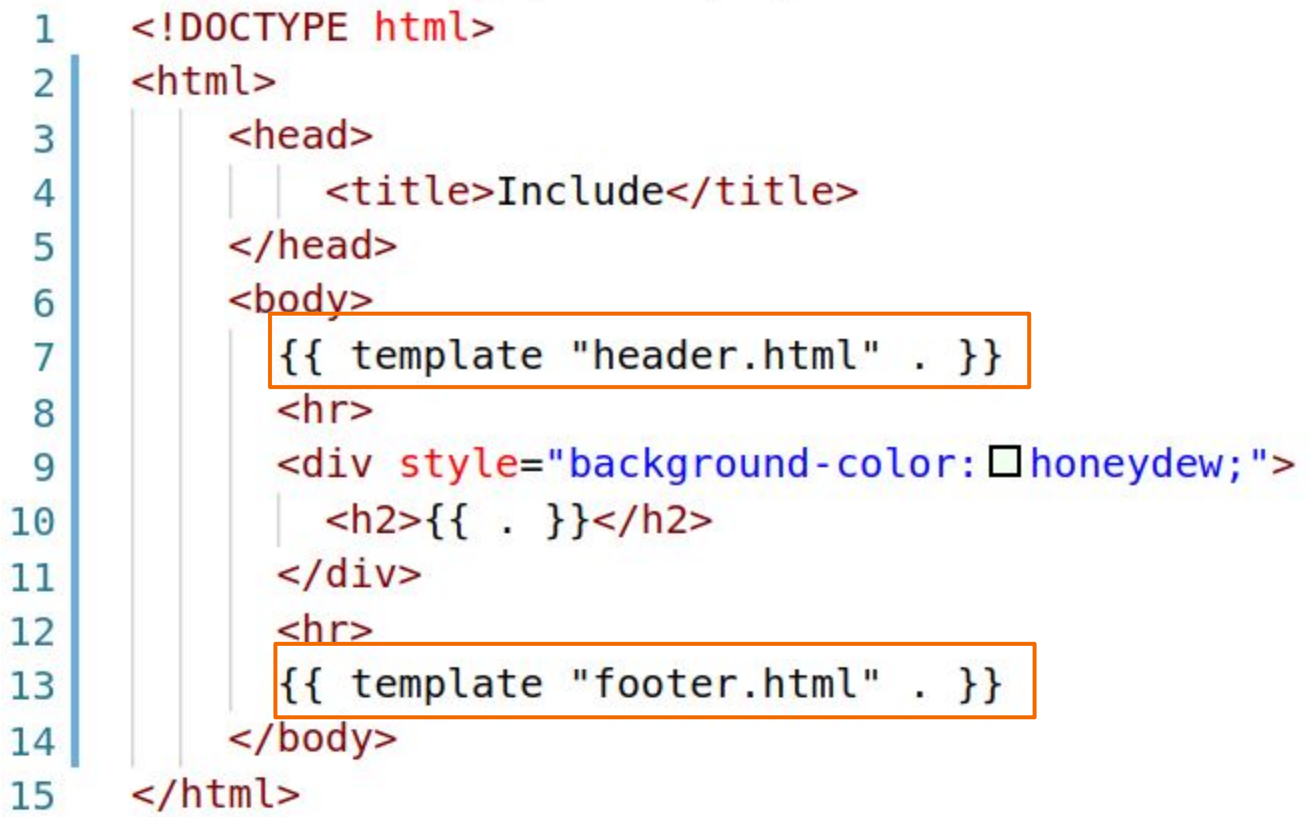


```
<div style="background-color:goldenrod;">
        @ITSC 2019 {{ . }}
</div>
```

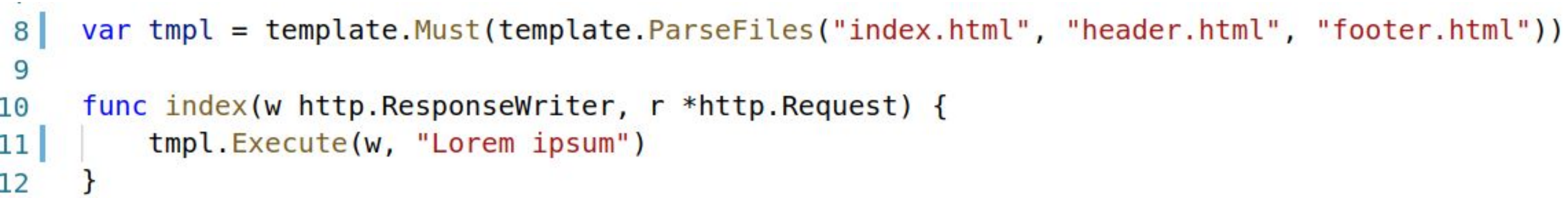
Include actions

How about now?





```html
<!DOCTYPE html>
<html>
    <head>
        <title>Include</title>
    </head>
    <body>
      {{ template "header.html" . }}
      <hr>
      <div style="background-color:□honeydew;">
        <h2>{{ . }}</h2>
      </div>
      <hr>
      {{ template "footer.html" . }}
    </body>
</html>
```



```go
var tmpl = template.Must(template.ParseFiles("index.html", "header.html", "footer.html"))

func index(w http.ResponseWriter, r *http.Request) {
    tmpl.Execute(w, "Lorem ipsum")
}
```

Arguments, variables, and pipelines

An **argument is a value that's used in a template**

It can be a `Boolean`, `integer`, `string`, and so on

It can be a `struct`, or a `field of a struct`, `map`, or `array`

It can be a `variable`, a `method` (which must return either one value, or a value and an error) or a `function`

It can also be a `dot (.)`, which is the value passed from the template engine

Arguments, variables, and pipelines

We can also set **variables** in the action

Variables start with the dollar sign `( $ )`

```
        $variable := value
```

Example Variable Usage

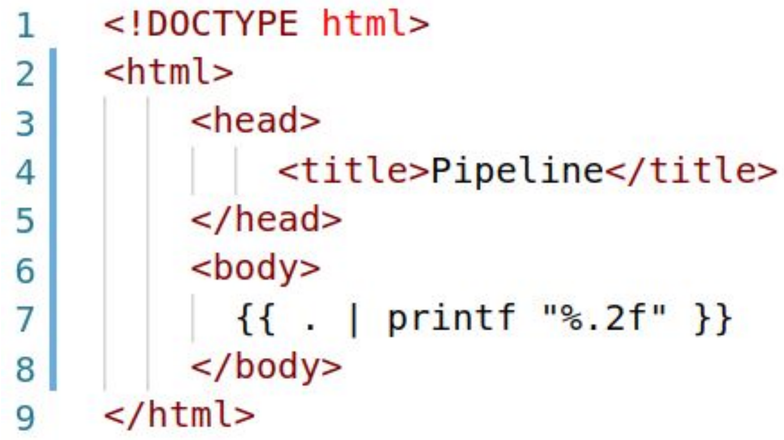
```
{{ range $key, $value := . }}
   The key is {{ $key }} and the value is {{ $value }}
{{ end }}
```

Arguments, variables, and pipelines

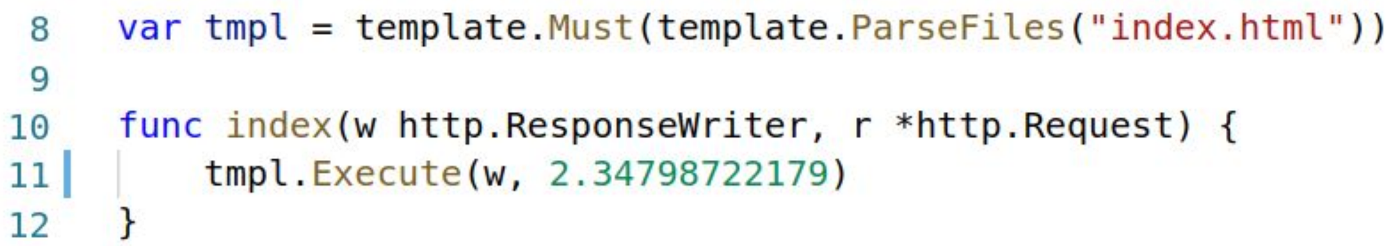
Pipelines are arguments, functions, and methods chained together in a sequence

This works much like the Unix pipeline

```
{{ p1 | p2 | p3 }}
```



```html
<!DOCTYPE html>
<html>
        <head>
                <title>Pipeline</title>
        </head>
        <body>
            {{ . | printf "%.2f" }}
        </body>
</html>
```



```go
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
	tmpl.Execute(w, 2.34798722179)
}
```

Functions

The Go template engine has a set of built-in functions such as `printf`

It also allows programmers to define their own custom functions

Although a function can take any number of input parameters, it must only return either one value, or two values only if the second value is an error

You can find all builtin functions in the link below

`https://golang.org/pkg/text/template/#hdr-Functions`

Functions

To define custom functions, you need to:

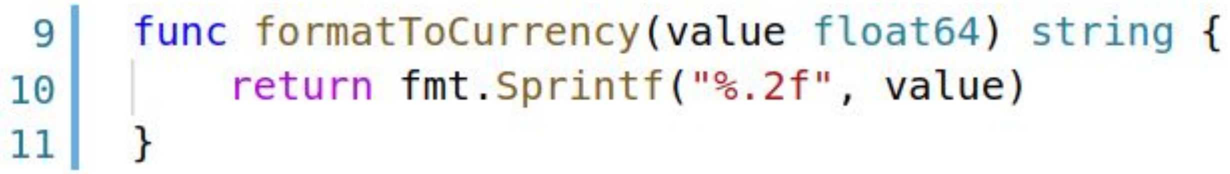
Create a `FuncMap` map, which has the **name of the function as the key** and the **actual function as the value**

Attach the `FuncMap` **to the template**

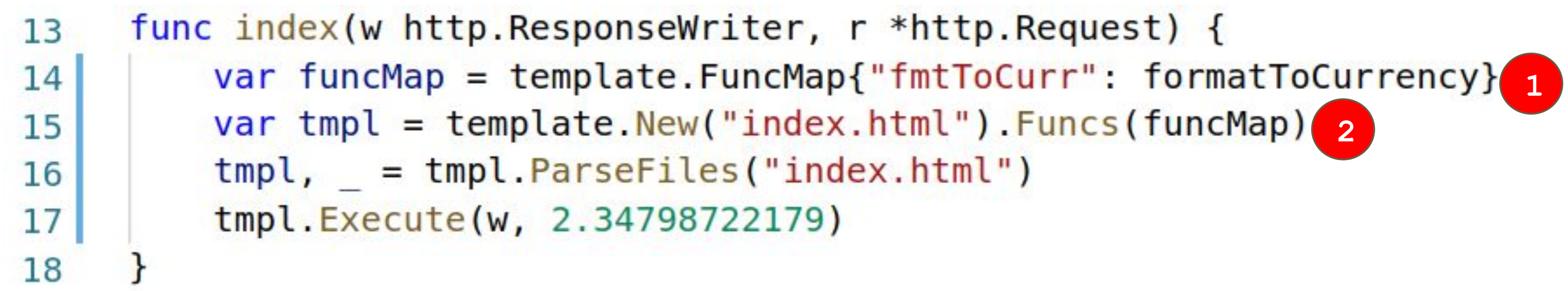
The signature of `FuncMap`

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

Functions



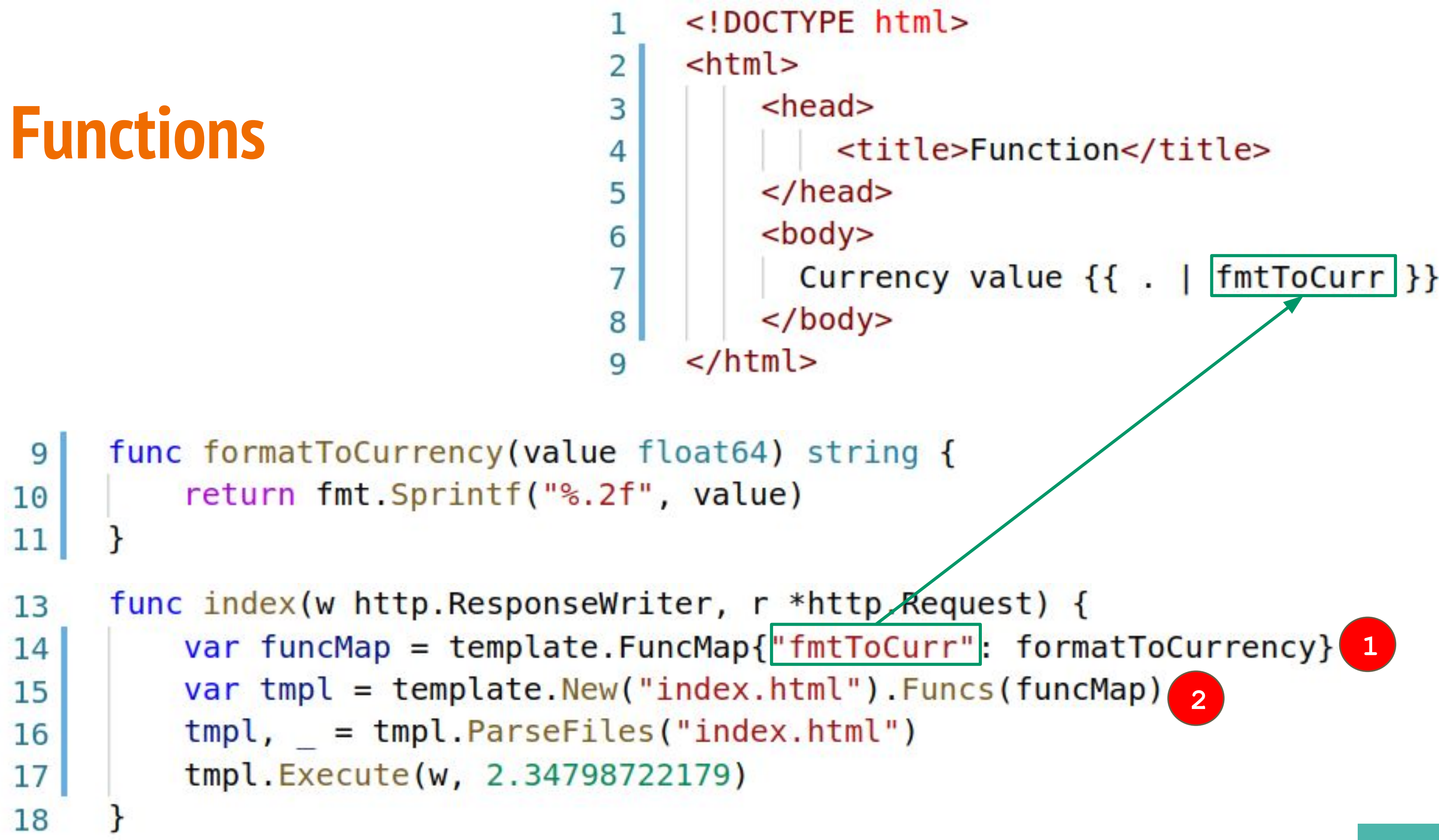
```go
func formatToCurrency(value float64) string {
    return fmt.Sprintf("%.2f", value)
}
```



```go
func index(w http.ResponseWriter, r *http.Request) {
    var funcMap = template.FuncMap{"fmtToCurr": formatToCurrency}
    var tmpl = template.New("index.html").Funcs(funcMap)
    tmpl, _ = tmpl.ParseFiles("index.html")
    tmpl.Execute(w, 2.34798722179)
}
```

You need to attach the `FuncMap` before you parse the template

Functions



```html
<!DOCTYPE html>
<html>
    <head>
        <title>Function</title>
    </head>
    <body>
      Currency value {{ . | fmtToCurr }}
    </body>
</html>
```

```go
func formatToCurrency(value float64) string {
    return fmt.Sprintf("%.2f", value)
}

func index(w http.ResponseWriter, r *http.Request) {
    var funcMap = template.FuncMap{"fmtToCurr": formatToCurrency}
    var tmpl = template.New("index.html").Funcs(funcMap)
    tmpl, _ = tmpl.ParseFiles("index.html")
    tmpl.Execute(w, 2.34798722179)
}
```

Context awareness

Go template engine is **context-aware**

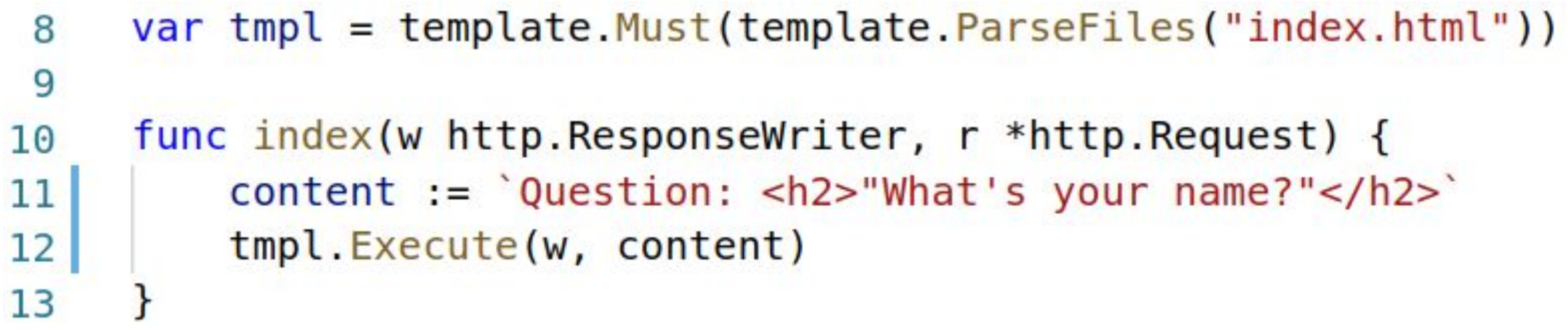
One use of this is to escape the displayed content properly

For example, If the content is **`HTML`** , it will be **`HTML`** escaped if it is **`JavaScript`**, it will be **`JavaScript`** escaped

It also recognizes content that’s part of a **`URL`** or is a **`CSS`** style

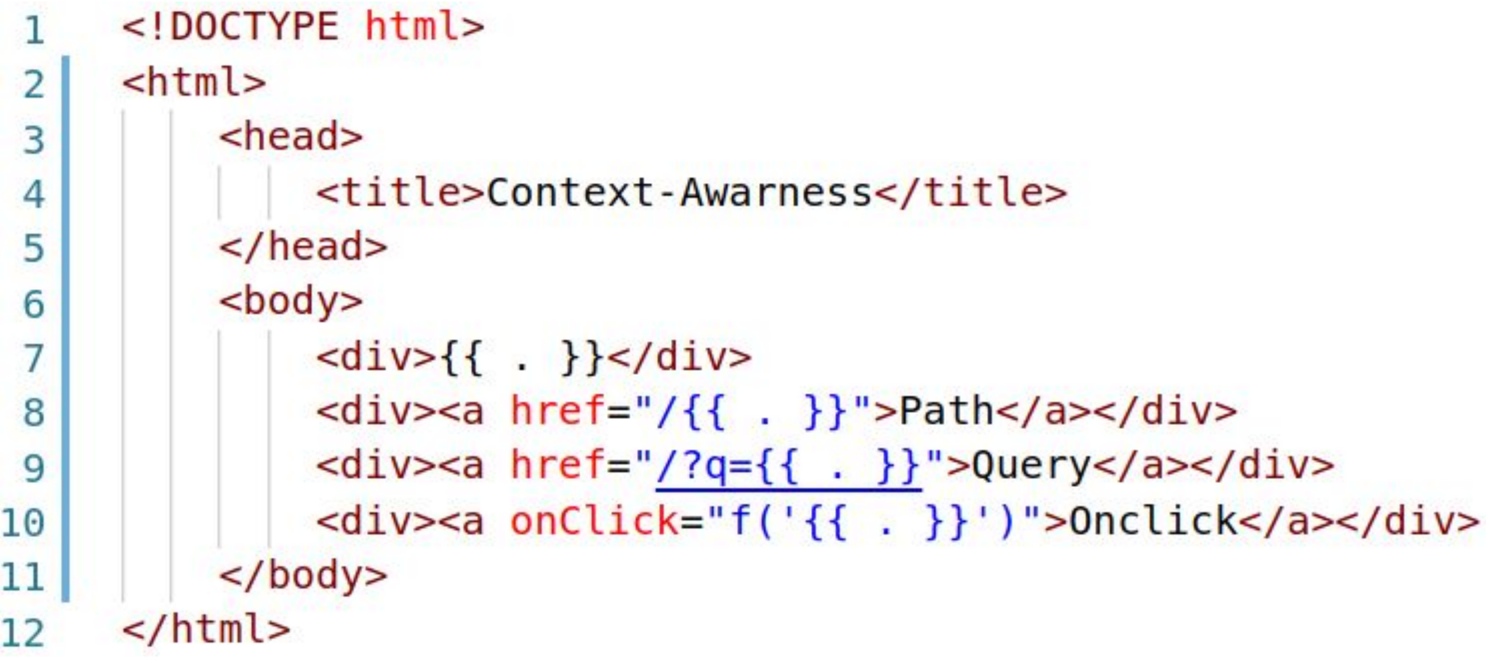
Context awareness

Example



```go
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
    content := `Question: <h2>"What's your name?"</h2>`
    tmpl.Execute(w, content)
}
```

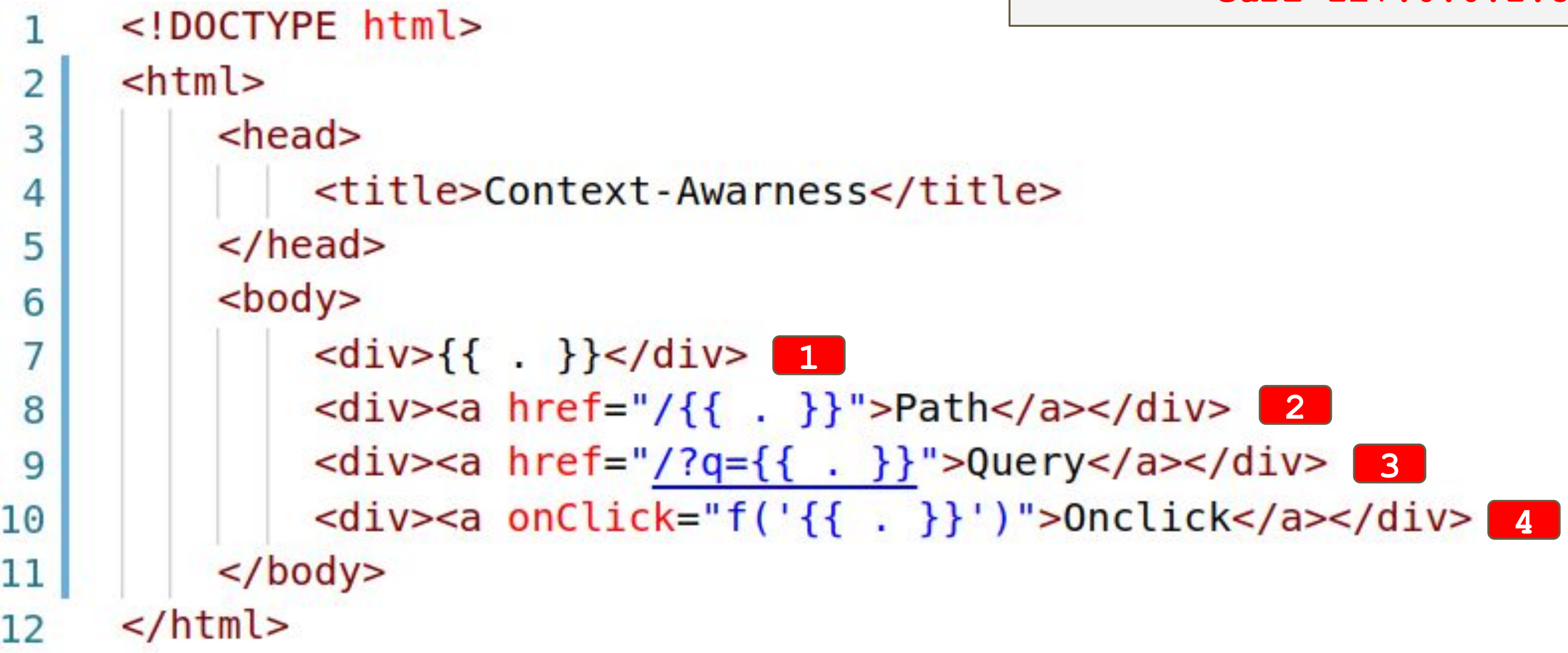
Context awareness



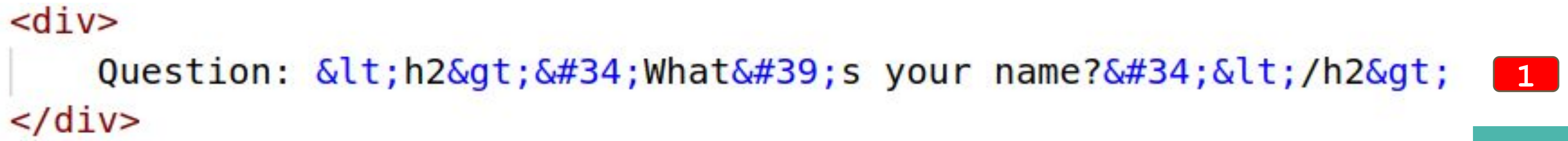
```
<!DOCTYPE html>
<html>
    <head>
        <title>Context-Awarness</title>
    </head>
    <body>
        <div>{{ . }}</div>
        <div><a href="/{{ . }}">Path</a></div>
        <div><a href="/?q={{ . }}">Query</a></div>
        <div><a onClick="f('{{ . }}')">Onclick</a></div>
    </body>
</html>
```

Context awareness

Type the following to see the output
`curl 127.0.0.1:8080/`



```
<!DOCTYPE html>
<html>
    <head>
        <title>Context-Awarness</title>
    </head>
    <body>
        <div>{{ . }}</div>  1
        <div><a href="/{{ . }}">Path</a></div>  2
        <div><a href="/?q={{ . }}">Query</a></div>  3
        <div><a onClick="f('{{ . }}')">Onclick</a></div>  4
    </body>
</html>
```



```
<div>
    Question: &lt;h2&gt;&#34;What&#39;s your name?&#34;&lt;/h2&gt;  1
</div>
```

Context awareness

Type the following to see the output
curl 127.0.0.1:8080/

```
<!DOCTYPE html>
<html>
    <head>
        <title>Context-Awarness</title>
    </head>
    <body>
        <div>{{ . }}</div>
        <div><a href="/{{ . }}">Path</a></div>  2
        <div><a href="/?q={{ . }}">Query</a></div>  3
        <div><a onClick="f('{{ . }}')">Onclick</a></div>  4
    </body>
</html>
```

2

```
<div>
    <a href="/Question:%20%3ch2%3e%22What%27s%20your%20name?%22%3c/h2%3e">Path</a>
</div>
```

Context awareness

Type the following to see the output
`curl 127.0.0.1:8080/`

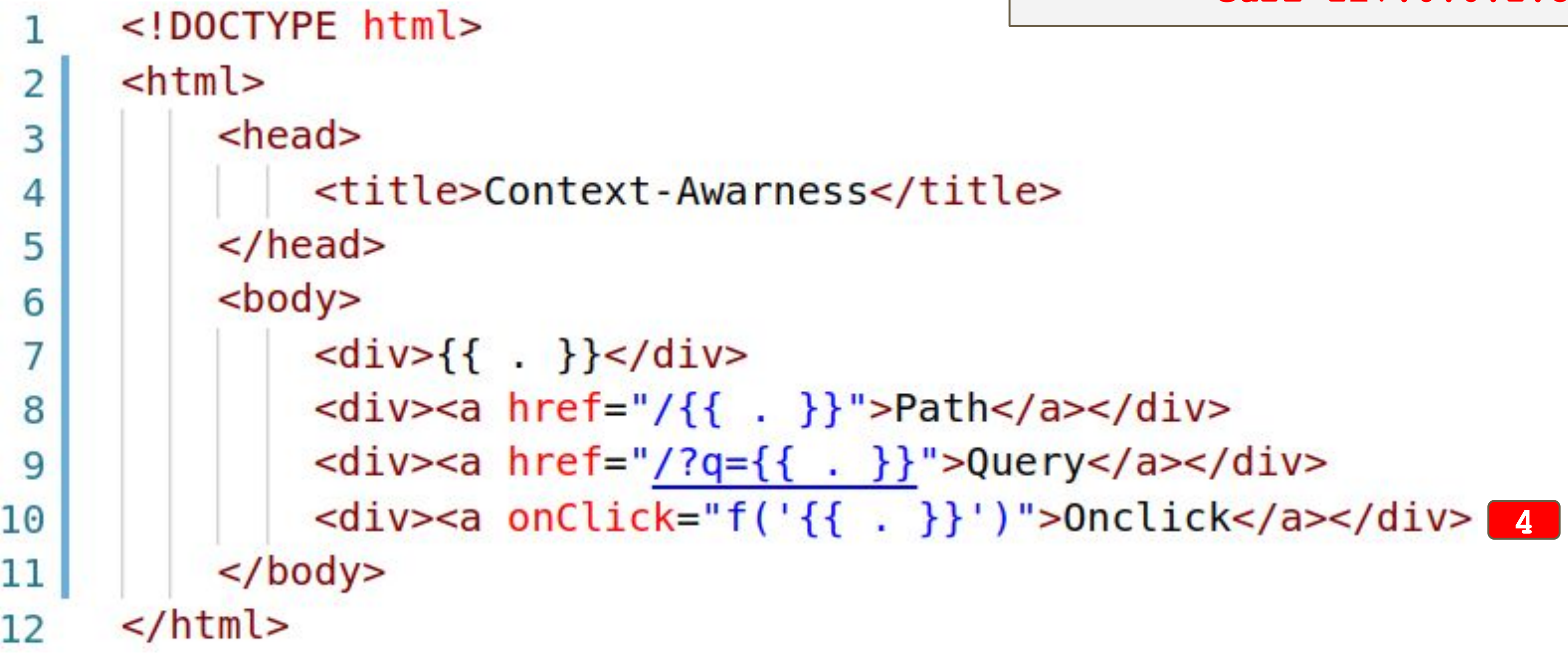
```
<!DOCTYPE html>
<html>
    <head>
        <title>Context-Awarness</title>
    </head>
    <body>
        <div>{{ . }}</div>
        <div><a href="/{{ . }}">Path</a></div>
        <div><a href="/?q={{ . }}">Query</a></div>   3
        <div><a onClick="f('{{ . }}')">Onclick</a></div>   4
    </body>
</html>
```

3

```
<div>
    <a href="/?q=Question%3a%20%3ch2%3e%22What%27s%20your%20name%3f%22%3c%2fh2%3e">Query</a>
</div>
```

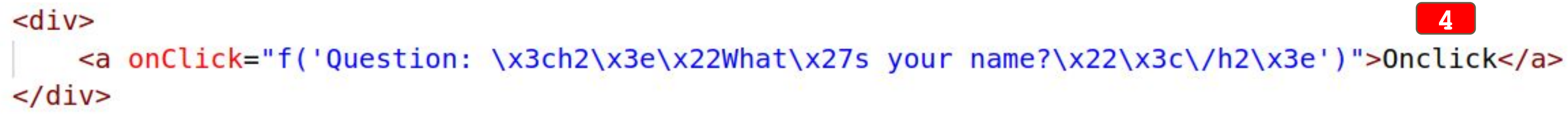
Context awareness

Type the following to see the output
curl 127.0.0.1:8080/



```
<!DOCTYPE html>
<html>
    <head>
        <title>Context-Awarness</title>
    </head>
    <body>
        <div>{{ . }}</div>
        <div><a href="/{{ . }}">Path</a></div>
        <div><a href="/?q={{ . }}">Query</a></div>
        <div><a onClick="f('{{ . }}')">Onclick</a></div>
    </body>
</html>
```

4

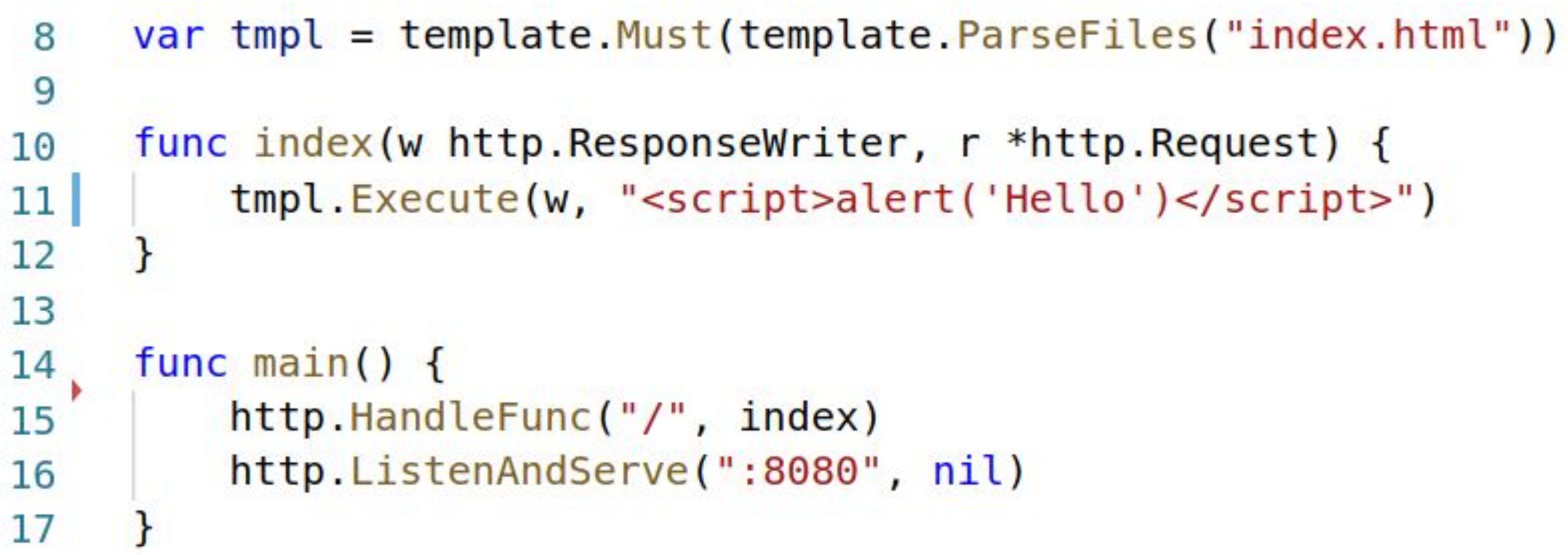


4

```
<div>
    <a onClick="f('Question: \x3ch2\x3e\x22What\x27s your name?\x22\x3c\/h2\x3e')">Onclick</a>
</div>
```

text/template vs html/template

Check the difference between the output of the following code with **text/template** package or **html/template** package



```go
var tmpl = template.Must(template.ParseFiles("index.html"))

func index(w http.ResponseWriter, r *http.Request) {
	tmpl.Execute(w, "<script>alert('Hello')</script>")
}

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