Exploring the template Package

This lesson provides in-depth knowledge on functionalities and what support the template package offers.

WE'LL COVER THE FOLLOWING

Introduction

The documentation for the `template` package is available here. In the last lesson, we used templates to merge data from a (data) struct(ure)s with HTML-templates. This is very useful, indeed, for building web applications, but template techniques are more general than this. Data-driven templates can be made for generating textual output, and HTML is only a special case of this.

A **template** is executed by merging it with a data structure, in many cases a struct or a slice of structs. It rewrites a piece of text on the fly by substituting elements derived from data items passed to `templ.Execute()`. Only the exported data items are available for merging with the template. Actions can be data evaluations or control structures and are delimited by **"{{" and "}}"**. Data items may be values or pointers. The interface hides the indirection.

Field substitution

To include the content of a field within a template, enclose it within double 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. This also works when `Name` is a key to a map. A new template is created with `template.New`, which takes the template name as a string parameter. As we already encountered previously, the `Parse` methods generate a template as an internal representation by parsing some template definition string. Use `ParseFile` when the parameter is the path to a file with the template definition. When there was a problem with the parsing, their second return parameter is an `Error != nil`. In the last step, the content of a data structure `p` is merged with the template through the `Execute` method, and written to its first argument, which is an `io.Writer`. Again, an error can be returned.

This is illustrated in the following program, where the output is written to the console through `os.Stdout`:

```
package main
import (
  "os"
  "fmt"
  "text/template"
)

type Person struct {
  Name string
  nonExportedAgeField string
}

func main() {
  t := template.New("hello")
  t, _ = t.Parse("hello {{.Name}}!")
  p := Person{Name:"Mary", nonExportedAgeField: "31"} // data
  if err := t.Execute(os.Stdout, p); err != nil {
    fmt.Println("There was an error:", err.Error())
  }
}
```

Parsing the Field

To use templates, we need to import the package `text/template` (**line 5**). We also define a struct `Person` at **line 8** with *two* fields. The first field is `Name`, which is exported (it begins with a capital letter) and the second is `nonExportedAgeField` which is not exported (at **line 10**).

`nonExportedAgeField` which is not exported (at **line 10**).

Line 14 creates a new template `t` with the name **hello**. The template definition is parsed at **line 15**. **Line 16** makes an instance of `Person`: data to be shown through the template.

Calling the `Execute` method on `t` at **line 17** with the data as the second parameter shows the output. It is combined with error-handling in the same line. If there was an error during parsing, it is shown through **line 18**.

Our data structure contains a non-exported field, and when we try to merge this through a definition string like:

```
t, _ = t.Parse("your age is {{.nonExportedAgeField}}!")
```

the following error occurs: `There was an error: template: hello:1:8: executing "hello" at <.nonExportedAgeField>: nonExportedAgeField is an unexported field of struct type main.Person`.

If you simply want to substitute the second argument of `Execute()`, use **{{.}}**. When this is done in a browser context, filter the content with the HTML filter, like this: `{{html .}}` or with a field `FieldName` `{{ .FieldName |html }}`

The `|html` part asks the template engine to pass the value of `FieldName` through the HTML-formatter before outputting it, which escapes special HTML characters (such as replacing `>` with `&gt`;). This will prevent user data from corrupting the form HTML.

Validation of the templates

To check whether the template definition syntax is correct, use the `Must` function executed on the result of the `Parse`.

```
package main
import (
"text/template"
"fmt"
)

func main() {
  tOk := template.New("ok")
  //a valid template, so no panic with Must:
  template.Must(tOk.Parse("/* and a comment */ some static text: {{ .Name }}"))
  fmt.Println("The first one parsed OK.")
  fmt.Println("The next one ought to fail.")
  tErr := template.New("error_template")

  template.Must(tErr.Parse(" some static text {{ .Name }}"))
}
```

Template Validation

In the code above, **line 8** creates a new template `tOk` with the name **ok**. The template definition is parsed at **line 10**. To check for errors while parsing the template, you can use the `template.Must` method, as is done at **line 10**. This panics if the error is non-nil.

Line 13 creates a new template `tErr` with the name **error_template**. The template definition is parsed at **line 14**. The `Must` function finds an error and panics because `{{ .Name }` lacks a closing brace.

If-else construct

The output from a template resulting from `Execute` contains static text, and text contained within **{{ }}**, which is called a *pipeline*. For example, running this code:

```
t := template.New("template test")
t = template.Must(t.Parse("This is just static text. \n{{\"This is pipeline data - because it is
evaluated within the double braces.\"}} {{`So is this, but within reverse quotes.`}}\n"))
t.Execute(os.Stdout, nil)
```

gives this output:

```
This is just static text.
This is pipeline data - because it is evaluated within the double braces. So is this, but within reverse quotes.
```

Now, we can condition the output of pipeline data with if-else-end, if the pipeline is empty, like in:

```
{{if ``}} Will not print. {{end}}
```

then, the if condition evaluates to false and nothing will be output, but with this:

```
{{if `anything`}} Print IF part. {{else}} Print ELSE part.{{end}}
```

Print IF part will be output.

This is illustrated in the following program:

```
package main

import (
        "os"
        "text/template"
)

func main() {
        tEmpty := template.New("template test")
        //empty pipeline following if
        tEmpty.Execute(os.Stdout, nil)
        tEmpty = template.Must(tEmpty.Parse("Empty pipeline if demo: {{if ``}} Will not print

        tWithValue := template.New("template test")
        //non empty pipeline following if condition
        tWithValue = template.Must(tWithValue.Parse("Non empty pipeline if demo: {{if `anythi
        tWithValue.Execute(os.Stdout, nil)

        tIfElse := template.New("template test")
        //non empty pipeline following if condition
        tIfElse = template.Must(tIfElse.Parse("if-else demo: {{if `anything`}} Print IF part.
        tIfElse.Execute(os.Stdout, nil)
}
```

Template with if-else

In the code above, at **line 9**, a new template `tEmpty` is made. When it is parsed at **line 12**, the `{{if ``}}` doesn't execute anything (` `` ` is false). So, this template doesn't produce any output when executed.

At **line 14**, a new template `tWithValue` is made. When it is parsed at **line 16**, the `{{if `anything`}}` executes (`` `anything` `` is true). So, this template does produce its output when executed.

At **line 19**, a new template `Else` is made. When it is parsed at **line 21**, the `{{if`

`` `anything`}} `` executes (`` `anything` `` is true). So, this template produces the

output of the `if` branch, not that of the `else` branch. This produces the following output:

```
Non-empty pipeline if demo:  Will print.
if-else demo:  Print IF part.
```

Dot and `with-end`

The dot (.) is used in Go templates. Its value **{{.}}** is set to the current pipeline value. The `with` statement sets the value of dot to the value of the pipeline. If the pipeline is empty, then whatever is between the `with-end` block is skipped; when nested, the dot takes the value according to the closest scope.

This is illustrated in the following program:

```
package main
import (
"os"
"text/template"
)

func main() {
  t := template.New("test")
  t, _ = t.Parse("{{with `hello`}}{{.}}{{end}}!\n")
  t.Execute(os.Stdout, nil)
  t, _ = t.Parse("{{with `hello`}}{{.}} {{with `Mary`}}{{.}}{{end}}{{end}}!\n")
  t.Execute(os.Stdout, nil)
}
```

Template with with-end Construct

In the code above, the template `t` defined at **line 8**, is parsed at **line 9**, and executed at **line 10**. Because of the with `hello`, the current pipeline has a value, namely **hello**, which is printed together with the **!**.

The template `t` is parsed at **line 9**, and executed at **line 11**, resulting in **hello Mary!** printed out.

Template variables `$`

You can create local variables for the pipelines within the template by

You can create local variables for the pipelines within the template by prefixing the variable name with a `$` sign. Variable names can be composed

of alphanumeric characters and the underscore. In the example below, we have used a few variations that work for variable names. It is necessary to use := in the assignment.

```
package main
import (
"os"
"text/template"
)

func main() {
  t := template.New("test")
  t = template.Must(t.Parse("{{with $3 := `hello`}}{{$3}}{{end}}!\n"))
  t.Execute(os.Stdout, nil)
  t = template.Must(t.Parse("{{with $x3 := `hola`}}{{$x3}}{{end}}!\n"))
  t.Execute(os.Stdout, nil)
  t = template.Must(t.Parse("{{with $x_1 := `hey`}}{{$x_1}} {{.}} {{$x_1}}{{end}}!\n"))
  t.Execute(os.Stdout, nil)
}
```

Local Variables

In the code above, **line 8** constructs a template `t`. At **line 9** and at **line 10**, a variable `$3` is displayed with its value given in the `with` clause. At **line 11** and **line 12**, the same is done with a variable `$x3`. At **line 13** and **line 14**, the same is done with a variable `$x_1`. The `.` at **line 13** also takes its value from this variable.

This produces the following output:

```
hello!
hola!
hey hey hey!
```

Range-end

This construct has the format:

```
{{range pipeline}} T1 {{else}} T0 {{end}}
```

The `range` is used for looping over collections, the value of the pipeline must be an array, slice, or map. If the value of the pipeline has a length of zero, dot

is unaffected and `T0` is executed; otherwise, dot is set to the successive elements of the array, slice, or map and `T1` is executed.

If `t` is the template:

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

then, this code:

```
s := []int{1,2,3,4}
t.Execute(os.Stdout, s)
```

will output:

```
1
2
3
4
```

Here is a more concrete example, where data from fields `Author`, `Content` and `Date` are shown:

```
{{range .}}
{{with .Author}}
<p><b>{{html .}}</b> wrote:</p>
{{else}}
<p>An anonymous person wrote:</p>
{{end}}
<pre>{{html .Content}}</pre>
<pre>{{html .Date}}</pre>
{{end}}
```

The `range .` here loops over a slice of structs, each containing an `Author`, `Content` and `Date` field.

Predefined template functions

There are also a few predefined template functions that you can use within your code, e.g., the `printf` function, which works similar to the function `fmt.Sprintf`:

```
package main
import (
        "os"
        "text/template"
)

func main() {
        t := template.New("test")
        t = template.Must(t.Parse("{{with $x := `hello`}}{{printf `%s %s` $x `Mary`}}{{end}}!
        t.Execute(os.Stdout, nil)
}
```

Predefined Functions

In the code above, the template `I`, created at **line 8**, is parsed at **line 9**, and displayed at **line 10**. It prints out **hello Mary!** because, in the format string `%s %s` of the `printf` function. The first `%s` is substituted by `$x`, given in the with clause, and the second `%s` is substituted by the constant string **Mary**.

That is about the details on the `template` package; the next lesson brings you a challenge to solve.