5 Displaying content

This chapter covers

- Templates and template engines
- The Go template libraries text/template and html/template
- Actions, pipelines, and functions in templates
- Nesting of templates and layouts

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. Web template engines are an important part of any web application framework, and most if not all full-fledged frameworks have one. Although a number of frameworks have embedded template engines, many frameworks use a mix-and-match strategy that allows programmers to choose the template engine they prefer.

Go is no exception. Although Go is a relatively new programming language, there are already a few template engines built on it. However the Go's standard library provides strong template support through the text/template and html/template libraries, and unsurprisingly, most Go frameworks support these libraries as the default template engine.

In this chapter we'll focus on these two libraries and show how they can be used to generate HTML responses.

5.1 Templates and template engines

Template engines often combine data with templates to produce the final HTML (see figure 5.1). Handlers usually call template engines to combine data with the templates and return the resultant HTML to the client.

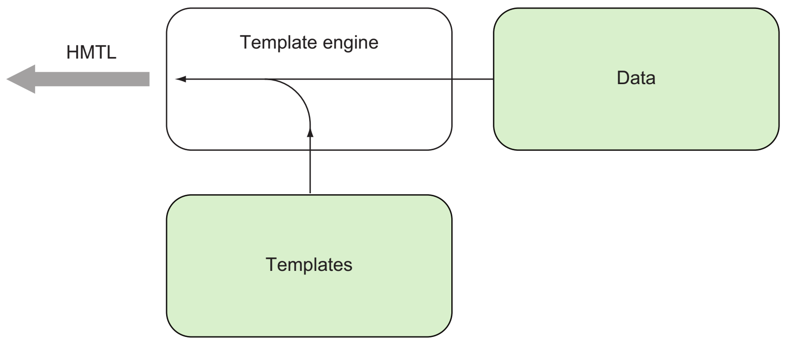


Figure 5.1 Template engines combine data and templates to produce the final displayed HTML.

Web template engines evolved from server-side includes (SSIs), which eventually branched out into web programming languages like PHP, ColdFusion, and JSP. As a result, no standards exist for template engines and the features of existing template engines vary widely, depending on why they were created. But there are roughly two ideal types of template engines, at opposite ends of the spectrum:

- *Logic-less template engines*—Dynamic data is substituted into the templates at specified placeholders. The template engine doesn't do any logic processing; it only does string substitutions. The idea behind having logic-less template engines is to have a clean separation between presentation and logic, where the processing is done by the handlers only.
- *Embedded logic template engines*—Programming language code is embedded into the template and executed during runtime by the template engine, including substitutions. This makes these types of template engines very powerful. Because the logic is embedded in the template itself, though, we often get the logic distributed between the handlers, making the code harder to maintain.

Logic-less template engines are usually faster to render because less processing is involved. Some template engines claim to be logic-less (such as Mustache), but the ideal of substitution-only is impossible to achieve. Mustache claims to be logic-less but has tags that behave like conditionals and loops.

Embedded logic template engines that are completely at the other end of the spectrum are indistinguishable from a computer program itself. We can see this with PHP.

PHP originated as a standalone web template engine, but today many PHP pages are written without a single line of HTML. It's difficult to even say that PHP is still a template engine. In fact, plenty of template engines, like Smarty and Blade, are built for PHP.

The biggest argument against embedded logic template engines is that presentation and logic are mixed up together and logic is distributed in multiple places, resulting in code that's hard to maintain. The counter-argument against logic-less template engines is that the ideal logic-less template engine would be impractical and that placing more logic into the handlers, especially for presentation, would add unnecessary complexity to the handlers.

In reality, most useful template engines would lie somewhere between these two ideal types, with some closer to being logic-less and others closer to having embedded logic. Go's template engine, mostly in text/template and the HTML-specific bits in html/template, is such a hybrid. It can be used as a logic-less template engine, but there's enough embedded features that make Go's template engine an interesting and powerful one.

5.2 The Go template engine

The Go template engine, like most template engines, lies somewhere along the spectrum between logic-less and embedded logic. In a web app, the handler normally triggers the template engine. Figure 5.2 shows how Go's template engine is called from a handler. The handler calls the template engine, passing it the template(s) to be used, usually as a list of template files and the dynamic data. The template engine then generates the HTML and writes it to the `ResponseWriter`, which adds it to the HTTP response sent back to the client.

Go templates are text documents (for web apps, these would normally be HTML files), with certain commands embedded in them, called *actions*. In the Go template

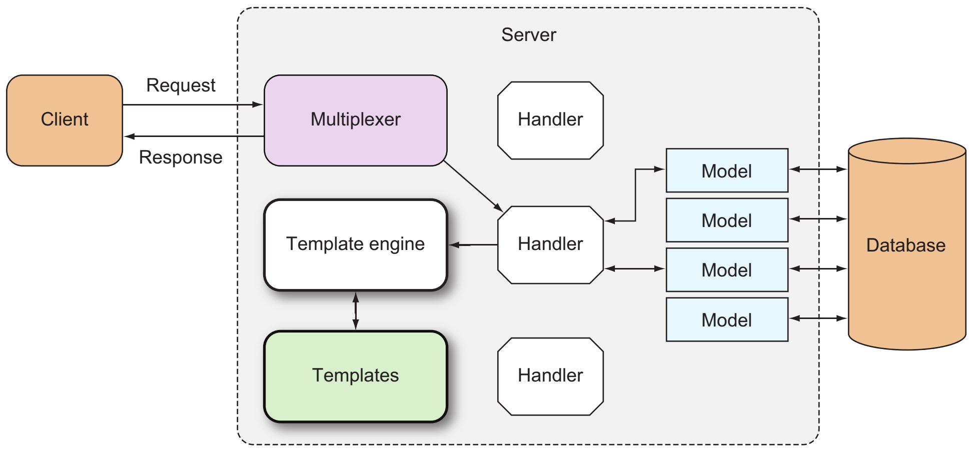


Figure 5.2 The Go template engine as part of a web application

engine, a template is text (usually in a template file) that has embedded actions. The text is parsed and executed by the template engine to produce another piece of text. Go has a text/template standard library that's a generic template engine for any type of text format, as well as an html/template library that's a specific template engine for HTML. Actions are added between two double braces, {{ and }} (although these delimiters can be changed programmatically). We'll get into actions later in this chapter. This listing shows an example of a very simple template.

Listing 5.1 A simple template

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    {{ . }}
  </body>
</html>
```

This template is placed in a template file named tmpl.html. Naturally we can have as many template files as we like. Template files must be in a readable text format but can have any extension. In this case because it generates HTML output, I used the .html extension.

Notice the dot (.) between the double braces. The dot is an action, and it's a command for the template engine to replace it with a value when the template is executed.

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. This triggers the template engine to combine the parsed template with the data to generate the final HTML that's passed to the `ResponseWriter`.

This listing provides a concrete, simple example.

Listing 5.2 Triggering a template engine from a handler function

```
package main

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

func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("tmpl.html")
    t.Execute(w, "Hello World!")
}
```

```
func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/process", process)
    server.ListenAndServe()
}
```

We're back to our server again. This time we have a handler function named `process`, which triggers the template engine. First, we parse the template file tmpl.html using the `ParseFiles` function which returns a parsed template of type `Template` and an error, which we conveniently ignore for brevity.

```
t, _ := template.ParseFiles("tmpl.html")
```

Then we call the `Execute` method to apply data (in this case, the string `Hello World!`) to the template:

```
t.Execute(w, "Hello World!")
```

We pass in the `ResponseWriter` along with the data so that the generated HTML can be passed into it. When you're running this example, the template file should be in the same directory as the binary (remember that we didn't specify the absolute path to the file).

This is the simplest way to use the template engine, and as expected, there are variations, which I'll describe later in the chapter.

5.2.1 Parsing templates

`ParseFiles` is a standalone function that parses template files and creates a parsed template struct that you can execute later. The `ParseFiles` function is actually a convenience function to the `ParseFiles` method on the `Template` struct. When you call the `ParseFiles` function, Go creates a new template, with the name of the file as the name of the template:

```
t, _ := template.ParseFiles("tmpl.html")
```

Then it calls the `ParseFiles` method on that template:

```
t := template.New("tmpl.html")
t, _ := t.ParseFiles("tmpl.html")
```

`ParseFiles` (both the function and the method) can take in one or more filenames as parameters (making it a *variadic* function—that is, a function that can take in a variable number of parameters). But it still returns just one template, regardless of the number of files it's passed. What's up with that?

When we pass in more than one file, the returned parsed template has the name and content of the first file. The rest of the files are parsed as a map of templates, which can be referred to later on during the execution. You can think of this as `ParseFiles` returning a template when you provide a single file and a template set when you provide more than one file. This fact is important when we look at including a template within a template, or nesting templates, later in this chapter.

Another way to parse files is to use the `ParseGlob` function, which uses pattern matching instead of specific files. Using the same example:

```
t, _ := template.ParseFiles("tmpl.html")
```

and

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

would be the same, if tmpl.html were the only file in the same path.

Parsing files is probably the most common use, but you can also parse templates using strings. In fact, all other ways of parsing templates ultimately call the `Parse` method to parse the template. Using the same example again:

```
t, _ := template.ParseFiles("tmpl.html")
```

and

```
tmpl := `<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    {{ . }}
  </body>
</html>
`
t := template.New("tmpl.html")
t, _ = t.Parse(tmpl)
t.Execute(w, "Hello World!")
```

are the same, assuming tmpl.html contains the same HTML.

So far we've been ignoring the error that's returned along with the parsed template. The usual Go practice is to handle the error, but Go provides another mechanism to handle errors returned by parsing templates:

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

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`. (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.)

5.2.2 Executing templates

The usual way to execute a template is to call the `Execute` function on a template, passing it the `ResponseWriter` and the data. This works well when the parsed template is a single template instead of a template set. If you call the `Execute` method on a template set, it'll take the first template in the set. But if you want to execute a different template in the template set and not the first one, you need to use the `ExecuteTemplate` method. For example:

```
t, _ := template.ParseFiles("t1.html", "t2.html")
```

The argument `t` is a template set containing two templates, the first named t1.html and the second t2.html (the name of the template is the name of the file, extension and all, unless you change it). If you call the `Execute` method on it:

```
t.Execute(w, "Hello World!")
```

it'll result in t1.html being executed. If you want to execute t2.html, you need to do this instead:

```
t.ExecuteTemplate(w, "t2.html", "Hello World!")
```

We've discussed how to trigger the template engine to parse and execute templates. Let's look at the templates next.

5.3 Actions

As mentioned earlier, *actions* are embedded commands in Go templates, placed between a set of double braces, {{ and }}. Go has an extensive set of actions, which are quite powerful and flexible. In this section, we'll discuss some of the important ones:

- Conditional actions
- Iterator actions
- Set actions
- Include actions

We'll discuss another important action, the define action, later in this chapter. You can look up the other actions in the text/template library documentation.

It might come as a surprise, but the dot (.) is an action, and it's the most important one. The dot is the evaluation of the data that's passed to the template. The other actions and functions mostly manipulate the dot for formatting and display.

5.3.1 Conditional actions

The conditional actions are ones that select one of many data evaluations depending on value of the argument. The simplest action is one with this format:

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

The other variant is

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

The arg in both formats is the argument to the action. We'll examine arguments in detail later in this chapter. For now, consider arguments to be values like a string constant, a variable, a function, or a method that returns a value. Let's see how it can be used in a template. First, you need to create a handler that generates a random integer between 0 and 10, shown in the next listing. Then you check if it's larger than 5 and return it as a Boolean to the template.

Listing 5.3 Generating a random number in the handler

```
package main

import (
    "net/http"
    "html/template"
    "math/rand"
    "time"
)

func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("tmpl.html")
    rand.Seed(time.Now().Unix())
    t.Execute(w, rand.Intn(10) > 5)
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/process", process)
    server.ListenAndServe()
}
```

Next, in the template file tmpl.html, you test the argument (which happens to be the dot, which is the value passed from the handler) and display either *Number is greater than 5!* or *Number is 5 or less!*

Listing 5.4 Template file tmpl.html for conditional action

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    {{ if . }}
      Number is greater than 5!
    {{ else }}
      Number is 5 or less!
    {{ end }}
  </body>
</html>
```

5.3.2 Iterator actions

Iterator actions are those that iterate through an array, slice, map, or channel. Within the iteration loop, the dot (.) is set to the successive elements of the array, slice, map, or channel. This is how it looks:

```
{{ range array }}
  Dot is set to the element {{ . }}
{{ end }}
```

The example in this listing shows the template file tmpl.html.

Listing 5.5 Iterator action

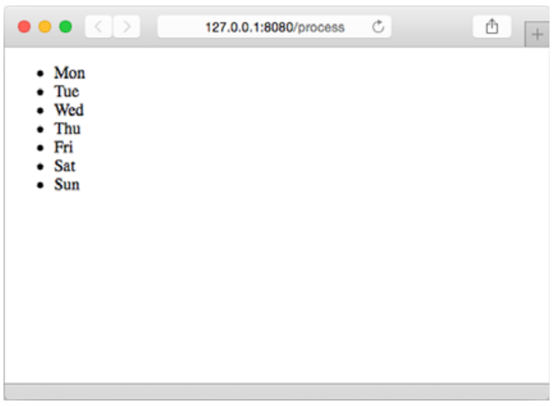
```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <ul>
    {{ range . }}
      <li>{{ . }}</li>
    {{ end}}
    </ul>
  </body>
</html>
```

Let's look at the triggering handler:

```
func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("tmpl.html")
    daysOfWeek := []string{"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"}
    t.Execute(w, daysOfWeek)
}
```

You're simply passing a slice of strings with the short names of the days in a week. This slice is then passed on to the dot (.) in the `{{ range . }}`, which loops through the elements of this slice.

The `{{ . }}` within the iterator loop is an element in the slice, so figure 5.3 shows what you'll see in the browser.



- Mon
- Tue
- Wed
- Thu
- Fri
- Sat
- Sun

Figure 5.3 Iterating with the iterator action

The following listing shows a variant of the iterator action that allows you to display a fallback in case the iterator is empty.

Listing 5.6 Iterator action with fallback

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <ul>
    {{ range . }}
      <li>{{ . }}</li>
    {{ else }}
      <li> Nothing to show </li>
    {{ end}}
    </ul>
  </body>
</html>
```

In the listing the content after `{{ else }}` and before `{{ end }}` will be displayed if the dot (.) is `nil`. In this case, we're displaying the text *Nothing to show*.

5.3.3 Set actions

The set actions allow us to set the value of dot (.) for use within the enclosed section. This is how it looks:

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

The dot (.) between `{{ with arg }}` and `{{ end }}` is now pointing to `arg`. In the next listing, let's look at something more concrete and make changes to tmpl.html again.

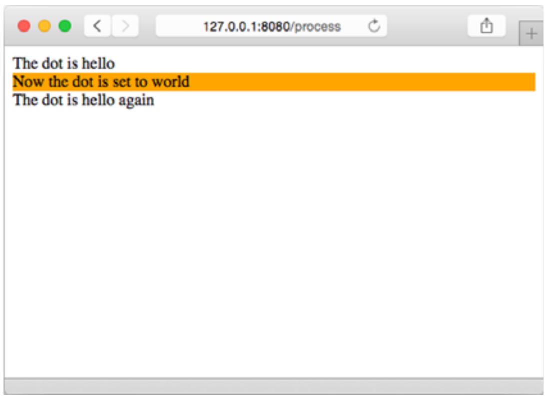
Listing 5.7 Setting the dot

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <div>The dot is {{ . }}</div>
    <div>
    {{ with "world"}}
      Now the dot is set to {{ . }}
    {{ end }}
    </div>
    <div>The dot is {{ . }} again</div>
  </body>
</html>
```

For the handler, we'll pass a string `hello` to the template:

```
func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("tmpl.html")
    t.Execute(w,"hello")
}
```

The dot before `{{ with "world"}}` is set to `hello`. The value is from the handler, but between `{{ with "world"}}` and `{{ end }}` it's set to `world`. After `{{ end }}` it'll revert to `hello` again, as shown in figure 5.4.



```
The dot is hello
Now the dot is set to world
The dot is hello again
```

Figure 5.4 Setting the dot with the set action

As with the iterator action, there's a variant of the set action that allows a fallback:

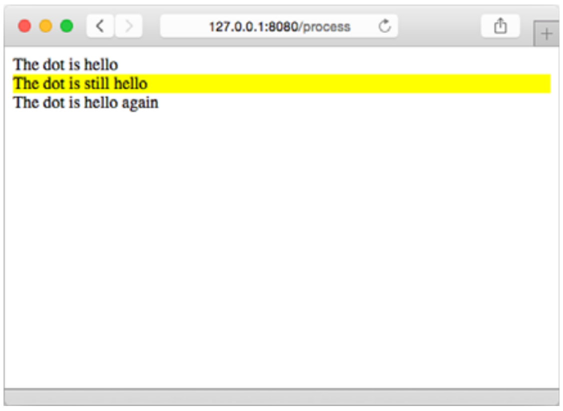
```
{{ with arg }}
  Dot is set to arg
{{ else }}
  Fallback if arg is empty
{{ end }}
```

This listing takes another look at how this is used in tmpl.html.

Listing 5.8 Setting the dot with fallback

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <div>The dot is {{ . }}</div>
    <div>
    {{ with "" }}
      Now the dot is set to {{ . }}
    {{ else }}
      The dot is still {{ . }}
    {{ end }}
    </div>
    <div>The dot is {{ . }} again</div>
  </body>
</html>
```

The argument next to the `with` action is an empty string, which means the content after `{{ else }}` will be displayed. The dot in the content is still `hello` because it's not affected. If you run the server again (you don't need to make any changes to the handler—in fact, you don't even need to stop the server), you'll see figure 5.5.



```
The dot is hello
The dot is still hello
The dot is hello again
```

Figure 5.5 Setting the dot with a fallback

5.3.4 *Include actions*

Next, we have the include actions, which allow us to include a template in another template. As you might realize, these actions let us nest templates. The format of the include action is `{{ template "name" }}`, where `name` is the name of the template to be included.

In our example, we have two templates: t1.html and t2.html. The template t1.html includes t2.html. Let's look at t1.html first.

Listing 5.9 t1.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=9">
    <title>Go Web Programming</title>
  </head>
  <body>
    <div> This is t1.html before</div>
    <div>This is the value of the dot in t1.html - [{{ . }}]</div>
    <hr/>
    {{ template "t2.html" }}
    <hr/>
    <div> This is t1.html after</div>
  </body>
</html>
```

As you can see, the name of the file is used as the name of the template. Remember that if we don't set the name of the template when it's created, Go will set the name of the file, including its extension, as the name of the template.

Let's look at t2.html now.

Listing 5.10 t2.html

```
<div style="background-color: yellow;">
  This is t2.html<br/>
  This is the value of the dot in t2.html - [{{ . }}]
</div>
```

The template t2.html is a snippet of HTML. The next listing shows the handler, which is just as short.

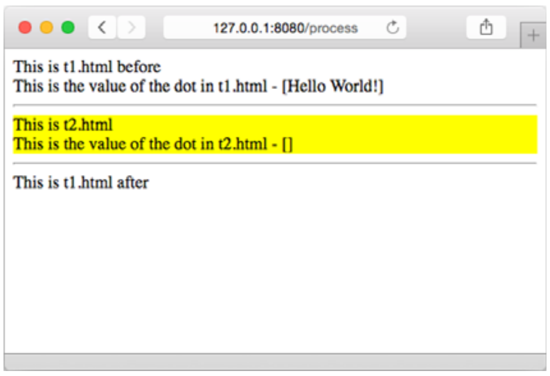
Listing 5.11 Handler that includes templates

```
func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("t1.html", "t2.html")
    t.Execute(w, "Hello World!")
}
```

Unlike with the previous handlers, we need to parse both the template files that we're going to use. This is extremely important because if we forget to parse our template files, we'll get nothing.

Because we didn't set the name of the templates, the templates in the template set use the name of the files. As mentioned earlier, the parameters for the `ParseFiles` function are position-sensitive for the first parameter. The first template file that's parsed is the main template, and when we call the `Execute` method on the template set, this is the template that'll be called.

Figure 5.6 shows what happens in the browser when you run the server.



```
This is t1.html before
This is the value of the dot in t1.html - [Hello World!]

This is t2.html
This is the value of the dot in t2.html - []

This is t1.html after
```

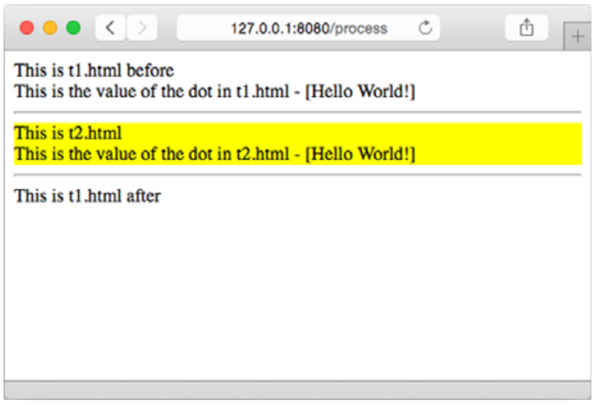
Figure 5.6 Including a template within a template

You can see that in t1.html the dot is replaced correctly with `Hello World!` And the contents of t2.html are inserted where `{{ template "t2.html" }}` appears. You can also see that the dot in t2.html is empty because the string `Hello World!` isn't passed to included templates. But there's a variant of the include action that allows this: `{{ template "name" arg }}`, where `arg` is the argument you want to pass on to the included template. This listing shows how this works in our example.

Listing 5.12 t1.html with an argument passed to t2.html

```
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=9">
    <title>Go Web Programming</title>
  </head>
  <body>
    <div> This is t1.html before</div>
    <div>This is the value of the dot in t1.html - [{{ . }}]</div>
    <hr/>
    {{ template "t2.html" . }}
    <hr/>
    <div> This is t1.html after</div>
  </body>
</html>
```

The change isn't obvious, but what we did here in t1.html is pass the dot to t2.html. If we go back to the browser now, we'll see figure 5.7.



```
This is t1.html before
This is the value of the dot in t1.html - [Hello World!]

This is t2.html
This is the value of the dot in t2.html - [Hello World!]

This is t1.html after
```

Figure 5.7 Passing data to included templates

Actions provide flexibility for programmers, but so far we've only seen the simpler and more straightforward use of templates. We'll get back into nested templates later in this chapter, and also talk about another action that we didn't cover in this section: the define action. But first, we need to talk about arguments, variables, and pipelines.

5.4 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 also be a struct, or a field of a struct, or the key of an array. Arguments can be a variable, a method (which must return either one value, or a value and an error) or a function. An argument can also be a dot (.), which is the value passed from the template engine.

In the example

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

the argument is `arg`.

We can also set variables in the action. Variables start with the dollar sign (`$`) and look like this:

```
$variable := value
```

Variables don't look too useful at first glance, but they can be quite invaluable in actions. Here's one way to use a variable as a variant of the iterator action:

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

In this snippet, the dot (.) is a map and `range` initializes the `$key` and `$value` variables with the key and value of the successive elements in the map.

Pipelines are arguments, functions, and methods chained together in a sequence. This works much like the Unix pipeline. In fact, the syntax looks very similar

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

where `p1`, `p2`, and `p3` are either arguments or functions. Pipelining allows us to send the output of an argument into the next one, which is separated by the pipeline (|). The next listing shows how this looks in a template.

Listing 5.13 Pipelining in a template

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    {{ 12.3456 | printf "%.2f" }}
  </body>
</html>
```

We often need to reformat data for display in our templates, and in the previous listing we want to display a floating point number with two decimal points accuracy. To do so, we can use the `fmt.Sprintf` function or the built-in `printf` function that wraps around `fmt.Sprintf`.

In listing 5.13, we pipe the number 12.3456 to the `printf` function, with the format specifier as the first parameter (we'll talk about functions later in this chapter). This will result in *12.35* being displayed in the browser.

Pipelining doesn't look terribly exciting, but when we talk about functions you'll understand why it can be a very powerful feature.

5.5 Functions

As mentioned earlier, an argument can be a Go function. The Go template engine has a set of built-in functions that are pretty basic, including a number of aliases for variants of `fmt.Sprint`.(Refer to the `fmt` package documentation to see the list.) What's more useful is the capability for programmers to define their own functions.

Go template engine functions are limited. 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.

To define custom functions, you need to:

1. Create a `FuncMap` map, which has the name of the function as the key and the actual function as the value.
2. Attach the `FuncMap` to the template.

Let's look at how you can create your own custom function. When writing web applications, you may need to convert a time or date object to an ISO8601 formatted time or date string, respectively. Unfortunately, this formatter isn't a built-in function, which gives you a nice excuse for creating it as a custom function, shown next.

Listing 5.14 Custom function for templates

```
package main

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

func formatDate(t time.Time) string {
    layout := "2006-01-02"
    return t.Format(layout)
}

func process(w http.ResponseWriter, r *http.Request) {
    funcMap := template.FuncMap { "fdate": formatDate }
    t := template.New("tmpl.html").Funcs(funcMap)
    t, _ = t.ParseFiles("tmpl.html")
    t.Execute(w, time.Now())
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/process", process)
    server.ListenAndServe()
}
```

The previous listing defines a function named `formatDate` that takes in a `Time` struct and returns an ISO8601 formatted string in the Year-Month-Day format. This is the function that we'll be using later.

In the handler, you first create a `FuncMap` struct named `funcMap`, mapping the name `fdate` to the `formatDate` function. Next, you create a template with the name tmpl.html using the `template.New` function. This function returns a template, so you can chain it with the `Funcs` function, passing it `funcMap`. This attaches `funcMap` to the template, which you can then use to parse your template file tmpl.html. Finally, you call `Execute` on the template, passing it the `ResponseWriter` as well as the current time.

There are potentially a few minor gotchas here. First you need to attach the `FuncMap` before you parse the template. This is because when you parse the template you must already know the functions used within the template, which you won't unless the template has the `FuncMap`.

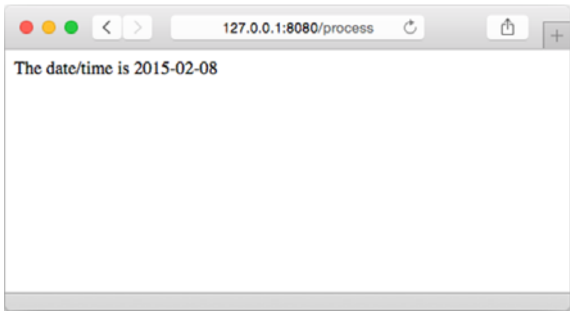
Second, remember that when you call `ParseFiles` if no templates are defined in the template files, it'll take the name of the file as the template name. When you create a new template using the `New` function, you'll have to pass in the template name. If this template name and the template name derived from the filename don't match, you'll get an error.

Now that you have the handler done, the following listing shows you how you can use it in tmpl.html.

Listing 5.15 Using a custom function by pipelining

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <div>The date/time is {{ . | fdate }}</div>
  </body>
</html>
```

You can use your function in a couple of ways. You can use it in a pipeline, piping the current time into the `fdate` function, which will produce figure 5.8.



The date/time is 2015-02-08

Figure 5.8 Using a custom function to format the date/time

An alternative is to use it as with a normal function, passing the dot as a parameter to the `fdate` function, as illustrated in this listing.

Listing 5.16 Using a custom function by passing parameters

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <div>The date/time is {{ fdate . }}</div>
  </body>
</html>
```

Both produce the same results, but you can see that pipelining is more powerful and flexible. If you define a number of custom functions, you can pipeline the output of one function to the input of another, as well as mix and match them. Although you can also do this with normal function calls, this approach is a lot more readable and creates simpler code.

5.6 Context awareness

One of the most interesting features of the Go template engine is that the content it displays can be changed according to its context. Yes, you read this correctly. The content that's displayed changes depending on where you place it within the document; that is, the display of the content is *context-aware*. Why would anyone want that and how is it useful?

One obvious use of this is to escape the displayed content properly. This means if the content is HTML, it will be HTML escaped; if it is JavaScript, it will be JavaScript escaped; and so on. The Go template engine also recognizes content that's part of a URL or is a CSS style. This listing demonstrates how this works.

Listing 5.17 Handler for context awareness in templates

```
package main

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

func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("tmpl.html")
    content := `I asked: <i>"What's up?"</i>`
    t.Execute(w, content)
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/process", process)
    server.ListenAndServe()
}
```

For the handler, we're going to send a text string `I asked: <i>"What's up?"</i>`. This string has a number of special characters that normally should be escaped beforehand. This listing contains the template file tmpl.html.

Listing 5.18 Context-aware template

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</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>
```

As you can see, we're placing it in various places within the HTML. The "control" text is within a normal `<div>` tag. When we run cURL to get the raw HTML (please refer to section 4.1.4), we get this:

```
curl -i 127.0.0.1:8080/process
HTTP/1.1 200 OK
Date: Sat, 07 Feb 2015 05:42:41 GMT
```

```
Content-Length: 505
Content-Type: text/html; charset=utf-8

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <div>I asked: &lt;i&gt;&#34;What&#39;s up?&#34;&lt;/i&gt;</div>
    <div>
      <a href="/I%20asked:%20%3ci%3e%22What%27s%20up?%22%3c/i%3e">
        Path
      </a>
    </div>
    <div>
      <a href="/?q=I%20asked%3a%20%3ci%3e%22What%27s%20up%3f%22%3c%2fi%3e">
        Query
      </a>
    </div>
    <div>
      <a onclick="f('I asked: \x3ci\x3e\x22What\x27s up?\x22\x3c\/i\x3e')">
        Onclick
         </a>
    </div>
  </body>
</html>
```

which looks a bit messy. Let's explore the differences in table 5.1.

Table 5.1 Context awareness in Go templates: different content is produced according to the location of the actions

Context	Content
Original text	`I asked: <i>"What's up?"</i>`
`{{ . }}`	`I asked: &lt;i&gt;&#34;What&#39;s up?&#34;&lt;/i&gt;`
`<a href="/{{ . }}">`	`I%20asked:%20%3ci%3e%22What%27s%20up?%22%3c/i%3e`
`<a href="/?q={{ . }}">`	`I%20asked%3a%20%3ci%3e%22What%27s%20up%3f%22%3c%2fi%3e`
`<a onclick="{{ . }}">`	`I asked: \x3ci\x3e\x22What\x27s up?\x22\x3c\/i\x3e`

This feature is pretty convenient, but its critical use is for automating defensive programming. By changing the content according to the context, we eliminate certain obvious and newbie programmer mistakes. Let's see how this feature can be used to defend against XSS (cross-site scripting) attacks.

5.6.1 Defending against XSS attacks

A common XSS attack is the persistent XSS vulnerability. This happens when data provided by an attacker is saved to the server and then displayed to other users as it is. Say there's a vulnerable forum site that allows its users to create posts or comments to be saved and read by other users. An attacker can post a comment that includes malicious JavaScript code within the `<script>` tag. Because the forum displays the comment as is and whatever is within the `<script>` tag isn't shown to the user, the malicious code is executed with the user's permissions but without the user's knowledge. The normal way to prevent this is to escape whatever is passed into the system before displaying or storing it. But as with most exploits and bugs, the biggest culprit is the human factor.

Rather than hardcoding data at the handler, you can put your newly acquired knowledge from chapter 4 to good use and create an HTML form, shown in the following listing, that allows you to submit data to your web application and place it in a form.html file.

Listing 5.19 Form for submitting XSS attack

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <form action="/process" method="post">
      Comment: <input name="comment" type="text">
     <hr/>
     <button id="submit">Submit</button>
    </form>
  </body>
</html>
```

Next, change your handler accordingly to process the data from the form shown in this listing.

Listing 5.20 Testing an XSS attack

```
package main

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

func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("tmpl.html")
    t.Execute(w, r.FormValue("comment"))
}
```

```
func form(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("form.html")
    t.Execute(w, nil)
}

func main() {
    server := http.Server{
        Addr: "127.0.0.1:8080",
    }
    http.HandleFunc("/process", process)
    http.HandleFunc("/form", form)
    server.ListenAndServe()
}
```

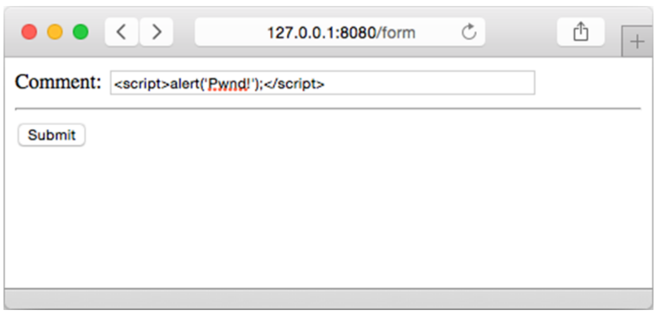
In your tmpl.html, clean up the output a bit to better see the results.

Listing 5.21 Cleaned-up tmpl.html

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <div>{{ . }}</div>
  </body>
</html>
```

Now compile the server and start it up, then go to http://127.0.0.1:8080/form. Enter the following into the text field and click the submit button, shown in figure 5.9:

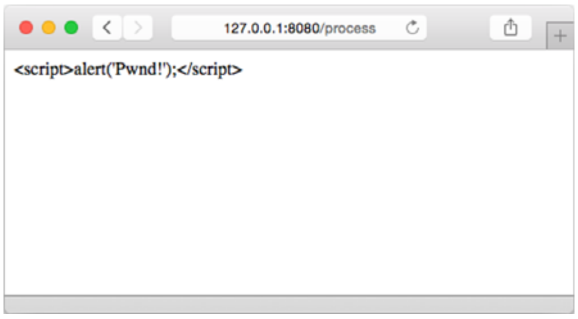
```
<script>alert('Pwnd!');</script>
```



Comment: <script>alert('Pwnd!');</script>

Figure 5.9 Form for creating an XSS attack

A web app using a different template engine that doesn't scrub the input, and displays user input directly on a web page, will get an alert message, or potentially any other malicious code that the attacker writes. As you probably realize, the Go template engine protects you from such mistakes because even if you don't scrub the input, when the input is displayed on the screen it'll be converted into escaped HTML, shown in figure 5.10.



```
<script>alert('Pwnd!');</script>
```

Figure 5.10 The input is escaped, thanks to the Go template engine.

If you inspect the source code of the page you'll see something like this:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    <div>&lt;script&gt;alert(&#39;Pwnd!&#39;);&lt;/script&gt;</div>
  </body>
</html>
```

Context-awareness isn't just for HTML, it also works on XSS attacks on JavaScript, CSS, and even URLs. Does this mean we're saved from ourselves if we use the Go template engine? Well, no, nothing really saves us from ourselves, and there are ways of getting around this. In fact, Go allows us to escape from being context-aware if we really want to.

5.6.2 Unescaping HTML

Say you really want this behavior, meaning you want the user to enter HTML or JavaScript code that's executable when displayed. Go provides a mechanism to "unescape" HTML. Just cast your input string to template.HTML and use that instead, and our code is happily unescaped. Let's see how to do this. First, make a minor change to the handler:

```
func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("tmpl.html")
    t.Execute(w, template.HTML(r.FormValue("comment")))
}
```

Notice that you've just typecast the comment value to the template.HTML type.

Now recompile and run the server, and then try the same attack. What happens depends on which browser you use. If you use Internet Explorer (8 and above), Chrome, or Safari, nothing will happen—you'll get a blank page. If you use Firefox, you'll get something like figure 5.11.

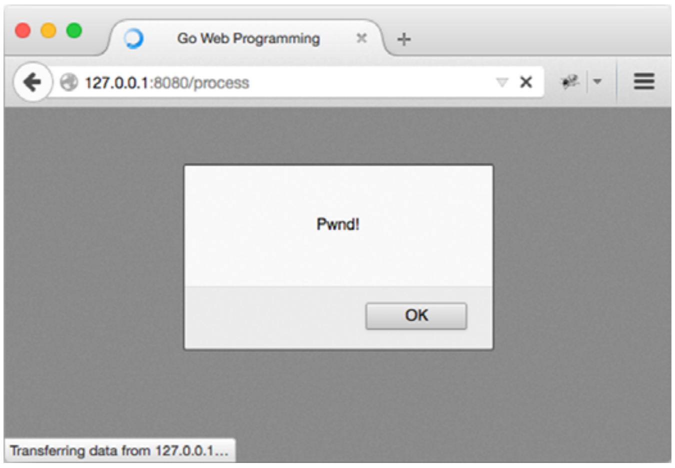


Figure 5.11 You are pwnd!

What just happened? By default Internet Explorer, Chrome, and Safari have built-in support for protection against certain types of XSS attacks. As a result, the simulated XSS attack doesn't work on these browsers. But you can turn off the protection by sending a special HTTP response header: *X-XSS-Protection* (originally created by Microsoft for Internet Explorer) will disable it. (Note: Firefox doesn't have default protection against these types of attacks.)

If you really want to stop the browser from protecting you from XSS attacks, you simply need to set a response header in our handler:

```
func process(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("X-XSS-Protection", "0")
    t, _ := template.ParseFiles("tmpl.html")
    t.Execute(w, template.HTML(r.FormValue("comment")))
}
```

Once you do this and try the same attack again, you'll find that the attack works for Internet Explorer, Chrome, and Safari.

5.7 Nesting templates

We went through quite a lot of features of the Go template engine in this chapter. Before we move on, I'd like to show you how you can use layouts in your web app.

So what exactly are layouts? *Layouts* are fixed patterns in web page design that can be reused for multiple pages. Web apps often use layouts, because pages in a web app need to look the same for a consistent UI. For example, many web application designs have a header menu as well as a footer that provides additional information such as status or copyright or contact details. Other layouts include a left navigation bar or multilevel navigation menus. You can easily make the leap that layouts can be implemented with nested templates.

In an earlier section, you learned that templates can be nested using the include action. If you start writing a complicated web app, though, you'll realize that you might end up with a lot of hardcoding in your handler and a lot of template files.

Why is this so?

Remember that the syntax of the include action looks like this:

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

where `name` is the name of the template and a string constant. This means that if you use the name of the file as the template name, it'll be impossible to have one or two common layouts, because every page will have its own layout template file—which defeats the purpose of having layouts in the first place. As an example, the file shown in the next listing won't work as a common layout template file.

Listing 5.22 An unworkable layout file

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    {{ template "content.html" }}
  </body>
</html>
```

The answer to this dilemma is that the Go template engine doesn't work this way. Although we can have each template file define a single template, with the name of the template as the name of the file, we can explicitly define a template in a template file using the define action. This listing shows our layout.html now.

Listing 5.23 Defining a template explicitly

```
{{ define "layout" }}

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    {{ template "content" }}
  </body>
</html>

{{ end }}
```

Notice that we start the file with `{{ define "layout" }}` and end it with `{{ end }}`. Anything within these two action tags is considered part of the layout template. This means if we have another define action tag after the `{{ end }}` we can define another template! In other words, we can define multiple templates in the same template file, as you can see in this listing.

Listing 5.24 Defining multiple templates in a single template file

```
{{ define "layout" }}

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    {{ template "content" }}
  </body>
</html>

{{ end }}

{{ define "content" }}

Hello World!

{{ end }}
```

The following listing shows how we use these templates in our handler.

Listing 5.25 Using explicitly defined templates

```
func process(w http.ResponseWriter, r *http.Request) {
    t, _ := template.ParseFiles("layout.html")
    t.ExecuteTemplate(w, "layout", "")
}
```

Parsing the template file is the same, but this time if we want to execute the template, we have to be more explicit and use the `ExecuteTemplate` method, with the name of the template we want to execute as the second parameter. The layout template nests the content template, so if we execute the layout template, we'll see `Hello World!` in the browser. Let's use cURL to get the actual HTML so that we can see it properly:

```
> curl -i http://127.0.0.1:8080/process
HTTP/1.1 200 OK
Date: Sun, 08 Feb 2015 14:09:15 GMT
Content-Length: 187
Content-Type: text/html; charset=utf-8

<html>
  <head>
```

```
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>

Hello World!

  </body>
</html>
```

We can also define the same template in the multiple template files. To see how this works, let's remove the definition of the content template in layout.html and place it in red_hello.html, as shown in this listing. Listing 5.27 shows how to create a blue_hello.html template file.

Listing 5.26 red_hello.html

```
{{ define "content" }}

<h1 style="color: red;">Hello World!</h1>

{{ end }}
```

Listing 5.27 blue_hello.html

```
{{ define "content" }}

<h1 style="color: blue;">Hello World!</h1>

{{ end }}
```

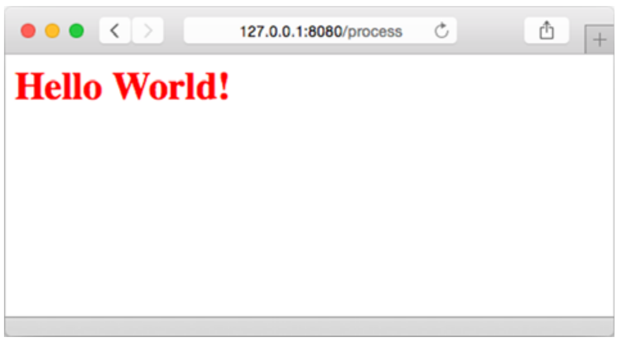
Notice that we've just defined the content template in two places. How can we use these two templates? This listing shows our modified handler.

Listing 5.28 A handler using the same template in different template files

```
func process(w http.ResponseWriter, r *http.Request) {
    rand.Seed(time.Now().Unix())
    var t *template.Template
    if rand.Intn(10) > 5 {
        t, _ = template.ParseFiles("layout.html", "red_hello.html")
    } else {
        t, _ = template.ParseFiles("layout.html", "blue_hello.html")
    }
    t.ExecuteTemplate(w, "layout", "")
}
```

Note that we're actually parsing different template files (either red_hello.html or blue_hello.html) according to the random number we create. We use the same layout template as before, which includes a *content* template. Remember that the content template is defined in two different files. Which template we use depends now on which template file we parse, because both of these template files define the same template. In other words, we can switch content by parsing different template files, while maintaining the same template to be nested in the layout.

If we now recompile our server, start it, and access it through the browser, we'll randomly see either a blue or red `Hello World!` showing up in the browser (see figure 5.12).



Hello World!

Figure 5.12 Switching templates

5.8 Using the block action to define default templates

Go 1.6 introduced a new block action that allows you to define a template and use it at the same time. This is how it looks:

```
{{ block arg }}
  Dot is set to arg
{{ end }}
```

To see how this works, I'll use the previous example and use the block action to replicate the same results. What I'll do is default to using the blue Hello World template if no templates are specified. Instead of parsing the layout.html and blue_hello.html files in the else block, as in listing 5.28, I will parse layout.html only as indicated in bold in the following listing.

Listing 5.29 Parsing layout.html only

```
func process(w http.ResponseWriter, r *http.Request) {
    rand.Seed(time.Now().Unix())
    var t *template.Template
    if rand.Intn(10) > 5 {
        t, _ = template.ParseFiles("layout.html", "red_hello.html")
    } else {
        t, _ = template.ParseFiles("layout.html")
    }
    t.ExecuteTemplate(w, "layout", "")
}
```

Without any further changes, this will result in a crash at random, because the template in the else block doesn't have a content template. Instead of passing it externally, I will use a block action and add it as a default content in layout.html itself, as in the code in bold in this listing.

Listing 5.30 Using the block action to add a default content template

```
{{ define "layout" }}

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Web Programming</title>
  </head>
  <body>
    {{ block "content" . }}
      <h1 style="color: blue;">Hello World!</h1>
    {{ end }}
  </body>
</html>

{{ end }}
```

The block action effectively defines a template named content and also places it in the layout. If no content template is available when the overall template is executed, the content template defined by the block will be used instead.

We're done with handling requests, processing them, and generating content to respond to the requests. In the next chapter, you'll learn how you can store data in memory, in files, and in databases using Go.

5.9 Summary

- In a web app, template engines combine templates and data to produce the HTML that is sent back to the client.
- Go's standard template engine is in the `html/template` package.
- Go's template engine works by parsing a template and then executing it, passing a `ResponseWriter` and some data to it. This triggers the template engine to combine the parsed template with the data and send it to the `ResponseWriter`.
- Actions are instructions about how data is to be combined with the template. Go has an extensive and powerful set of actions.
- Besides actions, templates can also contain arguments, variables, and pipelines. Arguments represent the data value in a template; variables are constructs used with actions in a template. Pipelines allow chaining of arguments and functions.
- Go has a default but limited set of template functions. Customized functions can also be created by making a function map and attaching it to the template.
- Go's template engine can change the content it displays according to where the data is placed. This context-awareness is useful in defending against XSS attacks.
- Web layouts are commonly used to design a web app that has a consistent look and feel. This can be implemented in Go using nested templates.