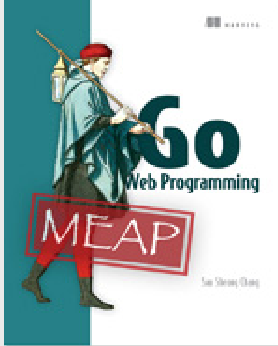

Exploring the Go template engine

By Sau Sheong Chang

Go's template engine is like a hybrid between a logic-less template engine and an embedded logic template engine. In this article, we'll talk about why Go's template engine is an interesting and powerful one.

The Go template engine, like most template engines, lies somewhere along the spectrum between logic-less and embedded logic. In a web application, the handler normally triggers the template engine. The following diagram 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.

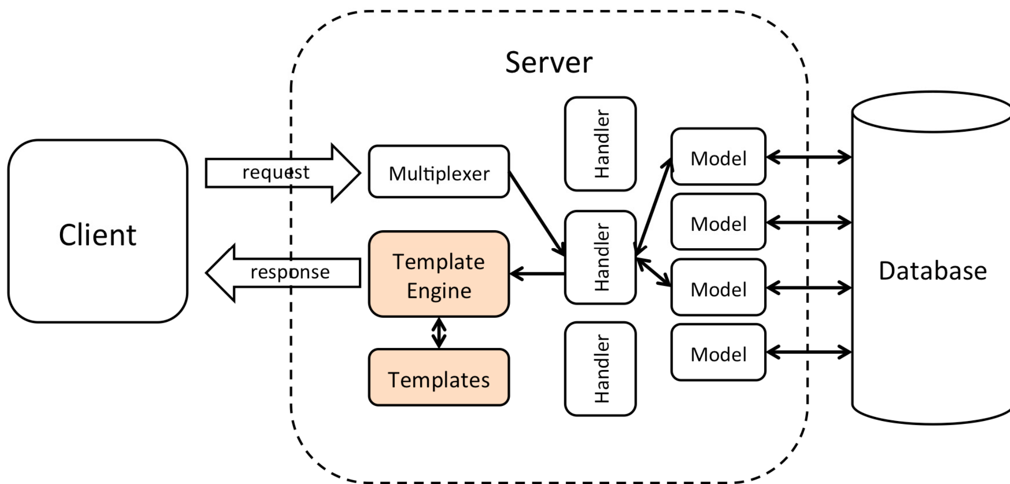

Figure 1 – Go template engine as part of a web application

Go templates are text documents (for web applications, these would normally be HTML files), with certain commands embedded in them, called *actions*. In the Go template engine, a template is text (usually in a template file) that has embedded actions that are parsed and executed by the template engine to produce another piece of text. Go has a text/template standard library that is a generic template engine for any type of text format, and an html/template library that is a specific template engine for HTML. Actions are added between two double braces ({{) and (}}). Here's an example of what a very simple template looks like:

Listing 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, but in this case since it generates HTML output, I used the `.html` extension.

Notice the dot (.) between the double braces. The dot is an action, and is 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; this 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 is passed to the *ResponseWriter*

Let's look at a concrete, simple example:

Listing 2 – Triggering template engine from 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. The function 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 HTML that will be generated can be passed back to the client. When you are 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. I'll describe those now.

Parsing templates

ParseFiles is a standalone function that parses template files and creates a parsed template struct that you can execute later. However, the *ParseFiles* function is actually a convenience function to the *ParseFiles* method on the *Template* struct. Basically what happens is that when you call *ParseFiles*, Go creates a new template, with the name of the file as the name of the template, then calls *ParseFiles* on that template.

In other words, these two:

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

and

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

ParseFiles can take in one or more file names as parameters (it is a *variadic* function; that is, a function that can take in a variable number of parameters). However, it still returns only one template, regardless of the number of file 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 it as *ParseFiles* returning a template when you provide a single file, and a template set when you provide more than one file.

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` was the only file in the same path.

Parsing files is probably the most common, 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 also the same, assuming `tmpl.html` contains the same HTML.

So far we've been ignoring the error that is returned along with the parsed template. The usual Go practice is, of course, 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.

Executing templates

The usual way to execute a template is to call the *Execute* function on a template, passing it *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 simply take the first template in the set. However 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, take this code:

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

This contains two templates, the first is named `t1.html` and the second is `t2.html` (the name of the template is the name of the file, extension and all, unless you change it). If we call the *Execute* method on it:

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

It will 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!")
```

This article took you through how to trigger the template engine to parse and execute templates. My book, [Go Web Programming](), explores Go's template engine even further.