

Templating

You can find all the code here

We live in a world where everyone wants to build web applications with the latest **flavour of the month frontend** framework built upon gigabytes of transpiled JavaScript, working with a Byzantine build system; **but maybe that's not always necessary**.

I'd say most Go developers value a simple, **stable & fast toolchain** but the frontend world frequently fails to deliver on this front.

Many websites do not need to be an **SPA**. **HTML and CSS are fantastic ways of delivering content** and you can use Go to make a website to deliver HTML.

If you wish to still have some dynamic elements, you can still sprinkle in some client side JavaScript, or you may even want to try experimenting with **Hotwire** which allows you to deliver a dynamic experience with a server-side approach.

You can generate your HTML in Go with elaborate usage of **fmt.Fprintf**, but in this chapter you'll learn that Go's standard library has some tools to generate HTML in a simpler and more maintainable way. You'll also learn more effective ways of testing this kind of code that you may not have run in to before.

What we're going to build

In the **Reading Files** chapter we wrote some code that would take an **fs.FS** (a file-system), and return a slice of **Post** for each markdown file it encountered.

```
1 posts, err := blogposts.NewPostsFromFS(os.DirFS("posts"))
```

Here is how we defined **Post**

```
1 type Post struct {  
2     Title, Description, Body string  
3     Tags []string
```

Here's an example of one of the markdown files that can be parsed.

```
1 Title: Welcome to my blog
2 Description: Introduction to my blog
3 Tags: cooking, family, live-laugh-love
4 ---
5 # First recipe!
6 Welcome to my amazing recipe blog. I am going to write about my family recipes, a
```

If we continue our journey of writing blog software, we'd take this data and generate HTML from it for our web server to return in response to HTTP requests.

For our blog, we want to generate two kinds of page:

1. **View post.** Renders a specific post. The `Body` field in `Post` is a string containing markdown so that should be converted to HTML.
2. **Index.** Lists all of the posts, with hyperlinks to view the specific post.

We'll also want a consistent look and feel across our site, so for each page we'll have the usual HTML furniture like `<html>` and a `<head>` containing links to CSS stylesheets and whatever else we may want.

When you're building blog software you have a few options in terms of approach of how you build and send HTML to the user's browser.

We'll design our code so it accepts an `io.Writer`. This means the caller of our code has the flexibility to:

- Write them to an `os.File`, so they can be statically served
- Write out the HTML directly to a `http.ResponseWriter`
- Or just write them to anything really! So long as it implements `io.Writer` the user can generate some HTML from a `Post`

Write the test first

As always, it's important to think about requirements before diving in too fast. How can we take this large-ish set of requirements and break it down in to a small, achievable step that we can focus on?

In my view, actually viewing content is higher priority than an index page. We could launch this product and share direct links to our wonderful content. An index page which cant link to the actual content isn't useful.

Still, rendering a post as described earlier still feels big. All the HTML furniture, converting the body markdown into HTML, listing tags, e.t.c.

At this stage I'm not overly concerned with the specific markup, and an easy first step would be just to check we can render the post's title as an `<h1>`. This *feels* like the smallest first step that can move us forward a bit.

```
1 package blogrenderer_test
2
3 import (
4     "bytes"
5     "github.com/quii/learn-go-with-tests/blogrenderer"
6     "testing"
7 )
8
9 func TestRender(t *testing.T) {
10     var (
11         aPost = blogrenderer.Post{
12             Title:      "hello world",
13             Body:       "This is a post",
14             Description: "This is a description",
15             Tags:       []string{"go", "tdd"},
16         }
17     )
18
19     t.Run("it converts a single post into HTML", func(t *testing.T) {
20         buf := bytes.Buffer{}
21         err := blogrenderer.Render(&buf, aPost)
22
23         if err != nil {
24             t.Fatal(err)
25         }
26
27         got := buf.String()
28         want := `

# 


```

```
31         } t.Errorf("got '%s' want '%s'", got, want)

32     })

33 }
```

Our decision to accept an `io.Writer` also makes testing simple, in this case we're writing to a `bytes.Buffer` which we can then later inspect the contents.

Try to run the test

If you've read the previous chapters of this book you should be well-practiced at this now. You won't be able to run the test because we don't have the package defined or the `Render` function. Try and follow the compiler messages yourself and get to a state where you can run the test and see that it fails with a clear message.

It's really important that you exercise your tests failing, you'll thank yourself when you accidentally make a test fail 6 months later that you put in the effort *now* to check it fails with a clear message.

Write the minimal amount of code for the test to run and check the failing test output

This is the minimal code to get the test running

```
1 package blogrenderer
2
3 // if you're continuing from the read files chapter, you shouldn't redefine this
4 type Post struct {
5     Title, Description, Body string
6     Tags []string
7 }
8
9 func Render(w io.Writer, p Post) error {
10     return nil
11 }
```

The test should complain that an empty string doesn't equal what we want.

Write enough code to make it pass

```
1 func Render(w io.Writer, p Post) error {  
2     _, err := fmt.Fprintf(w, "<h1>%s</h1>", p.Title)  
3     return err  
4 }
```

Remember, software development is primarily a learning activity. In order to discover and learn as we work, we need to work in a way that gives us frequent, high-quality feedback loops, and the easiest way to do that is work in small steps.

So we're not worrying about using any templating libraries right now. You can make HTML just with "normal" string templating just fine, and by skipping the template part we can validate a small bit of useful behaviour and we've done a small bit of design work for our package's API.

Refactor

Not much to refactor yet, so let's move to the next iteration

Write the test first

Now we have a very basic version working, we can now iterate on the test to expand on the functionality. In this case, rendering more information from the `Post`.

```
1 t.Run("it converts a single post into HTML", func(t *testing.T) {  
2     buf := bytes.Buffer{}  
3     err := blogrenderer.Render(&buf, aPost)  
4  
5     if err != nil {  
6         t.Fatal(err)  
7     }
```

```

8
9         got := buf.String()
10        want := `

# 


```

Notice that writing this, *feels* awkward. Seeing all that markup in the test feels bad, and we haven't even put the body in, or the actual HTML we'd want with all of the `<head>` content and whatever page furniture we need.

Nonetheless, let's put up with the pain *for now*.

Try to run the test

It should fail, complaining it doesn't have the string we expect, as we're not rendering the description and tags.

Write enough code to make it pass

Try and do this yourself rather than copying the code. What you should find is that making this test pass *is a bit annoying!* When I tried, my first attempt got this error

```

1 === RUN    TestRender
2 === RUN    TestRender/it_converts_a_single_post_into_HTML
3     renderer_test.go:32: got '<h1>hello world</h1><p>This is a description</p><ul><li>
4         <p>This is a description</p>
5         Tags: <ul><li>go</li><li></li></ul>'

```

New lines! Who cares? Well, our test does, because it's matching on an exact string value. Should it? I removed the newlines for now just to get the test passing.

```

1 func Render(w io.Writer, p Post) error {
2     _, err := fmt.Fprintf(w, "<h1>%s</h1><p>%s</p>", p.Title, p.Description)
3     if err != nil {
4         return err
5     }
6
7     _, err = fmt.Fprint(w, "Tags: <ul>")
8     if err != nil {
9         return err
10    }
11
12    for _, tag := range p.Tags {
13        _, err = fmt.Fprintf(w, "<li>%s</li>", tag)
14        if err != nil {
15            return err
16        }
17    }
18
19    _, err = fmt.Fprint(w, "</ul>")
20    if err != nil {
21        return err
22    }
23
24    return nil
25 }

```

Yikes. Not the nicest code i've written, and we're still only at a very early implementation of our markup. We'll need so much more content and things on our page, we're quickly seeing that this approach is not appropriate.

Crucially though, we have a passing test; we have working software.

Refactor

With the safety-net of a passing test for working code, we can now think about changing our implementation approach at the refactoring stage.

Introducing templates

Go has two templating packages [text/template](#) and [html/template](#) and they share the same interface. What they both do is allow you to combine a template and some data to produce a string.

What's the difference with the HTML version?

Package `template` (`html/template`) implements data-driven templates for generating HTML output safe against code injection. It provides the same interface as package `text/template` and should be used instead of `text/template` whenever the output is HTML.

The templating language is very similar to [Mustache](#) and allows you to dynamically generate content in a very clean fashion with a nice separation of concerns. Compared to other templating languages you may have used, it is very constrained or "logic-less" as Mustache likes to say. This is an important, **and deliberate** design decision.

Whilst we're focusing on generating HTML here, if your project is doing complex string concatenations and incantations, you might want to reach for `text/template` to clean up your code.

Back to the code

Here is a template for our blog:

```
<h1>{{.Title}}</h1><p>{{.Description}}</p>Tags: <ul>{{range .Tags}}<li>{{.}}</li>{{end}}</ul>
```

Where do we define this string? Well, we have a few options, but to keep the steps small, let's just start with a plain old string

```
1 package blogrenderer
2
3 import (
4     "html/template"
5     "io"
6 )
7
8 const (
```




```

9      postTemplate = `

# {{.Title}}

<p>{{.Description}}

```

We create a new template with a name, and then parse our template string. We can then use the `Execute` method on it, passing in our data, in this case the `Post`.

The template will substitute things like `{{.Description}}` with the content of `p.Description`. Templates also give you some programming primitives like `range` to loop over values, and `if`. You can find more details in the [text/template documentation](#).

This should be a pure refactor. We shouldn't need to change our tests and they should continue to pass. Importantly, our code is easier to read and has far less annoying error handling to contend with.

Frequently people complain about the verbosity of error handling in Go, but you might find you can find better ways to write your code so it's less error-prone in the first place, like here.

More refactoring

Using the `html/template` has definitely been an improvement, but having it as a string constant in our code isn't great:

- It's still quite difficult to read.
- It's not IDE/editor friendly. No syntax highlighting, ability to reformat, refactor, e.t.c.
- It looks like HTML, but you can't really work with it like you could a "normal" HTML file

What we'd like to do is have our templates live in separate files so we can better organise them, and work with them as if they're HTML files.

Create a folder called "templates" and inside it make a file called `blog.gohtml`, paste our template into the file.

Now change our code to embed the file systems using the [embedding functionality included in go 1.16](#).

```
1 package blogrenderer
2
3 import (
4     "embed"
5     "html/template"
6     "io"
7 )
8
9 var (
10     //go:embed "templates/*"
11     postTemplates embed.FS
12 )
13
14 func Render(w io.Writer, p Post) error {
15     templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
16     if err != nil {
17         return err
18     }
19
20     if err := templ.Execute(w, p); err != nil {
21         return err
22     }
23
24     return nil
25 }
```

By embedding a "file system" into our code, we can load multiple templates and combine them freely. This will become useful when we want to share rendering logic across different templates, such as a header for the top of the HTML page and a footer.

Embed?

Embed was lightly touched on in [reading files](#). The [documentation from the standard library](#) explains

Package embed provides access to files embedded in the running Go program.

Go source files that import "embed" can use the `//go:embed` directive to initialize a variable of type `string`, `[]byte`, or `FS` with the contents of files read from the package directory or subdirectories at compile time.

Why would we want to use this? Well the alternative is that we *can* load our templates from a "normal" file system. However this means we'd have to make sure that the templates are in the correct file path wherever we want to use this software. In your job you may have various environments like development, staging and live. For this to work, you'd need to make sure your templates are copied to the correct place.

With `embed`, the files are included in your Go program when you build it. This means once you've built your program (which you should only do once), the files are always available to you.

What's handy is you can not only embed individual files, but also file systems; and that filesystem implements `io/fs` which means your code doesn't need to care what kind of file system it is working with.

If you wish to use different templates depending on configuration though, you may wish to stick to loading templates from disk in the more conventional way.

Next: Make the template "nice"

We don't really want our template to be defined as a one line string. We want to be able to space it out to make it easier to read and work with, something like this:

```
1 <h1>{{.Title}}</h1>
2
3 <p>{{.Description}}</p>
4
5 Tags: <ul>{{range .Tags}}<li>{{.}}</li>{{end}}</ul>
```

But if we do this, our test fails. This is because our test is expecting a very specific string to be returned.

But really, we don't actually care about whitespace. Maintaining this test will become a nightmare if we have to keep painstakingly updating the assertion string every time we make minor changes to the markup. As the template grows, these kind of edits become harder to manage and the costs of work will spiral out of control.

Introducing Approval Tests

Go Approval Tests

ApprovalTests allows for easy testing of larger objects, strings and anything else that can be saved to a file (images, sounds, CSV, etc...)

The idea is similar to "golden" files, or snapshot testing. Rather than awkwardly maintaining strings within a test file, the approval tool can compare the output for you with an "approved" file you created. You then simply copy over the new version if you approve it. Re-run the test and you're back to green.

Add a dependency to `"github.com/approvals/go-approval-tests"` to your project and edit the test to the following

```
1 func TestRender(t *testing.T) {
2     var (
3         aPost = blogrenderer.Post{
4             Title:      "hello world",
5             Body:        "This is a post",
6             Description: "This is a description",
7             Tags:         []string{"go", "tdd"},
8         }
9     )
10
11     t.Run("it converts a single post into HTML", func(t *testing.T) {
12         buf := bytes.Buffer{}
13
14         if err := blogrenderer.Render(&buf, aPost); err != nil {
15             t.Fatal(err)
16         }
17
18         approvals.VerifyString(t, buf.String())
19     })
20 }
```

The first time you run it, it will fail because we haven't approved anything yet

```
2 === RUN   TestRender/it_converts_a_single_post_into_HTML
3     renderer_test.go:29: Failed Approval: received does not match approved.
```

It will have created two files, that look like the following

- `renderer_test.TestRender.it_converts_a_single_post_into_HTML.received.txt`
- `renderer_test.TestRender.it_converts_a_single_post_into_HTML.approved.txt`

The received file has the new, unapproved version of the output. Copy that into the empty approved file and re-run the test.

By copying the new version you have "approved" the change, and the test now passes.

To see the workflow in action, edit the template to how we discussed to make it easier to read (but semantically, it's the same).

```
1 <h1>{{.Title}}</h1>
2
3 <p>{{.Description}}</p>
4
5 Tags: <ul>{{range .Tags}}<li>{{.}}</li>{{end}}</ul>
```

Re-run the test. A new "received" file will be generated because the output of our code differs to the approved version. Give them a look, and if you're happy with the changes, simply copy over the new version and re-run the test. Be sure to commit the approved files to source control.

This approach makes managing changes to big ugly things like HTML far simpler. You can use a diff tool to view and manage the differences, and it keeps your test code cleaner.

renderer_test.TestRender.it_converts_a_single_post_into_HTML.approved.txt (/...)		renderer_test.TestRender.it_converts_a_single_post_into_HTML.receive
home	13	home
about	14	about
archive	15	archive
	16	
</div>	17	</div>
</nav>	18	</nav>
<main>	19	<main>
hello world</h1>	21	<h1>hello world oops i mashed the keyboard</h1>
This is a description</p>	23	<p>This is a description</p>
gotdd	25	Tags: gotdd

```

er>
ul>
<li><a href="https://twitter.com/quii">Twitter</a>
<li><a href="https://github.com/quii">GitHub</a></li>
</ul>
</body>
</html>

```

Use diff tool to manage changes

This is actually a fairly minor usage of approval tests, which are an extremely useful tool in your testing arsenal. [Emily Bache](#) has an [interesting video where she uses approval tests to add an incredibly extensive set of tests to a complicated codebase that has zero tests](#). "Combinatorial Testing" is definitely something worth looking into.

Now that we have made this change, we still benefit from having our code well-tested, but the tests won't get in the way too much when we're tinkering with the markup.

Are we still doing TDD?

An interesting side-effect of this approach is it takes us away from TDD. Of course you *could* manually edit the approved files to the state you want, run your tests and then fix the templates so they output what you defined.

But that's just silly! TDD is a method for doing work, specifically designing; but that doesn't mean we have to **dogmatically use it for everything**.

The important thing is, we've done the right thing and used TDD as a **design tool to design our package's API**. For templates changes our process can be:

- Make a small change to the template
- Run the approval test
- **Eyeball the output to check it looks correct**
- **Make the approval**
- Repeat

We still shouldn't give up the value of working in small achievable steps. Try to find ways to make the changes small and keep re-running the tests to get real feedback on what you're doing.

If we start doing things like changing the code *around* the templates, then of course that may warrant going back to our TDD method of work.

Expand the markup

Most websites have richer HTML than we have right now. For starters, a `html` element, along with a `head`, perhaps some `nav` too. Usually there's an idea of a footer too.

If our site is going to have different pages, we'd want to define these things in one place to keep our site looking consistent. Go templates support us defining sections which we can then import in to other templates.

Edit our existing template to import a top and bottom template

```
1 {{template "top" .}}
2 <h1>{{.Title}}</h1>
3
4 <p>{{.Description}}</p>
5
6 Tags: <ul>{{range .Tags}}<li>{{.}}</li>{{end}}</ul>
7 {{template "bottom" .}}
```

Then create `top.gohtml` with the following

```
1 {{define "top"}}
2 <!DOCTYPE html>
3 <html lang="en">
4 <head>
5     <title>My amazing blog!</title>
6     <meta charset="UTF-8"/>
7     <meta name="description" content="Wow, like and subscribe, it really helps the ch
8 </head>
9 <body>
10 <nav role="navigation">
11     <div>
12         <h1>Budding Gopher's blog</h1>
13         <ul>
14             <li><a href="/">home</a></li>
15             <li><a href="about">about</a></li>
16             <li><a href="archive">archive</a></li>
17         </ul>
18     </div>
19 </nav>
20 <main>
21 {{end}}
```

And `bottom.gohtml`

```
1 {{define "bottom"}}
2 </main>
3 <footer>
4     <ul>
5         <li><a href="https://twitter.com/qui" >Twitter</a></li>
6         <li><a href="https://github.com/qui" >GitHub</a></li>
7     </ul>
8 </footer>
9 </body>
10 </html>
11 {{end}}
```

(Obviously, feel free to put whatever markup you like!)

Re-run your test. A new "received" file should be made and the test will fail. Check it over and if you're happy, approve it by copying it over the old version. Re-run the test again and it should pass.

An excuse to mess around with Benchmarking

Before pressing on, let's consider what our code does.

```
1 func Render(w io.Writer, p Post) error {
2     templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
3     if err != nil {
4         return err
5     }
6
7     if err := templ.Execute(w, p); err != nil {
8         return err
9     }
10
11     return nil
12 }
```

- Parse the templates

- Use the template to render a post to an `io.Writer`

Whilst the performance impact of re-parsing the templates for each post in most cases will be fairly negligible, the effort to *not* do this is also pretty negligible and should tidy the code up a bit too.

To see the impact of not doing this parsing over and over, we can use the benchmarking tool to see how fast our function is.

```
1 func BenchmarkRender(b *testing.B) {
2     var (
3         aPost = blogrenderer.Post{
4             Title:      "hello world",
5             Body:         "This is a post",
6             Description: "This is a description",
7             Tags:          []string{"go", "tdd"},
8         }
9     )
10
11     b.ResetTimer()
12     for i := 0; i < b.N; i++ {
13         blogrenderer.Render(io.Discard, aPost)
14     }
15 }
```

On my computer, here are the results

```
1 BenchmarkRender-8 22124 53812 ns/op
```

To stop us having to re-parse the templates over and over, we'll create a type that'll hold the parsed template, and that'll have a method to do the rendering

```
1 type PostRenderer struct {
2     templ *template.Template
3 }
4
5 func NewPostRenderer() (*PostRenderer, error) {
6     templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
7     if err != nil {
8         return nil, err
9     }
10 }
```

```

10
11     return &PostRenderer{templ: templ}, nil
12 }
13
14 func (r *PostRenderer) Render(w io.Writer, p Post) error {
15
16     if err := r.templ.Execute(w, p); err != nil {
17         return err
18     }
19
20     return nil
21 }

```

This does change the interface of our code, so we'll need to update our test

```

1 func TestRender(t *testing.T) {
2     var (
3         aPost = blogrenderer.Post{
4             Title:      "hello world",
5             Body:        "This is a post",
6             Description: "This is a description",
7             Tags:        []string{"go", "tdd"},
8         }
9     )
10
11     postRenderer, err := blogrenderer.NewPostRenderer()
12
13     if err != nil {
14         t.Fatal(err)
15     }
16
17     t.Run("it converts a single post into HTML", func(t *testing.T) {
18         buf := bytes.Buffer{}
19
20         if err := postRenderer.Render(&buf, aPost); err != nil {
21             t.Fatal(err)
22         }
23
24         approvals.VerifyString(t, buf.String())
25     })
26 }

```

And our benchmark

```
1 func BenchmarkRender(b *testing.B) {
2     var (
3         aPost = blogrenderer.Post{
4             Title:      "hello world",
5             Body:         "This is a post",
6             Description: "This is a description",
7             Tags:         []string{"go", "tdd"},
8         }
9     )
10
11     postRenderer, err := blogrenderer.NewPostRenderer()
12
13     if err != nil {
14         b.Fatal(err)
15     }
16
17     b.ResetTimer()
18     for i := 0; i < b.N; i++ {
19         postRenderer.Render(io.Discard, aPost)
20     }
21 }
```

The test should continue to pass. How about our benchmark?

BenchmarkRender-8 362124 3131 ns/op . The old NS per op were 53812 ns/op , so this is a decent improvement! As we add other methods to render, say an Index page, it should simplify the code as we don't need to duplicate the template parsing.

Back to the real work

In terms of rendering posts, the important part left is actually rendering the `Body` . If you recall, that should be markdown that the author has written, so it'll need converting to HTML.

We'll leave this as an exercise for you, the reader. You should be able to find a Go library to do this for you. Use the approval test to validate what you're doing.

On testing 3rd party libraries

Note. Be careful not to worry too much about explicitly testing how a 3rd party library behaves in unit tests.

Writing tests against code you don't control is wasteful and adds maintenance overhead. Sometimes you may wish to use [dependency injection](#) to control a dependency and mock its behaviour for a test.

In this case though, I view converting the markdown into HTML as implementation detail of rendering, and our approval tests should give us enough confidence.

Render index

The next bit of functionality we're going to do is rendering an Index, listing the posts as a HTML ordered list.

We're expanding upon our API, so we'll put our TDD hat back on.

Write the test first

On the face of it an index page seems simple, but writing the test still prompts us to make some design choices

```
1 t.Run("it renders an index of posts", func(t *testing.T) {
2     buf := bytes.Buffer{}
3     posts := []blogrenderer.Post{{Title: "Hello World"}, {Title: "Hello World 2"}}
4
5     if err := postRenderer.RenderIndex(&buf, posts); err != nil {
6         t.Fatal(err)
7     }
8
9     got := buf.String()
10    want := `


<li><a href="/post/hello-world">Hello World</a></li><li><a href=
11
12    if got != want {
13        t.Errorf("got %q want %q", got, want)
14    }
15 })
```

1. We're using the `Post` 's title field as a part of the path of the URL, but we don't really want spaces in the URL so we're replacing them with hyphens.
2. We've added a `RenderIndex` method to our `PostRenderer` that again takes an `io.Writer` and a slice of `Post`.

If we had stuck with a test-after, approval tests approach here we would not be answering these questions in a controlled environment. **Tests give us space to think.**

Try to run the test

```
1 ./renderer_test.go:41:13: undefined: blogrenderer.RenderIndex
```



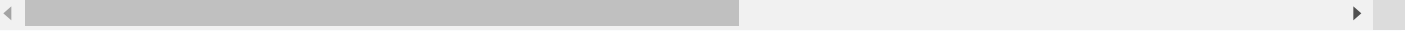

Write the minimal amount of code for the test to run and check the failing test output

```
1 func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {  
2     return nil  
3 }
```



The above should get the following test failure

```
1 === RUN    TestRender  
2 === RUN    TestRender/it_renders_an_index_of_posts  
3     renderer_test.go:49: got "" want "<ol><li><a href=\"/post/hello-world\">Hello Wor  
4 --- FAIL: TestRender (0.00s)
```



Write enough code to make it pass

Even though this *feels* like it should be easy, it is a bit awkward. I did it in multiple steps

```
1 func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
2     indexTemplate := `<ol>{{range .}}<li><a href="/post/{{.Title}}">{{.Title}}</a>
3
4     templ, err := template.New("index").Parse(indexTemplate)
5     if err != nil {
6         return err
7     }
8
9     if err := templ.Execute(w, posts); err != nil {
10         return err
11     }
12
13     return nil
14 }
```

I didn't want to bother with separate template files at first, I just wanted to get it working. I view the upfront template parsing and separation as refactoring I can do later.

This doesn't pass, but it's close.

```
1 === RUN   TestRender
2 === RUN   TestRender/it_renders_an_index_of_posts
3     renderer_test.go:49: got "<ol><li><a href=\"/post/Hello%20World\">Hello World</a>
4 --- FAIL: TestRender (0.00s)
5     --- FAIL: TestRender/it_renders_an_index_of_posts (0.00s)
```

You can see that the templating code is escaping the spaces in the `href` attributes. We need a way to do a string replace of spaces with hyphens. We can't just loop through the `[]Post` and replace them in-memory because we still want the spaces displayed to the user in the anchors.

We have a few options. The first one we'll explore is passing a function in to our template.

Passing functions into templates

```
1 func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
2     indexTemplate := `<ol>{{range .}}<li><a href="/post/{{sanitiseTitle .Title}}"`
```

```

3
4     templ, err := template.New("index").Funcs(template.FuncMap{
5
6         "sanitiseTitle": func(title string) string {
7             return strings.ToLower(strings.Replace(title, " ", "-", -1))
8         },
9     }).Parse(indexTemplate)
10    if err != nil {
11        return err
12    }
13    if err := templ.Execute(w, posts); err != nil {
14        return err
15    }
16
17    return nil
18 }

```

Before you parse a template you can add a `template.FuncMap` into your template, which allows you to define functions that can be called within your template. In this case we've made a `sanitiseTitle` function which we then call inside our template with `{{sanitiseTitle .Title}}`.

This is a powerful feature, being able to send functions in to your template will allow you to do some very cool things, but, should you? Going back to the principles of Mustache and logic-less templates, why did they advocate for logic-less? **What is wrong with logic in templates?**

As we've shown, in order to test our templates, *we've had to introduce a whole different kind of testing.*

Imagine you introduce a function into a template which has a few different permutations of behaviour and edge cases, **how will you test it?** With this current design, your only means of testing this logic is by *rendering HTML and comparing strings*. This is not an easy or sane way of testing logic, and definitely not what you'd want for *important* business logic.

Even though the approval tests technique has reduced the cost of maintaining these tests, they're still more expensive to maintain than most unit tests you'll write. They're still sensitive to any minor markup changes you might make, it's just we've made it easier to manage. We should still strive to architect our code so we don't have to write many tests around our templates, and try and separate concerns so any logic that doesn't need to live inside our rendering code is properly separated.

What Mustache-influenced templating engines give you is a useful constraint, don't try to circumvent it too often; **don't go against the grain**. Instead, embrace the idea of [view models](#),

where you construct specific types that contain the data you need to render, in a way that's convenient for the templating language.

This way, whatever important business logic you use to generate that bag of data can be unit tested separately, away from the messy world of HTML and templating.

Separating concerns

So what could we do instead?

Add a method to `Post` and then call that in the template

We can call methods in our templating code on the types we send, so we could add a `SanitisedTitle` method to `Post`. This would simplify the template and we could easily unit test this logic separately if we wish. This is probably the easiest solution, although not necessarily the simplest.

A downside to this approach is that this is still *view* logic. It's not interesting to the rest of the system but it now becomes a part of the API for a core domain object. This kind of approach over time can lead to you creating [God Objects](#).

Create a dedicated view model type, such as `PostViewModel` with exactly the data we need

Rather than our rendering code being coupled to the domain object, `Post`, it instead takes a view model.

```
1 type PostViewModel struct {  
2     Title, SanitisedTitle, Description, Body string  
3     Tags []string  
4 }
```

Callers of our code would have to map from `[]Post` to `[]PostView`, generating the `SanitizedTitle`. A way to keep this clean would be to have a `func NewPostView(p Post) PostView` which would encapsulate the mapping.

This would keep our rendering code logic-less and is probably the strictest separation of concerns we could do, but the trade-off is a slightly more convoluted process to get our posts rendered.

Both options are fine, in this case I am tempted to go with the first. As you evolve the system you should be wary of adding more and more ad-hoc methods just to grease the wheels of rendering;

dedicated view models become more useful when the transformation between the domain object and view becomes more involved.

So we can add our method to `Post`

```
1 func (p Post) SanitisedTitle() string {
2     return strings.ToLower(strings.Replace(p.Title, " ", "-", -1))
3 }
```

And then we can go back to a simpler world in our rendering code

```
1 func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
2     indexTemplate := `<ol>{{range .}}<li><a href="/post/{{.SanitisedTitle}}">{{.Title}}</a></li>{{end}}</ol>`
3
4     templ, err := template.New("index").Parse(indexTemplate)
5     if err != nil {
6         return err
7     }
8
9     if err := templ.Execute(w, posts); err != nil {
10        return err
11    }
12
13    return nil
14 }
```

Refactor

Finally the test should be passing. We can now move our template into a file (`templates/index.gohtml`) and load it once, when we construct our renderer.

```
1 package blogrenderer
2
3 import (
4     "embed"
5     "html/template"
6     "io"
```

```

7 )
8
9 var (
10     //go:embed "templates/*"
11     postTemplates embed.FS
12 )
13
14 type PostRenderer struct {
15     templ *template.Template
16 }
17
18 func NewPostRenderer() (*PostRenderer, error) {
19     templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
20     if err != nil {
21         return nil, err
22     }
23
24     return &PostRenderer{templ: templ}, nil
25 }
26
27 func (r *PostRenderer) Render(w io.Writer, p Post) error {
28     return r.templ.ExecuteTemplate(w, "blog.gohtml", p)
29 }
30
31 func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
32     return r.templ.ExecuteTemplate(w, "index.gohtml", posts)
33 }

```

By parsing more than one template into `templ` we now have to call `ExecuteTemplate` and specify *which* template we wish to render as appropriate, but hopefully you'll agree the code we've arrived at looks great.

There is a *slight* risk if someone renames one of the template files, it would introduce a bug, but our fast to run unit tests would catch this quickly.

Now we're happy with our package's API design and got some basic behaviour driven out with TDD, let's change our test to use approvals.

```

1     t.Run("it renders an index of posts", func(t *testing.T) {
2         buf := bytes.Buffer{}
3         posts := []blogrenderer.Post{{Title: "Hello World"}, {Title: "Hello W
4

```

```

6         if err := postRenderer.RenderIndex(&buf, posts); err != nil {
7             t.Fatal(err)
8         }
9         approvals.VerifyString(t, buf.String())
10    }

```

Remember to run the test to see it fail, and then approve the change.

Finally we can add our page furniture to our index page:

```

1 {{template "top" .}}
2 <ol>{{range .}}<li><a href="/post/{{.SanitisedTitle}}">{{.Title}}</a></li>{{end}}</ol>
3 {{template "bottom" .}}

```

Re-run the test, approve the change and we're done with the index!

Rendering the markdown body

I encouraged you to try it yourself, here's the approach I ended up taking.

```

1 package blogrenderer
2
3 import (
4     "embed"
5     "github.com/gomarkdown/markdown"
6     "github.com/gomarkdown/markdown/parser"
7     "html/template"
8     "io"
9 )
10
11 var (
12     //go:embed "templates/*"
13     postTemplates embed.FS
14 )
15
16 type PostRenderer struct {
17     templ    *template.Template
18     mdParser *parser.Parser

```

```

19 }
20
21 func NewPostRenderer() (*PostRenderer, error) {
22     templ, err := template.ParseFS(postTemplates, "templates/*.gohtml")
23     if err != nil {
24         return nil, err
25     }
26
27     extensions := parser.CommonExtensions | parser.AutoHeadingIDs
28     parser := parser.NewWithExtensions(extensions)
29
30     return &PostRenderer{templ: templ, mdParser: parser}, nil
31 }
32
33 func (r *PostRenderer) Render(w io.Writer, p Post) error {
34     return r.templ.ExecuteTemplate(w, "blog.gohtml", newPostVM(p, r))
35 }
36
37 func (r *PostRenderer) RenderIndex(w io.Writer, posts []Post) error {
38     return r.templ.ExecuteTemplate(w, "index.gohtml", posts)
39 }
40
41 type postViewModel struct {
42     Post
43     HTMLBody template.HTML
44 }
45
46 func newPostVM(p Post, r *PostRenderer) postViewModel {
47     vm := postViewModel{Post: p}
48     vm.HTMLBody = template.HTML(markdown.ToHTML([]byte(p.Body), r.mdParser, nil))
49     return vm
50 }

```

I used the excellent [gomarkdown](#) library which worked exactly how I'd hope.

If you tried to do this yourself you may have found that your body render had the HTML escaped. This is a security feature of Go's `html/template` package to stop malicious 3rd-party HTML being outputted.

To circumvent this, in the type you send to the render, you'll need to wrap your trusted HTML in `template.HTML`

HTML encapsulates a known safe HTML document fragment. It should not be used for HTML

from a third-party, or HTML with unclosed tags or comments. The outputs of a sound HTML sanitiser and a template escaped by this package are fine for use with HTML.

Use of this type presents a security risk: the encapsulated content should come from a trusted source, as it will be included verbatim in the template output.

So I created an **unexported** view model (`postViewModel`), because I still viewed this as internal implementation detail to rendering. I have no need to test this separately and I don't want it polluting my API.

I construct one when rendering so I can parse the `Body` into `HTMLBody` and then I use that field in the template to render the HTML.

Wrapping up

If you combine your learnings of the [reading files](#) chapter and this one, you can comfortably make a well-tested, simple, static site generator and spin up a blog of your own. Find some CSS tutorials and you can make it look nice too.

This approach extends beyond blogs. Taking data from any source, be it a database, an API or a file-system and converting it into HTML and returning it from a server is a simple technique spanning many decades. People like to bemoan the complexity of modern web development but are you sure you're not just inflicting the complexity on yourself?

Go is wonderful for web development, especially when you think clearly about what your real requirements are for the website you're making. Generating HTML on the server is often a better, simpler and more performant approach than creating a "web application" with technologies like React.

What we've learned

- How to create and render HTML templates.
- How to compose templates together and [DRY](#) up related markup and help us keep a consistent look and feel.
- How to pass functions into templates, and why you should think twice about it.
- How to write "Approval Tests", which help us test the big ugly output of things like template

renderers.

On logic-less templates

As always, this is all about **separation of concerns**. It's important we consider what the responsibilities are of the various parts of our system. Too often people leak important business logic into templates, mixing up concerns and making systems difficult to understand, maintain and test.

Not just for HTML

Remember that go has `text/template` to generate other kinds of data from a template. If you find yourself needing to transform data into some kind of structured output, the techniques laid out in this chapter can be useful.

References and further material

- [John Calhoun's 'Learn Web Development with Go'](#) has a number of excellent articles on templating.
- [Hotwire](#) - You can use these techniques to create Hotwire web applications. It has been built by Basecamp who are primarily a Ruby on Rails shop, but because it is server-side, we can use it with Go.