Writing Web Applications

Table of Contents

Introduction
Getting Started
Data Structures
Introducing the net/http package (an interlude)
Using net/http to serve wiki pages
Editing Pages
The html/template package
Handling non-existent pages
Saving Pages
Error handling
Template caching
Validation
Introducing Function Literals and Closures
Try it out!
Other tasks

Introduction

Covered in this tutorial:

- Creating a data structure with load and save methods
- Using the `net/http` package to build web applications
- Using the `html/template` package to process HTML templates
- Using the `regexp` package to validate user input
- Using closures

Assumed knowledge:

- Programming experience
- Understanding of basic web technologies (HTTP, HTML)
- Some UNIX/DOS command-line knowledge

Getting Started

At present, you need to have a FreeBSD, Linux, macOS, or Windows machine to run Go. We will use $ to represent the command prompt.

Install Go (see the Installation Instructions).

Make a new directory for this tutorial inside your `GOPATH` and cd to it:

```
$ mkdir gowiki
$ cd gowiki
```

Create a file named `wiki.go`, open it in your favorite editor, and add the following lines:

```
package main

import (
        "fmt"
        "os"
)
```

We import the `fmt` and `os` packages from the Go standard library. Later, as we implement additional functionality, we will add more packages to this `import` declaration.

Data Structures

Let's start by defining the data structures. A wiki consists of a series of interconnected pages, each of which has a title and a body (the page content). Here, we define `Page` as a struct with two fields representing the title and body.

```
type Page struct {
    Title string
    Body  []byte
}
```

The type `[]byte` means "a `byte` slice". (See Slices: usage and internals for more on slices.) The `Body` element is a `[]byte` rather than `string` because that is the type expected by the `io` libraries we will use, as you'll see below.

The `Page` struct describes how page data will be stored in memory. But what about persistent storage? We can address that by creating a `save` method on `Page`:

```
func (p *Page) save() error {
    filename := p.Title + ".txt"
    return os.WriteFile(filename, p.Body, 0600)
}
```

This method's signature reads: "This is a method named `save` that takes as its receiver `p`, a pointer to `Page` . It takes no parameters, and returns a value of type `error`."

This method will save the `Page`'s `Body` to a text file. For simplicity, we will use the `Title` as the file name.

The `save` method returns an `error` value because that is the return type of `WriteFile` (a standard library function that writes a byte slice to a file). The `save` method returns the error value, to let the application handle it should anything go wrong while writing the file. If all goes well, `Page.save()` will return `nil` (the zero-value for pointers, interfaces, and some other types).

The octal integer literal `0600`, passed as the third parameter to `WriteFile`, indicates that the file should be created with read-write permissions for the current user only. (See the Unix man page `open(2)` for details.)

In addition to saving pages, we will want to load pages, too:

```
func loadPage(title string) *Page {
    filename := title + ".txt"
    body, _ := os.ReadFile(filename)
    return &Page{Title: title, Body: body}
}
```

The function `loadPage` constructs the file name from the title parameter, reads the file's contents into a new variable `body`, and returns a pointer to a `Page` literal constructed with the proper title and body values.

Functions can return multiple values. The standard library function `os.ReadFile` returns `[]byte` and `error`. In `loadPage`, error isn't being handled yet; the "blank identifier" represented by the underscore (_) symbol is used to throw away the error return value (in essence, assigning the value to nothing).

But what happens if `ReadFile` encounters an error? For example, the file might not exist. We should not ignore such errors. Let's modify the function to return `*Page` and `error`.

```
func loadPage(title string) (*Page, error) {
    filename := title + ".txt"
    body, err := os.ReadFile(filename)
    if err != nil {
        return nil, err
    }
    return &Page{Title: title, Body: body}, nil
}
```

Callers of this function can now check the second parameter; if it is `nil` then it has successfully loaded a Page. If not, it will be an `error` that can be handled by the caller (see the language specification for details).

At this point we have a simple data structure and the ability to save to and load from a file. Let's write a `main` function to test what we've written:

```
func main() {
    p1 := &Page{Title: "TestPage", Body: []byte("This is a sample Page.")}
    p1.save()
    p2, _ := loadPage("TestPage")
    fmt.Println(string(p2.Body))
}
```

After compiling and executing this code, a file named `TestPage.txt` would be created, containing the contents of `p1`. The file would then be read into the struct `p2`, and its `Body` element printed to the screen.

You can compile and run the program like this:

```
$ go build wiki.go
$ ./wiki
This is a sample Page.
```

(If you're using Windows you must type "`wiki`" without the "`./`" to run the program.)

Click here to view the code we've written so far.

Introducing the `net/http` package (an interlude)

Here's a full working example of a simple web server:

```
//go:build ignore

package main

import (
    "fmt"
    "log"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hi there, I love %s!", r.URL.Path[1:])
}

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

The `main` function begins with a call to `http.HandleFunc`, which tells the `http` package to handle all requests to the web root (`"/"`) with `handler`.

It then calls `http.ListenAndServe`, specifying that it should listen on port 8080 on any interface (`":8080"`). (Don't worry about its second parameter, `nil`, for now.) This function will block until the program is terminated.

`ListenAndServe` always returns an error, since it only returns when an unexpected error occurs. In order to log that error we wrap the function call with `log.Fatal`.

The function `handler` is of the type `http.HandlerFunc`. It takes an `http.ResponseWriter` and an `http.Request` as its arguments.

An `http.ResponseWriter` value assembles the HTTP server's response; by writing to it, we send data to the HTTP client.

An `http.Request` is a data structure that represents the client HTTP request. `r.URL.Path` is the path component of the request URL. The trailing `[1:]` means "create a sub-slice of `Path` from the 1st character to the end." This drops the leading "/" from the path name.

If you run this program and access the URL:

```
http://localhost:8080/monkeys
```

the program would present a page containing:

```
Hi there, I love monkeys!
```

Using `net/http` to serve wiki pages

To use the `net/http` package, it must be imported:

```
import (
        "fmt"
```

```
        "os"
        "log"
        "net/http"
)
```

Let's create a handler, `viewHandler` that will allow users to view a wiki page. It will handle URLs prefixed with "/view/".

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/view/"):]
    p, _ := loadPage(title)
    fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>", p.Title, p.Body)
}
```

Again, note the use of `_` to ignore the `error` return value from `loadPage`. This is done here for simplicity and generally considered bad practice. We will attend to this later.

First, this function extracts the page title from `r.URL.Path`, the path component of the request URL. The `Path` is re-sliced with `[len("/view/"):]` to drop the leading `"/view/"` component of the request path. This is because the path will invariably begin with `"/view/"`, which is not part of the page's title.

The function then loads the page data, formats the page with a string of simple HTML, and writes it to `w`, the `http.ResponseWriter`.

To use this handler, we rewrite our `main` function to initialize `http` using the `viewHandler` to handle any requests under the path `/view/`.

```
func main() {
    http.HandleFunc("/view/", viewHandler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Click here to view the code we've written so far.

Let's create some page data (as `test.txt`), compile our code, and try serving a wiki page.

Open `test.txt` file in your editor, and save the string "Hello world" (without quotes) in it.

```
$ go build wiki.go
$ ./wiki
```

(If you're using Windows you must type "`wiki`" without the "`./`" to run the program.)

With this web server running, a visit to `http://localhost:8080/view/test` should show a page titled "test" containing the words "Hello world".

Editing Pages

A wiki is not a wiki without the ability to edit pages. Let's create two new handlers: one named `editHandler` to display an 'edit page' form, and the other named `saveHandler` to save the data entered via the form.

First, we add them to `main()`:

```
func main() {
    http.HandleFunc("/view/", viewHandler)
    http.HandleFunc("/edit/", editHandler)
    http.HandleFunc("/save/", saveHandler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

The function `editHandler` loads the page (or, if it doesn't exist, create an empty `Page` struct), and displays an HTML form.

```
func editHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/edit/"):]
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    fmt.Fprintf(w, "<h1>Editing %s</h1>"+
        "<form action=\"/save/%s\" method=\"POST\">"+
        "<textarea name=\"body\">%s</textarea><br>"+
        "<input type=\"submit\" value=\"Save\">"+
        "</form>",
        p.Title, p.Title, p.Body)
}
```

This function will work fine, but all that hard-coded HTML is ugly. Of course, there is a better way.

The `html/template` package

The `html/template` package is part of the Go standard library. We can use `html/template` to keep the HTML in a separate file, allowing us to change the layout of our edit page without modifying the underlying Go code.

First, we must add `html/template` to the list of imports. We also won't be using `fmt` anymore, so we have to remove that.

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

Let's create a template file containing the HTML form. Open a new file named `edit.html`, and add the following lines:

```
<h1>Editing {{.Title}}</h1>

<form action="/save/{{.Title}}" method="POST">
<div><textarea name="body" rows="20" cols="80">{{printf "%s" .Body}}</textarea></
<div><input type="submit" value="Save"></div>
</form>
```

Modify `editHandler` to use the template, instead of the hard-coded HTML:

```
func editHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/edit/"):]
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    t, _ := template.ParseFiles("edit.html")
    t.Execute(w, p)
}
```

The function `template.ParseFiles` will read the contents of `edit.html` and return a `*template.Template`.

The method `t.Execute` executes the template, writing the generated HTML to the `http.ResponseWriter`. The `.Title` and `.Body` dotted identifiers refer to `p.Title` and `p.Body`.

Template directives are enclosed in double curly braces. The `printf "%s" .Body` instruction is a function call that outputs `.Body` as a string instead of a stream of bytes, the same as a call to `fmt.Printf`. The `html/template` package helps guarantee that only safe and correct-looking HTML is generated by template actions. For instance, it automatically escapes any greater than sign (>), replacing it with `>`, to make sure user data does not corrupt the form HTML.

Since we're working with templates now, let's create a template for our `viewHandler` called `view.html`:

```
<h1>{{.Title}}</h1>

<p>[<a href="/edit/{{.Title}}">edit</a>]</p>

<div>{{printf "%s" .Body}}</div>
```

Modify `viewHandler` accordingly:

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/view/"):]
    p, _ := loadPage(title)
    t, _ := template.ParseFiles("view.html")
    t.Execute(w, p)
}
```

Notice that we've used almost exactly the same templating code in both handlers. Let's remove this duplication by moving the templating code to its own function:

```
func renderTemplate(w http.ResponseWriter, tmpl string, p *Page) {
    t, _ := template.ParseFiles(tmpl + ".html")
    t.Execute(w, p)
}
```

And modify the handlers to use that function:

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/view/"):]
    p, _ := loadPage(title)
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/edit/"):]
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    renderTemplate(w, "edit", p)
}
```

If we comment out the registration of our unimplemented save handler in `main`, we can once again build and test our program. Click here to view the code we've written so far.

Handling non-existent pages

What if you visit `/view/APageThatDoesntExist`? You'll see a page containing HTML. This is because it ignores the error return value from `loadPage` and continues to try and fill out the template with no data. Instead, if the requested Page doesn't exist, it should redirect the client to the edit Page so the content may be created:

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/view/"):]
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}
```

The `http.Redirect` function adds an HTTP status code of `http.StatusFound` (302) and a `Location` header to the HTTP response.

Saving Pages

The function `saveHandler` will handle the submission of forms located on the edit pages. After uncommenting the related line in `main`, let's implement the handler:

```
func saveHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/save/"):]
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    p.save()
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

The page title (provided in the URL) and the form's only field, `Body`, are stored in a new `Page`. The `save()` method is then called to write the data to a file, and the client is

redirected to the `/view/` page.

The value returned by `FormValue` is of type `string`. We must convert that value to `[]byte` before it will fit into the `Page` struct. We use `[]byte(body)` to perform the conversion.

Error handling

There are several places in our program where errors are being ignored. This is bad practice, not least because when an error does occur the program will have unintended behavior. A better solution is to handle the errors and return an error message to the user. That way if something does go wrong, the server will function exactly how we want and the user can be notified.

First, let's handle the errors in `renderTemplate`:

```
func renderTemplate(w http.ResponseWriter, tmpl string, p *Page) {
    t, err := template.ParseFiles(tmpl + ".html")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    err = t.Execute(w, p)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

The `http.Error` function sends a specified HTTP response code (in this case "Internal Server Error") and error message. Already the decision to put this in a separate function is paying off.

Now let's fix up `saveHandler`:

```
func saveHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/save/"):]
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    err := p.save()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

Any errors that occur during `p.save()` will be reported to the user.

Template caching

There is an inefficiency in this code: `renderTemplate` calls `ParseFiles` every time a page is rendered. A better approach would be to call `ParseFiles` once at program initialization,

parsing all templates into a single `*Template`. Then we can use the `ExecuteTemplate` method to render a specific template.

First we create a global variable named `templates`, and initialize it with `ParseFiles`.

```
var templates = template.Must(template.ParseFiles("edit.html", "view.html"))
```

The function `template.Must` is a convenience wrapper that panics when passed a non-nil `error` value, and otherwise returns the `*Template` unaltered. A panic is appropriate here; if the templates can't be loaded the only sensible thing to do is exit the program.

The `ParseFiles` function takes any number of string arguments that identify our template files, and parses those files into templates that are named after the base file name. If we were to add more templates to our program, we would add their names to the `ParseFiles` call's arguments.

We then modify the `renderTemplate` function to call the `templates.ExecuteTemplate` method with the name of the appropriate template:

```
func renderTemplate(w http.ResponseWriter, tmpl string, p *Page) {
    err := templates.ExecuteTemplate(w, tmpl+".html", p)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

Note that the template name is the template file name, so we must append `".html"` to the `tmpl` argument.

Validation

As you may have observed, this program has a serious security flaw: a user can supply an arbitrary path to be read/written on the server. To mitigate this, we can write a function to validate the title with a regular expression.

First, add `"regexp"` to the `import` list. Then we can create a global variable to store our validation expression:

```
var validPath = regexp.MustCompile("^/(edit|save|view)/([a-zA-Z0-9]+)$")
```

The function `regexp.MustCompile` will parse and compile the regular expression, and return a `regexp.Regexp`. `MustCompile` is distinct from `Compile` in that it will panic if the expression compilation fails, while `Compile` returns an `error` as a second parameter.

Now, let's write a function that uses the `validPath` expression to validate path and extract the page title:

```
func getTitle(w http.ResponseWriter, r *http.Request) (string, error) {
    m := validPath.FindStringSubmatch(r.URL.Path)
    if m == nil {
        http.NotFound(w, r)
```

```
        return "", errors.New("invalid Page Title")
    }
    return m[2], nil // The title is the second subexpression.
}
```

If the title is valid, it will be returned along with a `nil` error value. If the title is invalid, the function will write a "404 Not Found" error to the HTTP connection, and return an error to the handler. To create a new error, we have to import the `errors` package.

Let's put a call to `getTitle` in each of the handlers:

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    renderTemplate(w, "edit", p)
}

func saveHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    err = p.save()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

Introducing Function Literals and Closures

Catching the error condition in each handler introduces a lot of repeated code. What if we could wrap each of the handlers in a function that does this validation and error checking?

Go's function literals provide a powerful means of abstracting functionality that can help us here.

First, we re-write the function definition of each of the handlers to accept a title string:

```
func viewHandler(w http.ResponseWriter, r *http.Request, title string)
func editHandler(w http.ResponseWriter, r *http.Request, title string)
func saveHandler(w http.ResponseWriter, r *http.Request, title string)
```

Now let's define a wrapper function that *takes a function of the above type*, and returns a function of type `http.HandlerFunc` (suitable to be passed to the function `http.HandleFunc`):

```
func makeHandler(fn func (http.ResponseWriter, *http.Request, string)) http.Handl
        return func(w http.ResponseWriter, r *http.Request) {
                // Here we will extract the page title from the Request,
                // and call the provided handler 'fn'
        }
}
```

The returned function is called a closure because it encloses values defined outside of it. In this case, the variable `fn` (the single argument to `makeHandler`) is enclosed by the closure. The variable `fn` will be one of our save, edit, or view handlers.

Now we can take the code from `getTitle` and use it here (with some minor modifications):

```
func makeHandler(fn func(http.ResponseWriter, *http.Request, string)) http.Handle
    return func(w http.ResponseWriter, r *http.Request) {
        m := validPath.FindStringSubmatch(r.URL.Path)
        if m == nil {
            http.NotFound(w, r)
            return
        }
        fn(w, r, m[2])
    }
}
```

The closure returned by `makeHandler` is a function that takes an `http.ResponseWriter` and `http.Request` (in other words, an `http.HandlerFunc`). The closure extracts the `title` from the request path, and validates it with the `validPath` regexp. If the `title` is invalid, an error will be written to the `ResponseWriter` using the `http.NotFound` function. If the `title` is valid, the enclosed handler function `fn` will be called with the `ResponseWriter`, `Request`, and `title` as arguments.

Now we can wrap the handler functions with `makeHandler` in `main`, before they are registered with the `http` package:

```
func main() {
    http.HandleFunc("/view/", makeHandler(viewHandler))
    http.HandleFunc("/edit/", makeHandler(editHandler))
    http.HandleFunc("/save/", makeHandler(saveHandler))
```

```
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Finally we remove the calls to `getTitle` from the handler functions, making them much simpler:

```
func viewHandler(w http.ResponseWriter, r *http.Request, title string) {
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request, title string) {
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    renderTemplate(w, "edit", p)
}

func saveHandler(w http.ResponseWriter, r *http.Request, title string) {
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    err := p.save()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

Try it out!

Click here to view the final code listing.

Recompile the code, and run the app:

```
$ go build wiki.go
$ ./wiki
```

Visiting http://localhost:8080/view/ANewPage should present you with the page edit form. You should then be able to enter some text, click 'Save', and be redirected to the newly created page.

Other tasks

Here are some simple tasks you might want to tackle on your own:

- Store templates in `tmpl/` and page data in `data/`.
- Add a handler to make the web root redirect to `/view/FrontPage`.

- Spruce up the page templates by making them valid HTML and adding some CSS rules.
- Implement inter-page linking by converting instances of `[PageName]` to `<a href="/view/PageName">PageName</a>`. (hint: you could use `regexp.ReplaceAllFunc` to do this)