Writing Web Application with Templates

This lesson shows how to build a wiki application which is more detailed than the previous one. To make this application quickly and effectively, a template package is used from the Go standard library.

Building a wiki application

The following program is a working web-application for a **wiki**, which is a collection of pages that can be viewed, edited, and saved in under 100 lines of code. It is the code lab wiki tutorial from the Go site, one of the best Go-tutorials in existence. It is certainly worthwhile to work through the complete code lab to see and better understand how the program builds up. Here, we will give a complimentary description of the program in its totality, from a top-down view.

The program is a web server, so it must be started on the command-line; let's say on port 3000. The browser can ask to view the content of a wiki-page with a URL, like https://1dkne4jl5mmmm.educative.run/view/page1.

The text of this page is then read from a file and shown on the web page; this includes a hyperlink to edit the wiki page (https://1dkne4jl5mmmm.educative.run/edit/page1). The edit page shows the contents in a text frame. The user can change the text and save it to its file with a submit button **Save**; then, the same page with the modified content is viewed. If the page that is asked for viewing does not exist (e.g., https://1dkne4jl5mmmm.educative.run/edit/page999), the program detects this and redirects immediately to the edit-page, so that the new wiki-page can be made and saved.

The wiki-page needs a title and text content. It is modeled in the program by the following struct, where the content is a slice of bytes named `Body` :

```
type Page struct {
Title string

Body []byte
}
```

In order to maintain our wiki-pages outside of the running program, we will use simple text-files as persistent storage.

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

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

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

> **Remark:** Change **line 92** to `log.Fatal(http.ListenAndServe(":8080", nil))` if you're running this program locally.

Let us go through the code:

- First we import the necessary packages, `http` of course, because we are going to construct a web server but also `io/ioutil` for easy reading and writing of the files, `regexp` for validating URL-input, and `template` for dynamically creating our Html-files. The package `template` (html/template) implements data-driven templates for generating HTML output, which are safe against code injection. It provides the same interface as the package `text/template` and should be used instead of `text/template` whenever the output is HTML.

- We want to stop hackers input which could harm our server, so we will check the user input in the browser URL (which is the title of the wiki-page) with the following regular expression:

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

This will be controlled in the function `makeHandler`.

- We must have a mechanism to insert our `Page` structs into the title and content of a web page; this is done as follows with the use of the `template` package:

  - First, make the html-templatefile(s) in an editor, e.g., **view.html**:

    ```
    <h1>{{.Title |html}}</h1>
    <p>[<a href="/edit/{{.Title |html}}">edit</a>]</p>
    <div>{{printf "%s" .Body |html}}</div>
    ```

    The fields, which are to be inserted from a data-structure, are put between **{{ }}**, here `{{.Title|html}}` and `{{printf "%s" .Body |html}}`, from the `Page` struct (of course this can be very complex html. However, it is simplified as much as possible to show the principle.

  - The `template.Must(template.ParseFiles("edit.html", "view.html"))` function transforms this into a `*template.Template`. For efficiency reasons, we only do this parsing once in our program.

  - In order to construct the page out of the template and the struct, we must use the function:

    ```
    templates.ExecuteTemplate(w, tmpl+".html", p)
    ```

    It is called upon a template, gets the `Page` struct `p` to be substituted in the template as a parameter, and writes to the `ResponseWriter w`. This function must be checked on its error-output. In case there is an error, we call `http.Error` to signal this. This code will be called many times in our application, so we extract it into a separate function `renderTemplate`.

- In `main()`, our web server is started with `ListenAndServe` on port 3000, but we first define some handler functions for URLs, which start with view, edit, and save after https://1dkne4jl5mmmm.educative.run/. In

most web server applications, this forms a series of URL-paths with handler-functions analogous to a routing table in MVC frameworks like Ruby and Rails, Django or ASP.NET MVC. The requested URL is matched with these paths. The longer paths match first; if not matched with anything else, the handler for `/` is called.

Here, we need to define 3 handlers, and because this setup contains repetitive code, we isolate it in a `makeHandler` function. This is a rather special higher-order function, which is well worth studying. It has a function as its first parameter, and it returns a function which is a closure:

```go
func makeHandler(fn func(http.ResponseWriter, *http.Request, string))
 http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    if !validPath.MatchString(r.URL.Path) {
      fmt.Fprintf(w, "Wrong URL!", r.URL.Path)
      return
    }
    title := r.URL.Path[lenPath:]
    if title == "" {
      http.NotFound(w, r)
        return
    }
    fn(w, r, title)
  }
}
```

This closure takes the enclosing function variable `fn` in order to construct its return value, but before doing that, it validates the URL input with `validPath.MatchString(r.URL.Path)`. If the URL does not start with edit, save or view, and does not consist out of letters and digits, a `NotFound` error is signaled (test this with, e.g., https://1dkne4jl5mmmm.educative.run/view/page++). The `viewhandler`, `edithandler` and `savehandler`, which are the parameters for `makeHandler` in `main()`, must all be of the same type as `fn`.

- The `viewhandler` tries to read a text file with the given title; this is done through the `load()` function which constructs the filename and reads the file with `ioutil.ReadFile`. If the file is found its contents go into a local string body. A pointer to a `Page` struct is made literally with it:

```
&Page{Title: title, Body: body}
```

This is returned to the caller together with *nil* for the error.

The struct is then merged with the template with `renderTemplate`. In case of an error, which means the wiki-page does not yet exist on disk, the error is returned to `viewHandler()`, where an automatic redirect is done to request an edit-page with that title.

- The `edithandler` is almost the same: try to load the file. If found, render the edit-template with it; in case of an error, make a new `Page` object with that title and render it also.

- Saving a page content is done through the **Save** button on the edit-page. This button resides in the Html form, which starts with:

  ```
  <form action="/save/{{.Title}}" method="POST">
  ```

  This means that when posting, a request with a URL of the form https://1dkne4jl5mmmm.educative.run/save/{Title} (with the title substituted through the template) is sent to the web server. For such a URL, we have defined a handler function `saveHandler()`. With the `FormValue()` method of the request, it extracts the contents of the text area-field named body, constructs a `Page` object with this info and tries to store this page with the `save()` function. In case this fails, an `http.Error` is returned to be shown in the browser. When it succeeds, the browser is redirected to view the same page. The `save()` function is very simple. Write the `Body` field of the `Page` struct in a file called filename with the function `ioutil.WriteFile()`. It uses `{{ printf "%s" .Body|html}}`.

> **Remark:** Try the following URLs; they should work if you're running the code locally: http://localhost:8080/view/page1, http://localhost:8080/edit/page1, http://localhost:8080/view/page999, http://localhost:8080/edit/page999, http://localhost/save/{Title}

That’s how the `template` package helps in devising such an application. Let’s explore further functionalities provided by this package in the next lesson.