

Temporada 02: Go web

Por [Uriel Ruelas](https://codingornot.com/author/zainiaz07)

En la [primera temporada de Go to go](http://codingornot.com/temporada-01-go-to-go) aprendimos la sintaxis general del lenguaje, palabras reservadas y tipos de datos, para crear nuestros primeros programas de prueba. En esta temporada aplicamos muchos de los conocimientos previamente adquiridos para crear nuestro primer servidor web.

[01. Go web: ¿cómo crear aplicaciones web con Go?](http://codingornot.com/01-go-web-como-crear-aplicaciones-web-con-go)  
[02. Go web: ¿cómo hacer peticiones HTTP con Golang?](http://codingornot.com/02-go-web-como-hacer-peticiones-http-con-golang)  
[03. Go web: ¿cómo servir páginas con Golang?](http://codingornot.com/03-go-web-como-servir-paginas-con-golang)  
[04. Go web: ¿cómo crear un editor de páginas con Golang? Parte 1 de 2](http://codingornot.com/04-go-web-como-crear-un-editor-de-paginas-con-golang-parte-1-de-2)  
[05. Go web: ¿cómo crear un editor de páginas con Go? Parte 2 de 2](http://codingornot.com/05-go-web-como-crear-un-editor-de-paginas-con-go-parte-2-de-2)  
[06. Go web: Cargar plantillas HTML eficientemente en Golang](http://codingornot.com/06-go-web-cargar-plantillas-html-eficientemente-en-golang)  
[07. Go web: expresiones regulares (regex) en Golang](http://codingornot.com/07-go-web-expresiones-regulares-regex-en-golang)  
[08. Go web: closures en Golang](http://codingornot.com/08-go-web-closures-en-golang)  
[09. Go web: crear y redireccionar a la página principal](http://codingornot.com/09-go-web-crear-y-redireccionar-a-la-pagina-principal)  
[10. Go web: enlaces a página anterior y siguiente](http://codingornot.com/10-go-web-enlaces-a-pagina-anterior-y-siguiente)

01. Go web: ¿cómo crear aplicaciones web con Go?

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

Gracias a la [primera temporada de Golang](http://codingornot.com/temporada-01-go-to-go/), ahora ya eres capaz de crear fácilmente pequeños programas de uso general. Muy bien, creo que es momento de retomar la programación con Go pero en esta ocasión para un fin en específico: crear aplicaciones web.

¿Quieres saber para qué es bueno Go con ejemplos reales? Ha llegado el momento de saberlo. Al crear aplicaciones web pondrás a prueba, entre otras cosas, los paquetes de Go, su concurrencia, su velocidad y la facilidad para crear programas que, en otros lenguajes, serían muy complicados de desarrollar.

Nota: en esta entrada no comenzaremos de lleno con temas web. Primero aprenderemos a crear y respaldar registros (algo fundamental para poder continuar).

Toda la información aquí presentada está basada en la documentación oficial del lenguaje. Puedes consultar la fuente original dando clic [aquí](https://golang.org/doc/articles/wiki/).

Conocimientos y requisitos previos

Para comprender estas entradas se requiere que tengas nociones básicas de programación y de tecnologías web. A cada momento asumo que tienes conocimientos básicos de comandos de terminal, si no es así, te recomiendo que revises [estas entradas](http://codingornot.com/category/serie-terminal-101/). Si no utilizas Linux o Mac, no te preocupes, cualquier cosa que se haga desde terminal la puedes hacer desde línea de comandos de Windows con pocos o ningún cambio.

Aunque resulte obvio, necesitas tener instalado Go en tu equipo y agregar la carpeta de sus binarios a tu PATH (puedes consultar más información al respecto [aquí](http://codingornot.com/01-go-to-go-iniciando-go-for-the-lulz/)).

Creación de páginas

Vamos a crear páginas estilo wiki para ejemplificar un poco el uso de algunas funciones disponibles en los paquetes de Go.

Lo primero que haremos es, desde terminal (o el gestor de archivos de tu preferencia), crearemos un directorio llamado ‘gowiki’ y nos moveremos dentro de él.

1. mkdir gowiki
2. cd gowiki

Ahora creamos un archivo llamado ‘wiki.go’ dentro de la carpeta ‘gowiki’. Ábrelo y agrega el siguiente código:

1. package main
2. import (
3. "fmt"
4. "io/ioutil"
5. )

Nota: el paquete fmt, como ya expliqué anteriormente, nos proporciona funcionalidades de entradas y salidas con formato. En cambio, ioutil nos brinda funciones de entrada y salida para archivos, directorios, etc..

Registros de datos

Las wiki consisten en páginas interconectadas que tienen un título y un cuerpo (o contenido). Utilizaremos registros (struct) con dos miembro para representar esos campos:

1. //Los registros representan páginas
2. type Pagina struct{
3. Titulo string
4. Cuerpo []byte
5. }

Nota: la razón por la que usamos un slice (trozo) en lugar de un string para el cuerpo, será aclarada un poco más adelante.

Las páginas se almacenarán en la memoria principal durante la ejecución. Si queremos mantener una copia en disco necesitamos crear el método guardar(); sería poco práctico que al cerrar nuestro programa no haya un respaldo de nuestras páginas:

1. //Este método almacenará páginas en disco duro
2. func (p\* Pagina) guardar() error{
3. nombre := p.Titulo + ".txt"
4. return ioutil.WriteFile(nombre, p.Cuerpo, 0600)
5. }

El método anterior puede ser llamado por registros tipo Pagina. No recibe parámetros y regresa un valor tipo error. En el cuerpo del método está codificado el algoritmo para almacenar el cuerpo de una página en un archivo de texto con el nombre del título que esta posee. La función WriteFile() escribe un slice a un archivo, por eso nuestro miembro Cuerpo de los registros Pagina también fue declarada con ese tipo (en lugar de string, como mencionaba anteriormente).

El método regresa un dato tipo error porque es, a su vez, el que retorna la función WriteFile(). Esto es muy útil para que sea procesado correctamente por nuestra aplicación si ocurre algo malo. Si la función escribe correctamente la información, regresará nil (el valor cero).

El octal 0600 de argumento del método es usado para indicar que el archivo creado tendrá permisos de lectura y escritura únicamente para el usuario actual. Si quieres leer más al respecto, puedes consultar el manual de Unix para más detalles.

Leer páginas

Si almacenamos páginas, es de suponer que también queremos leerlas. La siguiente función realiza ese trabajo:

1. func cargarPagina(titulo string) (\*Pagina, error) {
2. nombre\_archivo = titulo + ".txt"
3. cuerpo, err := ioutil.ReadFile(nombre\_archivo)
4. if err != nil {
5. return nil, err
6. }
7. return &Pagina{Titulo: titulo, Cuerpo: cuerpo}, nil
8. }

Esta función recibe de parámetro el nombre de la página a cargar; lee al archivo correspondiente en disco y guarda la información leída junto con un código de error. Si ocurrió algo malo, regresa nil y el dato de lo que falló . Si todo se realizó correctamente, regresa el apuntador a una literal tipo Pagina construida con sus respectivos título y cuerpo en ese momento, además de también regresar nil para señalar que no hubo error.

Recordarás que en la [entrada de funciones](http://codingornot.com/08-go-to-go-funciones-y-ambitos/) detallé que es posible que una función regrese dos valores en Golang. Es exactamente eso lo que hace la función ReadFile(); regresa los datos leídos y un código de error. Si aún no has validado los errores (o no quieres hacerlo), puedes usar el identificador en blanco (\_) para desecharlos ya que todo lo que se intente guardar en él será eliminado, sin embargo, lo recomendable es que al menos valides si ocurrió un error (sin distinción de cuál fue) porque eso significa que no se pudo leer el archivo correctamente. Si la variable devuelta err no es igual nil, no se creará la página y la función regresará el código del error ocurrido. Más adelante puedes crear sistemas de manejo de errores más eficientes y robustos pero por el momento nos limitaremos a detectar un error para evitar que falle nuestra aplicación.

Hasta este punto ya tenemos una estructura de datos simple que almacena y carga páginas desde archivos. El siguiente ejemplo muestra todo lo visto en esta entrada en un único código funcional:

1. package main
2. import ("fmt"; "io/ioutil")
3. //Registros que guardan páginas
4. type Pagina struct{
5. Titulo string
6. Cuerpo []byte
7. }
8. func main(){
9. //Creamos y guardamos una página en disco.
10. pag1 := &Pagina{Titulo: "Prueba 1", Cuerpo: []byte("Este es el cuerpo")}
11. pag1.guardar()
12. //Cargamos página guardada
13. pag2, \_ := cargarPagina("Prueba 1")
14. fmt.Println(string(pag2.Cuerpo))
15. }
16. //Método de registros Pagina, almacena el cuerpo
17. func ( p\* Pagina ) guardar() error {
18. nombre := p.Titulo + ".txt"
19. return ioutil.WriteFile( nombre, p.Cuerpo, 0600)
20. }
21. //Función que carga una página desde disco
22. func cargarPagina( titulo string ) (\*Pagina, error) {
23. nombre\_archivo := titulo + ".txt"
24. cuerpo, err := ioutil.ReadFile( nombre\_archivo )
25. if err != nil {
26. return nil, err
27. }
28. return &Pagina{Titulo: titulo, Cuerpo: cuerpo}, nil
29. }

Salida:  
Este es el cuerpo

Finalizando…

Este es solamente el principio para aprender a programar aplicaciones web con Go. En la siguiente entrada revisaremos un poco de puertos y peticiones HTTP.

02. Go web: ¿cómo hacer peticiones HTTP con Golang?

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

Es hora de comenzar a usar los paquetes web que vienen incluidos con Golang. En esta entrada crearemos un pequeño servidor que atenderá peticiones HTTP. Sin más que agregar, comencemos.

Creando un servidor

Lo primero que haremos es crear un archivo con el nombre ‘**codingornot.go’** (nombre opcional), que será nuestro servidor en cuestión. En seguida, importamos los paquetes necesarios para nuestro programa, como esta será una versión de prueba con la única finalidad de familiarizarnos un poco con las peticiones y respuestas HTTP, solamente necesitaremos el paquete de salidas con formato (**fmt**) y un paquete para las peticiones HTTP (**net/http**):

1. import (
2. "fmt"
3. "net/http"
4. )

En seguida creamos una función que será la encargada de manejar las peticiones. En este caso la llamaré **manejador()** pero tú puedes usar el nombre que creas conveniente:

1. func manejador(w http.ResponseWriter, r \*http.Request){
2. fmt.Fprintf(w,"Hola, %s, ¡este es un servidor!", r.URL.Path)
3. }

Esta función recibe como primer argumento un dato tipo **http.ResponseWriter** que forma parte del paquete **net/http**. Este se encarga de construir la respuesta del servidor, es decir, es el que envía los mensajes al cliente HTTP.

El segundo argumento de **manejador()** es un tipo **http.Request**, es decir, son las peticiones del cliente HTTP.

Dentro del cuerpo de la función escribimos un mensaje a **w** (la respuesta para el cliente) por medio de **fmt.Fprintf()**. En este caso escribimos un saludo junto con la ruta de la solicitud, la cual puede ser consultada con **r.URL.Path**. La ruta en este caso será:“**/codingornot”**, si quisieras que solamente apareciera el nombre, puedes crear un *slice* con el nombre de la ruta a partir del segundo caracter (**r.URL.Path[1:]** devolvería “**codingornot”** en lugar de “**/codingornot”**).

Ahora creamos la función principal (**main()**) desde la cual haremos el llamado a **manejador()**:

1. func main(){
2. http.HandleFunc("/", manejador)
3. http.ListenAndServe(":8080", nil)
4. }

Primero hacemos llamado a **http.HandleFunc()** para que el paquete **http** se encargue de las peticiones hechas a la web raíz (“**/”**) por medio de nuestra, previamente construida, función **manejador()**.

En seguida, llamamos a la función **http.ListenAndServe()** para que el programa se mantenga bloqueado escuchando peticiones en el puerto 8080 (“**:8080”**) hasta que termine su ejecución. El uso del parámetro **nil** será explicado en entradas posteriores, por el momento dejemos esa parte así.

Programa funcional

El siguiente programa reúne todo lo visto en esta entrada:

1. package main
2. import (
3. "fmt"
4. "net/http"
5. )
6. func manejador(w http.ResponseWriter, r \*http.Request){
7. fmt.Fprintf(w,"Hola, %s, ¡este es el servidor!", r.URL.Path)
8. }
9. func main(){
10. http.HandleFunc("/", manejador)
11. fmt.Println("El servidor se encuentra en ejecución")
12. http.ListenAndServe(":8080", nil)
13. }

Compilamos y ejecutamos el programa:

1. go build codingornot.go
2. ./codingornot

Para verificar que funciona correctamente, accedemos a la siguiente URL:

http://localhost:8080/codingornot

Se abrirá una página con el siguiente mensaje:

Hola, /codingornot, ¡este es el servidor!

Finalizando…

En la siguiente entrada aprenderemos a servir las páginas *wiki* que creamos en la [entrada anterior](http://codingornot.com/01-go-web-como-crear-aplicaciones-web-con-go/) usando el paquete **net/http**.

03. Go web: ¿cómo servir páginas con Golang?

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

En la primera entrada aprendimos a crear páginas estilo *wiki* con Golang, mientras que en la segunda entrada creamos un pequeño servidor web. En esta ocasión vamos a combinar lo visto en ambas entradas para servir páginas *wiki*.

Servidor de páginas *wiki*

Comenzamos creando un archivo con extensión **.go** que será el servidor (eres libre de nombrarlo como quieras, en mi caso le llamaré ‘**servidor\_wikis’**. Lo abrimos e importamos los 3 paquetes que vamos a necesitar:

1. import (
2. "fmt"
3. "io/ioutil"
4. "net/http"
5. )

Si no recuerdas por qué importamos alguno de esos paquetes, te recomiendo que revises las entradas anteriores para mayor información (**io/ioutil** [aquí](http://codingornot.com/01-go-web-como-crear-aplicaciones-web-con-go/), **net/http** [aquí](http://codingornot.com/02-go-web-como-hacer-peticiones-http-con-golang/)).

En seguida vamos a crear la función manejadora (*handler*) que se encargará de mostrar las páginas *wiki*.

1. func manejadorMostrarPagina(w http.ResponseWriter, r \*http.Request){
2. titulo := r.URL.Path
3. pagina, \_ := cargarPagina(titulo)
4. fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>",
5. pagina.Titulo[len("/view/"):], pagina.Cuerpo)
6. }

Con **r.URL.Path** extraemos la ruta y el título de la página desde la URL que pide el cliente. Yo escogí la ruta “***/view”*** para mis páginas, tú puedes usar cualquier otra que desees.

En seguida, la función carga la página, le agrega etiquetas HTML para dar formato y, finalmente, la guarda en **w**(la respuesta para el cliente). En mi caso creo un *slice* con el título de la página porque, de otro modo, incluiría el prefijo “***/view/”*** en referencia a la ruta de acceso.

Para hacer el llamado a este manejador desde la función principal (**main()**) inicializamos el paquete **http** para que escuche peticiones hechas a la ruta “***/view/”*** y las sirva con **manejadorMostrarPagina()**. El puerto que escuchará nuestro servidor será el **8080**:

1. func main(){
2. http.HandleFunc("/view/", manejadorMostrarPagina)
3. http.ListenAndServe(":8080", nil)
4. }

Para compilar y ejecutar desde terminal de Linux:

1. go build servidor\_wikis.go
2. ./servidor\_wikis

El siguiente es el código completo del servidor junto con las *wiki*:

1. package main
2. import (
3. "fmt"
4. "io/ioutil"
5. "net/http"
6. )
7. type Pagina struct{
8. Titulo string
9. Cuerpo []byte
10. }
11. func main(){
12. //Creamos y guardamos una página para que el cliente la pida
13. pag1 := &Pagina{ Titulo: "Ejemplo", Cuerpo:[]byte(
14. "¡Hola personita! Este es el cuerpo de tu página.") }
15. pag1.guardar()
16. http.HandleFunc("/view/", manejadorMostrarPagina)
17. fmt.Println("El servidor se encuentra en ejecución");
18. http.ListenAndServe(":8080", nil)
19. }
20. //Manejador de peticiones
21. func manejadorMostrarPagina(w http.ResponseWriter, r \*http.Request){
22. titulo := r.URL.Path
23. pagina, \_ := cargarPagina(titulo)
24. fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>",
25. pagina.Titulo[len("/view/"):], pagina.Cuerpo)
26. fmt.Println("¡Página servida!")
27. }
28. //Método para guardar página
29. func ( p\* Pagina ) guardar() error {
30. nombre := p.Titulo + ".txt"
31. return ioutil.WriteFile( "./view/" + nombre, p.Cuerpo, 0600)
32. }
33. //Método para cargar página
34. func cargarPagina( titulo string ) (\*Pagina, error) {
35. nombre\_archivo := titulo + ".txt"
36. fmt.Println("El cliente ha pedido:" + nombre\_archivo)
37. cuerpo, err := ioutil.ReadFile( "." + nombre\_archivo )
38. if err != nil {
39. return nil, err
40. }
41. return &Pagina{Titulo: titulo, Cuerpo: cuerpo}, nil
42. }

**Nota importante:** recuerda que es necesario crear una carpeta con el nombre ‘***view’*** para que este ejemplo funcione. Si tu usaste otra ruta, será indispensable que modifiques el código además de crear las respectivas carpetas.

Para mostrar nuestra página, accedemos al siguiente enlace (ten cuidado de ingresar la URL tal como aparece a continuación):

<http://localhost:8080/view/Ejemplo>

Finalizando…

Espero que te haya servido lo que estudiamos en esta entrada. A pesar de ser bastante sencillo crear un servidor, te recomiendo que practiques un poco para dominarlo. En la siguiente entrada te mostraré cómo crear un editor simple para tus páginas *wiki*.

04. Go web: ¿cómo crear un editor de páginas con Golang? Parte 1 de 2

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

En esta entrada vamos a crear un editor simple para nuestras *wiki*. Utilizaremos un nuevo paquete que nos proporciona Golang para trabajar con etiquetas HTML.

Comenzando…

Ya no es necesario utilizar el paquete **fmt** para imprimir en consola, puedes eliminarlo si lo deseas; en mi caso, lo seguiré utilizando para imprimir un par de líneas de ayuda durante la ejecución del servidor pero queda a tu consideración.

Haremos uso del paquete **html/template** para crear plantillas de etiquetas HTML en archivos separados de nuestro código fuente. Los paquetes que importaremos quedarán como a continuación:

1. import (
2. "fmt"
3. "io/ioutil"
4. "net/http"
5. "html/template"
6. )

Vamos a necesitar dos nuevos manejadores de eventos: uno mostrará un área de texto para editar *wikis*, el otro nos permitirá guardar los datos del área de texto. El primero se podrá acceder por medio de la ruta “*/edit/”*, mientras que el segundo a través de“*/save/”*. En esta entrada nos centraremos en el editor, en la siguiente entrada agregaremos la función de guardar los cambios.

Nuestra función principal lucirá así:

1. func main(){
2. pag1 := &Pagina{Titulo: "Ejemplo", Cuerpo:[]byte(
3. "¡Hola personita! Este es el cuerpo de tu página.")}
4. pag1.guardar()
5. http.HandleFunc("/view/", manejadorMostrarPagina)
6. http.HandleFunc("/edit/", manejadorEditar)
7. //http.HandleFunc("/save/", manejadorGuardar)
8. http.ListenAndServe(":8080", nil)
9. }

Comentamos el llamado al manejador guardar para evitar errores. En la segunda parte vamos a quitar las diagonales y a desarrollar la función **manejadorGuardar()**. La primera plantilla HTML que necesitamos crear se llamará **edit.html**. Creamos el archivo y agregamos las siguientes líneas:

1. <h1>Editando: {{.Titulo}}</h1>
2. <form action="/save/{{.Titulo}}" method="POST">
3. <div><textarea name="body" rows="20" cols="80">{{printf "%s" .Cuerpo}}</textarea></div>
4. <div><input type="submit" value="Guardar"></div>
5. </form>

Las directivas de las plantillas son encerradas entre dos llaves (esa información provendrá de nuestras *wikis*). La instrucción **printf “%s” .Cuerpo** llama al cuerpo de la página (**p.Cuerpo**) como una cadena (*string*) en lugar de una cadena de bytes, similar a llamar a **fmt.Printf()**. Las demás son etiquetas HTML de uso general.

Ahora creamos el cuerpo de la función **manejadorEditar()**:

1. func manejadorEditar(w http.ResponseWriter, r \*http.Request){
2. titulo := r.URL.Path[len("/edit/"):]
3. p, err := cargarPagina(titulo)
4. if err != nil{
5. p = &Pagina{Titulo: titulo}
6. }
7. t, \_ := template.ParseFiles("edit.html")
8. t.Execute(w, p)
9. }

La función va a leer el contenido de **edit.html** y regresar un **\*template.Template**. El método **t.Execute()** ejecuta la plantilla y escribe el HTML generado a **http.ResponseWriter**.

El paquete **html/template** nos ayuda a garantizar que se genera un HTML correcto y seguro; por ejemplo, automáticamente reemplaza algunos signos como el mayor que (**>**) por **&gt;** para garantizar que los datos del usuario no corrompan el HTML (quienes hayan usado **WordPress** alguna vez, entenderán este problema de reemplazo de caracteres), el paquete nos evita algunos problemas con la representación de caracteres.

En seguida, vamos a modificar el **manejadorMostrarPagina()** que hicimos en entradas anteriores para que haga uso de un archivo HTML como el editor (no quedaría muy bien que solo algunos los usaran y otros no):

1. func manejadorMostrarPagina(w http.ResponseWriter, r \*http.Request){
2. titulo := r.URL.Path[len("/view/"):]
3. p, \_ := cargarPagina(titulo)
4. t, \_ := template.ParseFiles(titulo + ".html")
5. t.Execute(w, p)
6. }

Creamos el archivo **view.html** correspondiente:

1. <h1>{{.Titulo}}</h1>
2. <p>[<a href="/edit/{{.Titulo}}">edit</a>]</p>
3. <div>{{printf "%s" .Cuerpo}}</div>

¿Notaste que utilizamos instrucciones similares en ambos manejadores? Movamos las instrucciones duplicadas a una función llamada **cargarPlantilla()**.

1. func cargarPlantilla(w http.ResponseWriter, plantilla string, p \*Pagina){
2. t, \_ := template.ParseFiles(plantilla + ".html")
3. t.Execute(w, p)
4. }

Modifiquemos los manejadores para que hagan uso de nuestra nueva función:

1. func manejadorMostrarPagina(w http.ResponseWriter, r \*http.Request){
2. titulo := r.URL.Path[len("/view/"):]
3. p, \_ := cargarPagina(titulo)
4. cargarPlantilla(w, "view", p)
5. }
6. func manejadorEditar(w http.ResponseWriter, r \*http.Request){
7. titulo := r.URL.Path[len("/edit/"):]
8. p, err := cargarPagina(titulo)
9. if err != nil{
10. p = &Pagina{Titulo: titulo}
11. }
12. cargarPlantilla(w, "edit", p)
13. }

Servidor funcional

El código del servidor completo quedaría de la siguiente forma:

1. package main
2. import (
3. "fmt"
4. "io/ioutil"
5. "net/http"
6. "html/template"
7. )
8. type Pagina struct{
9. Titulo string
10. Cuerpo []byte
11. }
12. func main(){
13. //Creamos y guardamos una página para que el cliente la pida
14. pag1 := &Pagina{Titulo: "Ejemplo", Cuerpo:[]byte(
15. "¡Hola personita! Este es el cuerpo de tu página.")}
16. pag1.guardar()
17. http.HandleFunc("/view/", manejadorMostrarPagina)
18. http.HandleFunc("/edit/", manejadorEditar)
19. //http.HandleFunc("/save/", manejadorGuardar)
20. fmt.Println("El servidor se encuentra en ejecución");
21. http.ListenAndServe(":8080", nil)
22. }
23. //Método para guardar página
24. func ( p\* Pagina ) guardar() error {
25. nombre := p.Titulo + ".txt"
26. return ioutil.WriteFile( "./view/" + nombre, p.Cuerpo, 0600)
27. }
28. //Método para cargar página
29. func cargarPagina( titulo string ) (\*Pagina, error) {
30. nombre\_archivo := titulo + ".txt"
31. fmt.Println("El cliente ha pedido:" + nombre\_archivo)
32. cuerpo, err := ioutil.ReadFile( "./view/" + nombre\_archivo )
33. if err != nil {
34. return nil, err
35. }
36. return &Pagina{Titulo: titulo, Cuerpo: cuerpo}, nil
37. }
38. //Carga las plantillas HTML
39. func cargarPlantilla(w http.ResponseWriter, plantilla string, p \*Pagina){
40. t, \_ := template.ParseFiles(plantilla + ".html")
41. t.Execute(w, p)
42. }
43. //Manejador de peticiones
44. func manejadorMostrarPagina(w http.ResponseWriter, r \*http.Request){
45. titulo := r.URL.Path[len("/view/"):]
46. p, \_ := cargarPagina(titulo)
47. cargarPlantilla(w, "view", p)
48. }
49. //Manejador para editar wikis
50. func manejadorEditar(w http.ResponseWriter, r \*http.Request){
51. titulo := r.URL.Path[len("/edit/"):]
52. p, err := cargarPagina(titulo)
53. if err != nil{
54. p = &Pagina{Titulo: titulo}
55. }
56. cargarPlantilla(w, "edit", p)
57. }

Compila y ejecuta el servidor como lo hemos hecho anteriormente:

1. go build servidor\_wikis.go
2. ./servidor\_wikis

Para verificar que funciona tu servidor, accede a la siguiente dirección:

<http://localhost:8080/view/Ejemplo>

Deberá de aparecer un enlace para editar tu *wiki* (si intentas guardar los cambios con el botón “*Guardar”* que aparecerá, ocurrirá un error ya que hasta la [segunda parte](http://codingornot.com/05-go-web-como-crear-un-editor-de-paginas-con-go-parte-2-de-2/) agregaremos el código de esa funcionalidad).

Finalizando…

En esta entrada aprendiste a utilizar plantillas HTML y a crear un editor simple de *wikis*. En la siguiente entrega te enseñaré a agregar la funcionalidad de guardar los cambios que hagas en tus *wikis*.

05. Go web: ¿cómo crear un editor de páginas con Go? Parte 2 de 2

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

En la entrada anterior creamos un pequeño editor para páginas *wiki* pero nos faltó agregar el código de la función que guardaría los cambios hechos; en esta entrada completaremos esa funcionalidad.

Comenzamos

Primeramente, vamos a quitar las diagonales del llamado al **manejadorGuardar()** en nuestra función principal(**main()**) del servidor de la [entrada anterior](http://codingornot.com/04-go-web-como-crear-un-editor-de-paginas-con-golang-parte-1-de-2/).

En seguida, creamos la función que se encargará de guardar *wikis*:

1. //Manejador para guardar wikis
2. func manejadorGuardar(w http.ResponseWriter, r \* http.Request) {
3. titulo := r.URL.Path[len("/save/"):]
4. cuerpo := r.FormValue("body")
5. p := &Pagina{Titulo: titulo, Cuerpo: []byte(cuerpo)}
6. fmt.Println("Guardando " + titulo + ".txt...")
7. p.guardar()
8. http.Redirect(w, r, "/view/" + titulo, http.StatusFound)
9. }

Además de extraer el nombre de la *wiki* que intentamos guardar, almacenamos el único componente“***body”***(cuerpo) del formulario(***form***) o área de texto que se mostraba para editar el texto de las páginas. A continuación creamos un registro de tipo Pagina con los miembros Titulo y Cuerpo correspondientes extraídos de la petición y el formulario respectivamente.

Acceso a páginas que no existen

Si intentas acceder a una página que no existe(por ejemplo, <http://localhost:8080/view/Prueba>) se cargará una página completamente en blanco en nuestro navegador debido a que el servidor simplemente ignora el error que regresa la función **cargarPagina()** y procede a utilizar datos en blanco. Algo razonable es que se te redireccione al editor para crear una página nueva cuando intentes acceder a una que no existe (en entradas posteriores trabajaremos con otras soluciones). Para lograr eso agregaremos unas cuantas líneas de código extra a la función **manejadorMostrarPagina()** para que quede de la siguiente forma:

1. //Manejador de peticiones
2. func manejadorMostrarPagina(w http.ResponseWriter, r \*http.Request){
3. titulo := r.URL.Path[len("/view/"):]
4. p, err := cargarPagina(titulo)
5. if err != nil {
6. http.Redirect(w, r, "/edit/" + titulo, http.StatusFound)
7. fmt.Println("La página solicitada no existía. Llamando al editor...")
8. return
9. }
10. cargarPlantilla(w, "view", p)
11. }

Cuando el servidor no puede localizar la página que le pidió el cliente, almacena en la variable **err** la información del error (en este caso, si se imprimiera en consola dicha variable, obtendríamos: ***open ./view/Prueba.txt: no such file or directory***). Al identificar que la variable **err** no está vacía, es decir, que algo falló, el servidor procede a llamar a la función **http.Redirect()** con una nueva URL como argumento y un código de estado **http.StatusFound** de **302** para “decirle” al navegador del cliente que el servidor le intenta redireccionar. A partir de ahí se hace llamado a la función **manejadorEditar()** puesto que es la encargada de atender las peticiones de edición“*/edit/”*.

Servidor funcional

El siguiente es el código fuente del servidor tal y como lo hemos programado hasta este punto:

1. package main
2. import (
3. "fmt"
4. "io/ioutil"
5. "net/http"
6. "html/template"
7. )
8. type Pagina struct{
9. Titulo string
10. Cuerpo []byte
11. }
12. func main(){
13. //Creamos y guardamos una página para que el cliente la pida
14. pag1 := &Pagina{Titulo: "Ejemplo", Cuerpo:[]byte(
15. "¡Hola personita! Este es el cuerpo de tu página.")}
16. pag1.guardar()
17. http.HandleFunc("/view/", manejadorMostrarPagina)
18. http.HandleFunc("/edit/", manejadorEditar)
19. http.HandleFunc("/save/", manejadorGuardar)
20. fmt.Println("El servidor se encuentra en ejecución.");
21. http.ListenAndServe(":8080", nil)
22. }
23. //Método para guardar página
24. func ( p\* Pagina ) guardar() error {
25. nombre := p.Titulo + ".txt"
26. return ioutil.WriteFile( "./view/" + nombre, p.Cuerpo, 0600)
27. }
28. //Método para cargar página
29. func cargarPagina( titulo string ) (\*Pagina, error) {
30. nombre\_archivo := titulo + ".txt"
31. fmt.Println("El cliente ha pedido: " + nombre\_archivo)
32. cuerpo, err := ioutil.ReadFile( "./view/" + nombre\_archivo )
33. if err != nil {
34. return nil, err
35. }
36. return &Pagina{Titulo: titulo, Cuerpo: cuerpo}, nil
37. }
38. //Carga las plantillas HTML
39. func cargarPlantilla(w http.ResponseWriter, plantilla string, p \*Pagina){
40. t, \_ := template.ParseFiles(plantilla + ".html")
41. t.Execute(w, p)
42. }
43. //Manejador de peticiones
44. func manejadorMostrarPagina(w http.ResponseWriter, r \*http.Request){
45. titulo := r.URL.Path[len("/view/"):]
46. p, err := cargarPagina(titulo)
47. if err != nil {
48. http.Redirect(w, r, "/edit/" + titulo, http.StatusFound)
49. fmt.Println("La página solicitada no existía. Llamando al editor...")
50. return
51. }
52. cargarPlantilla(w, "view", p)
53. }
54. //Manejador para editar wikis
55. func manejadorEditar(w http.ResponseWriter, r \*http.Request){
56. titulo := r.URL.Path[len("/edit/"):]
57. p, err := cargarPagina(titulo)
58. if err != nil{
59. p = &Pagina{Titulo: titulo}
60. }
61. cargarPlantilla(w, "edit", p)
62. }
63. //Manejador para guardar wikis
64. func manejadorGuardar(w http.ResponseWriter, r \* http.Request) {
65. titulo := r.URL.Path[len("/save/"):]
66. cuerpo := r.FormValue("body")
67. p := &Pagina{Titulo: titulo, Cuerpo: []byte(cuerpo)}
68. fmt.Println("Guardando " + titulo + ".txt...")
69. p.guardar()
70. http.Redirect(w, r, "/view/" + titulo, http.StatusFound)
71. }

Para verificar que funciona lo compilamos y ejecutamos como siempre:

1. go build servidor\_wikis.go
2. ./servidor\_wikis

Y accedemos a la siguiente URL en la que aparecerá el editor puesto que la página solicitada no existe (ten cuidado de ingresar la URL correctamente):

<http://localhost:8080/view/PaginaQueNoExiste>

Finalizando…

Espero que esta entrada te haya servido y que hayas logrado completar el servidor con lo aprendido hasta aquí. En las siguientes entrada continuaremos haciendo más robusto nuestro servidor.

06. Go web: Cargar plantillas HTML eficientemente en Golang

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

Hasta ahora, cada vez que el cliente pedía una página a nuestro servidor, la plantilla correspondiente necesitaba ser cargada desde el disco. Esto puede llegar a ser contraproducente si, por ejemplo, muchos clientes realizaran peticiones al mismo tiempo, se necesitarían cargar las mismas plantillas una y otra vez, ocasionando retrasos en los tiempos de respuesta. En esta entrada aprenderemos cómo almacenar las plantillas en memoria para cargarlas solamente una vez, al inicio del programa.

Comenzamos

Haremos uso del, previamente incluido, paquete html/template para cargar las plantillas al principio.

Crearemos una variable tipo \*Template (apuntador a Template), tipo de dato incluido con el paquete html/template que nos servirá para almacenar las plantillas. Haremos uso del método template.ParseFiles() de la siguiente forma:

var plantillas, err = template.ParseFiles( "edit.html", "view.html" )

La variable err es necesaria porque el método template.ParseFiles() devuelve dos valores: el apuntador y una variable tipo error; sin almacenar esa segunda variable, nuestro programa no compilaría. Es buena práctica validar los errores de nuestros programas, en entradas posteriores te enseñaré a hacerlo pero por ahora no será necesario.

Cuando intentamos acceder a una plantilla que no existe, ocurre un error fatal y se despliega una serie de códigos de error en consola, pero esto sucede en ejecución y no hay forma de prevenirlo con el código que agregamos anteriormente. Si quieres que tu programa se detenga al inicio de su ejecución cuando no puede cargar una plantilla en lugar de esperar hasta que se haga una petición, puedes utilizar el método template.Must() también incluido en el paquete html/template:

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

Este método envuelve los dos datos que nos regresa template.ParseFiles(). Al momento que detecta que algo falló al cargar las plantillas, pone al programa en estado de “pánico” (panic) e inmediatamente termina su ejecución. Dejamos de utilizar la variable err porque el método template.Must() es el encargado de procesar los errores regresados por template.ParseFiles() y “filtrar” solamente el \*Template.

Finalmente, haremos unos cambios a la función cargarPlantilla() para que utilice la variable plantillas. Quedaría de la siguiente manera:

1. func cargarPlantilla( w http.ResponseWritter, nombre\_plantilla string, pagina \*Pagina) {
2. plantillas.ExecuteTemplate(w, nombre\_plantilla + ".html", pagina)
3. }

Programa completo

El siguiente es el código fuente del servidor con todas las modificaciones que hemos hecho hasta ahora:

1. package main
2. import (
3. "fmt"
4. "io/ioutil"
5. "net/http"
6. "html/template"
7. )
8. type Pagina struct{
9. Titulo string
10. Cuerpo []byte
11. }
12. var plantillas = template.Must(template.ParseFiles("edit.html", "view.html"))
13. func main(){
14. //Creamos y guardamos una página para que el cliente la pida
15. pag1 := &Pagina{Titulo: "Ejemplo", Cuerpo:[]byte(
16. "¡Hola personita! Este es el cuerpo de tu página.")}
17. pag1.guardar()
18. http.HandleFunc("/view/", manejadorMostrarPagina)
19. http.HandleFunc("/edit/", manejadorEditar)
20. http.HandleFunc("/save/", manejadorGuardar)
21. fmt.Println("El servidor se encuentra en ejecución.");
22. http.ListenAndServe(":8080", nil)
23. }
24. //Método para guardar página
25. func ( p\* Pagina ) guardar() error {
26. nombre := p.Titulo + ".txt"
27. return ioutil.WriteFile( "./view/" + nombre, p.Cuerpo, 0600)
28. }
29. //Método para cargar página
30. func cargarPagina( titulo string ) (\*Pagina, error) {
31. nombre\_archivo := titulo + ".txt"
32. fmt.Println("El cliente ha pedido: " + nombre\_archivo)
33. cuerpo, err := ioutil.ReadFile( "./view/" + nombre\_archivo )
34. if err != nil {
35. return nil, err
36. }
37. return &Pagina{Titulo: titulo, Cuerpo: cuerpo}, nil
38. }
39. //Carga las plantillas HTML
40. func cargarPlantilla(w http.ResponseWriter, nombre\_plantilla string, pagina \*Pagina){
41. plantillas.ExecuteTemplate(w, nombre\_plantilla + ".html", pagina)
42. }
43. //Manejador de peticiones
44. func manejadorMostrarPagina(w http.ResponseWriter, r \*http.Request){
45. titulo := r.URL.Path[len("/view/"):]
46. p, err := cargarPagina(titulo)
47. if err != nil {
48. http.Redirect(w, r, "/edit/" + titulo, http.StatusFound)
49. fmt.Println("La página solicitada no existía. Llamando al editor...")
50. return
51. }
52. cargarPlantilla(w, "view", p)
53. }
54. //Manejador para editar wikis
55. func manejadorEditar(w http.ResponseWriter, r \*http.Request){
56. titulo := r.URL.Path[len("/edit/"):]
57. p, err := cargarPagina(titulo)
58. if err != nil{
59. p = &Pagina{Titulo: titulo}
60. }
61. cargarPlantilla(w, "edit", p)
62. }
63. //Manejador para guardar wikis
64. func manejadorGuardar(w http.ResponseWriter, r \* http.Request) {
65. titulo := r.URL.Path[len("/save/"):]
66. cuerpo := r.FormValue("body")
67. p := &Pagina{Titulo: titulo, Cuerpo: []byte(cuerpo)}
68. fmt.Println("Guardando " + titulo + ".txt...")
69. p.guardar()
70. http.Redirect(w, r, "/view/" + titulo, http.StatusFound)
71. }

Después de compilar tu servidor y ponerlo en ejecución, recuerda acceder a la siguiente dirección desde tu navegador para realizar pruebas:

localhost:8080/view/Ejemplo

Finalizando…

Espero que te haya servido lo visto en esta entrada. Cualquier duda puedes dejarla en la sección de comentarios y a la brevedad te ayudaré a resolverla.

En la siguiente entrada aprenderemos a utilizar regex (expresiones regulares) en nuestro servidor para validar las URL. Hasta la próxima.

07. Go web: expresiones regulares (regex) en Golang

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

El servidor que hemos desarrollado hasta esta entrada tiene un grave problema de seguridad: el cliente puede hacer peticiones a cualquier ruta de lectura o escritura sin ninguna clase de filtro. En esta entrada vamos a solucionar ese problema.

Expresiones regulares

Una expresión regular, expresión racional, regex o regexp es una secuencia de caracteres que definen un patrón de búsqueda. Con ellas es posible agilizar y estandarizar los procesos de búsqueda de texto.

Golang cuenta con un paquete dedicado a las regex: regexp.

En esta entrada vamos a utilizar la siguiente expresión regular:

^/(edit|save|view)/([a-zA-Z0-9]+)$

De forma general esa expresión se puede leer como: verifica la coincidencia de las rutas de edición (edit), guardado (save) y visualización (view), que vayan acompañadas del nombre de una página que esté conformado por mínimo una letra mayúscula, o minúscula o un número, o bien, una combinación de esos caracteres.

Ruta válida: /view/Ejemplo  
Ruta inválida: /view

Comenzamos

Importamos los paquetes regexp y errors al servidor; el primero de ellos ya hemos explicado para qué nos servirá. Por otro lado, el paquete errors incluye las definiciones de varios métodos para trabajar con errores, en nuestro caso nos servirá para avisar a los manejadores (handlers) cuando haya algo malo con la ruta de la petición. Los paquetes del servidor quedarían de la siguiente forma:

1. import (
2. "fmt"
3. "io/ioutil"
4. "net/http"
5. "html/template"
6. "regexp"
7. "errors"
8. )

En seguida, vamos a crear una variable global que almacenará la expresión regular:

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

El método regex.MustCompile() es el encargado de analizar y compilar la expresión regular para generar una variable tipo regexp. Existe otro método para compilar expresiones regulares, regexp.Compile(); este último, además de regresar la variable regexp, también devuelve un dato tipo error que señala si hay algún problema con la expresión que se suministró. Si usamos regexp.MustCompile(), podemos prescindir de algunas líneas de código puesto que el método automáticamente hará que el programa entre en estado de pánico y termine su ejecución si no puede compilar la regex.

En seguida, creamos la función que retornará el título y la variable de error (de ser necesario):

1. func dameTitulo(w http.ResponseWriter, r \*http.Request) (string, error) {
2. m := regex\_ruta.FindStringSubmatch(r.URL.Path)
3. if m == nil {
4. http.NotFound(w, r)
5. return "", errors.New("Ruta inválida")
6. }
7. return m[2], nil
8. }

El método FindStringSubmatch() regresa un [slice](http://codingornot.com/12-go-to-go-slices-trozos-en-golang/) con la primera ocurrencia de la expresión buscada (de izquierda a derecha) y sus subgrupos. Si imprimimos el slice generado al intentar acceder a /view/Ejemplo, obtenemos:

[“/view/Ejemplo”, “view”, “Ejemplo”]

Cuando la URL de la petición del cliente no concuerda con el conjunto que acepta la regex, la variable m será nil y la función dameTitulo() regresará una cadena vacía junto con un error. En cambio, si es correcta, significa que el nombre de la wiki estará en el índice 2 del slice, es por eso que regresamos m[2] junto con nil para indicar que no hubo ningún problema.

Finalmente, agregamos el llamado a la función dameTitulo() en los 3 manejadores que necesitan hacer uso de ella: manejadorMostrar(), manejadorEditar() y manejadorGuardar().

Servidor funcional

El siguiente es el código fuente del servidor con los cambios que hemos hecho hasta esta entrada:

1. package main
2. import (
3. "fmt"
4. "io/ioutil"
5. "net/http"
6. "html/template"
7. "regexp"
8. "errors"
9. )
10. type Pagina struct{
11. Titulo string
12. Cuerpo []byte
13. }
14. var plantillas = template.Must(template.ParseFiles("edit.html", "view.html"))
15. var regex\_ruta = regexp.MustCompile("^/(edit|save|view)/([a-zA-Z0-9]+)$")
16. func main(){
17. //Creamos y guardamos una página para que el cliente la pida
18. pag1 := &Pagina{Titulo: "Ejemplo", Cuerpo:[]byte(
19. "¡Hola personita! Este es el cuerpo de tu página.")}
20. pag1.guardar()
21. http.HandleFunc("/view/", manejadorMostrar)
22. http.HandleFunc("/edit/", manejadorEditar)
23. http.HandleFunc("/save/", manejadorGuardar)
24. fmt.Println("El servidor se encuentra en ejecución.");
25. http.ListenAndServe(":8080", nil)
26. }
27. //Método para guardar página
28. func ( p\* Pagina ) guardar() error {
29. nombre := p.Titulo + ".txt"
30. return ioutil.WriteFile( "./view/" + nombre, p.Cuerpo, 0600)
31. }
32. //Función para cargar página
33. func cargarPagina( titulo string ) (\*Pagina, error) {
34. nombre\_archivo := titulo + ".txt"
35. fmt.Println("El cliente ha pedido: " + nombre\_archivo)
36. cuerpo, err := ioutil.ReadFile( "./view/" + nombre\_archivo )
37. if err != nil {
38. return nil, err
39. }
40. return &Pagina{Titulo: titulo, Cuerpo: cuerpo}, nil
41. }
42. //Función para validar ruta y regresar nombre de la página solicitada
43. func dameTitulo(w http.ResponseWriter, r \*http.Request) (string, error) {
44. m := regex\_ruta.FindStringSubmatch(r.URL.Path)
45. if m == nil {
46. http.NotFound(w, r)
47. return "", errors.New("Ruta inválida")
48. }
49. return m[2], nil
50. }
51. //Función para cargar las plantillas HTML
52. func cargarPlantilla(w http.ResponseWriter, nombre\_plantilla string, pagina \*Pagina){
53. plantillas.ExecuteTemplate(w, nombre\_plantilla + ".html", pagina)
54. }
55. //Manejador para visualizar wikis
56. func manejadorMostrar(w http.ResponseWriter, r \*http.Request){
57. titulo, err := dameTitulo(w, r)
58. if err != nil{
59. return
60. }
61. p, err := cargarPagina(titulo)
62. if err != nil {
63. http.Redirect(w, r, "/edit/" + titulo, http.StatusFound)
64. fmt.Println("La página solicitada no existía. Llamando al editor...")
65. return
66. }
67. cargarPlantilla(w, "view", p)
68. }
69. //Manejador para editar wikis
70. func manejadorEditar(w http.ResponseWriter, r \*http.Request){
71. titulo, err := dameTitulo(w, r)
72. if err != nil{
73. return
74. }
75. p, err := cargarPagina(titulo)
76. if err != nil{
77. p = &Pagina{Titulo: titulo}
78. }
79. cargarPlantilla(w, "edit", p)
80. }
81. //Manejador para guardar wikis
82. func manejadorGuardar(w http.ResponseWriter, r \* http.Request) {
83. titulo, err := dameTitulo(w, r)
84. if err != nil {
85. return
86. }
87. cuerpo := r.FormValue("body")
88. p := &Pagina{Titulo: titulo, Cuerpo: []byte(cuerpo)}
89. fmt.Println("Guardando " + titulo + ".txt...")
90. p.guardar()
91. http.Redirect(w, r, "/view/" + titulo, http.StatusFound)
92. }

Nota: he renombrado la función manejadorMostrarPagina() a manejadorMostrar() para que concuerde con el nombre de los otros 2 manejadores.

Finalizando…

En esta entrada aprendiste a solucionar un grave problema de seguridad que tenía tu servidor.

08. Go web: closures en Golang

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

Hasta ahora, cada vez que nuestro servidor invoca un manejador, es necesario extraer el título de la wiki que pide el cliente y verificar que no hay errores. Para lograrlo agregamos las siguientes líneas a cada manejador:

1. titulo, err := dameTitulo(w, r)
2. if err != nil {
3. return
4. }

En esta entrada te voy a mostrar cómo modificar tu servidor para prescindir de esas líneas.

Conceptos previos

Antes de continuar es necesario que conozcas lo que es una closure (clausura, en español). Si ya conoces el término, puedes saltar a la siguiente sección.

Jaakko Järvi, John Freeman y Lawrence Crowl nos dicen en su publicación [Lambda Expressions and Closures](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2550.pdf): “una closure consiste en el código del cuerpo de la función lambda y el ámbito en que esta fue creada”. Podemos definirlas de forma muy burda como funciones “almacenadas” que recuerdan el ámbito o entorno en que fueron hechas.

Para comprender este concepto es necesario poner unos ejemplos. Analicemos la siguiente función:

1. func suma(a int) func(int) int{
2. return func(b int) int{
3. return a + b
4. }
5. }

Esa es la forma de crear closures en Golang; la función suma() recibe un argumento a int y devuelve una función anónima (sin identificador) que a su vez esta espera un argumento b int y devuelve un int. Podemos utilizar la función suma() para crear y llamar closures como a continuación:

1. suma01 := suma(5) //Creamos una closure
2. suma02 := suma(10) //Creamos otra
3. fmt.Println("Suma 01: ", suma01(2)) //Suma 5 + 2
4. fmt.Println("Suma 02: ", suma02(100)) //Suma 10 + 100

Salida:

1. Suma 01: 7
2. Suma 02: 110

En las primeras dos líneas hacemos que suma01 y suma02 almacenen una closure, es decir, una función y el ámbito en que fue creada. A pesar de haber salido del ámbito de suma(), ambas closures almacenan la variable a int correspondiente al momento de su creación (suma01 tiene a=5, suma02 tiene a=10), cuando se invoca la función de las closures sumamos a + b (el a del momento de la creación, y el b que pasamos al invocarlas). Es posible reutilizar y hacer la cantidad de llamados que queramos a una closure.

Otra gran ventaja es que si la función que necesitamos devolver es muy corta o solo se repite su llamado un par de veces en el programa, podemos crearlas en una sola línea como en seguida te muestro:

1. saludar := func(nombre string) string { return "¡Hola, " + nombre + "!"}

Cada que llamemos la closure saludar le necesitamos pasar una string:

1. fmt.Println(saludar("José"))
2. fmt.Println(saludar("María"))

Así logramos imprimir mensajes como los siguientes:

1. ¡Hola, José!
2. ¡Hola, María!

Closures en el servidor

Después de repasar rápidamente lo que es una closure, es momento de agregarlas a nuestro servidor para eliminar las repeticiones innecesarias de código. Comenzamos por modificar los manejadores para que reciban un nuevo argumento tipo string, además eliminamos los llamados a dameTitulo() y la validación de posibles errores al hacerlo. Los manejadores quedarían así:

* Manejador view
* Manejador edit
* Manejador save

1. //Manejador para mostrar wikis
2. func manejadorMostrar(w http.ResponseWriter, r \*http.Request, titulo string){
3. p, err := cargarPagina(titulo)
4. if err != nil {
5. http.Redirect(w, r, "/edit/" + titulo, http.StatusFound)
6. fmt.Println("La página solicitada no existía. Llamando al editor...")
7. return
8. }
9. cargarPlantilla(w, "view", p)
10. }

Ahora creamos una función que recibirá una función del tipo de los manejadores como argumento y devolverá una closure de tipo http.HandlerFunc() la cual será pasada a la función http.HandleFunc():

1. func llamarManejador(manejador func (http.ResponseWriter, \*http.Request, string)) http.HandlerFunc {
2. return func(w http.ResponseWriter, r \*http.Request) {
3. titulo, err := dameTitulo(w, r)
4. if err != nil {
5. http.NotFound(w, r)
6. return
7. }
8. manejador(w, r, titulo)
9. }
10. }

Para poder utilizar funciones como argumentos, necesitamos especificar su tipo, es decir, el tipo de los argumentos y de los valores que devuelve. La mayoría del cuerpo de la función anterior ya ha sido revisado en notas anteriores: con la función dameTitulo() [extraemos el título de la petición](http://codingornot.com/03-go-web-como-servir-paginas-con-golang) y lo validamos con la [expresión regular (regex) que añadimos anteriormente](http://codingornot.com/07-go-web-expresiones-regulares-regex-en-golang), si es un título inválido escribimos un error a ResponseWriter, en cambio, si el título es correcto, se hará un llamado a la función adjunta que se le pasó de argumento (al manejador de edición, de guardado o de mostrar).

Modificamos la función principal para que haga llamados a llamarManejador() con el manejador que corresponda para cada caso (para hacer más limpio el código, quitamos las líneas con las que creábamos una página de ejemplo en las versiones anteriores del servidor):

1. func main() {
2. http.HandleFunc("/view/", llamarManejador(manejadorMostrar))
3. http.HandleFunc("/save/", llamarManejador(manejadorGuardar))
4. http.HandleFunc("/edit/", llamarManejador(manejadorEditar))
5. http.ListenAndServe(":8080", nil)
6. }

Para compilar y ejecutar:

1. go build servidor\_wikis.go
2. ./servidor\_wikis

Para verificar que el servidor funciona ingresamos a:

http://localhost:8080/cualquier\_pagina\_que\_desees

Finalizando…

Espero que lo visto en esta entrada te haya servido, cualquier duda puedes dejarla en la sección de comentarios.

09. Go web: crear y redireccionar a la página principal

Publicado por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

En esta entrada vamos a realizar unos cambios para tener un poco más organizado nuestro servidor y que además pueda cumplir con otro requisito importante: redireccionar a una página principal.

Organización de plantillas

En las notas anteriores almacenamos las plantillas HTML en la misma carpeta que el código fuente del servidor. Para organizar un poco nuestros archivos, vamos a colocar todas las plantillas en su propia carpeta dentro del proyecto. Comenzamos creando la carpeta tmpl en la raíz del servidor. Movemos los archivos .html que creamos anteriormente (view.html y edit.html) dentro de dicha carpeta.

Abrimos el código fuente del servidor y agregamos el prefijo “tmpl/” a las plantillas de la siguiente línea:

1. var plantillas = template.Must(template.ParseFiles("edit.html", "view.html"))//Antes

Debería quedar de este modo:

1. var plantillas = template.Must(template.ParseFiles("tmpl/edit.html", "tmpl/view.html"))

Redireccionar a la página principal

Nuestro servidor necesita tener una página principal desde la cual permita a sus visitantes moverse entre las demás páginas que hemos creado. Por ahora te voy a mostrar cómo crear dicha página y en las siguientes vamos a añadir el código para las funcionalidades que tendrá.

Primero, vamos a crear la plantilla de la página principal dentro de la carpeta tmpl. Yo la nombraré front.html. La plantilla contendrá las siguientes líneas:

1. <h1>Has sido redireccionado a: {{.Titulo}}</h1>
2. {{printf "%s" .Cuerpo}}

Por el momento solamente mostrará dos mensajes: un aviso de que ha sido redireccionado y un mensaje adicional que tú desees.

En seguida, modificamos la variable plantillas para que acepte “tmpl/front.html”:

1. var plantillas = template.Must(template.ParseFiles("tmpl/edit.html", "tmpl/view.html", "tmpl/front.html"))

La página principal se mostrará cuando el usuario intente acceder a la raíz del servidor “/”, por tanto, modificamos la [regex](http://codingornot.com/07-go-web-expresiones-regulares-regex-en-golang) de rutas de acceso para que considere “/” como parte de las rutas permitidas:

1. var regex\_ruta = regexp.MustCompile("^/|((edit|save|view)/([a-zA-Z0-9]+))?$")

Opcional: puedes crear una variable pagina\_principal (o el nombre que desees darle) que almacene el nombre del archivo que tendrá la información de la página principal. Esto es buena práctica de programación porque disminuyes la cantidad de literales en tu código y, si quisieras cambiar la página a la que te redireccione el servidor, solamente modificarías la cadena de la variable. En mi caso quedaría así:

1. var pagina\_principal = "Principal" //también puede llamarse "Home", por ejemplo

Ahora, creamos un manejador para la página principal. Yo lo llamaré manejadorRaiz():

1. func manejadorRaiz(w http.ResponseWriter, r \*http.Request, titulo string) {
2. p, err := cargarPagina(pagina\_principal)
3. if err != nil {
4. http.Redirect(w, r, "edit/" + pagina\_principal, http.StatusFound)
5. fmt.Println("Error")
6. return
7. }
8. cargarPlantilla(w, "front", p)
9. }

El manejador hace lo mismo que los otros tres que ya tenemos hechos:

* Carga la página solicitada
* Si no existe, crear una nueva
* Si ya existe, la devuelve al cliente con el formato de la plantilla correspondiente.

Como puedes darte cuenta, con la función cargarPlantilla() cargamos la plantilla front.html que recién creamos. Cuando intentemos acceder a “/”, el servidor abrirá la página “/view/Principal” en su lugar.

Como modificamos la expresión regular (regex), necesitamos actualizar la función dameTitulo() puesto que cambiaron los submatches que devuelve FindStringSubmatch, ahora el título ya no estará en la posición 2:

1. func dameTitulo(w http.ResponseWriter, r \*http.Request) (string, error) {
2. peticion := regex\_ruta.FindStringSubmatch(r.URL.Path)
3. if peticion == nil {
4. http.NotFound(w, r)
5. return "", errors.New("Ruta inválida")
6. }
7. return peticion[len(peticion) - 1], nil
8. }

La estructura de las peticiones que validamos nos permite saber que el título de las wiki siempre estará en la posición tamaño-1 del slice, por tanto, aprovechamos y cambiamos la posición absoluta (2) por una relativa (tamaño – 1) para no tener este problema de nuevo.

Finalmente, añadimos el manejador a la función principal para que escuche las peticiones a “/”. El código completo del servidor quedaría así:

1. 09. Go web: crear y redireccionar a la página principal

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

En esta entrada vamos a realizar unos cambios para tener un poco más organizado nuestro servidor y que además pueda cumplir con otro requisito importante: redireccionar a una página principal.

Organización de plantillas

En las notas anteriores almacenamos las plantillas HTML en la misma carpeta que el código fuente del servidor. Para organizar un poco nuestros archivos, vamos a colocar todas las plantillas en su propia carpeta dentro del proyecto. Comenzamos creando la carpeta tmpl en la raíz del servidor. Movemos los archivos .html que creamos anteriormente (view.html y edit.html) dentro de dicha carpeta.

Abrimos el código fuente del servidor y agregamos el prefijo “tmpl/” a las plantillas de la siguiente línea:

1. var plantillas = template.Must(template.ParseFiles("edit.html", "view.html"))//Antes

Debería quedar de este modo:

1. var plantillas = template.Must(template.ParseFiles("tmpl/edit.html", "tmpl/view.html"))

Redireccionar a la página principal

Nuestro servidor necesita tener una página principal desde la cual permita a sus visitantes moverse entre las demás páginas que hemos creado. Por ahora te voy a mostrar cómo crear dicha página y en las siguientes vamos a añadir el código para las funcionalidades que tendrá.

Primero, vamos a crear la plantilla de la página principal dentro de la carpeta tmpl. Yo la nombraré front.html. La plantilla contendrá las siguientes líneas:

1. <h1>Has sido redireccionado a: {{.Titulo}}</h1>
2. {{printf "%s" .Cuerpo}}

Por el momento solamente mostrará dos mensajes: un aviso de que ha sido redireccionado y un mensaje adicional que tú desees.

En seguida, modificamos la variable plantillas para que acepte “tmpl/front.html”:

1. var plantillas = template.Must(template.ParseFiles("tmpl/edit.html", "tmpl/view.html", "tmpl/front.html"))

La página principal se mostrará cuando el usuario intente acceder a la raíz del servidor “/”, por tanto, modificamos la [regex](http://codingornot.com/07-go-web-expresiones-regulares-regex-en-golang) de rutas de acceso para que considere “/” como parte de las rutas permitidas:

1. var regex\_ruta = regexp.MustCompile("^/|((edit|save|view)/([a-zA-Z0-9]+))?$")

Opcional: puedes crear una variable pagina\_principal (o el nombre que desees darle) que almacene el nombre del archivo que tendrá la información de la página principal. Esto es buena práctica de programación porque disminuyes la cantidad de literales en tu código y, si quisieras cambiar la página a la que te redireccione el servidor, solamente modificarías la cadena de la variable. En mi caso quedaría así:

1. var pagina\_principal = "Principal" //también puede llamarse "Home", por ejemplo

Ahora, creamos un manejador para la página principal. Yo lo llamaré manejadorRaiz():

1. func manejadorRaiz(w http.ResponseWriter, r \*http.Request, titulo string) {
2. p, err := cargarPagina(pagina\_principal)
3. if err != nil {
4. http.Redirect(w, r, "edit/" + pagina\_principal, http.StatusFound)
5. fmt.Println("Error")
6. return
7. }
8. cargarPlantilla(w, "front", p)
9. }

El manejador hace lo mismo que los otros tres que ya tenemos hechos:

* Carga la página solicitada
* Si no existe, crear una nueva
* Si ya existe, la devuelve al cliente con el formato de la plantilla correspondiente.

Como puedes darte cuenta, con la función cargarPlantilla() cargamos la plantilla front.html que recién creamos. Cuando intentemos acceder a “/”, el servidor abrirá la página “/view/Principal” en su lugar.

Como modificamos la expresión regular (regex), necesitamos actualizar la función dameTitulo() puesto que cambiaron los submatches que devuelve FindStringSubmatch, ahora el título ya no estará en la posición 2:

1. func dameTitulo(w http.ResponseWriter, r \*http.Request) (string, error) {
2. peticion := regex\_ruta.FindStringSubmatch(r.URL.Path)
3. if peticion == nil {
4. http.NotFound(w, r)
5. return "", errors.New("Ruta inválida")
6. }
7. return peticion[len(peticion) - 1], nil
8. }

La estructura de las peticiones que validamos nos permite saber que el título de las wiki siempre estará en la posición tamaño-1 del slice, por tanto, aprovechamos y cambiamos la posición absoluta (2) por una relativa (tamaño – 1) para no tener este problema de nuevo.

Finalmente, añadimos el manejador a la función principal para que escuche las peticiones a “/”. El código completo del servidor quedaría así:

1. package main
2. import (
3. "fmt"
4. "io/ioutil"
5. "net/http"
6. "html/template"
7. "regexp"
8. "errors"
9. )
10. type Pagina struct{
11. Titulo string
12. Cuerpo []byte
13. }
14. var plantillas = template.Must(template.ParseFiles("tmpl/edit.html", "tmpl/view.html", "tmpl/front.html"))
15. var regex\_ruta = regexp.MustCompile("^(/|(/(edit|save|view)/([a-zA-Z0-9]+)))$")
16. var pagina\_principal = "Principal"
17. func main() {
18. http.HandleFunc("/", llamarManejador(manejadorRaiz)) //Nuevo manejador
19. http.HandleFunc("/view/", llamarManejador(manejadorMostrar))
20. http.HandleFunc("/save/", llamarManejador(manejadorGuardar))
21. http.HandleFunc("/edit/", llamarManejador(manejadorEditar))
22. http.ListenAndServe(":8080", nil)
23. }
24. //Método para guardar página
25. func ( p\* Pagina ) guardar() error {
26. nombre := p.Titulo + ".txt"
27. return ioutil.WriteFile( "./view/" + nombre, p.Cuerpo, 0600)
28. }
29. //Función para cargar página
30. func cargarPagina( titulo string ) (\*Pagina, error) {
31. nombre\_archivo := titulo + ".txt"
32. fmt.Println("El cliente ha pedido: " + nombre\_archivo)
33. cuerpo, err := ioutil.ReadFile( "./view/" + nombre\_archivo )
34. if err != nil {
35. return nil, err
36. }
37. return &Pagina{Titulo: titulo, Cuerpo: cuerpo}, nil
38. }
39. //Función para validar ruta y regresar nombre de la página solicitada
40. func dameTitulo(w http.ResponseWriter, r \*http.Request) (string, error) {
41. peticion := regex\_ruta.FindStringSubmatch(r.URL.Path)
42. if peticion == nil {
43. http.NotFound(w, r)
44. return "", errors.New("Ruta inválida")
45. }
46. return peticion[len(peticion) - 1], nil
47. }
48. //Función para cargar las plantillas HTML
49. func cargarPlantilla(w http.ResponseWriter, nombre\_plantilla string, pagina \*Pagina){
50. plantillas.ExecuteTemplate(w, nombre\_plantilla + ".html", pagina)
51. }
52. func llamarManejador(manejador func (http.ResponseWriter, \*http.Request, string)) http.HandlerFunc {
53. return func(w http.ResponseWriter, r \*http.Request) {
54. titulo, err := dameTitulo(w, r)
55. fmt.Println(titulo)
56. if err != nil {
57. http.NotFound(w, r)
58. return
59. }
60. manejador(w, r, titulo)
61. }
62. }
63. //Manejador para mostrar página principal
64. func manejadorRaiz(w http.ResponseWriter, r \*http.Request, titulo string) {
65. p, err := cargarPagina(pagina\_principal)
66. if err != nil {
67. http.Redirect(w, r, "edit/" + pagina\_principal, http.StatusFound)
68. fmt.Println("Error")
69. return
70. }
71. cargarPlantilla(w, "front", p)
72. }
73. //Manejador para visualizar wikis
74. func manejadorMostrar(w http.ResponseWriter, r \*http.Request, titulo string){
75. p, err := cargarPagina(titulo)
76. if err != nil {
77. http.Redirect(w, r, "/edit/" + titulo, http.StatusFound)
78. fmt.Println("La página solicitada no existía. Llamando al editor...")
79. return
80. }
81. cargarPlantilla(w, "view", p)
82. }
83. //Manejador para editar wikis
84. func manejadorEditar(w http.ResponseWriter, r \*http.Request, titulo string){
85. p, err := cargarPagina(titulo)
86. if err != nil{
87. p = &Pagina{Titulo: titulo}
88. }
89. cargarPlantilla(w, "edit", p)
90. }
91. //Manejador para guardar wikis
92. func manejadorGuardar(w http.ResponseWriter, r \* http.Request, titulo string) {
93. cuerpo := r.FormValue("body")
94. p := &Pagina{Titulo: titulo, Cuerpo: []byte(cuerpo)}
95. fmt.Println("Guardando " + titulo + ".txt...")
96. p.guardar()
97. http.Redirect(w, r, "/view/" + titulo, http.StatusFound)
98. }

Para compilar y ejecutar:

1. go build servidor\_wikis.go
2. ./servidor\_wikis

Para verificar que funciona ingresamos a:

1. http://localhost:8080/

Finalizando…

Espero que esta entrada te haya servido.

1. package main
2. import (
3. "fmt"
4. "io/ioutil"
5. "net/http"
6. "html/template"
7. "regexp"
8. "errors"
9. )
10. type Pagina struct{
11. Titulo string
12. Cuerpo []byte
13. }
14. var plantillas = template.Must(template.ParseFiles("tmpl/edit.html", "tmpl/view.html", "tmpl/front.html"))
15. var regex\_ruta = regexp.MustCompile("^(/|(/(edit|save|view)/([a-zA-Z0-9]+)))$")
16. var pagina\_principal = "Principal"
17. func main() {
18. http.HandleFunc("/", llamarManejador(manejadorRaiz)) //Nuevo manejador
19. http.HandleFunc("/view/", llamarManejador(manejadorMostrar))
20. http.HandleFunc("/save/", llamarManejador(manejadorGuardar))
21. http.HandleFunc("/edit/", llamarManejador(manejadorEditar))
22. http.ListenAndServe(":8080", nil)
23. }
24. //Método para guardar página
25. func ( p\* Pagina ) guardar() error {
26. nombre := p.Titulo + ".txt"
27. return ioutil.WriteFile( "./view/" + nombre, p.Cuerpo, 0600)
28. }
29. //Función para cargar página
30. func cargarPagina( titulo string ) (\*Pagina, error) {
31. nombre\_archivo := titulo + ".txt"
32. fmt.Println("El cliente ha pedido: " + nombre\_archivo)
33. cuerpo, err := ioutil.ReadFile( "./view/" + nombre\_archivo )
34. if err != nil {
35. return nil, err
36. }
37. return &Pagina{Titulo: titulo, Cuerpo: cuerpo}, nil
38. }
39. //Función para validar ruta y regresar nombre de la página solicitada
40. func dameTitulo(w http.ResponseWriter, r \*http.Request) (string, error) {
41. peticion := regex\_ruta.FindStringSubmatch(r.URL.Path)
42. if peticion == nil {
43. http.NotFound(w, r)
44. return "", errors.New("Ruta inválida")
45. }
46. return peticion[len(peticion) - 1], nil
47. }
48. //Función para cargar las plantillas HTML
49. func cargarPlantilla(w http.ResponseWriter, nombre\_plantilla string, pagina \*Pagina){
50. plantillas.ExecuteTemplate(w, nombre\_plantilla + ".html", pagina)
51. }
52. func llamarManejador(manejador func (http.ResponseWriter, \*http.Request, string)) http.HandlerFunc {
53. return func(w http.ResponseWriter, r \*http.Request) {
54. titulo, err := dameTitulo(w, r)
55. fmt.Println(titulo)
56. if err != nil {
57. http.NotFound(w, r)
58. return
59. }
60. manejador(w, r, titulo)
61. }
62. }
63. //Manejador para mostrar página principal
64. func manejadorRaiz(w http.ResponseWriter, r \*http.Request, titulo string) {
65. p, err := cargarPagina(pagina\_principal)
66. if err != nil {
67. http.Redirect(w, r, "edit/" + pagina\_principal, http.StatusFound)
68. fmt.Println("Error")
69. return
70. }
71. cargarPlantilla(w, "front", p)
72. }
73. //Manejador para visualizar wikis
74. func manejadorMostrar(w http.ResponseWriter, r \*http.Request, titulo string){
75. p, err := cargarPagina(titulo)
76. if err != nil {
77. http.Redirect(w, r, "/edit/" + titulo, http.StatusFound)
78. fmt.Println("La página solicitada no existía. Llamando al editor...")
79. return
80. }
81. cargarPlantilla(w, "view", p)
82. }
83. //Manejador para editar wikis
84. func manejadorEditar(w http.ResponseWriter, r \*http.Request, titulo string){
85. p, err := cargarPagina(titulo)
86. if err != nil{
87. p = &Pagina{Titulo: titulo}
88. }
89. cargarPlantilla(w, "edit", p)
90. }
91. //Manejador para guardar wikis
92. func manejadorGuardar(w http.ResponseWriter, r \* http.Request, titulo string) {
93. cuerpo := r.FormValue("body")
94. p := &Pagina{Titulo: titulo, Cuerpo: []byte(cuerpo)}
95. fmt.Println("Guardando " + titulo + ".txt...")
96. p.guardar()
97. http.Redirect(w, r, "/view/" + titulo, http.StatusFound)
98. }

Para compilar y ejecutar:

1. go build servidor\_wikis.go
2. ./servidor\_wikis

Para verificar que funciona ingresamos a:

1. http://localhost:8080/

Finalizando…

Espero que esta entrada te haya servido.

10. Go web: enlaces a página anterior y siguiente

Por [Uriel Ruelas](http://codingornot.com/author/zainiaz07)

En las entradas anteriores necesitábamos escribir la URL con la ruta de acceso completa porque no nos era posible navegar entre las diferentes páginas que ya teníamos almacenadas a menos que supiéramos de antemano que existían. En esta entrada vamos a agregar un par de enlaces al final de las wiki cuya función será direccionar a las demás. Cabe señalar que para ejemplificar este funcionamiento, opté por enlazar a las páginas anterior y siguiente en orden alfabético pero tú podrías hacerlo, por ejemplo, con enlaces a páginas de temas relacionados o por fecha.

Enlazando wikis

Comenzamos por agregar 4 miembros nuevos a nuestro registro (struct) Pagina:

1. type Pagina struct{
2. Titulo string
3. Cuerpo []byte
4. Siguiente string
5. Visibilidad\_A string
6. Anterior string
7. Visibilidad\_S string
8. }

Las cadenas (string) Anterior y Siguiente son las que contendrán los nombres de las respectivas notas anterior y siguiente. Agregué dos miembros más, Visibilidad\_A y Visibilidad\_S, para permitir esconder los enlaces cuando no haya ninguna página a la cual direccionar, esto por medio del atributo style. Para que estos cambios sean visibles, modificamos la plantilla view.html así:

1. <h1>{{.Titulo}}</h1>
2. [<a href="/edit/{{.Titulo}}">edit</a>]
3. <div>{{printf "%s" .Cuerpo}}</div>
4. <div>[<a style="visibility: {{.Visibilidad\_A}};" href="/view/{{.Anterior}}">Anterior</a>]
5. [<a style="visibility: {{.Visibilidad\_S}};" href="/view/{{.Siguiente}}">Siguiente</a>]</div>

Como puedes darte cuenta, en la plantilla solamente agregamos un par de enlaces al final con estilo modificado. A este estilo se le conoce como propiedades CSS, en nuestro caso modificamos la visibilidad (visibility) de cada enlace por separado.

En seguida, agregamos un [slice](http://codingornot.com/12-go-to-go-slices-trozos-en-golang) global que será el encargado de almacenar los nombres de las páginas del servidor. Hay muchas otras alternativas para hacer esta tarea, en mi caso opté por almacenar los nombres de las páginas en memoria para no tener que leer muchas veces el disco.

1. var enlaces []string

A continuación, creamos la función encargada de leer los nombres de las wikis al inicio de la ejecución del servidor, yo la llamaré iniciarEnlaces():

1. func iniciarEnlaces(){
2. paginas, \_ := ioutil.ReadDir("./view/")
3. enlaces = make([]string, 0, 0)
4. for \_, actual := range paginas {
5. if(actual.Name() != (pagina\_principal + ".txt")){
6. enlaces = append(enlaces, actual.Name()[ : len(actual.Name()) - 4 ])
7. }
8. }
9. }

Con ayuda de ioutil.ReadDir() consultamos los nombres de las páginas que actualmente existen en la carpeta donde las guardamos y, además, aprovechamos para inicializar el slice que los contendrá en memoria. Iteramos entre las lecturas de ioutil.ReadDir(), extraemos el nombre de los archivos que fueron leídos y los guardamos en el slice (los nombres contendrán la extensión .txt, por ello eliminamos los últimos 4 caracteres antes de agregarlos).

La función anterior nos permite leer los nombres de las páginas que teníamos antes de la ejecución del servidor pero no nos permite agregar nuevos nombres en tiempo de ejecución, para ello creamos la siguiente función:

1. func agregarEnlace(pagina string){
2. for \_, actual := range enlaces{
3. if (actual == pagina || actual == pagina\_principal) {
4. return
5. }
6. }
7. enlaces = append(enlaces, pagina)
8. sort.Strings(enlaces)
9. }

Cuando intentamos guardar el nombre de una página, es necesario verificar que no existía previamente o que no sea la página principal ya que esta es tratada de forma distinta, en cuyo caso no se agrega el nombre y la función regresa sin hacer cambios. En lugar de un slice puedes usar un [map](http://codingornot.com/13-go-to-go-maps-en-golang) si así lo deseas, la ventaja de estos es que te permiten garantizar que los elementos no se repiten sin programar nada. Utilizamos el método sort.Strings() ubicado dentro del paquete sort para ordenar las cadenas del slice alfabéticamente, para ello lo importamos junto con los demás paquetes que ya teníamos de entradas anteriores:

1. import (
2. "fmt"
3. "io/ioutil"
4. "net/http"
5. "html/template"
6. "regexp"
7. "errors"
8. "sort" //Nuevo
9. )

Para continuar, necesitamos obtener los nombres de las páginas a las que vamos a direccionar al cliente desde la página actual. Para lograrlo construimos la siguiente función:

1. func obtenerEnlaces(pagina string) (string, string) {
2. var ant, sig string
3. for i, actual := range enlaces {
4. if(actual == pagina){
5. if(i > 0){
6. ant = enlaces[i - 1]
7. } else{
8. ant = ""
9. }
10. if(i < (len(enlaces)-1) ){
11. sig = enlaces[i + 1]
12. } else {
13. sig = ""
14. }
15. break
16. }
17. }
18. return ant, sig
19. }

Resumiendo, esta función obtiene la posición dentro del slice del nombre que se le pasa como argumento y devuelve los nombres de las páginas anterior y siguiente. Si la página con la que se trabaja es la primera o la última, utilizamos la cadena vacía para especificar que no hay nada antes o después según corresponda (tú puedes optar por otras alternativas como direccionar a la primera página cuando se alcance la última).

Construimos un método para los registros Pagina que sea el encargado de asignar los valores de visibilidad a los enlaces:

1. func ( p\* Pagina ) asignarVisibilidad() {
2. if(p.Anterior == ""){
3. p.Visibilidad\_A = "hidden"
4. }else {
5. p.Visibilidad\_A = "visible"
6. }
7. if(p.Siguiente == ""){
8. p.Visibilidad\_S = "hidden"
9. }else {
10. p.Visibilidad\_S = "visible"
11. }
12. }

La propiedad CSS visibility del atributo style nos dice que cuando esta es hidden, entonces el contenido no se mostrará en pantalla; en cambio, si esta es visible (por defecto), entonces el contenido sí se mostrará en pantalla.

Finalizando los cambios en nuestro código fuente, agregamos los llamados a las funciones que creamos anteriormente según corresponda:

* Guardar página
* Cargar página
* Mostrar página

1. //Cuando guardamos una página, intentamos agregar un nombre nuevo
2. func ( p\* Pagina ) guardar() error {
3. nombre := p.Titulo + ".txt"
4. agregarEnlace(p.Titulo) //Nuevo
5. return ioutil.WriteFile( "./view/" + nombre, p.Cuerpo, 0600)
6. }

Finalizando…

Esto ha sido todo por esta vez, espero que te haya servido