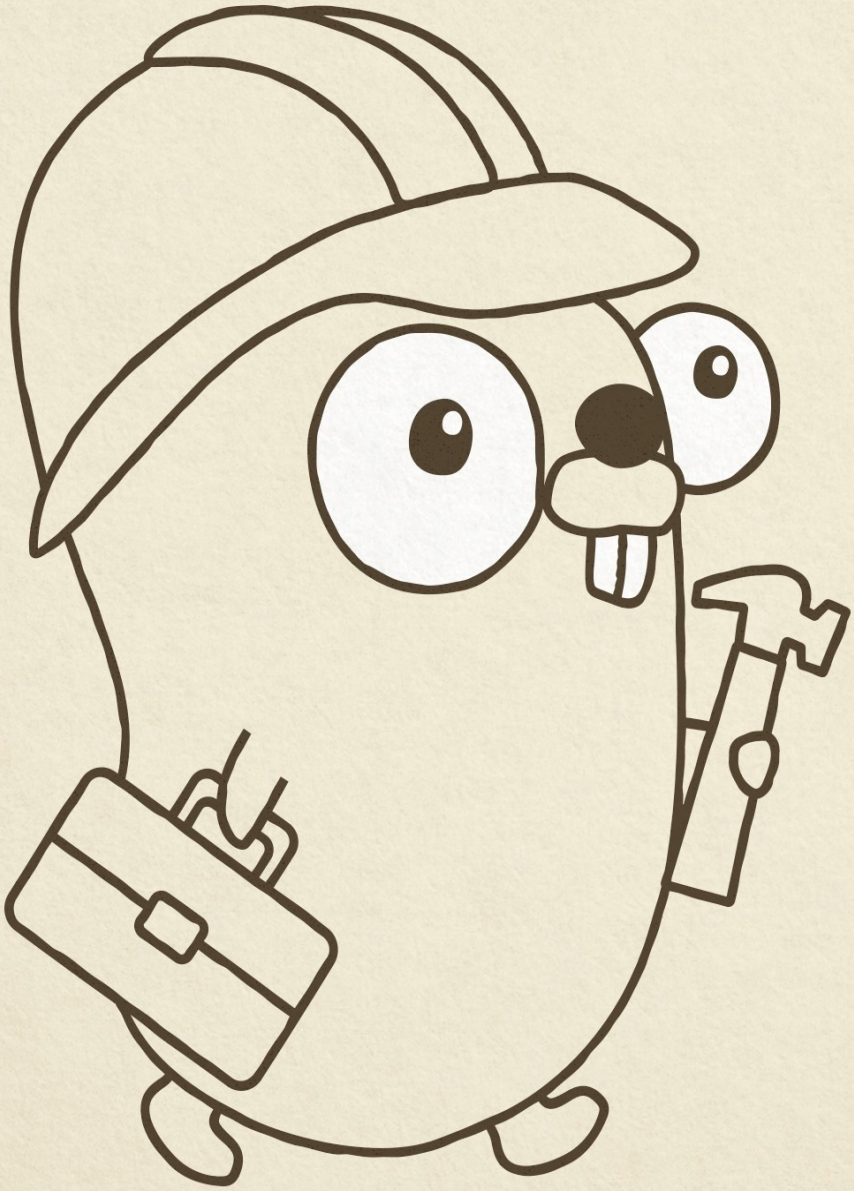
BUILDING WEB APPS WITH GO



Jeremy Saenz

Table of Contents

Construyendo aplicaciones web con Go

¡Bienvenido a la **construcción de aplicaciones web con Go**! Si estás leyendo esto, entonces acabas de comenzar tu viaje desde novato hasta profesional. No, en serio, la programación web en Go es tan divertida y fácil que ¡ni siquiera te darás cuenta de la cantidad de información que sin proponértelo estás aprendiendo!

Ten en cuenta que todavía hay partes de este libro que están incompletas y es necesario que las veas con un poco de cariño. La belleza de la publicación de código abierto es que te puedo entregar un libro incompleto y todavía te resulta útil.

Antes de profundizar en los detalles de los aspectos esenciales, vamos a empezar con algunas reglas básicas:

Requisitos previos

Para mantener este tutorial conciso y enfocado, estoy asumiendo que estás preparado de la siguiente manera:

1. Instalaste el Lenguaje de programación Go.
2. Configuraste la variable de entorno `GOPATH` siguiendo el tutorial cómo escribir código Go.
3. Estás familiarizado con los conceptos básicos de Go. (El paseo por Go es un muy buen lugar para empezar)
4. Tienes instalados todos los paquetes necesarios
5. Instalaste el Heroku Toolbelt
6. Tienes una cuenta Heroku

Paquetes necesarios

En su mayor parte vamos a utilizar los paquetes integrados de la librería estándar para construir nuestras aplicaciones web. Ciertas lecciones tal como la de bases de datos, semiware y el enrutamiento URL requerirán un paquete de terceros. Aquí está una lista de todos los paquetes `go` que debes instalar antes de comenzar:

Nombre	Ruta de importación	Descripción
Gorilla Mux	github.com/gorilla/mux	Un potente enrutador URL y despachador
Negroni	github.com/codegangsta/negroni	Idiomático semiware HTTP
Black Friday	github.com/russross/blackfriday	Un procesador de `markdown`
Render	gopkg.in/unrolled/render.v1	Fácil renderizado para JSON, XML y HTML
SQLite3	github.com/mattn/go-sqlite3	Controlador sqlite3 para Go

Puedes instalar (o actualizar) estos paquetes ejecutando la siguiente orden en la consola

```
go get -u <ruta_de_importación>
```

Por ejemplo, si quieres instalar Negroni, debes usar la siguiente orden:

```
go get -u github.com/codegangsta/negroni
```

Go facilita las cosas

Si has construido una aplicación web antes, seguramente sabes que hay un buen montón de conceptos a tener en cuenta. HTTP, HTML, CSS, JSON, bases de datos, sesiones, `cookies`, formularios, semiwares, enrutamiento y controladores son sólo algunas de las muchas cosas que tu aplicación web puede necesitar para poder interactuar.

Si bien cada una de esas cosas puede ser importante en la construcción de tus aplicaciones web, no cada una de ellas es importante para alguna determinada aplicación. Por ejemplo, una API web puede simplemente usar `JSON` como formato de serialización, con lo cual conceptos tales como HTML no son relevantes para esa aplicación web en particular.

El camino de Go

La comunidad de Go entiende este dilema. En lugar de confiar en las grandes y pesadas plataformas que tratan de cubrir todas las bases, los programadores Go apuestan a las necesidades básicas para realizar el trabajo. Este enfoque minimalista en la programación web puede ser una experiencia desagradable al principio, pero el resultado de este esfuerzo es un programa mucho más simple al final.

Go hace las cosas simples, es tan fácil como eso. Si nos entrenamos para alinearnos al "camino de Go" en la programación para la web, vamos a terminar con aplicaciones web más simples, flexibles y fáciles de mantener.

El poder en la simplicidad

A medida que avancemos a través de los ejercicios de este libro, creo que te sorprenderás por lo simple que algunos de estos programas pueden ser mientras que todavía proporcionan un montón de funcionalidad.

Al sentarte a diseñar tus propias aplicaciones Web en Go, piensa mucho acerca de los componentes y conceptos en que se centra tu aplicación, y utiliza sólo esas piezas. Este libro cubre una amplia gama de temas de Internet, pero no te sientas obligado a utilizarlos todos. En palabras de nuestro amigo Lonestar, "sólo toma lo que necesitas para sobrevivir".


TAKE ONLY WHAT YOU NEED TO SURVIVE.

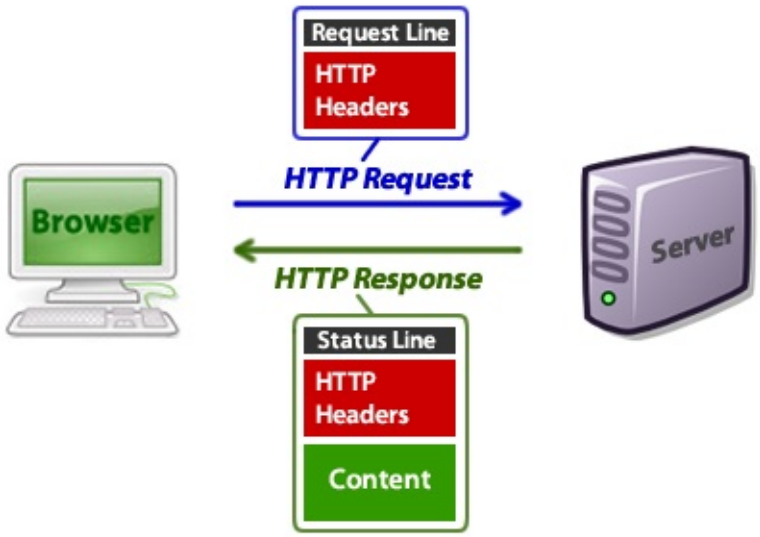
El paquete net/http

Probablemente has escuchado que Go es fantástico para crear aplicaciones web de todas las formas y tamaños. Esto se debe en parte al fantástico esfuerzo que se ha puesto en hacer una biblioteca estándar limpia, consistente y fácil de usar.

Tal vez uno de los paquetes más importantes para cualquier desarrollador web en ciernes Go es el paquete `net/http`. Este paquete te permite construir servidores HTTP en Go con sus poderosas construcciones de composición. Antes de comenzar a codificar, hagamos un muy rápido resumen de HTTP.

Conceptos HTTP básicos

Cuando hablamos de la construcción de aplicaciones web, normalmente queremos decir que estamos construyendo servidores HTTP. HTTP es un protocolo que originalmente fue diseñado para transportar documentos HTML desde un servidor hasta un navegador web cliente. Hoy, HTTP se utiliza para el transporte de un mucho más amplio conjunto de cosas que simplemente HTML.



Lo importante a notar en este diagrama son los dos puntos de interacción entre el servidor y el navegador. El navegador realiza una petición HTTP con un poco de información, el servidor procesa esa petición y devuelve una respuesta.

Este modelo de petición-respuesta es uno de los puntos focales clave en la construcción de aplicaciones web en Go. De hecho, la pieza más importante del paquete es la interfaz `http.Handler`.

La Interfaz http.Handler

A medida que te familiarices más con Go, te darás cuenta de cuanto impactan las interfaces en el diseño de tus programas. La interfaz `net/http` encapsula en un método el patrón petición-respuesta:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

Se espera que los implementadores de esta interfaz inspeccionen y procesen datos procedentes del objeto `http.Request` y escriban una respuesta en el objeto `http.ResponseWriter`.

La interfaz `http.ResponseWriter` tiene esta apariencia:

```
type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(int)
}
```

Componiendo servicios Web

Debido a que gran parte del paquete `net/http` está construido de tipos interfaz bien definidos, puedes (y se espera que) construyas tus aplicaciones web con la composición en mente. Cada implementación de `http.Handler` se puede considerar como su propio servidor web.

Puedes encontrar muchos patrones en esa simple pero poderosa suposición. A lo largo de este libro cubriremos algunos de estos patrones y cómo podemos utilizarlos para resolver problemas del mundo real.

Ejercicio: Un servidor de ficheros en 1 línea

Vamos a resolver un problema del mundo real en 1 línea de código.

La mayoría de las veces la gente sólo tiene que servir archivos estáticos. Tal vez tienes una página HTML estática y sólo quieres servir un poco de HTML, imágenes y CSS y llamarla un día. Claro, podrías ponerla bajo el servidor Apache o `SimpleHTTPServer` de Python, pero Apache es demasiado para este pequeño sitio y `SimpleHTTPServer` es, bueno, demasiado lento.

Empecemos creando un nuevo proyecto en nuestro `GOPATH`.

```
cd GOPATH/src
mkdir servidorfichero && cd servidorfichero
```

Crea un fichero **main.go** con nuestro típico y repetitivo texto modelo:

```
package main

import "net/http"

func main() {
}
```

Todo lo que necesitamos importar para que esto funcione es el paquete `net/http`. Recuerda que todo esto es parte de la biblioteca estándar de Go.

Escribamos nuestro código del servidor de ficheros:

```
http.ListenAndServe(":8080" , http.FileServer(http.Dir(".")))
```

La función `http.ListenAndServe` se utiliza para iniciar el servidor, se vinculará a la dirección que le dimos (`:8080`) y cuando reciba una petición HTTP, la entregaremos al `http.Handler` que suministramos como segundo argumento. En nuestro caso es la función incorporada `http.FileServer`.

La función `http.FileServer` construye un `http.Handler` que servirá un directorio de ficheros completo y averiguará cual fichero entregar basándose en la ruta de la petición. Con `http.Dir(".")` especificamos que el servidor de ficheros sirve el directorio de trabajo actual.

Todo el programa se ve así:

```
package main

import "net/http"

func main() {
    http.ListenAndServe(":8080", http.FileServer(http.Dir(".")))
}
```

Ahora construyamos y ejecutemos nuestro programa servidor de ficheros:

```
go build
./servidorfichero
```

Si visitas localhost:8080/main.go en tu navegador web, deberías ver el contenido de tu fichero **main.go**. Puedes ejecutar este programa desde cualquier directorio y servir el árbol como un servidor de ficheros estáticos. Todo esto en 1 sola línea de código Go.

Creando una aplicación web básica

Ahora que terminamos de repasar los conceptos básicos de HTTP, vamos a crear una sencilla pero útil aplicación web en Go.

Mejorando nuestro programa servidor de ficheros que implementamos en el capítulo anterior, vamos a implementar un pequeño editor de texto en el cual podemos introducir marcas del formato `markdown`, a partir del cual generaremos código HTML con el paquete `github.com/russross/blackfriday`.

Un formulario HTML

Para empezar, necesitamos un formulario HTML básico para introducir texto:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <title>Generador de HTML</title>
    <link href="/css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    <div class="container">
      <div class="page-title">
        <h1>Editor de texto en formato markdown</h1>
        <p class="lead">Go genera HTML a partir de markdown</p>
        <hr />
      </div>

      <form action="/markdown" method="POST">
        <div class="form-group">
          <textarea class="form-control" name="cuerpo" cols="30" rows="10"></textarea>
        </div>

        <div class="form-group">
          <input type="submit" class="btn btn-primary pull-right" />
        </div>
      </form>
    </div>
    <script src="/js/bootstrap.min.js"></script>
  </body>
</html>
```

Pon este código HTML en un fichero llamado `index.html` en el directorio `"publico"` de la aplicación y el `bootstrap.min.css` de `http://getbootstrap.com/` en el directorio `"publico/css"`. Observa que el formulario hace un `POST` HTTP al destino final `"/markdown"` de nuestra aplicación. Realmente en este momento todavía no manejamos esta ruta, por lo tanto la vamos a añadir.

La ruta "/markdown"

El programa para manejar la ruta `'/markdown'` y para servir el fichero `index.html` al público en general se ve así:

```
package main

import (
    "net/http"

    "github.com/russross/blackfriday"
)

func main () {
    http.HandleFunc("/markdown", GeneraDesdeMarkdown)
    http.Handle("/", http.FileServer(http.Dir("publico")))
    http.ListenAndServe(":8080", nil)
```

```
    }

    func GeneraDesdeMarkdown(rw http.ResponseWriter, r *http.Request) {
        html := blackfriday.MarkdownCommon([]byte (r.FormValue("cuerpo")))
        rw.Write(html)
    }
```

Dividamos este código en pequeñas piezas para tener una mejor idea de lo que está pasando:

```
http.HandleFunc("/markdown", GeneraDesdeMarkdown)
http.Handle("/", http.FileServer(http.Dir("publico")))
```

Estamos utilizando los métodos `http.HandleFunc` y `http.Handle` para definir algún sencillo enrutamiento para nuestra aplicación. Es importante señalar que llamar a `http.Handle` en el patrón `"/"` actuará como una ruta comodín que captura todo, motivo por el cual definimos esa ruta al último. `http.FileServer` devuelve un `http.Handler` así que usamos `http.Handle` para asignar una cadena de caracteres como patrón a un controlador. El método alternativo, `http.HandleFunc`, utiliza un `http.HandlerFunc` en lugar de un `http.Handler`. Este posiblemente sea más conveniente, para pensar en el manejo de las rutas a través de una función en lugar de con un objeto.

```
func GeneraDesdeMarkdown(rw http.ResponseWriter, r *http.Request) {
    html := blackfriday.MarkdownCommon([]byte (r.FormValue("cuerpo")))
    rw.Write (html)
}
```

Nuestra función `GeneraDesdeMarkdown` implementa la interfaz `http.HandlerFunc` estándar y renderiza código HTML de un campo del formulario que contiene texto con formato `markdown`. En este caso, el contenido se recupera con `r.FormValue("cuerpo")`. Es muy común conseguir entrada desde el objeto `http.Request` que recibe el `http.HandlerFunc` como argumento. Algunos otros ejemplos de entrada son los miembros `r.Header`, `r.Body` y `r.URL`.

Finalizamos la petición escribiendo a nuestro `http.ResponseWriter`. Nótese que no enviamos explícitamente un código de respuesta. Si escribimos a la respuesta sin un código, el paquete `net/http` asumirá que la respuesta es `200 OK`. Esto significa que si sucedió algo malo para Go, debemos establecer el código de respuesta a través del método `rw.WriteHeader()`.

```
http.ListenAndServe(":8080", nil)
```

La última parte de este programa inicia el servidor, le pasamos `nil` como nuestro controlador, con lo cual asume que las peticiones HTTP serán manejadas de manera predeterminada por los paquetes `net/http` y `http.ServeMux`, que se configuran utilizando `http.Handle` y `http.HandleFunc`, respectivamente.

Y eso es todo lo que se necesita para poder generar código HTML a partir de `markdown` con un servicio en Go. Es una sorprendentemente pequeña cantidad de código para la cantidad de trabajo pesado que hace. En el próximo capítulo aprenderemos cómo desplegar esta aplicación en la web usando `Heroku`.

Desplegando

Heroku facilita el despliegue de aplicaciones. Es una plataforma perfecta para pequeñas aplicaciones web de mediano tamaño que están dispuestas a sacrificar un poco de flexibilidad en infraestructura, para ganar un entorno bastante indoloro para desplegar y mantener las aplicaciones web.

Elegí desplegar nuestra aplicación web en Heroku en beneficio de este tutorial porque en mi experiencia, esta ha sido la forma más rápida de tener una aplicación web en marcha y funcionando en poco tiempo. Recuerda que el objetivo de este tutorial es cómo construir aplicaciones web en Go y no quedar atrapados en toda la distracción del aprovisionamiento, configuración, implementación y mantenimiento de las máquinas que ejecutan nuestro código Go.

Configuración inicial

Si aún no tienes una cuenta Heroku, crea una en id.heroku.com/signup. Es rápido, fácil y gratuito.

La gestión y configuración de aplicaciones se realiza a través del cinturón de herramientas de Heroku, que es una herramienta de línea de órdenes libre mantenida por Heroku. La usaremos para crear nuestra aplicación en Heroku. Lo puedes conseguir desde toolbelt.heroku.com.

Cambiando el código

Para asegurarte de que la aplicación del capítulo anterior trabajará en Heroku, tendremos que hacer algunos cambios. Heroku nos da una variable de entorno llamada `PORT` y espera que nuestra aplicación web se vincule a ella. Empezaremos importando el paquete " `os` " para que podamos enganchar esa variable de entorno `PORT` :

```
import (
    "net/http"
    "os"

    "github.com/russross/blackfriday"
)
```

A continuación, tenemos que enganchar la variable de entorno, comprobando si se ha definido, y si es que deberíamos vincurlanos a ella en lugar de a nuestro puerto definido en el código (`8080`).

```
puerto := os.getenv("PORT")
if puerto == "" {
    puerto = "8080"
}
```

Por último, queremos vincular ese puerto en nuestra llamada a `http.ListenAndServe` :

```
http.ListenAndServe(":" + puerto, nil)
```

El código final debe ser similar a este:

```
package main

import (
    "net/http"
    "os"

    "github.com/russross/blackfriday"
)
```

```
func main () {
    puerto: = os.getenv("PORT")
    if puerto == "" {
        puerto = "8080"
    }

    http.HandleFunc("/markdown" , GeneraDesdeMarkdown)
    http.Handle("/", http.FileServer(http.Dir("publico")))
    http.ListenAndServe(":" + puerto, nil)
}

func GeneraDesdeMarkdown (rw http.ResponseWriter, r *http.Request) {
    html := blackfriday.MarkdownCommon([]byte (r.FormValue("cuerpo")))
    rw.Write(html)
}
```

Configuración

Necesitamos un par de pequeños ficheros de configuración para decirle a Heroku cómo debe ejecutar nuestra aplicación. El primero de ellos es `Procfile`, el cual nos permite definir qué procesos deben funcionar para nuestra aplicación. De manera predeterminada, Go nombrará el ejecutable después del directorio que contiene tu paquete principal. Por ejemplo, si mi aplicación web vive en `GOPATH/github.com/codegangsta/bwag/despliegue`, mi `Procfile` se ve así:

```
web: despliegue
```

Específicamente para ejecutar aplicaciones Go, también necesitamos especificar un fichero `.godir` para decirle a Heroku cual carpeta de hecho es el directorio base de nuestro paquete.

```
despliegue
```

Despliegue

Una vez que todas estas cosas están en su lugar, Heroku facilita la implementación.

Inicia el proyecto como un repositorio Git:

```
git init
git add -A
git commit -m " inicial"
```

Crea tu aplicación Heroku (especificando el paquete de construcción Go):

```
heroku create -b https://github.com/kr/heroku-buildpack-go.git
```

Súbelo a Heroku y ¡ve cómo se despliega tu aplicación!

```
git push heroku master
```

Ve tu aplicación en todo su esplendor en tu navegador:

```
heroku open
```

Enrutamiento URL

Para algunas aplicaciones simples, el valor predeterminado `http.ServeMux` te puede llevar muy lejos. Si necesitas más poder en la forma de analizar sintácticamente los puntos finales del URL y encaminarlos al controlador adecuado, posiblemente tengas que utilizar una biblioteca de enrutamiento de terceros. Para este tutorial, vamos a utilizar la conocida biblioteca `github.com/gorilla/mux` como nuestro enrutador. `github.com/gorilla/mux` es una gran opción para un enrutador, ya que tiene una interfaz que le es familiar a los usuarios de `http.ServeMux`, sin embargo, tiene un montón de características adicionales construidas en torno a la idea de encontrar el `http.Handler` adecuado para la ruta al URL dado.

En este ejemplo, crearemos algún enrutamiento para un recurso `REST` llamado "mensajes". A continuación definiremos mecanismos para ver el índice, mostrar, crear, actualizar, destruir y editar mensajes. Afortunadamente con `github.com/gorilla/mux`, no tenemos que hacer demasiadas operaciones copiar/pegar para lograrlo.

```
package main

import (
    "fmt"
    "net/http"

    "github.com/gorilla/mux"
)

func main () {
    r:. = mux.NewRouter()StrictSlash(false)
    r.HandleFunc("/", ControladorInicio)

    // Colección de mensajes
    mensajes: = r.Path("/mensajes").Subrouter()
    mensajes.Methods("GET").HandlerFunc(ControladorIndiceMensajes)
    mensajes.Methods("POST").HandlerFunc(ControladorCreaMensajes)

    // Mensaje singular
    mensaje: = r.PathPrefix("/mensajes/{id}").Subrouter()
    mensaje.Methods("GET").Path("/editar").HandlerFunc(ControladorEditarMensaje)
    mensaje.Methods("GET").HandlerFunc(ControladorMostrarMensaje)
    mensaje.Methods("PUT", "POST").HandlerFunc(ControladorActualizaMensaje)
    mensaje.Methods("DELETE").HandlerFunc(ControladorEliminaMensaje)

    fmt.Println("Iniciando el servidor en :8080")
    http.ListenAndServe(":8080", r)
}

func ControladorInicio(rw http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(rw, "Inicio")
}

func ControladorIndiceMensajes(rw http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(rw, "índice de mensajes")
}

func ControladorCreaMensajes(rw http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(rw, "crear mensajes")
}

func ControladorMostrarMensaje(rw http.ResponseWriter, r *http.Request) {
    id: = mux.Vars(r)["id"]
    fmt.Fprintln(rw, "mostrando mensaje", id)
}

func ControladorActualizaMensaje(rw http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(rw, "actualizando mensaje")
}

func ControladorEliminaMensaje(rw http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(rw, "borrar mensaje")
}

func ControladorEditarMensaje(rw http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(rw, "editar mensaje")
}
```

Ejercicios

1. Explora la documentación de `github.com/gorilla/mux`.
2. Juega con los diferentes métodos de cadenas de caracteres para crear filtros y subenrutadores.
3. Averigua qué tan bien juega `github.com/gorilla/mux` con el `http.Handler` existente tal como `http.FileServer`.

Semiware

Si tienes algo de código que necesitas ejecutar en cada petición, independientemente de la ruta que eventualmente se acabará invocando, necesitas alguna manera de apilar `http.Handlers` uno encima del otro y ejecutarlos en secuencia. Este problema se resuelve elegantemente a través de paquetes de semiware. `Negroni` es un paquete de semiware popular que facilita la construcción y apilamiento de semiware, manteniendo intacta la naturaleza composible del ecosistema web de Go.

`Negroni` viene con algún semiware predeterminado tal como el registro cronológico de eventos, recuperación de errores, y la porción de fichero estático. Así que fuera de la caja `Negroni` te proporcionará un gran valor sin gastar demasiado.

El siguiente ejemplo muestra cómo utilizar una pila `Negroni` con su semiware integrado y cómo crear tu propio semiware personalizado.

```
package main

import (
    "log"
    "net/http"

    "github.com/codegangsta/negroni"
)

func main() {
    // Semiware pila
    n := negroni.New(
        negroni.NewRecovery(),
        negroni.HandlerFunc(MiSemiware)
        negroni.NewLogger(),
        negroni.NewStatic(http.Dir("publico"))
    )

    n.Run(":8080")
}

func MiSemiware(rw http.ResponseWriter, r *http.Request, siguiente http.HandlerFunc) {
    log.Println("Inicio de sesión en la ruta...")

    if r.URL.Query().Get("password") == "secreto123" {
        siguiente(rw, r)
    } else {
        http.Error(rw, "no autorizado", 401)
    }

    log.Println("De vuelta inicia sesión en la ruta..." )
}
```

Ejercicios

1. Piensa en algunas buenas ideas de semiware y trata de implementarlas utilizando `Negroni` .
2. Explora cómo puedes mejorar `Negroni` con `github.com/gorilla/mux` utilizando la interfaz `http.Handler` .
3. Juega creando pilas `Negroni` para ciertos grupos de rutas en lugar de crear la pila de toda la aplicación.

Renderizando

El renderizado es el proceso de tomar datos de la aplicación o desde la base de datos y reproducirlos en el cliente. El cliente puede ser un navegador que consume HTML, o puede ser otra aplicación que utiliza `JSON` como formato de serialización. En este capítulo aprenderás cómo representar estos dos formatos utilizando los métodos que nos ofrece Go en la biblioteca estándar.

JSON

Rápidamente JSON se ha convirtido en el formato de serialización ubicua para las API web, por lo que puede ser el más relevante a la hora de aprender a construir aplicaciones web usando Go. Afortunadamente, Go facilita el trabajo con JSON -es muy fácil convertir estructuras Go existentes a JSON usando el paquete `encoding/json` de la biblioteca estándar.

```
package main

import (
    "encoding/json"
    "net/http"
)

type Libro struct {
    Título   string  `json:"título"`
    Autor    string  `json:"autor"`
}

func main() {
    http.HandleFunc("/", MuestraLibros)
    http.ListenAndServe(":8080", nil)
}

func MuestraLibros(w http.ResponseWriter, r *http.Request) {
    libro: = {Libro "Construyendo Aplicaciones Web con Go", "Jeremy Sáenz"}

    js, err := json.Marshal(libro)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.Header().Set("Content-Type", "application/json")
    w.Write(js)
}
```

Ejercicios

1. Lee la documentación de la API JSON y averigua cómo cambiar el nombre e ignorar campos para serialización JSON.
2. En lugar de utilizar el método `json.Marshal`, intenta utilizar la API `json.Encoder`.
3. Averigua cómo imprimir nuestro JSON con el paquete `encoding/json.`

Plantillas HTML

Servir HTML es una tarea importante para algunas aplicaciones web. Go cuenta con uno de mis lenguajes de plantilla favorito hasta la fecha. No por sus características, sino por su sencillez y marco de seguridad. Renderizar plantillas HTML es casi tan fácil como representar JSON usando el paquete `net/html/template` de la biblioteca estándar.

Este es el código fuente para la prestación de plantillas HTML:

```
package main

import (
    "html/template"
    "net/http"
    "path"
)

type Libro struct {
    Título string
    Autor  string
}

func main() {
    http.HandleFunc("/", MuestraLibros)
    http.ListenAndServe(":8080", nil)
}

func MuestraLibros(w http.ResponseWriter, r *http.Request) {
    libro: = {Libro "Construyendo Aplicaciones Web con Go", "Jeremy Sáenz"}

    fp: = path.Join("plantillas", "index.html")
    plant, err: = template.ParseFiles(fp)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    if err := plant.Execute(w, libro); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

La siguiente es la plantilla que usaremos. Se debe colocar en el fichero `plantillas/index.html` del directorio desde donde se ejecuta tu programa:

```
<html>
  <h1>{{.Título}}</h1>
  <h3>por {{.Autor}}</h3>
</html>
```

Ejercicios

1. Mira la documentación de los paquetes `text/template` y `html/template`. Juega un poco con el lenguaje de plantillas para darte una idea de sus metas, fortalezas y debilidades.
2. En el ejemplo procesamos el fichero en cada petición, lo cual puede ser un montón de sobrecarga afectando el rendimiento. Experimenta procesando los archivos al inicio de tu programa y ejecutalo en tu `http.Handler` (sugerencia: usa el método `Copy()` de `html.Template`).
3. Experimenta procesando y utilizando varias plantillas.

Usando el paquete render

Si quieres que renderizar JSON y HTML sea aún más simple, está el paquete `github.com/unrolled/render`. Este paquete fue inspirado por el `martini-contrib/render` y es mi amparo cuando se trata de procesar datos para su presentación en mis aplicaciones web.

```
package main

import (
    "net/http"

    "gopkg.in/unrolled/render.v1"
)

func main() {
    r: = render.New(render.Options{})
    mux: = http.NewServeMux()

    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte ("Bienvenido, ahora visita nuestras subpáginas."))
    })

    mux.HandleFunc ("/datos", func(w http.ResponseWriter, r *http.Request) {
        r.Data(w, http.StatusOK, []byte ("Algunos datos binarios aquí."))
    })

    mux.HandleFunc("/json", func(w http.ResponseWriter, r *http.Request) {
        r.JSON(w, http.StatusOK, map[string]string {"hola": "json"})
    })

    mux.HandleFunc("/html", func(w http.ResponseWriter, r *http.Request) {
         // asume que tienes una plantilla en ./plantillas llamada "ejemplo.tmpl"
        // $ mkdir -p plantillas && echo "<h1>Hola {{.}}.</h1>"> plantillas/ejemplo.tmpl
        r.HTML(w, http.StatusOK, "ejemplo", nil)
    })

    http.ListenAndServe(":8080", mux)
}
```

Ejercicios

1. Diviértete jugando con todas las opciones disponibles al llamar a `render.New()`.
2. Trata de usar la función auxiliar `{{.yield}}` y un diseño con plantillas HTML.

Bases de datos

Una de las preguntas más frecuentes que recibo sobre el desarrollo web en Go es cómo conectarse a una base de datos SQL. Afortunadamente, Go tiene un fantástico paquete de SQL en la biblioteca estándar que nos permite utilizar una gran cantidad de controladores para diferentes bases de datos SQL.

En este ejemplo vamos a conectarnos a una base de datos SQLite, pero la sintaxis (salvo alguna pequeña semántica SQL) es la misma para una base de datos MySQL o PostgreSQL.

```
package main

import (
    "database/sql"
    "fmt"
    "log"
    "net/http"

    _ "github.com/mattn/go-sqlite3"
)

func main() {
    bd := NuevaBD()
    log.Println("Escucha en: 8080")
    http.ListenAndServe(":8080", MuestraLibros(bd))
}

func MuestraLibros(bd *sql.DB) http.Handler {
     return http.HandlerFunc(func (rw http.ResponseWriter, r *http.Request) {
         var título, autor string
        err: = bd.QueryRow("select title, author from libros").Scan(&título, &autor)
         if err != nil {
             panic(err)
        }

        fmt.Fprintf(rw, "El primer libro es '%s' de '%s'", título, autor)
    })
}

func NuevaBD() *sql.DB {
    bd, err := sql.Open("sqlite3", "ejemplo.sqlite")
     if err != nil {
         panic(err)
    }

    _, err = bd.Exec("create table if not exists books(titulo text, autor text)" )
     if err != nil {
         panic(err)
    }

    return bd
}
```

Ejercicios

1. Haz uso de la función `Query` en nuestro ejemplo `sql.DB` para extraer una colección de filas y asignarlos a estructuras.
2. Añade la capacidad de insertar nuevos registros en la base de datos mediante el uso de un formulario HTML.
3. Busca en `github.com/jmoiron/sqlx` y observa las mejoras realizadas sobre el paquete `database/sql` existente en la biblioteca estándar.

Controladores

Los controladores son un tema bastante familiar en otras comunidades de desarrollo web. Como la mayoría de los desarrolladores web está alrededor de una interfaz de poderosos red/http, ni muchas implementaciones del regulador han pegado fuerte. Sin embargo, hay un gran beneficio en el uso de un modelo controlador. Permite limpias y bien definidas abstracciones más allá de lo de la interfaz de controlador de red/http solo puede proporcionar.

Dependencias con Handler

En este ejemplo vamos a experimentar con la construcción de nuestra propia implementación de controlador utilizando algunas de las características estándar en Go. Pero primero, vamos a empezar con los problemas que estamos tratando de resolver.

Digamos que estamos usando el paquete `render` de la biblioteca estándar del que hablamos en capítulos anteriores:

```
var render = render.New(render.Options{})
```

Si queremos que nuestros `http.Handler`s puedan acceder a nuestra instancia `render.Render`, tenemos un par de opciones.

1. **Utilizar una variable global**: Esto no es demasiado malo para programas pequeños, pero cuando el programa crece esto rápidamente se convierte en una pesadilla de mantenimiento.
2. **Pasar la variable a través de un cierre al `http.Handler`**: Esta es una gran idea, y la deberíamos estar usando la mayor parte del tiempo.

La aplicación termina pareciéndose a esto:

```
func MiControlador (r *render.Render) http.Handler {
   return http.HandlerFunc(func (rw http.ResponseWriter, r *http.Request) {
     // ahora podemos acceder a 'r'
  })
}
```

Caso para controladores

Cuando el programa crece en tamaño, comenzarás a notar que muchos de tus `http.Handler`s compartirán las mismas dependencias y tendrás una gran cantidad de estos cierres `http.Handler`s con los mismos argumentos. La forma en que me gusta limpiar esto es escribiendo una implementación del controlador base que me ofrezca un par de pequeñas victorias:

1. Me permite compartir las dependencias entre `http.Handler`s que tienen objetivos o conceptos similares.
2. Evita las variables globales y funciones para facilitar las pruebas/simulaciones.
3. Me proporciona un mecanismo más centralizada e idiomatico de Go -tal como el manejo de errores.

¡Gran parte de los controladores nos proporcionan todas estas cosas sin importar un paquete externo! La mayor parte de esta funcionalidad viene de un uso inteligente del conjunto de características Go, a saber estructuras Go e incrustación. Echemos un vistazo a la implementación.

```
package main

import "net/http"

// Acción define una firma de función estándar para que la podamos utilizar
```

```
// al crear acciones del controlador.  Una acción del controlador básicamente
// es un método asociado a un controlador.
type Acción func(rw http.ResponseWriter, r *http.Request) error

// Este es nuestro controlador base
type ControladorAplic struct {}

// La función Acción ayuda con el manejo de errores en un controlador
func (c *ControladorAplic) Acción(a Acción) http.Handler {
     return http.HandlerFunc(func (rw http.ResponseWriter, r *http.Request) {
         if err := a(rw, r); err != nil {
            http.Error(rw, err.Error(), 500)
        }
    })
}
```

¡Eso es todo! Esa es toda la aplicación en que debemos tener el poder de los controladores a nuestro alcance. Todo lo que resta por hacer es implementar un controlador de ejemplo:

```
package main

import (
     "net/http"

    "gopkg.in/unrolled/render.v1"
)

type miControlador struct {
    ControladorAplic
    *render.Render
}

func (c *miControlador) Índice(rw http.ResponseWriter, r *http.Request) error {
    c.JSON(rw, 200, map[string]string {"Hola": "JSON"})
     return nil
}

func main() {
    c := &miControlador {Render: render.New(render.Options{})}
    http.ListenAndServe(":8080", c.Acción(c.Índice))
}
```

Ejercicios

1. Extiende miControlador para que tenga múltiples acciones para diferentes rutas de la aplicación.
2. Juega con más implementaciones de controladores, sé creativo.
3. Anula el método Acción en miControlador al presentar una página de error HTML.

Consejos y trucos

Envuelve un `http.HandlerFunc` en un cierre.

A veces, al iniciar quieres pasar datos a un `http.HandlerFunc`. Esto se puede hacer fácilmente mediante la creación de un cierre de la `http.HandlerFunc`:

```
func MiControlador(basededatos *sql.DB) http.Handler {
   return http.HandlerFunc(func (rw http.ResponseWriter, r *http.Request) {
     // ahora, aquí tienes acceso a *sql.DB
  })
}
```

Utilizando gorila / contexto para datos de una petición específica

Es bastante frecuente que necesitemos almacenar y recuperar datos que son específicos a la petición HTTP actual. Utiliza gorila / contexto para asignar valores y recuperarlos más tarde. Contiene un mutex global sobre un mapa del objeto petición.

```
func MiControlador(w http.ResponseWriter, r *http.Request) {
    valor: = context.Get(r, "miClave")

    // devuelve ("bar", true)
    valor, bien: = context.GetOk(r, "miClave")
     // ...
}
```

Avanzando

¡Lo lograste! Has visto una muestra de las herramientas de desarrollo web y bibliotecas de Go. Al momento de escribir estas líneas, este libro todavía está en proceso de desarrollo.

Esta sección está reservada para más recursos en la web sobre Go para continuar tu aprendizaje.

Glossary

markdown

Es una herramienta de conversión de texto plano a HTML para los escritores web. Markdown te permite escribir texto plano usando una herramienta fácil de leer, fácil de escribir y después convertirlo a código XHTML (o HTML) estructuralmente válido.

semiware

`middleware` en inglés, es el software "pegamento" que ayuda a los programas y bases de datos (que pueden estar en diferentes equipos) a trabajar juntos. Su función más básica es la de permitir la comunicación entre diferentes piezas de software.