Tecnologías para desarrollos en internet
Manual CRUD: Beego

Kihui-DEV

Fecha: 18/09/16
Facultad de Ciencias UNAM

Índice

1. Introducción

Hola

2. Instalación de Go 1.7

Paquetes de instalación para *Apple OS X, Microsoft Windows y Linux* son provistos en la página oficial de descargas de Go. También viene incluido entre las opciones el código fuente del compilador del lenguaje junto con instrucciones para llegar a una instalación tan completa como las demás.

A continuación, presentamos la instalación para *Linux y OS X.* [1]

2.1. Descarga

Linux

Para obtener el paquete de Golang[2]
Ejecutar:

```
$ wget https://golang.org/doc/install?download=go1.7.1.linux-amd64.tar.gz
```

O bien descargarlo directamente desde este enlace:

Go 1.7 para Linux y OS X.

Mac OS X

Alternativa a las siguientes instrucciones, existe la opción de descargar el fichero *.pkg* instalable de *Go* que automatiza la configuración para este sistema operativo.[3]

Para obtener una configuración inicial personalizada *Mac OS X* utilice el paquete descargable para *Linux* disponible en la sección anterior.

El resto de la configuración se sigue de la misma manera para ambas plataformas.

2.2. Configuración

Primero extraemos los archivos del paquete comprimido sobre algún directorio.
Para descomprimir el paquete sobre */usr/local* como es usual:

```
# tar -C /usr/local -xzf go1.7.1.linux-amd64.tar.gz
```

Al finalizar la extracción, procedemos a establecer una ruta a los binarios de las herramientas de *Go*, y así tenerlas disponibles en cada sesión.

Agregar esta línea en el script de inicio (típicamente sobre */user/local/profile* si se desea hacer una instalación general en el sistema operativo o en particular para el usuario en curso, usar en cambio *~/.profile*):

```
export PATH=$PATH:/usr/local/go/bin
```

[1] Si se desea revisar la configuración para *Windows*, hacer uso del siguiente enlace: MSI Installer
[2] Actualmente la versión 1.7.1
[3] Instalador para Mac OS X

2.3. Prueba

Crear un directorio que haga de workspace para la prueba.
Por ejemplo:

```
$ mkdir ~/go
```

Asignar la variable **GOPATH**[4] para que apunte a tal dirección:

```
$ export GOPATH=$HOME/go
```

Bien podemos hacer persistente este cambio agregando la misma línea al script de inicio que editamos en la sección anterior (ir a Descarga 2.1 y configuración 2.2).

A continuación, creamos dentro de ese directorio *src/hola*. Y dentro de *hola/* un fichero nuevo de nombre *hola.go*:

```
package main

import "fmt"

func main() {
    fmt.Printf("hola, mundo\n")
}
```

Luego, desde cualquier ubicación podemos ejecutar
esto:

```
$ go install hola
```

Esto producirá un ejecutable *hola* dentro de el directorio *go/bin/*, que podemos ejecutar utilizando lo siguiente:

```
$ $GOPATH/bin/hola
```

o directamente sobre el directorio donde se encuentre el ejecutable:

```
$ ./hola
```

Si produce la salida “hola, mundo”, quiere decir que nuestra instalación fue exitosa.

[4]La misma línea puede agregarse al script de inicio *profile* manejado en la sección de configuracion para evitar ejecutarla y mantener los proyectos y aplicaciones de *Go* ubicados en un sólo directorio.

3. Instalación de Beego

Para instalar la última versión de Beego[5] utilizamos el siguiente comando:

```
$ go get github.com/astaxie/beego
```

Para compilar y correr nuestros proyectos necesitaremos instalar Bee[6] también :

```
$ go get github.com/beego/bee
$ go install github.com/beego/bee
```

Para poder utilizar *bee* sin necesidad de ir a la carpeta de binarios de *Go*, podemos crear un enlace simbólico que apunte precisamente al ejecutable.

```
# ln -s $GOPATH/bin/bee /usr/bin/bee
```

4. Creación de proyecto

Para crear un proyecto en Beego, necesitamos ir al directorio de nuestro $GOPATH, donde escribimos el siguiente comando:

```
$ bee new beego-crud
```

Podremos ver que se han creado las siguientes carpetas y archivos necesarios para nuestra aplicación:

```
beego-crud/
|-- conf/
|   |__ app.conf
|-- controllers/
|   |__ default.go
|-- main.go
|-- models/
|-- routers/
|   |__ router.go
|-- static/
|   |--- css/
|   |--- img/
|   |___ js/
|-- tests/
|   |__ default_test.go
|-- views/
    |__ index.tpl
```

[5] A la fecha de elaboración de este manual: 1.7.1

[6] A la fecha de elaboración de este manual: 1.5.2

4.1. Estructura del proyecto - MVC

Modelo

1) *conf/*

2) *models/*

Vista

3) *views/*

Controlador

4) *controllers/*

5) *routers/*

Extra

6) *static/*

7) *tests/*

4.2. Prueba del servidor local

Finalmente para correr el nuevo proyecto que hemos creado, hacemos lo siguiente:

```
$ cd $GOPATH/src/beego-crud
$ bee run
```

Ingresamos *localhost:8080* como dirección en el navegador para encontrarnos con la siguiente página de inicio:

Welcome to Beego

Beego is a simple & powerful Go web framework which is inspired by tornado and sinatra.

Official website: beego.me / Contact me: astaxie@gmail.com

5. Elaboración del CRUD bajo MVC

En esta sección revisaremos la implementación de un CRUD (*Create-Read-Update-Delete*) sobre la aplicación que creamos en la sección anterior (s. 4). Comenzaremos explicando los documentos que tenemos que modificar para tener una configuración exitosa y la forma de escribir el código de acuerdo al MVC (*Modelo Vista Controlador*), siguiendo la estructura que nos proporciona *Beego* para éste.

5.1. Preparación

Todo proyecto de Beego, como ya vimos, está organizado segun el MVC. Es precisamente sobre el archivo de **models.go** que escribiremos nuestras relaciones expresadas con atributos asignados de acuerdo a los tipos de dato que maneja Go para después verlos reflejados en tablas en una base datos ya explorable. Bien podría ser en sentido inverso, desde un esquema *SQL* generar los modelos para nuestro proyecto de Beego, pero en nuestro caso decidimos implementarlo en esta dirección porque nuestra intención es enfocarnos en Beego, no en SQL.

A continuación presentamos la configuración de la base de datos y un mapeo de una sola relación con la que trabajaremos todas las operaciones.

Base de datos

Beego-ORM es la herramienta de Mapeo Objeto-Relacional de *Beego* escrita en *Go*. Según la página oficial está inspirada en *Django ORM* y *SQLAlchemy*. Se advierte que al estar en desarrollo, no se garantiza un 100 % de compatibilidad y es posible encontrar algunos "bugs", que el equipo de desarrollo de Beego promete solucionar apenas se reporten.

Lo primero que hay que considerar es la conexión con la base de datos que necesitamos para nuestro modelo. Beego provee soporte para tres sistemas manejadores de bases de datos con sus respectivos "drivers":

- MySQL
- Sqlite
- Postgres

Por su sencilla configuración y mayoría de ejemplos de implementación junto con Beego, escogimos MySQL por lo que los siguientes puntos se ejemplificarán con el supuesto de que trabajamos con este SMBD[7] y que ya se cuenta con una instalación funcional[8].

Lo primero que haremos será crear nuestra base de datos:

[7]Para revisar más ampliamente las opciones de bases de datos recomendamos visitar este apartado de la documentación: ORM Usage

[8]En el manual de MySQL v5.7 viene una amplia guía de instalación. Recomendamos los siguientes apartados para distintos sistemas operativos: Windows | OS X | Linux

```
$ mysql -u root -p
MariaDB [(none)]> create database beego;
MariaDB [(none)]> exit
```

Si no experimentamos ninguna dificultad con esto, podemos continuar con la configuración de nuestro proyecto.

Configuración

Todo proyecto de *Beego* cuenta con un archivo de configuración. Por defecto se interpreta el archivo **app.conf** bajo el directorio *conf/* de la aplicación. Inicialmente, dicho archivo está escrito en formato INI [9], una sintáxis sencilla, compuesta de secciones, propiedades y valores sobre archivos de texto plano. Por ejemplo, en ella podremos escribir configuraciones del estilo:

```
[seccion]
nombre_propiedad = valor
```

Nosotros conservaremos el nombre de la aplicación (appname), el puerto http (httport) y el modo de ejecución (runmode); propiedades que ya vienen incluídas al momento de creación de nuestra nueva aplicación. Adicionalmente escribiremos las siguientes propiedades:[10]

```
mysqluser = "root" ;el nombre del usuario de MySQL que es dueño de la base
mysqlpass = "magdario" ;la contraseña de acceso del usuario MySQL
mysqldb = "beego" ;el nombre de la base de datos que creamos anteriormente
mysqlurls = "127.0.0.1" ;la dirección del host de la base de datos (localhost)
```

Gracias a esta configuración la aplicación podrá establecer conexión con nuestra base de datos para llevar a cabo las operaciones.

Modelos

Como ya señalamos previamente, para realizar el CRUD, nos auxiliaremos de una sola relación en la base de datos. Para registrarla, basta definir una estructura sobre el archivo de Go **models.go**, ubicado en la carpeta *models/* de nuestra aplicación. Aquí mostramos un ejemplo de cómo podría ser un modelo de usuario[11]:

```
type Usuario struct {
    Id          int
    Nombre      string
    Edad        int
}
```

[9] También se permiten los siguientes formatos: XML, JSON y YAML

[10] El caracter ";" denota una línea de comentario. No es necesario al final de cada línea.

[11] Para revisar con más detenimiento el código, pruebe visualizar el archivo del proyecto en GitHub: models.go

Para continuar con el mapeo y ver nuestro nuevo modelo reflejado en la base de datos que configuramos previamente, ahora nos dirigimos al archivo **main.go** en la raíz del proyecto y agregamos las siguientes configuraciones a los "imports" del programa:

```
"github.com/astaxie/beego/orm"
_ "github.com/go-sql-driver/mysql"
models "beego-crud/models"
```

Esto con el fin de que podamos utilizar Beego-ORM, el driver del SMBD y los modelos que declaremos.
Luego, creamos un nuevo método método *init()*, en el que escribiremos:

```
func init() {
    orm.RegisterDriver("mysql", orm.DRMySQL)
    orm.RegisterDataBase("default", "mysql", "root:magdario@/beego?charset=utf8")
    orm.RegisterModel(new(models.Usuario))
    orm.RunCommand()
}
```

Con esto, registramos tanto el driver que utilizaremos como la base de datos sobre la que se realizaran las operaciones de los modelos que "demos de alta" en este método. Es importante señalar que si no registramos los modelos aquí o en su archivo correspondiente, no serán "visibles" para nuestra aplicación y no será posible efectuar cambios sobre ellos. Nosotros elegimos registrar el modelo de *usuario* en el método de inicialización, por la forma explícita que nos provee.

Recomendamos revisar la forma resultante del archivo: main.go

Finalmente, compilamos **main.go** y lo ejecutamos con los parámetros necesarios para realizar el mapeo y la sincronización con la base de datos.

Desde *$GOPATH/beego-crud* ejecutamos:

```
$ go build main.go
$ ./main orm syncdb
```

Para revisar que todo marcha bien podemos pedirle al "prompt" de MySQL que nos muestre las tablas de nuestra base de datos:

```
MariaDB [(none)]> use beego;
MariaDB [beego]> show tables;
```

Que debe resultarnos en algo similar a:

```
+-----------------+
| Tables_in_beego |
+-----------------+
| usuario         |
+-----------------+
```

Si en efecto nos encontramos con las tablas correspondientes a los modelos que registramos, ya podemos dedicarnos a escribir el controlador y las vistas pertinentes para nuestro CRUD, sin preocuparnos por la conexión con la base de datos.

A continuación presentamos las operaciones de un CRUD implementado en nuestro Framework Beego. Cada una viene separada con su sección correspondiente y los apartados que contiene se refieren a los archivos de la estructura del proyecto y cómo debemos modificarlos para elaborar la operación en cuestión.

5.2. Creación

Manejaremos esta operación como un registro de usuarios para alguna plataforma. Este registro será lo primero que tendremos en el "index" de la página que estamos desarrollando.

5.2.1. Controllers

Lo primero que debemos considerar es la función del controlador que manejara nuestra operación de creación. En el archivo controllers/**default.go**[12] encontraremos la estructura del controlador principal, sus métodos serán los que manejen las respuestas del servidor ante las peticiones de la interfaz. Antes de comenzar con nuestra función de agregado o creación de usuario, importaremos los modelos de la aplicación que ya hemos considerado[13], la herramienta *ORM* (mapeo y conexión con la base de datos), una herramienta de validación de datos y la herramienta estándar de Go *fmt*:

```
import (
  "github.com/astaxie/beego"
  models "beego-crud/models"
  "github.com/astaxie/beego/orm"
  "github.com/astaxie/beego/validation"
  "fmt"
)
```

Posteriormente, crearemos efectivamente nuestra función de agregado, que llamaremos *Add()*. Prestemos particular atención a la estructura (recordemos que Go, como C, es un lenguaje estructurado) llamada MainController.

```
// Método que permite registrar un nuevo usuario
func (this *MainController) Add() {
  this.TplName = "index.tpl"
  o := orm.NewOrm()
```

[12]Podemos cambiar este nombre de archivo si nos parece pertinente, pero por simplicidad lo dejaremos así, basta con que nuestras funciones de controlador se ubiquen en esa carpeta.

[13]Preparación - Modelos 5.1

```
    o.Using("default")

    // Se crea un nuevo usuario
    u := models.Usuario{}
    // Se guardan los datos del formulario en el nuevo usuario
    if err := this.ParseForm(&u); err != nil {
      // Si hay errores en el formulario
      beego.Error("Error: ", err)
    } else {
      this.Data["Usuarios"] = u
      // Se validan los datos del usuario
      valid := validation.Validation{}
      isValid, _ := valid.Valid(u)
      if !isValid {
        this.Data["Errors"] = valid.ErrorsMap
        beego.Error("Datos inválidos")
      } else {
        // Se inserta el usuario nuevo en la base de datos
        id, err := o.Insert(&u)
        if err == nil {
          msg := fmt.Sprintf("Nuevo usuario con id: ", id)
          beego.Debug(msg)
        } else {
          msg := fmt.Sprintf("No se pudo registrar usuario: ", err)
          beego.Debug(msg)
        }
      }
    }

    var usuarios []*models.Usuario
    num, err := o.QueryTable("usuario").All(&usuarios)

    if err != orm.ErrNoRows && num > 0 {
      this.Data["records"] = usuarios
    }
  }
```

Veamos que una de las primeras instrucciones que se dan es definir un template para la función de agregado del controlador, ello lo consideraremos más adelante (sección 5.2.3). A continuación, agregaremos una configuración en los *routers* para poder hacer uso de nuestra nueva función.

5.2.2. Routers

Como ya hemos señalado anteriormente, el archivo de las rutas o "urls" de nuestra aplicación web, se encuentra bajo el directorio *routers/* con el nombre de **router.go**. En él nos encontraremos con una ruta por defecto, esta es la que nos muestra apenas visitamos el servidor de *Bee* luego de crear nuestra nueva aplicación (sección 4.2).

```
beego.Router("/", &controllers.MainController{})
```

Ahora, modificaremos esa línea para que sea acorde a nuestra operación de creación.

```
beego.Router("/", &controllers.MainController{}, "post:Add")
```

Y listo, nuestra aplicación llamará a la función de agregado en cuanto se haga una petición al servidor por esa ruta.

5.2.3. Views

Con el fin de que nuestra ruta previamente creada funcione, debemos crear un archivo *.tpl* que corresponda con la plantilla (template) que indicamos en la función del controlador que llama la ruta del servidor. En este caso la ruta *localhost:8080/*, y la función **Add()**.

Toda plantilla debe ubicarse bajo el directorio *views/* de la aplicación. Básicamente, consisten de un documento HTML con anotaciones propias del framework que hacen referencia a estructuras que maneja el controlador. Nosotros indicamos el template de la función de agregado como **index.tpl**, por lo que crearemos la plantilla correspondiente con dicho nombre. Esta parte la dejaremos a su consideración[14], por la cuestión del estilo, sin embargo, les dejamos un ejemplo en HTML[15] sin estilo para guiarlos en el proceso de elaboración:

```
<!DOCTYPE html>
 <html>
 <head>
   <title>Beego-CRUD</title>
   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
 </head>

 <body>
   <h3 class="beego">CRUD de Beego</h3>
   <h4>Agregar Usuario</h4>
   <form role="form" id="usuario" method="POST">
     <div class="">
```

[14]Es posible encontrar nuestro template funcional con *MaterialDesign* como framework de frontend aquí. Es importante recordar que la aplicación de Beego cuenta con un directorio *static/* para manejar los estilos de los templates.

[15]Nótese que el pattern del correo electrónico no está definido, por lo que recomendamos dirigirse al FINAL de la siguiente sección del documento para conocer las razones: 5.4.3

```
        <input id="nombre" name="nombre" value="{{.Usuario.Nombre}}" type="text" />
        <label for="nombre">Nombre</label>
      </div>
      <br/>
      <div>
        <input id="app" name="app" value="{{.Usuario.App}}" type="text" />
        <label for="app">Apellido Paterno</label>
      </div>
      <br/>
      <div class="">
        <input id="apm" name="apm" value="{{.Usuario.Apm}}" type="text" />
        <label for="apm">Apellido Materno</label>
      </div>
      <br/>
      <div class="">
        <input id="correo" name="correo" pattern="" value="{{.Usuario.Correo}}" type="text
        <label for="correo">Correo Electrónico</label>
      </div>
      <br/>
      <div class="">
        <input id="edad" pattern="[0-9]*?" name="edad" value="{{.Usuario.Edad}}" type="tex
        <label for="edad">Edad</label>
      </div>
      <br/>
      <input type="submit" value="Aceptar" />
    </form>
  </div>
</body>
</html>
```

5.3. Lectura

Esta operación será implementada para la página principal de nuestro CRUD, junto con la operación de creación. Esto con el fin de que sea observable que ambas operaciones funcionan sin necesidad de hacer consultas en la base de datos. Como tal, el usuario no tendrá interacción con esta operación, eventualmente será "modificable", o mejor dicho observable (en los cambios) con las operaciones restantes.

A continuación, se presenta la misma lógica para la operación de creación previa.

5.3.1. Controllers

Para este punto suponemos que ya se han expresado las importaciones (5.2.1) necesarias en el archivo de *default.go* que tiene nuestro controlador principal. Continuaremos con la elaboración del método que manejará la lectura o consulta de los usuarios en la base de datos que llamaremos **View()**:

```
// Método que muestra todos los usuarios
func (c *MainController) View() {
  // Nombre del template
  c.TplName = "index.tpl"
  // Se le asigna un modelo al formulario de la vista
  c.Data["Form"] = &models.Usuario{}
  // Se crea una conexión a la base
  o := orm.NewOrm()
  // Se indica que se usará la base por default
  o.Using("default")

  // Arreglo donde se guardarán los usuarios
  var usuarios []*models.Usuario
  // Se guardan los usuarios de la base en el arreglo
  num, err := o.QueryTable("usuario").All(&usuarios)

  if err != orm.ErrNoRows && num > 0 {
    // Se pasa el arreglo al contexto de la vista
    c.Data["records"] = usuarios
  }
}
```

Como se había mencionado, la funcionalidad de este método será observable desde la página principal, es por eso que asignamos el mismo template que para la operación de creación anterior. Es importante que tengamos presente el nombre de nuestra función, pues la utilizaremos en el siguiente apartado.

5.3.2. Routers

Con el fin de hacer disponible nuestra función, es debido incluirla en alguna de las rutas de la aplicación. Modificando el archivo **routers.go** podemos asignar una función a una o más rutas. Dado que esperamos que nuestra operación de lectura esté disponible desde la página principal al igual que nuestra operación de creación, simplemente modificaremos la línea que habíamos agregado previamente para incluír nuestra nueva operación de lectura. Por tanto, de tener está línea:

```
beego.Router("/", &controllers.MainController{}, "post:Add")
```

Agregaremos la función **View()** de la siguiente manera:

```
beego.Router("/", &controllers.MainController{}, "get:View;post:Add")
```

Finalmente, para poder probar nuestra nueva configuración será necesario editar el template de la página principal.

5.3.3. Views

El template que elegimos para la función de lectura, referente al archivo **index.tpl** del directorio de vistas de la aplicación, ya debe existir si hemos implementado la operación de creación del CRUD previamente. Simplemente agregaremos un contenedor pertinente para mostrar la información del modelo que nos devuelve nuestra función recién realizada:

```
<div>
<h4>Usuarios</h4>
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Correo</th>
      <th>Edad</th>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    {{range $record := .records}}
    <tr>
      <td>{{$record.Nombre}} {{$record.App}} {{$record.Apm}}</td>
      <td>{{$record.Correo}}</td>
      <td>{{$record.Edad}}</td>
    </tr>
    {{end}}
  </tbody>
</table>
</div>
```

Con estos cambios será posible visualizar los nuevos usuarios apenas los agreguemos sin necesidad de cambiar de página. Todo desde la ruta principal de la aplicación *localhost:8080/*.

5.4. Actualización

Supondremos que ya se ha implementado la operación de lectura como se describe en este documento y que ya contamos con registros existentes de usuarios en la base de datos. Continuaremos

con la descripción de los pasos en la lógica de la aplicación; los mismos seguidos en el desarrollo de las demás operaciones del CRUD.

5.4.1. Controllers

Como ya deben sospechar, luego de haber repetido los pasos dos veces con las operaciones anteriores, lo primero será escribir una función en el archivo de controlador principal **routers.go**. Por motivos de simplicidad, el nombre ideal nos parece ser **Update()**, pero es preciso repetir que el nombre no afectará su funcionamiento, siempre que lo respetemos para las implementaciones de los apartados siguientes. De lo contrario, podríamos confundir un simple error de referencia con uno inmerso en la lógica de nuestra aplicación. A continuación nuestra propuesta de código para el método de actualización:

```
// Método que permite modificar los datos de un usuario
func (this *MainController) Update() {
  this.TplName = "update.tpl"
  this.Data["Form"] = &models.Usuario{}
  o := orm.NewOrm()
  o.Using("default")

  // Se obtiene el parámetro id del url
  if id, err := this.GetInt("id"); err == nil {
    // Se obtiene el usuario con el id del parámetro
    u := models.Usuario{Id: id}
    if o.Read(&u) != nil {
      beego.Error("El usuario no existe")
    } else {
      // Se pasan los datos del usuario al contexto de la vista
      this.Data["Nom"] = u.Nombre
      this.Data["App"] = u.App
      this.Data["Apm"] = u.Apm
      this.Data["Correo"] = u.Correo
      this.Data["Edad"] = u.Edad
    }
    // Si se efectúa el método POST en la vista
    if this.Ctx.Input.Method() == "POST" {
      // Se guardan los datos del formulario en el usuario
      if err := this.ParseForm(&u); err != nil {
        beego.Error("Error: ", err)
      } else {
        this.Data["Usuarios"] = u
        valid := validation.Validation{}
        isValid, _ := valid.Valid(u)
        if !isValid {
```

```
            this.Data["Errors"] = valid.ErrorsMap
            beego.Error("Datos inválidos")
          } else {
            // Se actualiza el usuario con los nuevos datos en la base
            id, err := o.Update(&u)
            if err == nil {
              msg := fmt.Sprintf("Usuario actualizado: ", id)
              beego.Debug(msg)
            } else {
              msg := fmt.Sprintf("No se pudo actulizar usuario: ", err)
              beego.Debug(msg)
            }
          }
        }
        this.Redirect("/", 302)
        return
      }
    } else {
      beego.Error("Parámetro inválido: ", err)
    }
  }
```

Con este método listo, podemos proceder a configurarlo en las rutas de la aplicación y dejarlo disponible. A propósito de hacer más explícita la contraparte visual que le corresponde, hemos asignado el template de esta función con el mismo nombre: **update.tpl**; archivo que deberá existir en el directorio de vistas de la aplicación.

5.4.2. Routers

Anteriormente agregamos y, posteriormente modificamos, una sola ruta de la aplicación en la función **init()** de nuestro archivo de configuración de rutas **router.go**. Justo debajo (o arriba) de la línea que define la ruta principal de la aplicación, colocaremos una que corresponda a la función de actualización recién presentada.

```
beego.Router("/update", &controllers.MainController{}, "*:Update")
```

Apreciemos que difiere en argumentos con la línea de la ruta principal. El primero nos indica que esta funcionalidad es accesible desde *localhost:8080/update*, sin embargo, para que el usuario pueda hacer uso de la función adecuadamente no es conveniente que acceda de forma "manual" a dicha ruta de a aplicación, pues necesitamos información de entrada, el usuario que será modificado -en nuestro caso, el id de dicho elemento.

5.4.3. Views

Con la intención de resolver la situación planteada al final del apartado anterior, alteraremos un poco la vista principal de la aplicación para hacer posible la modificación de usuarios existentes

en la base de datos. Añadiremos una línea en la vista principal **index.tpl** (dentro de la definición de la tabla).

```
<tr>
  <td>{{$record.Nombre}} {{$record.App}} {{$record.Apm}}</td>
  <td>{{$record.Correo}}</td>
  <td>{{$record.Edad}}</td>
</tr>
```

Resultando en lo siguiente:

```
<tr>
  <td>{{$record.Nombre}} {{$record.App}} {{$record.Apm}}</td>
  <td>{{$record.Correo}}</td>
  <td>{{$record.Edad}}</td>
  <td><a href="/update?id={{$record.Id}}"><i>editar</i></a></td>
</tr>
```

Posteriormente, ya que hemos hecho accesible la ruta de la operación de actualización con el parámetro necesario para la función del controlador que le corresponde, obligatoriamente debemos incluir el template que exige en el directorio de vistas.
*views/***update.tpl**:

```
<!DOCTYPE html>
 <html>
 <head>
  <title>Beego-CRUD</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
 </head>

 <body>
  <div>
    <h3 class="beego">CRUD de Beego</h3>
    <h4>Actualizar Usuario</h4>
    <form role="form" id="usuario" method="POST">
     <div>
       <input id="nombre" name="nombre" value="{{.Nom}}" type="text" />
       <label for="nombre">Nombre</label>
     </div>
     <br/>
     <div>
       <input id="app" name="app" value="{{.App}}" type="text" />
       <label for="app">Apellido Paterno</label>
     </div>
     <br/>
```

```
      <div>
        <input id="apm" name="apm" value="{{.Apm}}" type="text" />
        <label for="apm">Apellido Materno</label>
      </div>
      <br/>
      <div>
        <input id="correo"  name="correo" pattern="" value="{{.Correo}}" type="text" />
        <label for="correo">Correo Electrónico</label>
      </div>
      <br/>
      <div>
        <input id="edad" pattern="[0-9]*?" name="edad" value="{{.Edad}}" type="text" />
        <label for="edad">Edad</label>
      </div>
      <br/>
      <input type="submit" value="Aceptar" />
    </form>
  </div>
 </body>
</html>
```

No pasemos por alto que el patrón del correo debe ser llenado con una expresión regular. Por limitantes en el ancho de página, no pudimos agregar nuestro original al HTML que presentamos. Pero he aquí el que nosotros utilizamos:

```
[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$
```

5.5. Eliminación

Para terminar con la implementación de nuestro CRUD en Beego, implementaremos la eliminación de usuarios. De manera similar a la operación de actualización anterior, estará disponible la función de eliminación en el listado de lectura de usuarios de la página principal. A un lado de la edición de un usuario en la tabla, tendremos la opción de eliminarlos.

5.5.1. Controllers

Dentro del archivo del controlador principal *controllers/***default.go** crearemos una nueva función de eliminación para la estructura que ahí reside. Le llamaremos de forma convencional con el nombre **Delete()**. Aquí su implementación:

```
// Método que permite eliminar un usuario
func (this *MainController) Delete() {
  this.TplName = "delete.tpl"
  // Se obtiene el parámetro id del url
```

```
    id, err := this.GetInt("id")
    if err != nil {
      msg := fmt.Sprintf("Parámetro inválido: ", err)
      beego.Debug(msg)
    }

    o := orm.NewOrm()
    o.Using("default")

    // Se obtiene el usuario con el id del parámetro
    if existe := o.QueryTable("usuario").Filter("Id", id).Exist(); existe {
      // Se elimina el usuario de la base de datos
      if num, err := o.Delete(&models.Usuario{Id: id}); err == nil {
        beego.Info("Usuario eliminado. ", num)
      } else {
        beego.Error("Usuario no pudo ser eliminado: ", err)
      }
    } else {
      beego.Info("El usuario no existe.")
    }

    var usuarios []*models.Usuario
    num, err := o.QueryTable("usuario").All(&usuarios)

    if err != orm.ErrNoRows && num > 0 {
      this.Data["records"] = usuarios
    }
    this.Redirect("/",302)
  }
```

Una vez hecho esto podemos configurar su uso a partir de una ruta de la aplicación.

5.5.2. Routers

Con la finalidad de que la función sea usable, es preciso definir una ruta o utilizar alguna que ya hayamos definido. Nosotros agregaremos la siguiente línea al archivo *routers/***router.go**, para que la operación de eliminación a utilizar por cualquier usuario de la aplicación tenga efecto con la función correspondiente.

```
beego.Router("/delete", &controllers.MainController{}, "*:Delete")
```

Luego procedemos a hacer accesible dicha ruta para el usuario desde la interfaz visual de la aplicación.

5.5.3. Views

Simulando un menú de opciones para cada entrada en la tabla de lectura de la base de datos de la página principal, luego de tener la posibilidad de editar cualquier usuario, agregaremos la alternativa de eliminarlo permanentemente. Si hicimos las modificaciones en **index.tpl** como se indica en el apartado de vistas de la operación de actualización, deberíamos contar con algo similar a esto dentro de las definiciones de la tabla de registros:

```
<tr>
  <td>{{$record.Nombre}} {{$record.App}} {{$record.Apm}}</td>
  <td>{{$record.Correo}}</td>
  <td>{{$record.Edad}}</td>
  <td><a href="/update?id={{$record.Id}}"><i>editar</i></a></td>
</tr>
```

Sección sobre la que efectuaremos un agregado de línea como sigue:

```
<tr>
 <td>{{$record.Nombre}} {{$record.App}} {{$record.Apm}}</td>
 <td>{{$record.Correo}}</td>
 <td>{{$record.Edad}}</td>
 <td><a href="/update?id={{$record.Id}}"><i>editar</i></a></td>
 <td><a href="/delete?id={{$record.Id}}"><i>borrar</i></a></td>
</tr>
```

Por si el redireccionado de la página no surte efecto, es necesario que exista el template que asignamos a la función de eliminación. Podemos elaborar una plantilla sencilla que notifique del éxito en la eliminación del usuario.[16]

```
<!DOCTYPE html>
 <html>
   <head>
    <title>Beego-CRUD - Crear</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
   </head>
   <body>
     <h2>Usuario eliminado</h2>
     <a href="/">Regresar</a>
   </body>
 </html>
```

Con esta operación finaliza la implementación del CRUD en Beego. A continuación los resultados esperados.

[16] Cabe mencionar que bajo condiciones normales no debería ser una página resultante en la navegación a no ser que accedamos directamente a esta ruta.

5.6. Resultados

Nota: agregar screenshots de la aplicación web

6. Referencias

Nota: agregar formato!

http://beego.me/docs/intro
https://golang.org/
https://www.mysql.com/
http://wiki.freepascal.org/Using_INI_Files
http://dspace.howest.be/bitstream/10046/1323/1/anton-van-eechaute-2016-03-25.pdf