REST-API with Golang, Mux & MySQL

**Creating our project:**

We are going to start by creating our project which is only going to contain one file.

mkdir rest-api-mysql && cd rest-api && touch main.go

This will create a new directory called “rest-api-mysql” and a file called “main.go” in it.

**We also need to install some packages:**

go get -u github.com/gorilla/mux

go get -u github.com/go-sql-driver/mysql

We will use a MySQL driver in order to connect to our database. The reason we do this is because the driver handles a lot of the stuff necessary to communicate with our database.

**Creating the database object**

First we will add our database packages and create our database object. This does not open any connection to the database, instead it prepares the object for future use. We also create two global variables, “db” and “err”.

package main

import (

"database/sql"

\_"github.com/go-sql-driver/mysql"

)

var db \*sql.DB

var err error

func main() {

db, err = sql.Open("mysql", "<user>:<password>@tcp(127.0.0.1:3306)/<dbname>")

if err != nil {

panic(err.Error())

}

defer db.Close()

Just replace <user> with you database user, <password> with the password and <dbname> with your database name. If your database doesn’t have a password replace the “sql.Open()” line with this:

**db ,err = sql.Open("mysql", "root:@/<database-name>")**

**Initialising the router**

Now we have database object and are ready to create our endpoints. We import Mux, initialise the router and run our server on port 8000.

package main

import (

"database/sql"

\_"github.com/go-sql-driver/mysql"

"github.com/gorilla/mux"

)

var db \*sql.DB

var err error

func main() {

db, err = sql.Open("mysql", "<user>:<password>@tcp(127.0.0.1:3306)/<dbname>")

if err != nil {

panic(err.Error())

}

defer db.Close()

router := mux.NewRouter()

http.ListenAndServe(":8000", router)

}

**Creating our endpoints**

Now we are going to establish the endpoints of our API, the way we will set this up is to create all of our endpoints in the main function, every endpoint needs a function to handle the request and we will define those below the main function.

Now I will create our endpoints and your file should look something like this:

package main

import (

"database/sql"

\_"github.com/go-sql-driver/mysql"

"github.com/gorilla/mux"

"net/http"

)

var db \*sql.DB

var err error

func main() {

db, err = sql.Open("mysql", "<user>:<password>@tcp(127.0.0.1:3306)/<dbname>")

if err != nil {

panic(err.Error())

}

defer db.Close()

router := mux.NewRouter()

router.HandleFunc("/posts", getPosts).Methods("GET")

router.HandleFunc("/posts", createPost).Methods("POST")

router.HandleFunc("/posts/{id}", getPost).Methods("GET")

router.HandleFunc("/posts/{id}", updatePost).Methods("PUT")

router.HandleFunc("/posts/{id}", deletePost).Methods("DELETE")

http.ListenAndServe(":8000", router)

}

**Defining our models (structs)**

A struct in Golang is very similar to a ES6 or Java class. We will use a struct to represent a post.

package main

import (

"database/sql"

\_"github.com/go-sql-driver/mysql"

"github.com/gorilla/mux"

"net/http"

)

type Post struct {

ID string `json:"id"`

Title string `json:"title"`

}

var db \*sql.DB

var err error

func main() {

db, err = sql.Open("mysql", "<user>:<password>@tcp(127.0.0.1:3306)/<dbname>")

if err != nil {

panic(err.Error())

}

defer db.Close()

router := mux.NewRouter()

router.HandleFunc("/posts", getPosts).Methods("GET")

router.HandleFunc("/posts", createPost).Methods("POST")

router.HandleFunc("/posts/{id}", getPost).Methods("GET")

router.HandleFunc("/posts/{id}", updatePost).Methods("PUT")

router.HandleFunc("/posts/{id}", deletePost).Methods("DELETE")

http.ListenAndServe(":8000", router)

}

**Route handlers**

Now we just need to define the functions that will handle the requests. You also need to import the packages “encoding/json”, “fmt” and “io/ioutil”.

import (

"encoding/json"

"fmt"

"io/ioutil"

)

The syntax for creating these functions looks like this:

func <your-function-name>(w http.ResponseWriter, r \*http.Request) {

**getPosts()**

func getPosts(w http.ResponseWriter, r \*http.Request) {

w.Header().Set("Content-Type", "application/json")

var posts []Post

result, err := db.Query("SELECT id, title from posts")

if err != nil {

panic(err.Error())

}

defer result.Close()

for result.Next() {

var post Post

err := result.Scan(&post.ID, &post.Title)

if err != nil {

panic(err.Error())

}

posts = append(posts, post)

}

json.NewEncoder(w).Encode(posts)

}

Here I have defined the function that gets all posts from our database, we’re just setting the header “Content-Type” to “application/json”. Then we define an array of posts called “posts”.

When we query our database we use “db.Query()”, this should not be used with actions that doesn’t return any rows. Exemples of such actions is INSERT, UPDATE AND DELETE. Instead we use “db.Prepare()” and “.Exec()”, but more on that later.

Finally, we encode our posts to JSON and send send them away.

**createPost()**

func createPost(w http.ResponseWriter, r \*http.Request) {

stmt, err := db.Prepare("INSERT INTO posts(title) VALUES(?)")

if err != nil {

panic(err.Error())

}

body, err := ioutil.ReadAll(r.Body)

if err != nil {

panic(err.Error())

}

keyVal := make(map[string]string)

json.Unmarshal(body, &keyVal)

title := keyVal["title"]

\_, err = stmt.Exec(title)

if err != nil {

panic(err.Error())

}

fmt.Fprintf(w, "New post was created")

}

Here we will not use “db.Query()” as we are inserting data which does not return any rows, this is very important. Instead we prepare a statement with “db.Prepare()” which we will then execute with “stmt.Exec()”. In this case we prepare a statement that will create new post. The syntax for placeholder parameters in prepared statements is database-specific, using MySQL it’s a question mark (?).

**getPost()**

func getPost(w http.ResponseWriter, r \*http.Request) {

w.Header().Set("Content-Type", "application/json")

params := mux.Vars(r)

result, err := db.Query("SELECT id, title FROM posts WHERE id = ?", params["id"])

if err != nil {

panic(err.Error())

}

defer result.Close()

var post Post

for result.Next() {

err := result.Scan(&post.ID, &post.Title)

if err != nil {

panic(err.Error())

}

}

json.NewEncoder(w).Encode(post)

}

Now you should be able to understand at least some of what I’m doing here. First we set the content type to “application/json” and as we need an ID from our url we define create a variable called “params” set to “mux.Vars(r)”.

We then create a query that gets the post with the id from our url, of course we check for errors as well as calling “defer result.Close()”.

Then we create a new Post object, iterate through the result (even if we know that there is only one post with that specific id) and scan the values into our object.

At last, we encode our post to JSON and send it away.

**updatePost()**

func updatePost(w http.ResponseWriter, r \*http.Request) {

params := mux.Vars(r)

stmt, err := db.Prepare("UPDATE posts SET title = ? WHERE id = ?")

if err != nil {

panic(err.Error())

}

body, err := ioutil.ReadAll(r.Body)

if err != nil {

panic(err.Error())

}

keyVal := make(map[string]string)

json.Unmarshal(body, &keyVal)

newTitle := keyVal["title"]

\_, err = stmt.Exec(newTitle, params["id"])

if err != nil {

panic(err.Error())

}

fmt.Fprintf(w, "Post with ID = %s was updated", params["id"])

}

For the update handler, we combine some of the things we already have done.

We get the url parameters, prepares a statement that will update the title of a post and check for errors.

Then we read the request body the same way as before because we need access to the data. We define a variable called “newTitle” and set it to the “title” value that was sent from the client.

Finally, we execute the prepared statement and pass in the new title as well as the id from the url. Then we just send a message which says that the post has been updated.

**deletePost()**

func deletePost(w http.ResponseWriter, r \*http.Request) {

params := mux.Vars(r)

stmt, err := db.Prepare("DELETE FROM posts WHERE id = ?")

if err != nil {

panic(err.Error())

}

\_, err = stmt.Exec(params["id"])

if err != nil {

panic(err.Error())

}

fmt.Fprintf(w, "Post with ID = %s was deleted", params["id"])

}

Now we only have the delete handler left, this should be pretty self explanatory. We get the url parameters, prepares a query that will delete a post and then executes this statement with the id from the url.

We then send a message that says that the post has been deleted.

**Testing our API**

I’m going to use Postman to test the API but curl is also a good option

Make sure you build and restart/run the server:

go build && ./rest-api-mysql