How to create a basic RESTful API in Go

Etienne Rouzeaud Follow
Mar 12, 2015 · 5 min read

. . .

Like any languages, it is possible to quickly code a basic RESTful API with Golang. In this example, datas will be accessible via standards HTTP methods (GET, POST, PUT & DELETE) in JSON format. Data rows will be storaged in a SQLite file.

Schema

Routes

- POST : http://127.0.0.1:8080/api/v1/users
- GET : http://127.0.0.1:8080/api/v1/users
- GET : http://127.0.0.1:8080/api/users/1
- PUT : http://127.0.0.1:8080/api/users/1
- DELETE : http://127.0.0.1:8080/api/users/1

Table "users"

- id — integer & auto-increment
- firstname — varchar(255)
- lastname — varchar(255)

Gin for routes

We’re gonna use Gin who is a micro framework routing. It’s friendly user and fast.

```
go get github.com/gin-gonic/gin
```

Ok, let’s start coding ! Firstly in a new “main.go” file, we call our libraries.

```
package main

import (
    "strconv"

     "github.com/gin-gonic/gin"
)
```

Secondly, we declare the structure “User” :

```
type Users struct {
    Id        int    `gorm:"AUTO_INCREMENT" form:"id" json:"id"`
    Firstname string `gorm:"not null" form:"firstname"
json:"firstname"`
    Lastname  string `gorm:"not null" form:"lastname"
json:"lastname"`
}
```

The “gorm” param will be used later, with the database connection…

Thirdly, in the *main()* function, we regoup our routes in a single group :

```
func main() {
    r := gin.Default()

    v1 := r.Group("api/v1")
    {
        v1.POST("/users", PostUser)
        v1.GET("/users", GetUsers)
        v1.GET("/users/:id", GetUser)
        v1.PUT("/users/:id", UpdateUser)
        v1.DELETE("/users/:id", DeleteUser)
    }

    r.Run(":8080")
}
```

Then we declare the five functions calling in the routes :

```
func PostUser(c *gin.Context) {
    // The futur code…
}

func GetUsers(c *gin.Context) {
    var users = []Users{
        Users{Id: 1, Firstname: "Oliver", Lastname: "Queen"},
        Users{Id: 2, Firstname: "Malcom", Lastname: "Merlyn"},
    }

    c.JSON(200, users)

    // curl -i http://localhost:8080/api/v1/users
}

func GetUser(c *gin.Context) {
    id := c.Params.ByName("id")
    user_id, _ := strconv.ParseInt(id, 0, 64)

    if user_id == 1 {
        content := gin.H{"id": user_id, "firstname": "Oliver",
"lastname": "Queen"}
        c.JSON(200, content)
    } else if user_id == 2 {
        content := gin.H{"id": user_id, "firstname": "Malcom",
"lastname": "Merlyn"}
        c.JSON(200, content)
    } else {
        content := gin.H{"error": "user with id#" + id + " not
found"}
        c.JSON(404, content)
    }

    // curl -i http://localhost:8080/api/v1/users/1
}

func UpdateUser(c *gin.Context) {
    // The futur code…
}

func DeleteUser(c *gin.Context) {
    // The futur code…
}
```

At this stage, let’s start our API server with the classic command :

```
go run main.go
```

As you can see, for reading users, our URL’s (GET) are working good with fake datas.

For all users :

```
[{"id":1,"firstname":"Oliver","lastname":"Queen"},
{"id":2,"firstname":"Malcom","lastname":"Merlyn"}]
```

For a user :

```
{"firstname":"Oliver","id":1,"lastname":"Queen"}
```

SQLite with the ORM Gorm

```
go get github.com/jinzhu/gorm
```

With Gorm, we're gonna use a **SQLite** database. You can also use **MySQL** (and **MariaDB**), **Postgres** et **FoundationDB** database instead.

```
go get github.com/mattn/go-sqlite3
```

Your compiler will be angry if you forgot to import the new librairies (and we don't need the "strconv" library).

```
import (
    "github.com/gin-gonic/gin"
    "github.com/jinzhu/gorm"
    _ "github.com/mattn/go-sqlite3"
)
```

No need to create the database file and the table “users” your-self, Gorm does the job for you. ☺

```
func InitDb() *gorm.DB {
    // Openning file
    db, err := gorm.Open("sqlite3", "./data.db")
    db.LogMode(true)


    // Error
    if err != nil {
        panic(err)
    }


    // Creating the table
    if !db.HasTable(&Users{}) {
        db.CreateTable(&Users{})
        db.Set("gorm:table_options",
"ENGINE=InnoDB").CreateTable(&Users{})
    }

    return db
}
```

As agreed, at the next restart of your server, the table “users” will be created with his fields in the futur file “data.db”.

CRUD

Create a user

```
func PostUser(c *gin.Context) {
    db := InitDb()
    defer db.Close()

    var user Users
    c.Bind(&user)

    if user.Firstname != "" && user.Lastname != "" {
        // INSERT INTO "users" (name) VALUES (user.Name);
        db.Create(&user)
        // Display error
        c.JSON(201, gin.H{"success": user})
    } else {
        // Display error
        c.JSON(422, gin.H{"error": "Fields are empty"})
    }

    // curl -i -X POST -H "Content-Type: application/json" -d "{
\"firstname\": \"Thea\", \"lastname\": \"Queen\" }"
http://localhost:8080/api/v1/users
}
```

After the CURL command, the file “data.db” is created :).

Result :

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8

{"success":{"id":1,"firstname":"Thea","lastname":"Queen"}}
```

Read all users

```
func GetUsers(c *gin.Context) {
    // Connection to the database
    db := InitDb()
    // Close connection database
    defer db.Close()

    var users []Users
    // SELECT * FROM users
    db.Find(&users)

    // Display JSON result
    c.JSON(200, users)

    // curl -i http://localhost:8080/api/v1/users
}
```

Result :

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

[{"id":1,"firstname":"Thea","lastname":"Queen"}]
```

Read a user

```
func GetUser(c *gin.Context) {
    // Connection to the database
    db := InitDb()
    // Close connection database
    defer db.Close()

    id := c.Params.ByName("id")
    var user Users
    // SELECT * FROM users WHERE id = 1;
    db.First(&user, id)

    if user.Id != 0 {
        // Display JSON result
        c.JSON(200, user)
    } else {
        // Display JSON error
        c.JSON(404, gin.H{"error": "User not found"})
    }

    // curl -i http://localhost:8080/api/v1/users/1
}
```

Result :

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":1,"firstname":"Thea","lastname":"Queen"}
```

Update a user

```
func UpdateUser(c *gin.Context) {
    // Connection to the database
    db := InitDb()
    // Close connection database
    defer db.Close()

    // Get id user
    id := c.Params.ByName("id")
    var user Users
    // SELECT * FROM users WHERE id = 1;
    db.First(&user, id)

    if user.Firstname != "" && user.Lastname != "" {

        if user.Id != 0 {
            var newUser Users
            c.Bind(&newUser)

            result := Users{
                Id:        user.Id,
                Firstname: newUser.Firstname,
                Lastname:  newUser.Lastname,
            }

            // UPDATE users SET firstname='newUser.Firstname',
lastname='newUser.Lastname' WHERE id = user.Id;
            db.Save(&result)
            // Display modified data in JSON message "success"
            c.JSON(200, gin.H{"success": result})
        } else {
            // Display JSON error
            c.JSON(404, gin.H{"error": "User not found"})
        }

    } else {
        // Display JSON error
        c.JSON(422, gin.H{"error": "Fields are empty"})
    }

    // curl -i -X PUT -H "Content-Type: application/json" -d "{
\"firstname\": \"Thea\", \"lastname\": \"Merlyn\" }"
http://localhost:8080/api/v1/users/1
}
```

Result :

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8


{"success":{"id":1,"firstname":"Thea","lastname":"Merlyn"}}
```

Delete a user

```
func DeleteUser(c *gin.Context) {
    // Connection to the database
    db := InitDb()
    // Close connection database
    defer db.Close()

    // Get id user
    id := c.Params.ByName("id")
    var user Users
    // SELECT * FROM users WHERE id = 1;
    db.First(&user, id)

    if user.Id != 0 {
        // DELETE FROM users WHERE id = user.Id
        db.Delete(&user)
        // Display JSON result
        c.JSON(200, gin.H{"success": "User #" + id + " deleted"})
    } else {
        // Display JSON error
        c.JSON(404, gin.H{"error": "User not found"})
    }

    // curl -i -X DELETE http://localhost:8080/api/v1/users/1
}
```

Result :

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"success":"User #1 deleted"}
```

CORS

You can use CORS directly in the concern route.

```
c.Writer.Header().Add("Access-Control-Allow-Origin", "*")
c.Next()
```

Or globally with a custom middleware in a function with "gin.HandlerFunc":

```
func Cors() gin.HandlerFunc {
    return func(c *gin.Context) {
        c.Writer.Header().Add("Access-Control-Allow-Origin", "*")
        c.Next()
    }
}
```

And before routes call :

```
r.Use(Cors())
```

If you don’t activated CORS, you will get message error like this with Chrome :

```
XMLHttpRequest cannot load http://localhost:8080/api/v1/users. No
‘Access-Control-Allow-Origin’ header is present on the requested
resource. Origin ‘http://localhost:8081' is therefore not allowed
access.
```

OPTIONS

If you are using “XMLHttpRequest” or “Fetch” with Javacript and CORS, you will need to use “OPTIONS” for requests POST, PUT, DELETE.

Firstly, you must add 2 routes.

```
v1.OPTIONS("/users", OptionsUser)      // POST
v1.OPTIONS("/users/:id", OptionsUser)  // PUT, DELETE
```

And declare the “OptionsUser” function.

```
func OptionsUser(c *gin.Context) {
    c.Writer.Header().Set("Access-Control-Allow-Methods",
"DELETE,POST, PUT")
    c.Writer.Header().Set("Access-Control-Allow-Headers", "Content-
Type")
    c.Next()
}
```

If you don’t using this method you will get a message error like this with Chrome :

```
XMLHttpRequest cannot load http://localhost:8080/api/v1/users/1.
Response to preflight request doesn’t pass access control check: No
‘Access-Control-Allow-Origin’ header is present on the requested
resource. Origin ‘http://localhost:8081' is therefore not allowed
access. The response had HTTP status code 404.
```

In fact, the navigator does not find the OPTIONS http://localhost:8080/api/v1/users/1 URL

Conclusion

We have an example of a simple and functionnal basic RESTful API. As you can see, structures are importants in Golang, especialy when you manipulate some Json datas.

Have a good fun with Go ;)

More informations about :

- Gin : https://github.com/gin-gonic/gin
- Gorm : http://jinzhu.me/gorm
- Go SQLITE 3 driver :https://github.com/mattn/go-sqlite3
- Go Sublime (a Sublime Text plugin) : https://github.com/DisposaBoy/GoSublime
- Code available : https://gist.github.com/EtienneR/ed522e3d31bc69a9dec3335e639fcf60
- Old Code still available (with Gorp and MySQL) : https://gist.github.com/EtienneR/5eb48ae7d849cec6f55a

Golang API Sqlite