REST-API with Golang and Mux.

Hugo Johnsson Mar 10, 2019 · 7 min read

In this article we are building a complete REST-API in Golang along with Mux which is a third party router.

Already know this? Learn how to integrate with a database in this article:

REST-API with Golang, Mux & MySQL

In this article we are building a complete REST-API in Golang along with Mux and MySQL.

medium.com

Introduction

Today we are building a simple REST-API with Golang a router package called Mux. We are not going to integrate with a database as this article is meant to teach you the basics but I may write another article soon, with database integration.

Creating our project

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

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

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

We also need to install Mux:

```
go get -u github.com/gorilla/mux
```

Our first lines of code

Before we write any code related to the REST-API we need to write som mandatory code to make the program run.

```
package main

import (
  "fmt"
)

func main() {
  fmt.Println("It works!")
}
```

Basically to import packages you need to write the “package main” and then you import statement, you also need a main function to make the it work.

To build and run the program:

```
go build && ./rest-api
```

If it prints “It works!” without any errors you are good to go!

Initialising the router

We are going to remove the line that prints out “It works!” as well as the “fmt” package, instead we will add the Mux package and initialise the router.

```
package main

import (
  "github.com/gorilla/mux"
)

func main() {
  router := mux.NewRouter()
}
```

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.

The syntax for creating an endpoint looks like this:

```
router.HandleFunc("/<your-url>", <function-name>).Methods("
<method>")
```

The" Methods" function is required because we need to tell it what type of request to accept.

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

```
package main

import (
  "github.com/gorilla/mux"
  "net/http"
)

func main() {
  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")

}
```

What I've done is to create five different routes, I expect you to know at least the basics of REST-APIS so you should understand what the different enpoints does. I also added another package ("net/http").

We will also add this line below to actually run the server:

```
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, I encourage you to change up some things to make it your own, this way you will learn more. Even changing “posts” to “users” helps much in my experience.

```
package main

import (
  "github.com/gorilla/mux"
  "net/http"
)

type Post struct {
  ID string `json:"id"`
  Title string `json:"title"`
  Body string `json:"body"`
}

var posts []Post

func main() {
  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)
}
```

You can see I defined a “Post” struct, below it I also initialised what is called a slice. A slice is similar to an array, the difference is that when you want to use arrays in Golang you need to define the length. This is why we use a slice, we also tell it that it will contain posts.

If you’re confused about the struct part, basically I’ve just defined the properties as well as the type they are, then right after I just tell it how the property will be represented in JSON.

Route handlers

Now we just need to define the functions that will handle the requests. You also need to import the package “encoding/json”.

```
import (
  "github.com/gorilla/mux"
  "net/http"
  "encoding/json"
)
```

The syntax for creating these functions looks like this:

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

}
```

You can read more about it here: https://github.com/gorilla/mux

getPosts()

```
func getPosts(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  json.NewEncoder(w).Encode(posts)
}
```

Here I have defined the function that gets all posts, we're just setting the header "Content-Type" to "application/json" . Then we use the encoding package to encode all the posts as well as returning it at the same line.

getPost()

```
func getPost(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  params := mux.Vars(r)
  for _, item := range posts {
    if item.ID == params["id"] {
      json.NewEncoder(w).Encode(item)
      break
    }
    return
  }
  json.NewEncoder(w).Encode(&Post{})
}
```

As we are getting a specific book we need the ID from the url, we do this with this line:

```
params := mux.Vars(r)
```

and then to extract params we do:

```
params["<param-name>"]
```

Then we loop though all of our books to find the one we want, when we do we are returning it the same way we did in the previous handler.

createPost()

Before we continue, we need to add two packages (“math/rand” and “strconv”).

```
import (
  "github.com/gorilla/mux"
  "net/http"
  "encoding/json"
  "math/rand"
  "strconv"
)
```

Now we can create the handler.

```
func createPost(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  var post Post
  _ = json.NewDecoder(r.Body).Decode(post)
  post.ID = strconv.Itoa(rand.Intn(1000000))
  posts = append(posts, post)
  json.NewEncoder(w).Encode(&post)
}
```

This is the handler that creates a new post, we start by creating a new instance of the struct Post. Then we decode the data that is sent with the request and insert it into the post. Next we create a random ID with the “math/rand” package as well as convert to a string. At last

we simply append the post to our posts array, this will save it to memory, and right after we return the new post.

updatePost()

```
func updatePost(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  params := mux.Vars(r)
  for index, item := range posts {
    if item.ID == params["id"] {
      posts = append(posts[:index], posts[index+1:]...)

      var post Post
      _ = json.NewDecoder(r.Body).Decode(post)
      post.ID = params["id"]
      posts = append(posts, post)
      json.NewEncoder(w).Encode(&post)

      return
    }
  }
  json.NewEncoder(w).Encode(posts)
}
```

For the update handler, we loop through our posts array to find the post to update. When it matches we remove that post from the array and create a new post with the same ID (using params[“id”]) with the new values from the request body.

deletePost()

```
func deletePost(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  params := mux.Vars(r)
  for _, item := range posts {
    if item.ID == params["id"] {
      posts= append(posts[:index], posts[index+1]...)
      break
    }
  }
  json.NewEncoder(w).Encode(books)
}
```

Here we are also looping through our posts array to find the post we want to delete. When we do, we delete the post with this line:

```
posts = append(posts[:index], posts[index+1]...)
```

and then we *break* out of our loop.

Adding mock data

Before we test we just need to add some mock data, this is done in the main function right after we initialise the router. Make sure it looks like this:

```
func main() {
  router := mux.NewRouter()

posts = append(posts, Post{ID: "1", Title: "My first post", Body:
"This is the content of my first post"})
posts = append(posts, Post{ID: "2", Title: "My second post", Body:
"This is the content of my second post"})

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)
}
```

Your file should now look something like this:

```
package main

import (
  "github.com/gorilla/mux"
  "net/http"
  "encoding/json"
  "math/rand"
  "strconv"
)

type Post struct {
  ID string `json:"id"`
  Title string `json:"title"`
  Body string `json:"body"`
}

var posts []Post
```

```
func getPosts(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  json.NewEncoder(w).Encode(posts)
}

func createPost(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  var post Post
  _ = json.NewDecoder(r.Body).Decode(&post)
  post.ID = strconv.Itoa(rand.Intn(1000000))
  posts = append(posts, post)
  json.NewEncoder(w).Encode(&post)
}

func getPost(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  params := mux.Vars(r)
  for _, item := range posts {
    if item.ID == params["id"] {
      json.NewEncoder(w).Encode(item)
      return
    }
  }
  json.NewEncoder(w).Encode(&Post{})
}

func updatePost(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  params := mux.Vars(r)
  for index, item := range posts {
    if item.ID == params["id"] {
      posts = append(posts[:index], posts[index+1:]...)

      var post Post
      _ = json.NewDecoder(r.Body).Decode(&post)
      post.ID = params["id"]
      posts = append(posts, post)
      json.NewEncoder(w).Encode(&post)

      return
    }
  }
  json.NewEncoder(w).Encode(posts)
}

func deletePost(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  params := mux.Vars(r)
  for index, item := range posts {
    if item.ID == params["id"] {
      posts = append(posts[:index], posts[index+1:]...)
      break
    }
  }
  json.NewEncoder(w).Encode(posts)
}
```

```
func main() {
  router := mux.NewRouter()

  posts = append(posts, Post{ID: "1", Title: "My first post",
Body:      "This is the content of my first post"})

  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)
}
```

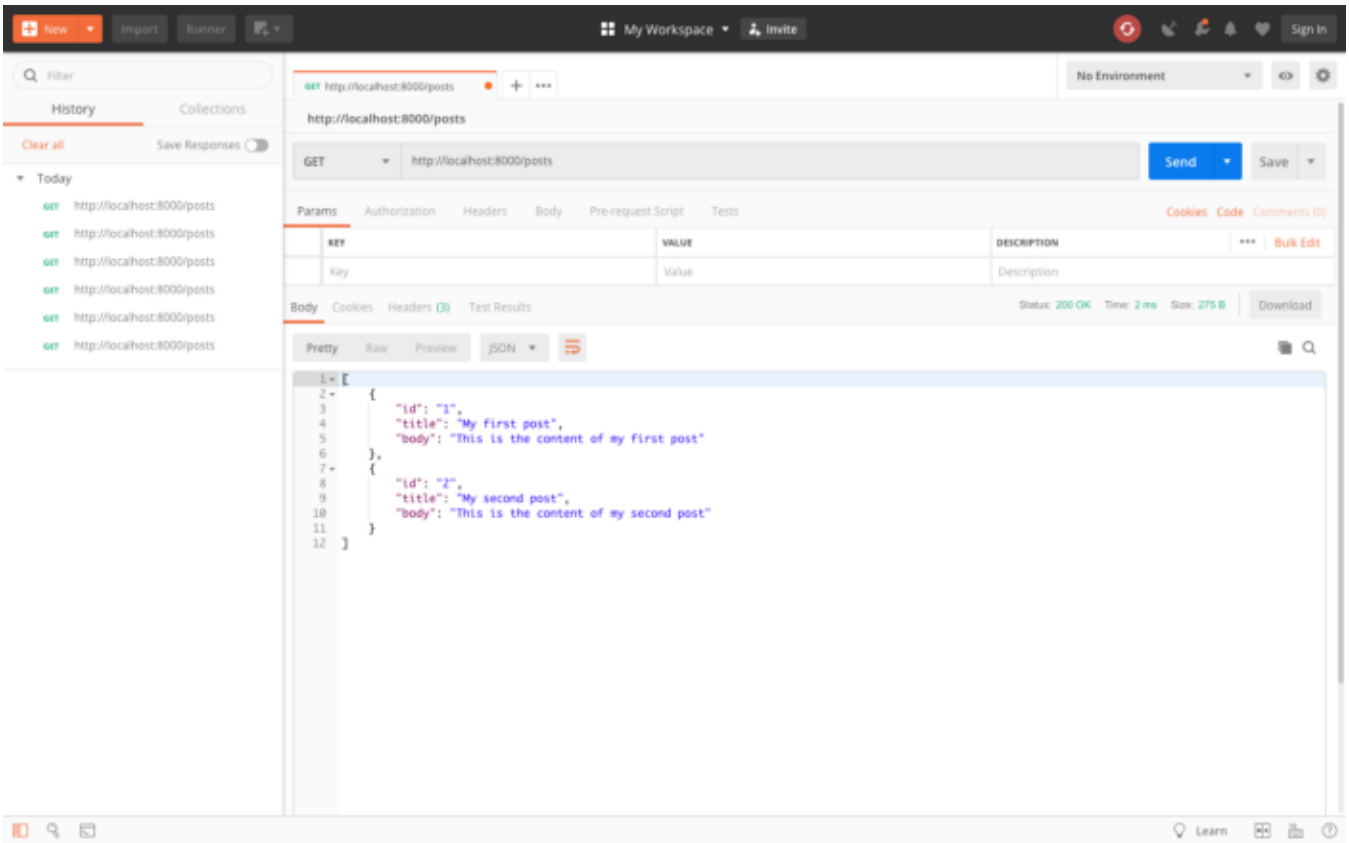
Testing our API

I'm going to use Postman to test the API but *curl* is also a good option, but I like Postman's graphical interface. You can get Postman here: https://www.getpostman.com

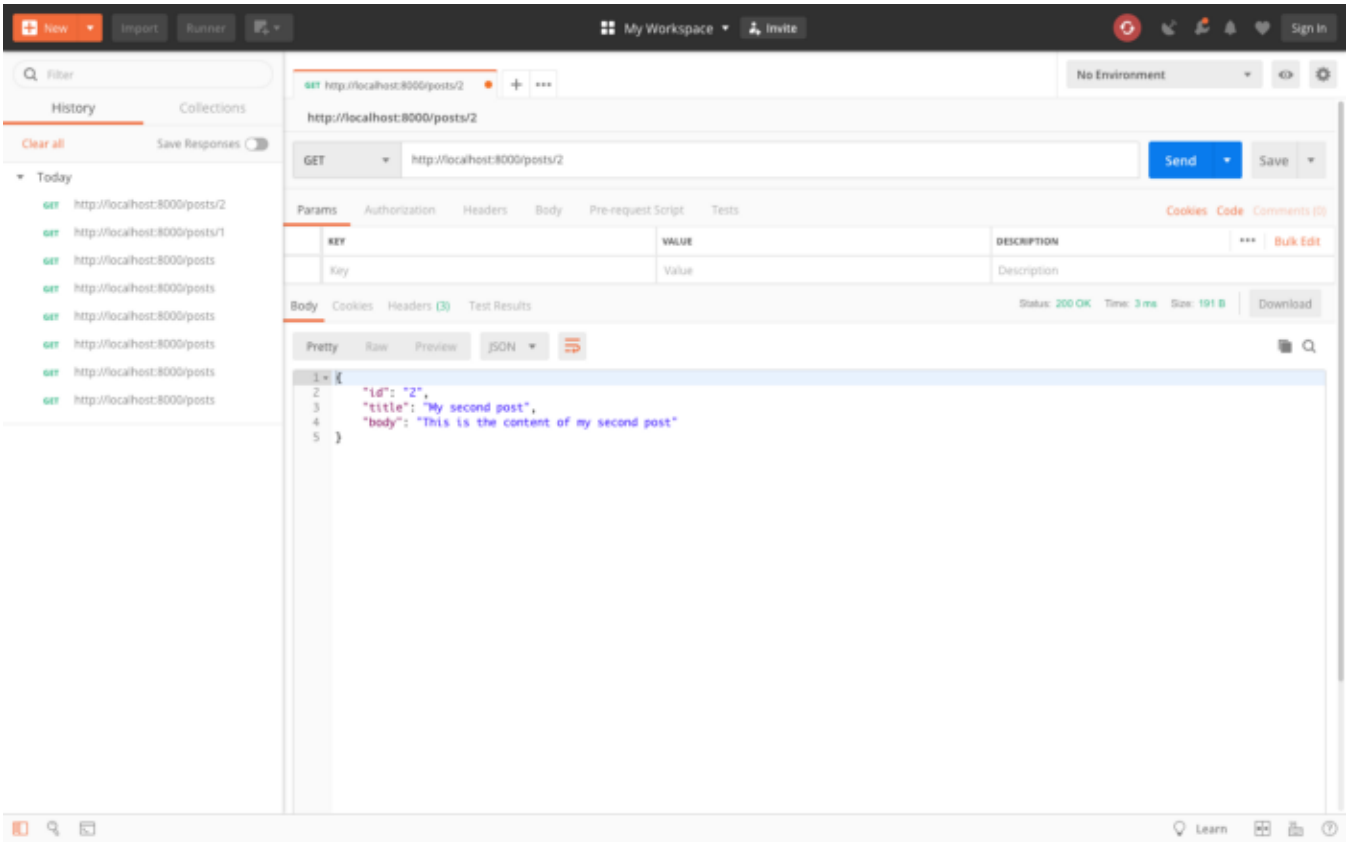
Make sure you build and restart/run the server:

```
go build && ./rest-api
```

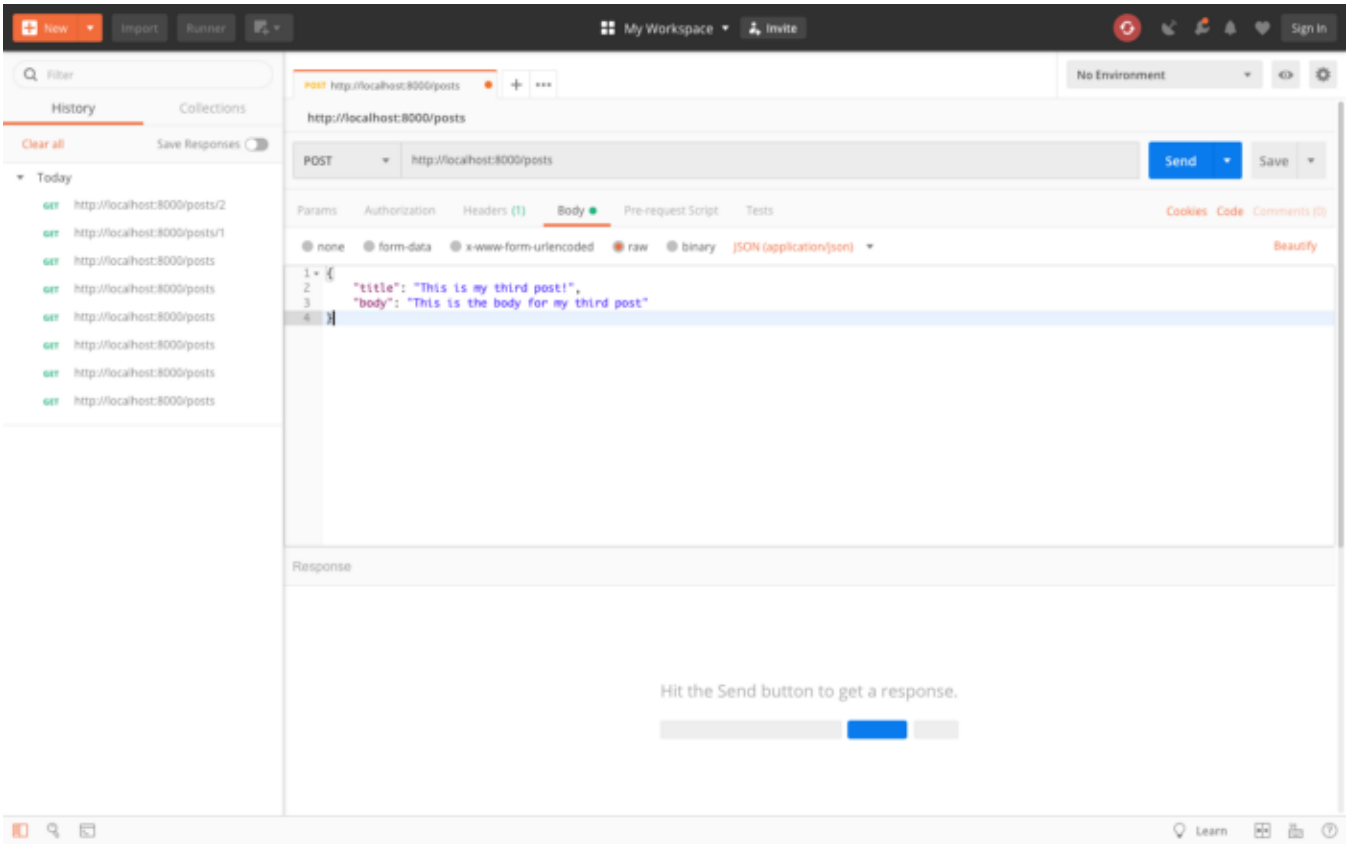
Getting all posts



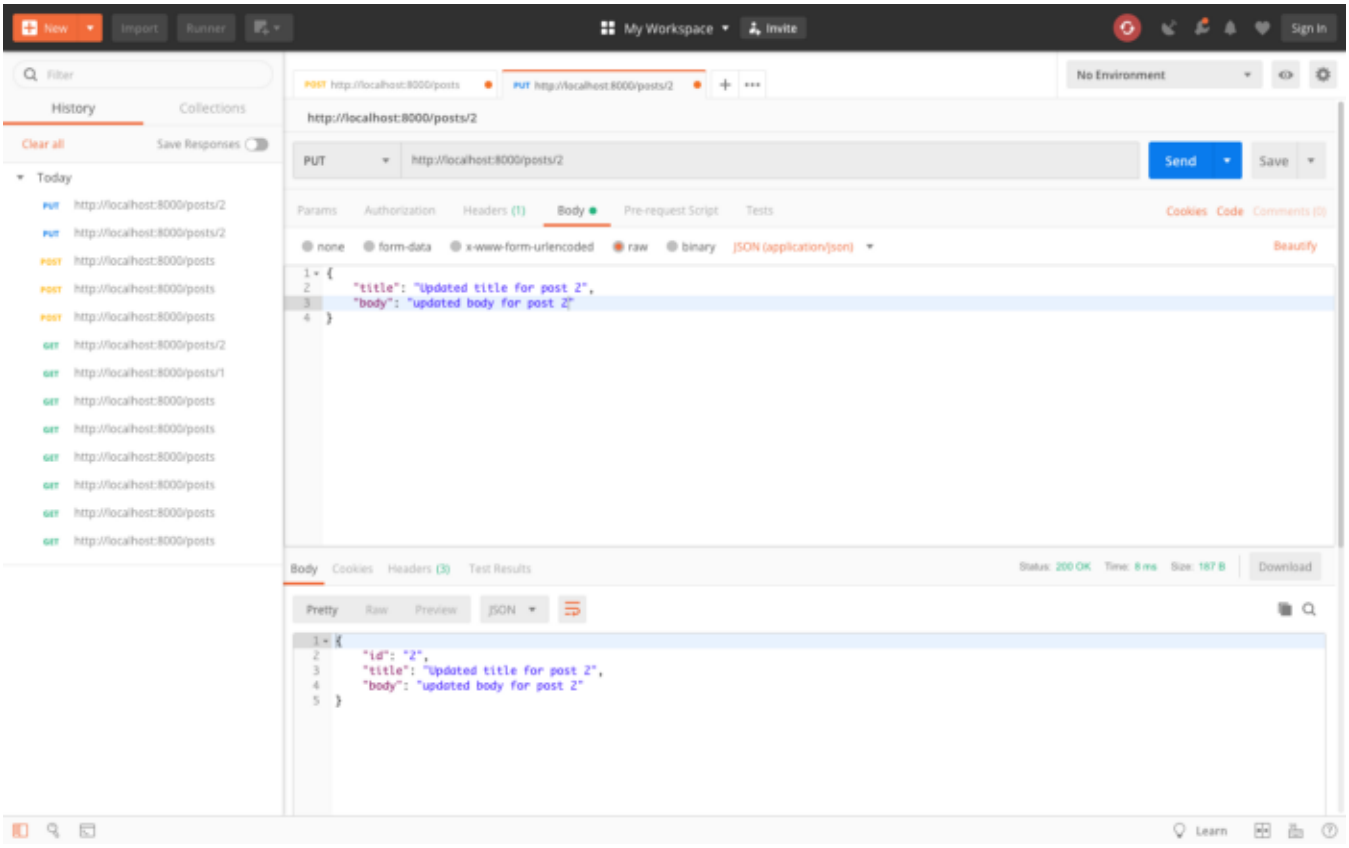
Getting a single post



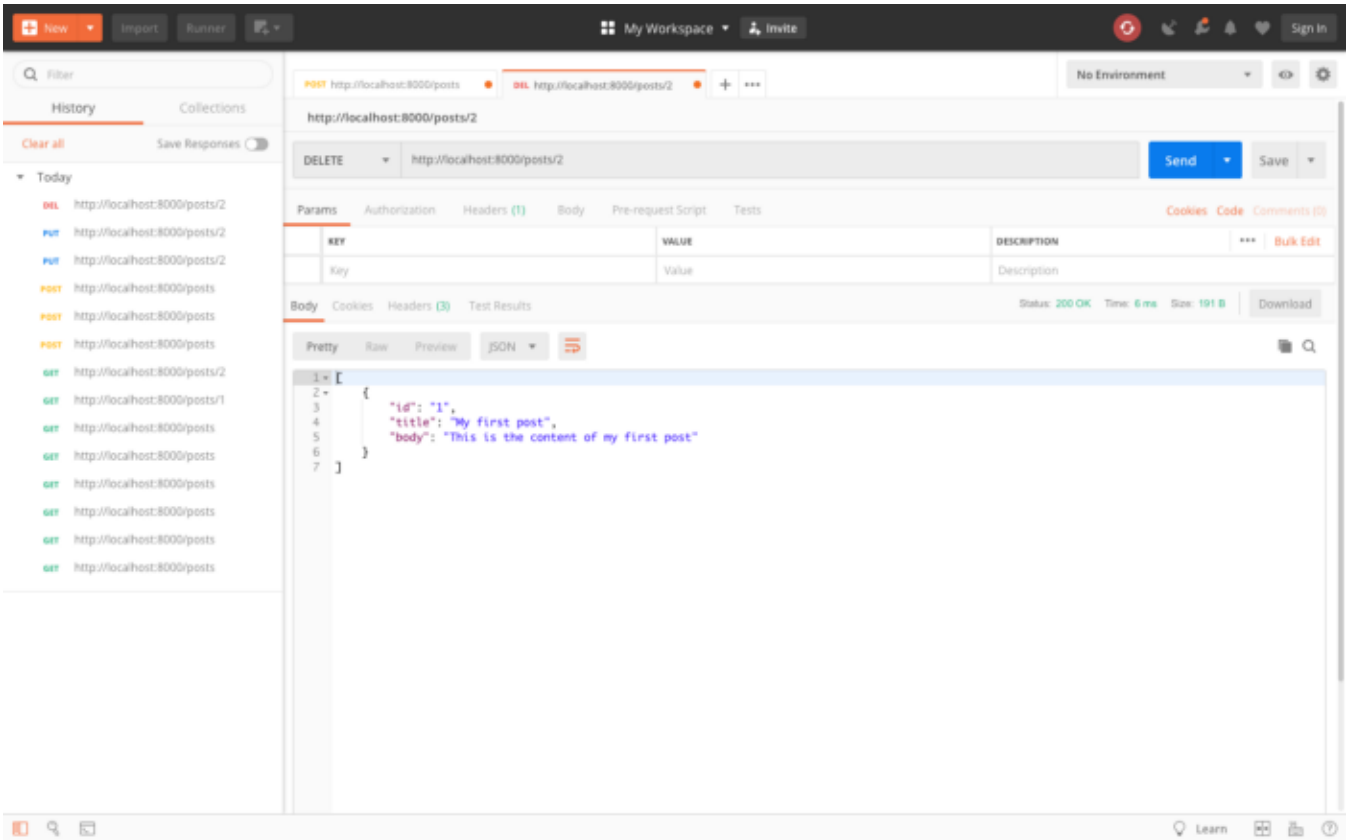
Creating a new post



Updating a post



Deleting a post



Thank you so much for reading!

I hope you found this article helpful, I really hope I helped at least one person get familiar with the basics of building REST-APIs with Golang and Mux. If you didn’t follow along and built it with me I encourage you to do so, you learn so much faster when you actually type the code:)