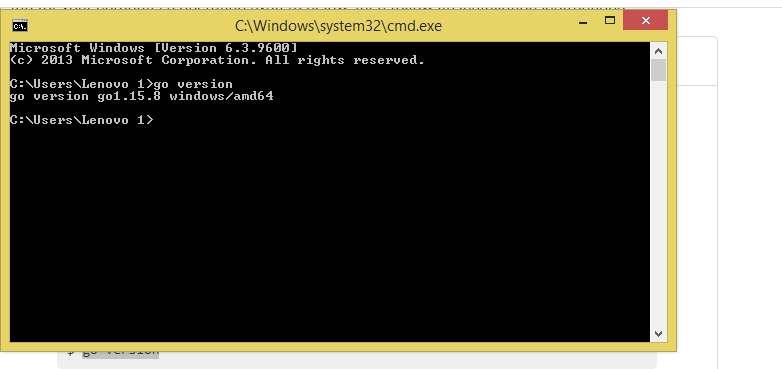
**Docs for reference which I have prepared for this assessment**

Install golang from

<https://golang.org/>

check the golang installed by below command:

go version



https://www.gorillatoolkit.org/--->is a powerful URL router and dispatcher.

Mux

C:\Program Files\Go\

<https://github.com/gorilla/mux> 🡪 repo for the mux

F:\golang >$ mkdir -p $GOPATH/src/github.com/shilpa/restapi

export GOPATH=$HOME/go

package main

import (

    "encoding/json"

    "log"

    "math/rand"

    "net/http"

    "strconv"

    "github.com/gorilla/mux"

)

// Book struct(Mode1)

type Book struct {

    ID      string  `json:"id"`

    Isbn    string  `json:" isbn"`

    Title   string  `json:"title"`

    Author  \*Author `json:"author"`

}

// Author

type Author struct {

    Firstname  string  `json:"firstname"`

    Lastname   string  `json:"lastname"`

}

//get all books

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

}

//get single book

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

}

//create book

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

}

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

}

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

}

func main() {

    //fmt.Println("HelloShilpa")

    r :=mux.NewRouter()

    // router handelers

    r.HandleFunc("/api/books", getBooks).Methods("GET")

    r.HandleFunc("/api/books/{id]", getBook).Methods("GET")

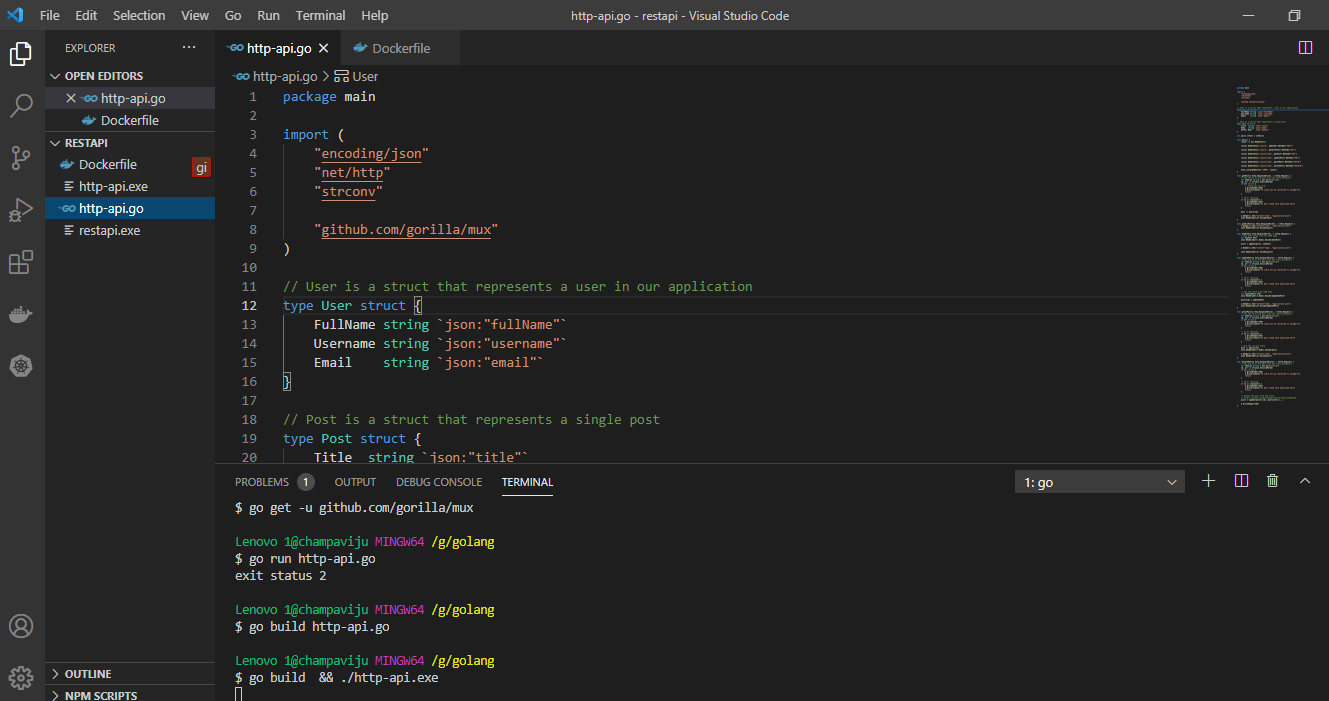
    r.HandleFunc("/api/books", getBooks).Methods("POST")

    r.HandleFunc("/api/books/{id}", getBooks).Methods("PUT")

    r.HandleFunc("/api/books/{id}", getBooks).Methods("DELETE")

    log.Fatal(http.ListenAndServe(":8000", r))

}



Docker build –it

"encoding/json"

This api will accept json in body of request and respond in json format

"github.com/gorilla/mux"

Install dependency for the mux(import gorilla mux library)

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

    // get Item value from the JSON body

    var newPost Post

    json.NewDecoder(r.Body).Decode(&newPost)

    posts = append(posts, newPost)

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

    json.NewEncoder(w).Encode(posts)

}

func main() {

    router := mux.NewRouter()

    router.HandleFunc("/api/encrypt", test).Methods("GET")

    http.ListenAndServe(":5000", router)

}

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

    w.Write([]byte("testing encrypt api"))

}

func main() {

    router := mux.NewRouter()

    router.HandleFunc("/api/encrypt", test).Methods("GET")

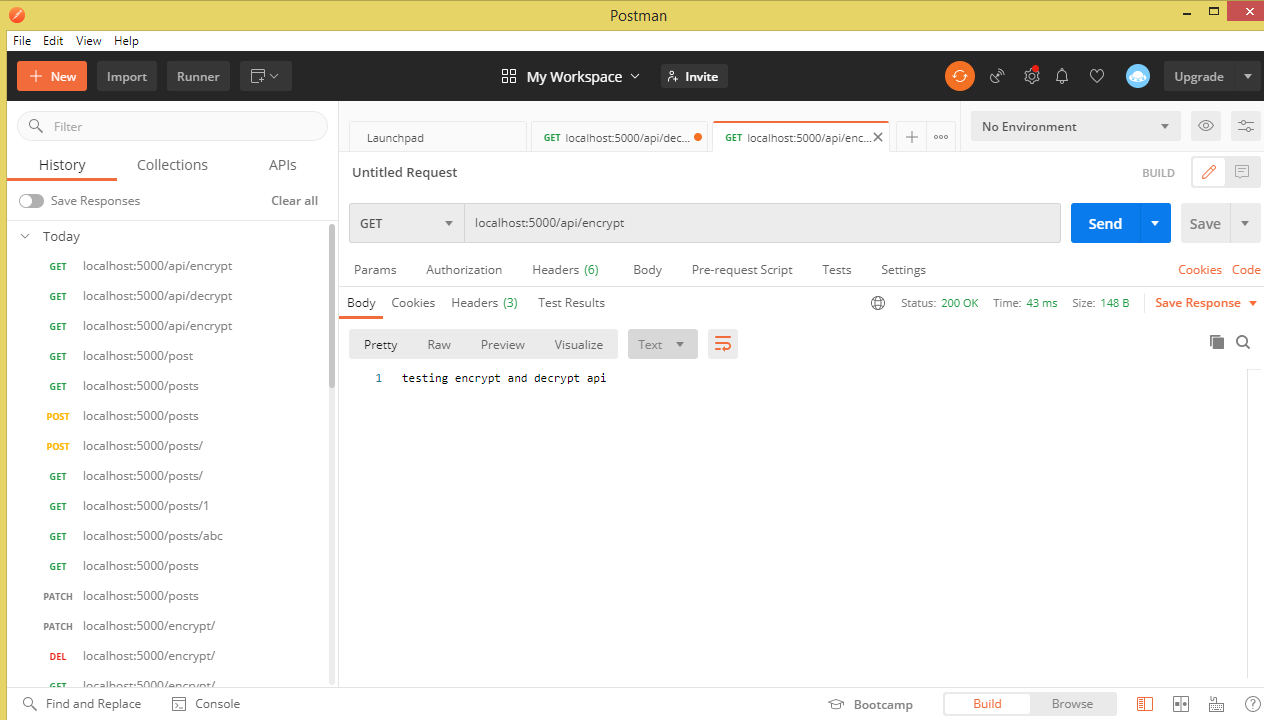
    http.ListenAndServe(":5000", router)

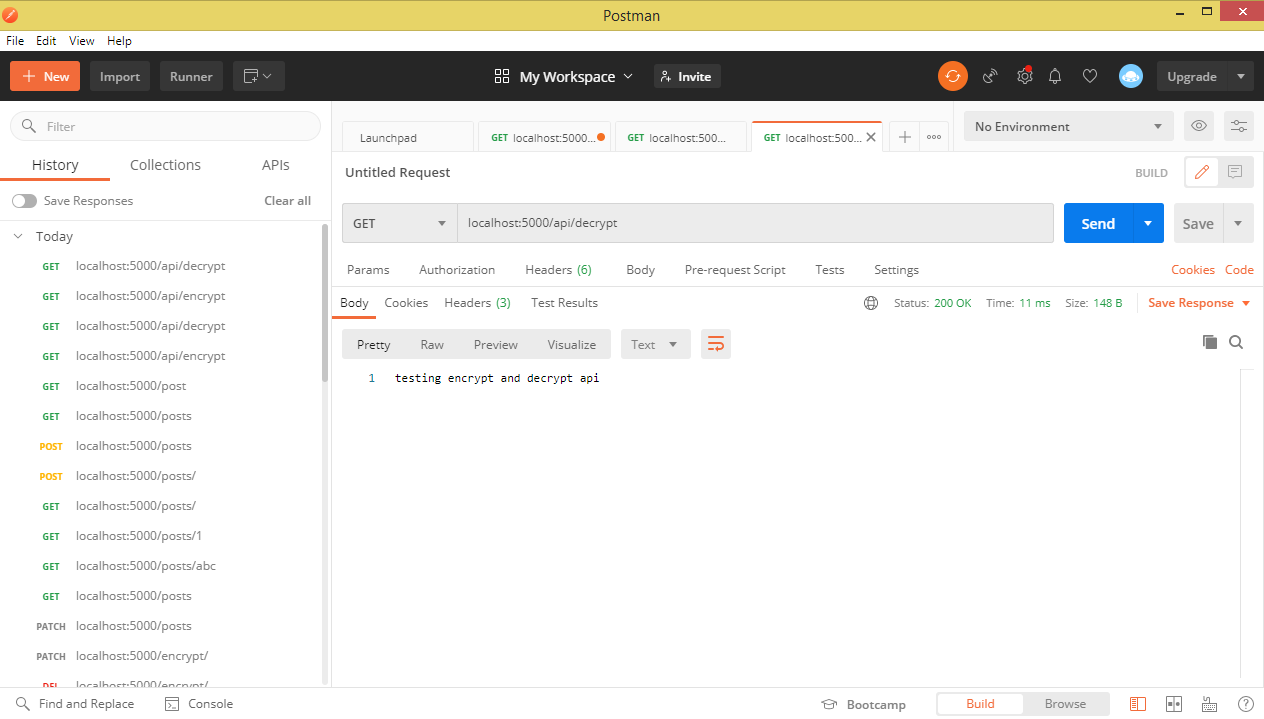
}

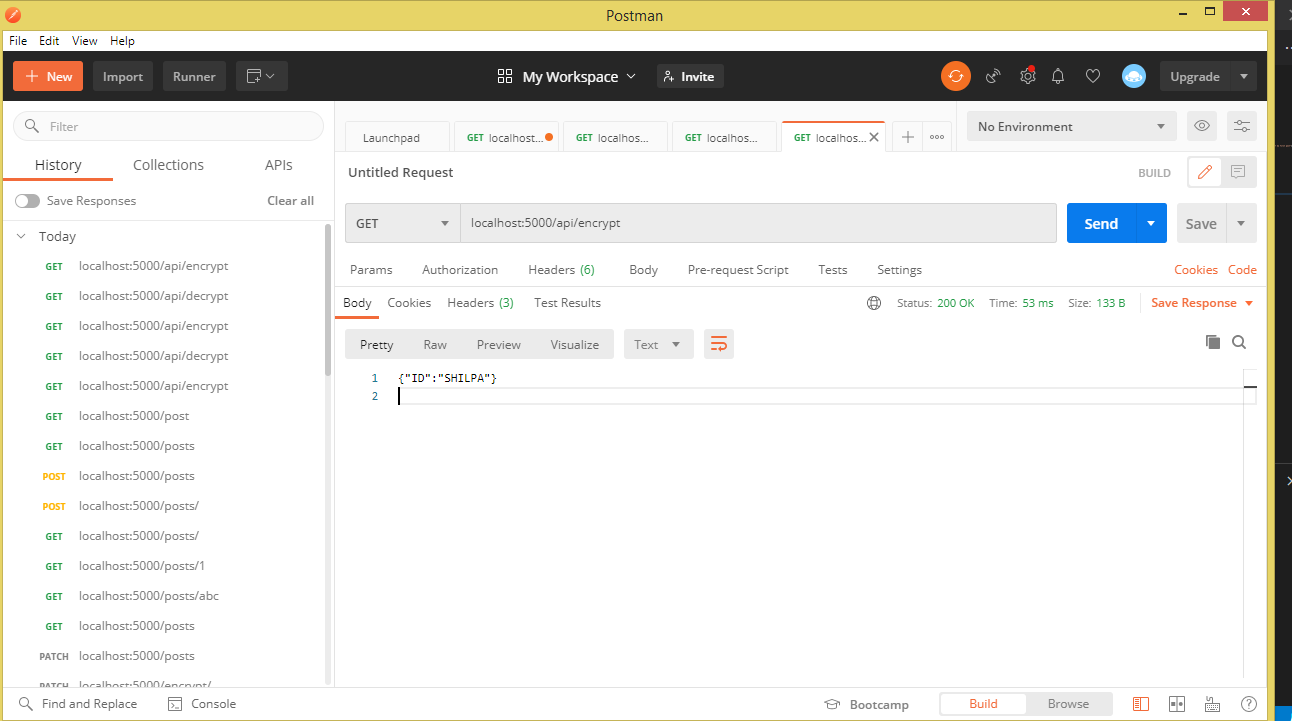
func test(w http.ResponseWriter, r \*http.Request) {

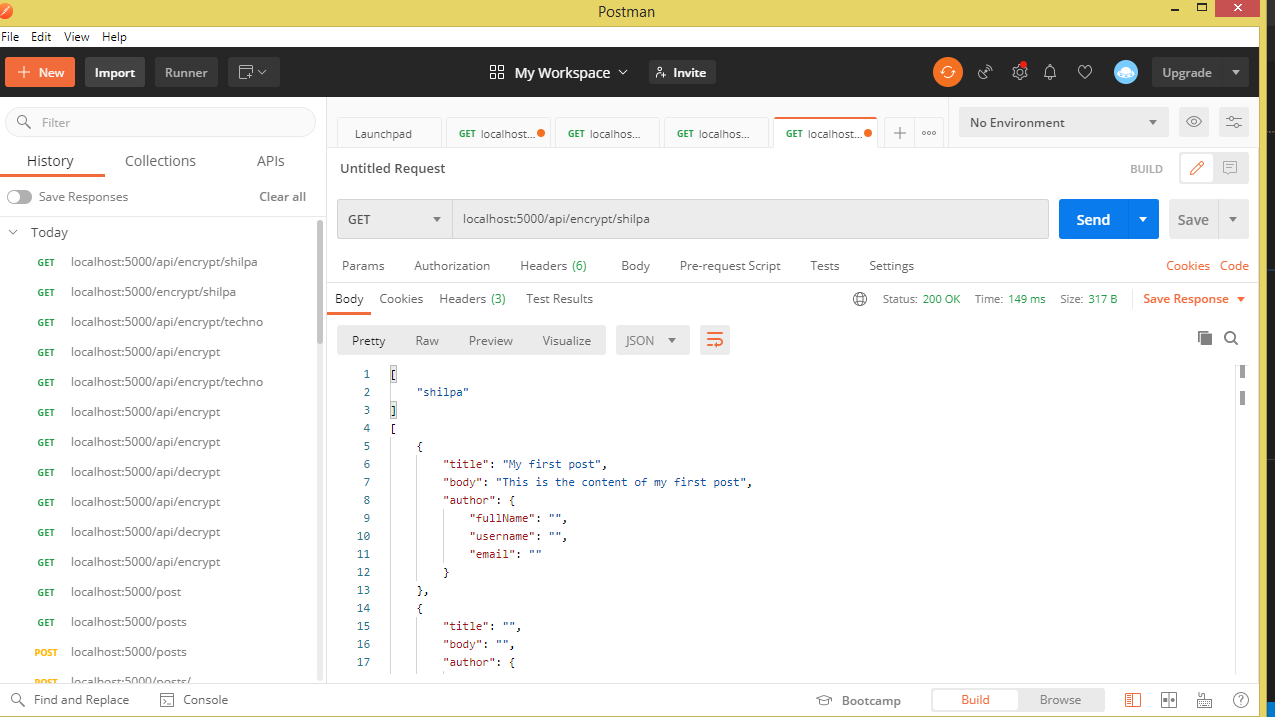
    w.Write([]byte("testing encrypt and decrypt api"))

}









APART from the

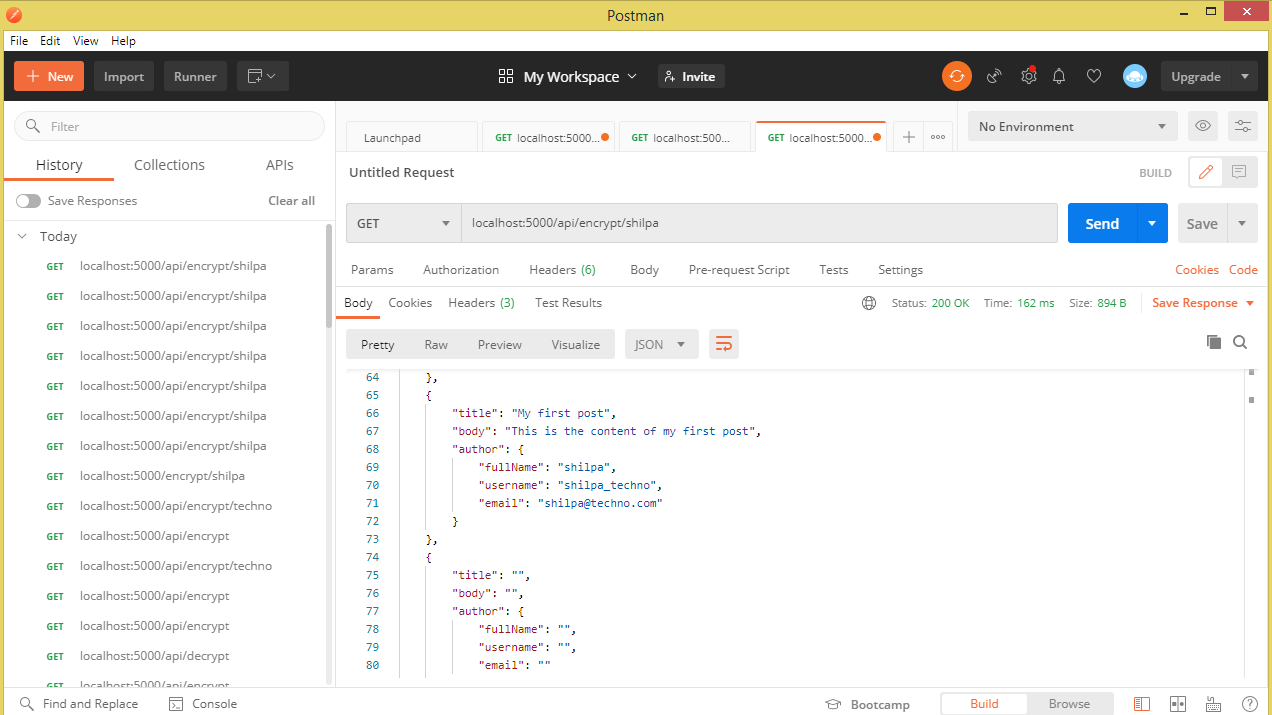
router.HandleFunc("/api/encrypt/{item}", addItem).Methods("GET", "DELETE", "PUT", "POST")

if we check for patch we get error 405 method not allowed

GET method 🡪 for getting the data

DELETE method 🡪 for deleting the data

POST method 🡪 for creating the data



Create,read,update and delete data function

id, err := strconv.Atoi(idParam)

Returns integer and error 🡪 Atoi is responsible for this

w.Write([]byte("ID could not be converted to integer"))

Does conversion from integer to slice of byte

if id >= len(posts) {

        w.WriteHeader(404)

        w.Write([]byte("No post found with specified ID"))

        return

    }

Id is invalid to handling this issue 404 🡪 page not found

For encrypting the data we need to import few packages and add functions

import (

"crypto/aes"

"crypto/cipher"

"crypto/md5"

"crypto/rand"

"encoding/hex"

"fmt"

"io"

"io/ioutil"

"os"

)

We can use MD5 hash for the encrypting:

func createHash(key string) string {

hasher := md5.New()

hasher.Write([]byte(key))

return hex.EncodeToString(hasher.Sum(nil))

}

Now that we have a key of an appropriate size, we can start the encryption process. We can be encrypting text, or any binary data, it doesn’t really matter.

func encrypt(data []byte, passphrase string) []byte {

block, \_ := aes.NewCipher([]byte(createHash(passphrase)))

gcm, err := cipher.NewGCM(block)

if err != nil {

panic(err.Error())

}

nonce := make([]byte, gcm.NonceSize())

if \_, err = io.ReadFull(rand.Reader, nonce); err != nil {

panic(err.Error())

}

ciphertext := gcm.Seal(nonce, nonce, data, nil)

return ciphertext

}

First we create a new block cipher based on the hashed passphrase. Once we have our block cipher, we want to wrap it in Galois Counter Mode (GCM) with a standard nonce length.

nonce := make([]byte, gcm.NonceSize())

if \_, err = io.ReadFull(rand.Reader, nonce); err != nil {

panic(err.Error())

}

There are a few strategies that can be used to make sure our decryption nonce matches the encryption nonce. One strategy would be to store the nonce alongside the encrypted data if it is going into a database. Another option is to prepend or append the nonce to the encrypted data. We’ll prepending the nonce.

ciphertext := gcm.Seal(nonce, nonce, data, nil)

and for decrypting:

func decrypt(data []byte, passphrase string) []byte {

key := []byte(createHash(passphrase))

block, err := aes.NewCipher(key)

if err != nil {

panic(err.Error())

}

gcm, err := cipher.NewGCM(block)

if err != nil {

panic(err.Error())

}

nonceSize := gcm.NonceSize()

nonce, ciphertext := data[:nonceSize], data[nonceSize:]

plaintext, err := gcm.Open(nil, nonce, ciphertext, nil)

if err != nil {

panic(err.Error())

}

return plaintext

}