UNIVERSIDAD DE CASTILLA LA MANCHAESCUELA SUPERIOR DE INFORMÁTICA

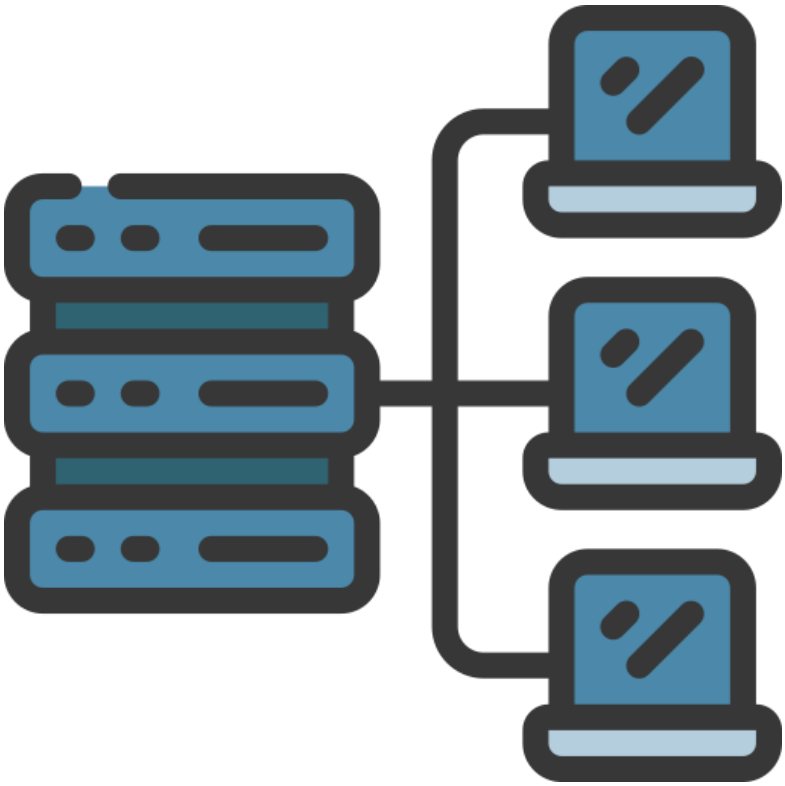
LABORATORIO 3

Seguridad en Redes

Alonso Villamayor Moreno

Alonso.Villamayor@alu.uclm.es

Curso 2023/2024



Contenido

Alonso Villamayor Moreno

Manual de usuario:

Descargar el proyecto:

Para ello debemos ejecutar el siguiente comando:

"git clone https://github.com/AlonsoVM/Seguridad_en_Redes_LAB3"

Ejecutar el proyecto:

Una vez descargado accedemos a la carpeta "Seguridad_en_Redes_LAB3".
Para la facilidad de uso y de pruebas de la API desarrollada, se ha creado un MakeFile que contiene reglas para el uso del proyecto.

make build:
Con esto vamos a crear al fichero "shadow" para las contraseñas, así como el directorio en el que se almacenará la información de los usuarios, además esto compilará el proyecto creando un binario llamado "main"

make clean:
Con esto vamos a elimar el fichero de contraseñas "shadow", el directorio en el que se almacena la información y el binario del programa.

make run:
Con esto vamos a ejecutar el programa, **es necesario ejecutar primero "make build"**.

make test:
Nos va a permitir lanzar automáticamente las pruebas de la API desarrollada.

make showTest:
Nos abrirá una pestaña en el navegador con los niveles de cobertura de los distintos ficheros en go que conforman la API.

Make cleanTest:
Se utiliza para eliminar la información obtenida al ejecutar la regla "**make test".**

Explicación del proyecto:

El código del programa se encuentra dividido en 3 fichero go, uno de ellos, **errors.go**, en el que se han creado errores personalizados para el correcto funcionamiento de la aplicación, **structs.go**, en el que se han creado estructuras para realizar las funciones

relacionadas con el dominio de la aplicación, y por último, el **main.go**, en el que se encuentra la especificación de las funciones que hacen de endpoints para la API. Adicionalmente, para el almacenamiento de usuario se ha utilizado la técnica de **fichero shadow** para no guardar las contraseñas de los usuarios del sistema en claro. También se ha desarrollado un fichero llamado **main_test.go** en el que se ha creado un conjunto de test para probar la API.

Errors.go:

En este archivo solo hemos definido estructuras que serán utilizadas durante la ejecución del programa para la comprobación de errores, para ello hemos creado por cada error una estructura y su correspondiente método que indica que hacer cuando la estructura quiere elevar un error, como se muestra a continuación:

```
type UserNotExists struct {
    User string
}

func (e *UserNotExists) Error() string {
    return "The user " + e.User + "do not exist in the system"
}
```

Esta estructura, es utilizada para manejar el caso en el que el usuario especificado no exista en el sistema.

Structs.go:

En este archivo vamos a encontrar la implementación de estructuras y métodos de estas estructuras que implementan la lógica adicional al funcionamiento de los endpoints de la API, para ello podemos distinguir entre 5 estructuras diferentes que se explica el funcionamiento de cada uno a continuación:

MemoryManager:

Esta estructura se utiliza para manejar la información que se ha almacenado o que se desea almacenar en el almacenamiento persistente, para ello cuenta con 4 funciones que pueden ser ejecutadas, estas funciones serán ejecutadas por los endpoint de la API en función del tipo de acción que se le requiera a esta. A continuación, explicamos los diferentes métodos:

- **saveInfo**:
  Este método acepta como entrada un username y filename que es de tipo string, y un “slice” de bytes que representa los datos a escribir y como salidas tenemos 2, un int que se utilizará para representar el número de bytes escritos y un error, que contendrá la información de los errores que hayan sucedido mientras se ejecutaba la función, en caso de que no exista error, error tomará el valor nil.

```go
func (Mem *MemoryManager) saveInfo(username string, filename string, data []byte) (int, error) {
    pathDir := fmt.Sprintf("%s/%s/", Mem.StorageDir, username)
    err := os.MkdirAll(pathDir, os.ModePerm)
    if err != nil {
        fmt.Println("Error creating directorys", err)
        return 0, err
    }
    pathfile := fmt.Sprintf("%s%s%s", pathDir, filename, ".json")
    _, err = os.Stat(pathfile)
    if err == nil {
        fmt.Println("The file already exits")
        return 0, &FileExits{filename}
    }
    archivo, err := os.OpenFile(pathfile, os.O_WRONLY|os.O_CREATE, 0666)
    if err != nil {
        fmt.Print("Error opening the file", filename)
        return 0, err
    }
    defer archivo.Close()
    bytesWritten, err := archivo.Write(data)
    if err != nil {
        fmt.Println("Error writting in the file :", filename)
        return 0, err
    }
    return bytesWritten, nil
}
```

Como se puede observar, primero creamos un directorio para el usuario que tendrá una ruta del tipo a “StorageDir/Username”, posteriormente, comprobamos que el archivo que vamos a crear no exista previamente en el sistema, si este no existe creamos el archivo y lo abrimos para posteriormente escribir los bytes que teníamos como entrada a la función y retornar el número de bytes escritos. Esta función será ejecutada al recibir un POST sobre el endpoint http://myserver.local:5000/username/filename creando un archivo llamado filename.json en el directorio “StorageDir/Username”.

- **updateInfo:**
  Este método, se utiliza para actualizar la información de un JSON ya almacenado, y al igual que el anterior tiene como entrada, “username” y “filename” que son de tipo string, y los datos que se quieren guardar que serán del tipo []byte, esta función devuelve un int que indica el numero de bytes escritos, y también en caso de que suceda un error durante la ejecución de la función, retorna el tipo de error, en caso de que no suceda ninguna retorna “nil”.

```go
func (Mem *MemoryManager) updateInfo(username string, filename string, data []byte) (int, error) {
    pathfile := fmt.Sprintf("%s/%s/%s%s", Mem.StorageDir, username, filename, ".json")
    archivo, err := os.OpenFile(pathfile, os.O_WRONLY, 0666)
    if err != nil {
        fmt.Print("Error opening the file", filename)
        return 0, err
    }
    defer archivo.Close()
    bytesWritten, err := archivo.Write(data)
    if err != nil {
        fmt.Println("Error writting in the file :", filename)
        return 0, err
    }
    return bytesWritten, nil
}
```

Como podemos ver el funcionamiento es muy similar al del caso anterior, no obstante, en este caso, no creamos el directorio, porque se supone que, si la

petición es correcta, el directorio del Usuario debería existir en el sistema y por lo tanto su correspondiente directorio donde almacenar los JSON. Esta función será utilizada al recibir un PUT sobre el endpoint http://myserver.local:5000/username/filename

- **removeInfo:**
  Este método, que se utiliza para eliminar un json del almacenamiento persistente, tiene como entradas el "username" y "filename" del archivo que deseamos eliminar del almacenamiento persistente, en este caso, como valores de salida tenemos el error que contendrá la información en caso de que suceda un error procesando la solicitud o "nil" en caso contrario.

```go
func (Mem *MemoryManager) removeInfo(username string, filename string) error {
    pathfile := fmt.Sprintf("%s/%s/%s%s", Mem.StorageDir, username, filename, ".json")
    err := os.Remove(pathfile)
    if err != nil {
        fmt.Print("Error removing", filename)
        return err
    }
    return nil
}
```

  Como se pude observar, la función intenta eliminar el archivo de usuario, en caso de suceda algún error, debido a que el archivo no exista, retornamos el tipo de error generado, en caso contrario retornamos "nil". Esta función será ejecutada al recibir un DELETE sobre el endpoint http://myserver.local:5000/username/filename

- **getInfo:**
  Esta función, que se utiliza para obtener la información de un json que se encuentra en el almacenamiento, tiene como entradas "username" y "filename" de tipo string, además como salida tenemos que retorna 2 valores, un interface{} que se utilizará para representar el JSON y error, que contendrá información de si se ha producido un error durante la ejecución de la función, en caso de que no suceda ningún error retorna "nil".

```go
func (Mem *MemoryManager) getInfo(username string, filename string) (interface{}, error) {
    var pathfile string
    var response interface{}
    if strings.HasSuffix(filename, ".json") {
        pathfile = fmt.Sprintf("%s/%s/%s", Mem.StorageDir, username, filename)
    } else {
        pathfile = fmt.Sprintf("%s/%s/%s%s", Mem.StorageDir, username, filename, ".json")
    }
    data, _ := os.ReadFile(pathfile)
    json.Unmarshal(data, &response)
    return response, nil
}
```

  Como se puede observar, primero leemos los bytes del archivo con "os.ReadFile" y luego utilizando la función "json.Unmarshal" lo deserializamos en objeto de tipo interface{} que será enviado como respuesta. Esta función se ejecutará cuando nos llegue una petición GET sobre el endpoint http://myserver.local:5000/username/filename

- **getAllDoc:**
  Esta función, que se utiliza para obtener todos los JSON que un usuario tiene almacenado, tiene como valores de entrada "username" que es de tipo string e indica el usuario del que obtener todos los JSON y como valores de salida, tenemos un map[string]interface{} que se utilizará para representar todos los JSON que un usuario tiene almacenado conforme a la especificación y un error que indica lo mismo que en casos anteriores.

```go
func (Mem *MemoryManager) getAllDoc(username string) (map[string]interface{}, error) {
    data := make(map[string]interface{})
    pathDir := fmt.Sprintf("%s/%s", Mem.StorageDir, username)
    dir, err := os.Open(pathDir)
    if err != nil {
        fmt.Println("Error opening the directory : ", pathDir)
        return data, err
    }
    defer dir.Close()

    files, err := dir.Readdirnames(0)
    if err != nil {
        fmt.Println("Error reading the entries of the directory : ", pathDir)
    }

    for i, file := range files {
        jsonData, _ := Mem.getInfo(username, file)
        data["id"+fmt.Sprint(i)] = jsonData
    }
    return data, nil
}
```

  Lo más representativo es que creamos un objeto de tipo map[string]interface{} para enviar la respuesta a la solicitud con el formato especificado, posteriormente, leemos los archivos que haya en la entrada de directorio del usuario y los añadimos al map[string]interface{} como se muestra en el bucle for, primero obtenemos la información de un JSON y eso lo asignamos al string idx dentro del map[string]interface{}, una vez recorridas todas las entradas del directorio lo retornamos. Esta función será ejecutada al recibir una operación GET sobre el endpoint http://myserver.local:5000/username/_all_docs

User:

Esta estructura se utiliza para manejar la información relativa al usuario como se muestra a continuación:

```go
type User struct {
    UserName string `json:"username"`
    Password string `json:"password"`
    Salt     string `json:"salt"`
}
```

UserName que indica el nombre el usuario en el sistema, Password que indica el hash de la contraseña del usuario para evitar guardarlas en claro ya que se utiliza un **"ficher**

shadow” para las contraseñas y por ultimo la Salt para evitar que dos usuarios con la misma contraseña tengan el mismo hash asociado.

UserManager:

Esta estructura se utiliza para manejar la información y métodos de los usuarios:

```
type UserManager struct {
    Users []User `json:"users"`
    file  string
}
```

Users que contiene una lista de los usuarios que existen en el sistema y file que contiene el nombre del **“fichero shadow”**.

A continuación, explicamos las funciones asociadas a la estructura UserManager:

- **UserExist**:
  Tiene como parámetros de entrada UserName de tipo string que indica el nombre del usuario que se debe comprobar si existe en el sistema.

```
func (UserL *UserManager) UserExist(UserName string) bool {
    for i := 0; i < len(UserL.Users); i++ {
        if UserName == UserL.Users[i].UserName {
            return true
        }
    }
    return false
}
```

  Retorna verdadero o falso dependiendo de si el usuario existe o no en el sistema.

- **getUserSalt:**
  Tiene como parámetros de entrada UserName de tipo string que indica el usuario del que obtener la Salt.

```
func (UserL *UserManager) getUserSalt(UserName string) string {
    for _, userAux := range UserL.Users {
        if userAux.UserName == UserName {
            return userAux.Salt
        }
    }
    return ""
}
```

  Retorna la sal de usuario en caso de que el usuario exista, en caso contrario devuelve una cadena vacía.

- **getUserPassword:**
  Tiene como parámetros de entrada UserName de tipo string que indica el usuario del que obtener la contraseña.

```go
func (UserL *UserManager) getUserPassword(UserName string) string {
    for _, userAux := range UserL.Users {
        if userAux.UserName == UserName {
            return userAux.Password
        }
    }
    return ""
}
```

Retorna la contraseña de usuario en caso de que el usuario exista, en caso contrario devuelve una cadena vacía.

- **saveUsers:**
  Esta función tiene como entrada NewUser que es de tipo User, esta estructura contiene la información del usuario a añadir al sistema.

```go
func (UserL *UserManager) saveUsers(NewUser User) {
    archivo, err := os.OpenFile(UserL.file, os.O_RDWR|os.O_CREATE, 0666)
    if err != nil {
        fmt.Print("Error opening the file", UserL.file)
    }
    defer archivo.Close()
    encoder := json.NewEncoder(archivo)
    UserL.Users = append(UserL.Users, NewUser)
    encoder.SetIndent("", " ")
    encoder.Encode(&UserL)
}
```

Como se puede observar se utiliza un objeto json.NewEncoder() para almacenar la información de usuario en el **"ficher Shadow".**

- **loadUsers:**
  Esta función se utiliza para cargar en memoria los usuarios del sistema cuando se inicia la aplicación.

```go
func (UserL *UserManager) loadUsers() {
    archivo, err := os.OpenFile(UserL.file, os.O_RDWR|os.O_CREATE, 0666)
    if err != nil {
        fmt.Print("Error opening the file", UserL.file)
    }
    defer archivo.Close()
    decoder := json.NewDecoder(archivo)
    decoder.Decode(&UserL)
}
```

- **createHashedPassword:**
  Esta función se utiliza para crear el hash que actuara como contraseña para los usuarios del sistema, para ello se combina la salt y la contraseña de un usuarios y se devuelve el hash sha256 de esa combinación.

```go
func createHashedPassword(salt string, pass string) string {
    tempPassword := fmt.Sprintf("%s%s", salt, pass)
    hash := sha256.New()
    return hex.EncodeToString(hash.Sum([]byte(tempPassword)))
}
```

- **createUser**:
  Esta función se utiliza para crear un usuario dentro del sistema, tiene como entradas un body de tipo io.ReadCloser que representa el cuerpo de la solicitud http con los parámetros del usuario a crear.

```go
func (UserL *UserManager) createUser(body io.ReadCloser) (User, error) {
    rand.NewSource(time.Now().UnixMilli())
    var datosJson, _ = io.ReadAll(body)
    var UserAux User

    json.Unmarshal(datosJson, &UserAux)
    if UserL.UserExist(UserAux.UserName) {
        return UserAux, &UserExists{"The user already exits"}
    }

    UserAux.Salt = fmt.Sprintf("%d", rand.Int())
    UserAux.Password = createHashedPassword(UserAux.Salt, UserAux.Password)
    UserL.saveUsers(UserAux)
    return UserAux, nil
}
```

  Esta función se apoya en la de saveUsers para guardar la información del nuevo usuario de manera persistente y también de createHashedPassword.

- **logUser**
  Esta función se utiliza para logear un usuario dentro del sistema, tiene como entradas un body de tipo io.ReadCloser que representa el cuerpo de la solicitud http con los parámetros del usuario a logear.

```go
func (UserL *UserManager) logUser(body io.ReadCloser) (User, error) {
    var datosJson, _ = io.ReadAll(body)
    var UserAux User
    json.Unmarshal(datosJson, &UserAux)
    if !UserL.UserExist(UserAux.UserName) {
        return UserAux, &UserNotExists{UserAux.UserName}
    }

    Password := UserL.getUserPassword(UserAux.UserName)
    Salt := UserL.getUserSalt(UserAux.UserName)
    tempPass := createHashedPassword(Salt, UserAux.Password)
    if Password != tempPass {
        return UserAux, &InvalidPassword{"The password provided is invalid"}
    }

    return UserAux, nil
}
```

  En caso de que la función se ejecute correctamente devuelva un Usuario válido y “nil” en el error, en caso contrario devuelve el tipo de error que a sucedido mientras se procesaba la solicitud.

VolatileToken:

Esta estructura se utiliza para manejar la información de los tokens temporales que se crean para autenticar las sesiones de usuarios. La estructura es la siguiente:

```
type VolatileToken struct {
    Token    string
    userName string
    Time     time.Time
}
```

Token que contiene el identificador del token, userName que indica el nombre del propietario del token y Time que indica la hora a la que el token fue creado.

TokenManager:

Esta estructura se utiliza para manejar la información de los tokens del sistema.

```
type TokenManager struct {
    VolatileTokens []VolatileToken
    mutex          sync.Mutex
}
```

Contiene una lista de “VolatileToken” y un mutex que será utilizado para la sincronización de operaciones simultaneas, ya que para eliminar los tokens con mayor antigüedad a 5 minutos se utiliza una go routine.
A continuación, explicamos las diferentes funciones de esta estructura:

- **getTokenOwner**:
  Esta función acepta como entrada un id del token del que queremos obtener el dueño, y como salida tenemos el nombre de usuario o una cadena vacía en caso de que el token no exista.

```
func (tokenList *TokenManager) getTokenOwner(id string) string {
    for _, token := range tokenList.VolatileTokens {
        if token.Token == id {
            return token.userName
        }
    }
    return ""
}
```

- **tokenExists**:
  Esta función se utiliza para comprobar si un token existe, devuelve verdadero en caso afirmativo.

```go
func (tokenList *TokenManager) tokenExists(id string) bool {
    for _, token := range tokenList.VolatileTokens {
        if token.Token == id {
            return true
        }
    }
    return false
}
```

- **deleteOldTokens**:
  Esta función es la que utilizará la rutina go para eliminar los tokens que lleven más de 5 minutos en el sistema.

```go
func (tokenList *TokenManager) deleteOldTokens() {
    for true {
        tokenList.mutex.Lock()
        for i, token := range tokenList.VolatileTokens {
            if time.Now().Sub(token.Time).Seconds() > 300 {
                tokenList.VolatileTokens = append(tokenList.VolatileTokens[:i], tokenList.VolatileTokens[i+1:]...)
                fmt.Println("Removing token : ", token.Token)
            }
        }
        tokenList.mutex.Unlock()
        time.Sleep(10 * time.Second)
    }
}
```

  Esta función esta hecha para que cada 10 segundos compruebe el tiempo que el token lleva en el sistema, si durante la comprobación un token lleva más de 5 minuto en el sistema lo eliminamos.

- **saveToken:**
  Esta función se utiliza para añadir un token a la lista de tokens validos en el sistema.

```go
func (tokenList *TokenManager) saveToken(tempToken string, userName string) {
    var token VolatileToken
    token.Token = tempToken
    token.userName = userName
    token.Time = time.Now()
    tokenList.mutex.Lock()
    tokenList.VolatileTokens = append(tokenList.VolatileTokens, token)
    tokenList.mutex.Unlock()
}
```

  Como se puede observar antes de añadir el token a la lista, tenemos que hacernos con el acceso exclusivo a la lista para evitar las condiciones de carrera.

- **removeSingleToken**:
  Esta función se utiliza para eliminar un solo token, esto se utilizara cuando un usuario con un token válido nos vuelva a pedir otro, ya que solo vamos a permitir una sesión por usuario en el sistema.

```go
func (tokenList *TokenManager) removeSingleToken(username string) {
    tokenList.mutex.Lock()
    for i, token := range tokenList.VolatileTokens {
        if token.userName == username {
            tokenList.VolatileTokens = append(tokenList.VolatileTokens[:i], tokenList.VolatileTokens[i+1:]...)
            fmt.Println("Removing token : ", token)
        }
    }
    tokenList.mutex.Unlock()
}
```

Al igual que en el caso anterior nos tenemos que hacer con el acceso exclusivo para evitar condiciones de carrera.

- **createToken**:
  Se utiliza para crear un token de manera aleatoria intentando evitar colisiones entre tokens.

```go
func (tokenList *TokenManager) createToken() string {
    rand.NewSource(time.Now().UnixMilli())
    random := rand.Int63()
    bytes := make([]byte, 8)
    binary.BigEndian.PutUint64(bytes, uint64(random))
    token := base64.StdEncoding.EncodeToString([]byte(bytes))
    return token
}
```

- **getToken**:
  Se utiliza para obtener un token para un usuario.

```go
func (tokenList *TokenManager) getToken(username string) string {
    token := tokenList.createToken()
    tokenList.removeSingleToken(username)
    tokenList.saveToken(token, username)
    return token

}
```

Como se puede observar, utiliza las funciones anteriormente mencionadas para crear el token, guardarlo y eliminarlo, esto último, en caso de que el usuario ya tuviese un token activo.

main.go:

Dentro del main, podemos distinguir dos tipos de funciones, las que implementan directamente la funcionalidad del endpoint y las que se utilizan para proporcionar funcionalidad adicional. También es importante resaltar que en el main se utilizan tres variables globales que son UserManager, MemoryManager y TokenManager para manejar la lógica subyacente de la API.

Funciones Adicionales:

Estas funciones se han utilizado con el objetivo de modularizar el código lo máximo posible, ya que muchas peticiones tienen a nuestra API tienen que realizar un tratamiento mi similar a la petición HTTP.

- **createDocResponse:** Se utiliza para crear la respuesta con el formato adecuado, a una petición de subir un JSON.

```go
func createDocResponse(bytesWritten int) map[string]interface{} {
    data := map[string]interface{}{
        "size": bytesWritten,
    }
    return data
}
```

- **createTokenResponse:** Se utiliza para crear la respuesta con el formato adecuado, a una petición de login o signup.

```go
func createTokenResponse(token string) map[string]interface{} {
    data := map[string]interface{}{
        "access_token": token,
    }
    return data
}
```

- **parseHeader:** Se utiliza para obtener el token que autentica la sesión de la cabecera de la petición HTTP. También comprueba que la cabecea se ajuste al formato indicado en el enunciado de la practica.

```go
func parseHeader(authHeader string) (string, error) {
    if authHeader == "" {
        return "", &MissingAuthHeader{"Missing Authorization Header in the request"}
    }
    words := strings.Split(authHeader, " ")
    if len(words) != 2 {
        return "", &BadAuthHeader{"Malformed Authorization header, type is token token_id"}
    }
    return words[1], nil
}
```

- **parseParams:** Se utiliza para obtener los parámetros dentro de la URL que puedan variar, por ejemplo, en la url http://myserver.local:5000/Alonso/fichero obtener los valores ”Alonso” y “fichero”. También se puede utiliza para comprobar errores, y se contempla el caso en el que la url es http://myserver.local:5000/Alonso/_all_docs en el que los parámetros solo es el nombre de usuario en este caso “Alonso”

```go
func parseParams(params gin.Params, token string) (string, string, error) {
    var username string
    var docId string
    if len(params) != 1 {
        username = params[0].Value
        docId = params[1].Value
    } else {
        username = params[0].Value
    }
    if !userManager.UserExist(username) {
        return "", "", &UserNotExists{username}
    }
    if !tokenManager.tokenExists(token) {
        return "", "", &TokenExpired{token}
    }
    if username != tokenManager.getTokenOwner(token) {
        return "", "", &NotOwner{username, token}
    }
    return username, docId, nil
}
```

- **parseBody:** Se utiliza para obtener el contenido del archivo a guardar, ya que el formato del cuerpo de una petición de subida de archivos es {“doc_content”: “contenido”} por lo tanto eliminamos el valor doc_content para guardar solo la información del JSON. Adicionalmente también comprueba los errores.

```go
func parseBody(body io.ReadCloser) ([]byte, error) {
    datosJson, _ := io.ReadAll(body)
    var jsonFormat map[string]interface{}
    var dataToSave []byte

    json.Unmarshal(datosJson, &jsonFormat)
    tempData := jsonFormat["doc_content"]
    if tempData == nil {
        return dataToSave, &MissingDocContent{"Missing doc_content"}
    }

    dataToSave, _ = json.Marshal(tempData)

    return dataToSave, nil

}
```

Función del Endpoint GET “/versión”:

En este caso la funcionalidad es muy sencilla ya que solo devuelve un string con el número de versión de la API.

```go
func VersionHandler(c *gin.Context) {
    c.String(http.StatusOK, "0.1.0")
}
```

Función del Endpoint POST "/signup":

Esta función, lo primero que hace es pedirle a la estructura UserManager que cree un usuario y le da el valor del cuerpo de la petición HTTP, que contiene el nombre de usuario y contraseña, y esta le devuelve una estructura de tipo usuario, posteriormente, con el usuario creado se llama al método getToken de tokenManager para crear un token temporal, por último, se adecua la respuesta al formato especificado y se envía al cliente.

```
func SignupHandler(c *gin.Context) {
    var UserAux, err = userManager.createUser(c.Request.Body)
    if err != nil {
        c.String(http.StatusConflict, err.Error())
        return
    }

    token := tokenManager.getToken(UserAux.UserName)
    response := createTokenResponse(token)
    c.IndentedJSON(http.StatusOK, response)

}
```

En caso de que el usuario ya exista retornamos con el código 409, en caso satisfactorio 200.

Función del Endpoint POST "/login":

La implementación de este endpoint es muy parecido al caso anterior, pero en lugar de llamar al método createUser de la estructura UserManager, llamamos al método logUser.

```
func LoginHandler(c *gin.Context) {
    var UserAux, err = userManager.logUser(c.Request.Body)
    if err != nil {
        c.String(http.StatusUnauthorized, err.Error())
        return
    }
    token := tokenManager.getToken(UserAux.UserName)
    response := createTokenResponse(token)
    c.IndentedJSON(http.StatusOK, response)
}
```

En caso de que el intento de log falle, devolvemos 401, en caso contrario 200.

Función del Endpoint GET "/<string: username>/_all_docs":

En este caso, lo que vamos a hacer primero es utilizar la función del main parseHeader para obtener el token de la cabecera HTTP, posteriormente utilizaremos la función parseParams para obtener los parámetros de la petición y por utlimo usamos la función getAllDocs de la estructura MemoryManager para obtener la información que será mandada como respuesta.

```go
func AllDocsHandler(c *gin.Context) {
    authHeader := c.Request.Header.Get("Authorization")
    token, err := parseHeader(authHeader)
    if err != nil {
        c.String(http.StatusBadRequest, err.Error())
        return
    }

    username, _, err := parseParams(c.Params, token)
    if err != nil {
        c.String(http.StatusUnauthorized, err.Error())
        return
    }

    data, err := MemManager.getAllDoc(username)
    if err != nil {
        c.String(http.StatusInternalServerError, err.Error())
        return
    }

    c.IndentedJSON(http.StatusOK, data)
}
```

En caso de que la cabecera este mal construida enviamos código 400, si el usuario no coincide con el token enviamos 401, en caso de fallo al ejecutar la acción relacionada a la petición enviamos error 500. Y en caso satisfactorio 200.

Función del Endpoint "/<string: username>/<string: doc_id>":

Este endpoint implementa todos los métodos (GET, PUT, POST, DELETE) dentro de la misma función ya que el tratamiento de la petición HTTP es muy similar en todos los casos.

Lo primero que vamos a hacer y que es común a todas las peticiones es usar las funciones parseHeader para obtener el token de la cabecera de la petición HTTP y luego la función parseParams, para obtener los parámetros.

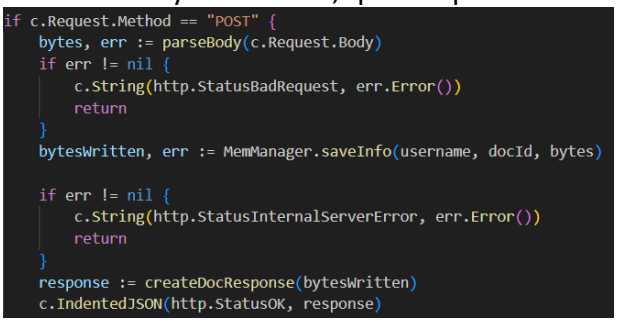
```go
func DocHandler(c *gin.Context) {
    authHeader := c.Request.Header.Get("Authorization")
    token, err := parseHeader(authHeader)
    if err != nil {
        c.String(http.StatusBadRequest, err.Error())
        return
    }
    username, docId, err := parseParams(c.Params, token)
    if err != nil {
        c.String(http.StatusUnauthorized, err.Error())
        return
    }
```

En caso de que la cabecera este mal construida enviamos código 400, si el usuario no coincide con el token enviamos 401.

Posteriormente apoyándonos en "**c.Request.Method**" diferenciamos el tipo de petición que estamos recibiendo y actuamos en consecuencia.

- POST: En este caso lo primero que hacemos es utilizar la función parseBody

para obtener la información del JSON a guardar, luego llamamos a la función saveInfo de MemoryManager para guardarlo, esta llamada nos retorno el numero de bytes escritos, que adaptaremos al formato de respuesta adecuado.

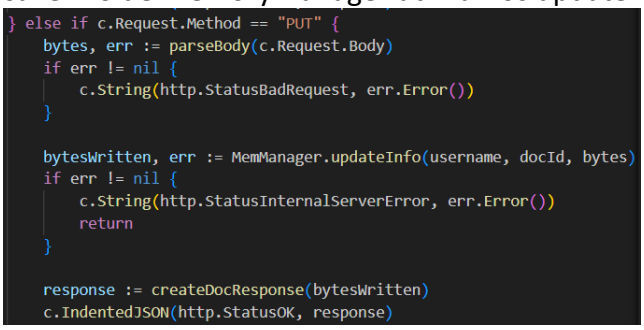


```
if c.Request.Method == "POST" {
    bytes, err := parseBody(c.Request.Body)
    if err != nil {
        c.String(http.StatusBadRequest, err.Error())
        return
    }
    bytesWritten, err := MemManager.saveInfo(username, docId, bytes)

    if err != nil {
        c.String(http.StatusInternalServerError, err.Error())
        return
    }
    response := createDocResponse(bytesWritten)
    c.IndentedJSON(http.StatusOK, response)
```

En caso de que el cuerpo de la petición no se adecue a la especificación enviamos 400, en caso de fallo al ejecutar la acción relacionada a la petición enviamos error 500. Y en caso satisfactorio 200.

- PUT: Este caso es muy similar al anterior ya que en lugar de utilizar la función saveInfo de MemoryManager utilizamos updateInfo.



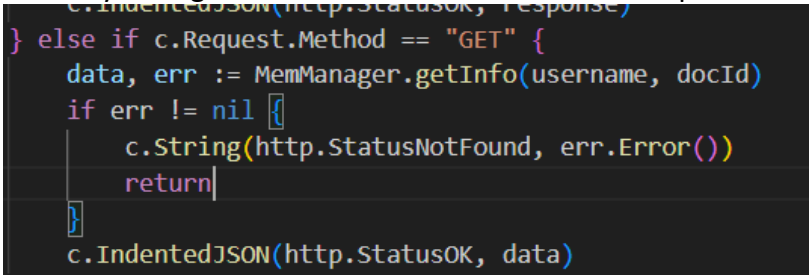
```
} else if c.Request.Method == "PUT" {
    bytes, err := parseBody(c.Request.Body)
    if err != nil {
        c.String(http.StatusBadRequest, err.Error())
    }

    bytesWritten, err := MemManager.updateInfo(username, docId, bytes)
    if err != nil {
        c.String(http.StatusInternalServerError, err.Error())
        return
    }

    response := createDocResponse(bytesWritten)
    c.IndentedJSON(http.StatusOK, response)
```

En caso de que el cuerpo de la petición no se adecue a la especificación enviamos 400, en caso de fallo al ejecutar la acción relacionada a la petición enviamos error 500. Y en caso satisfactorio 200.

- GET: En este caso directamente llamamos a la función getInfo de MemoryManager con las entradas obtenidas de los parámetros de la petición.



```
} else if c.Request.Method == "GET" {
    data, err := MemManager.getInfo(username, docId)
    if err != nil {
        c.String(http.StatusNotFound, err.Error())
        return
    }
    c.IndentedJSON(http.StatusOK, data)
```

- DELETE: En este caso llamamos a la función removeInfo de MemoryManager con las entradas obtenidas de los parámetros de la petición.

```
else if c.Request.Method == "DELETE" {
    err := MemManager.removeInfo(username, docId)
    if err != nil {
        c.String(http.StatusNotFound, err.Error())
        return
    }
    c.IndentedJSON(http.StatusOK, "{}")
```

Ya por último en la función main inicializamos las variables globales y enrutamos los diferentes endpoints a las funciones mencionadas anteriormente.

```
func main() {

    r := gin.Default()
    ruta := "Usuarios.json"
    storage := "StorageDir"
    userManager.file = ruta
    MemManager.StorageDir = storage
    userManager.loadUsers()
    go tokenManager.deleteOldTokens()
    r.POST("/signup", SignupHandler)
    r.POST("/login", LoginHandler)
    r.GET("/version", VersionHandler)
    r.POST("/:username/:doc_id", DocHandler)
    r.PUT("/:username/:doc_id", DocHandler)
    r.GET("/:username/:doc_id", DocHandler)
    r.DELETE("/:username/:doc_id", DocHandler)
    r.GET("/:username/_all_docs", AllDocsHandler)
    r.Run("myserver.local:5000") // listen and ser
}
```

Disponibilidad del servicio:

Como medidas para hacer un servicio más disponible, se podrían crear distintas instancias de del mismo servicio de tal modo que con un balanceador de carga podamos distribuir el trabajo equitativamente entre las distintas instancias del servicio, para el almacenamiento de datos podríamos considerar un sistema de memoria centralizado, de tal modo que las distintas instancias del servicio accedan a la misma memoria asegurando la exclusión entre operaciones.
Además, también se podría implementar un log del sistema para poder saber que operaciones pueden o han causado fallos en el sistema.

También se ha desarrollado medidas para asegurar un mejor servicio como un conjunto de pruebas, de tal modo que, un alto grado de cobertura puede indicar alta calidad del código y por lo tanto más resiliencia a fallos que pueden surgir durante su ejecución. Además, esto nos puede ayudar en el caso de que en un futuro se necesite realizar cambios en la aplicación ya que la aplicación con los cambios debería de ser capaz de pasar todos los test de nuevo. Y también se ha tenido en cuenta aspectos relacionados con la gestión de errores consiguiendo evitar así errores que tumben el sistema.