Go in Action 2 Assignment Report

Title: Go Security, idiomatic Go and Go documentation for Home Baker App

Submitted by: Tho Yan Bo

Table of Contents

1. Application Overview.

Home Baker is a simple application that is used to manage the orders of a home based bakery for the upcoming week (Monday to Sunday).

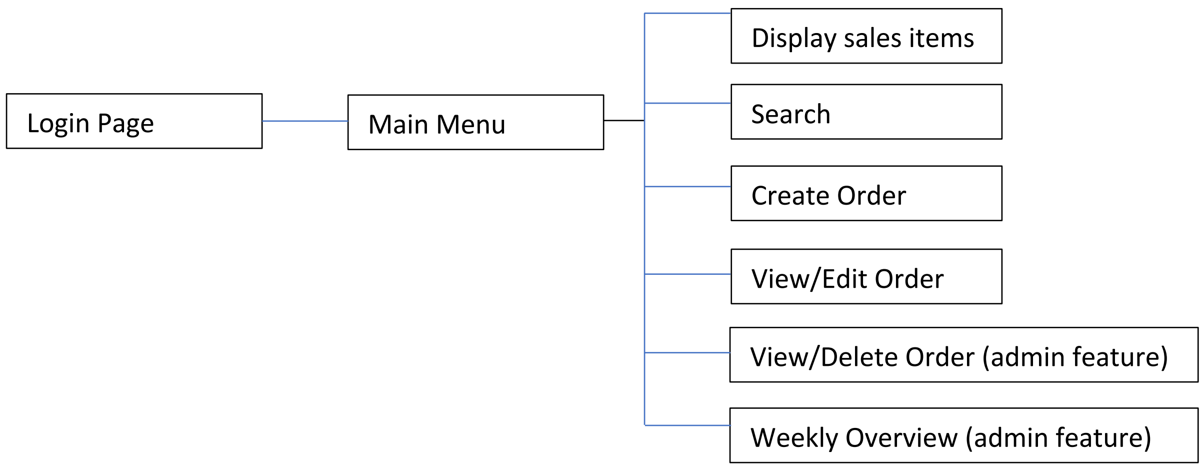
Below is an overview of the client and server structure for the home baker application.

Client	Server
Login Feature - New customer sign up - Existing customer login - Admin login	Login - Issue Cookie - Store session-user information - Store user information
Customer Page - Display all sales items - Search function - Create a new order - View or edit order	Sales item data - Sales item availability (derived from max weekly capacity and weekly order) - Item data are stored in array based list Delivery days data - Delivery day availability (based on pre-defined daily delivery cap) Booked orders data - Booked orders are stored into a linked list
Admin Page - Display sales items - Search function - Create a new order - View or edit order - Delete order (admin exclusive) - Order overview (admin exclusive)	Manages weekly overview by returning information to admin through backend handling of item and booking data: - Weekly Revenue - Daily Orders details - Item order details

For this assignment, the incorporation of the server and client is using a localhost at port 5221.

Application Features

The Home Baker application consists of the following features.



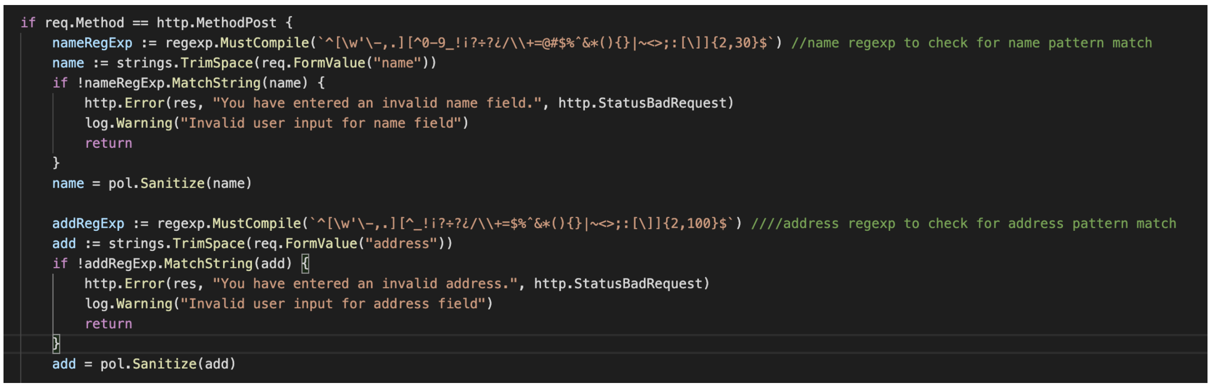
2. Discussion of Security concepts discussed in Go in Action 2 and applied.

2.1 Input validation

User input is encouraged to be checked against a set of conditions in order to guarantee that user is entering expected data. Unchecked user input and data can pose a security risk. Throughout the home baker web application, input validation or input sanitization is applied whenever user input is submitted to tackle this security risk.

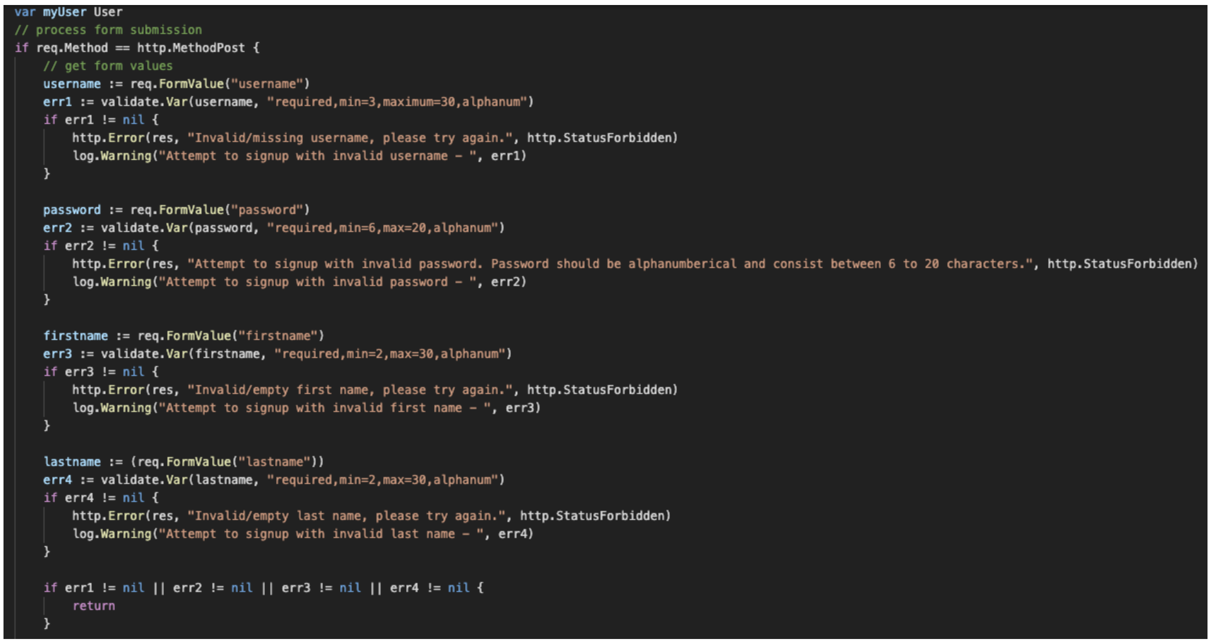
If validation of the input fails, the input is rejected by the application.

For this assignment, third party Validator package (https://github.com/go-playground/validator) and custom regular expressions were compiled to validate user input. Invalidated inputs are rejected by the application.



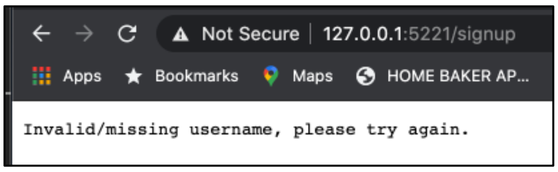
```go
if req.Method == http.MethodPost {
    nameRegExp := regexp.MustCompile(`^[\w'\-,.][^0-9_!¡?÷?¿/\\+=@#$%ˆ&*(){}|~<>;:[\]]{2,30}$`) //name regexp to check for name pattern match
    name := strings.TrimSpace(req.FormValue("name"))
    if !nameRegExp.MatchString(name) {
        http.Error(res, "You have entered an invalid name field.", http.StatusBadRequest)
        log.Warning("Invalid user input for name field")
        return
    }
    name = pol.Sanitize(name)

    addRegExp := regexp.MustCompile(`^[\w'\-,.][^_!¡?÷?¿/\\+=$%ˆ&*(){}|~<>;:[\]]{2,100}$`) ////address regexp to check for address pattern match
    add := strings.TrimSpace(req.FormValue("address"))
    if !addRegExp.MatchString(add) {
        http.Error(res, "You have entered an invalid address.", http.StatusBadRequest)
        log.Warning("Invalid user input for address field")
        return
    }
    add = pol.Sanitize(add)
```

Example: When creating an order, usage of custom compiled regular expression to validate user input against expected patterns for name and address field is performed.



```go
var myUser User
// process form submission
if req.Method == http.MethodPost {
    // get form values
    username := req.FormValue("username")
    err1 := validate.Var(username, "required,min=3,maximum=30,alphanum")
    if err1 != nil {
        http.Error(res, "Invalid/missing username, please try again.", http.StatusForbidden)
        log.Warning("Attempt to signup with invalid username - ", err1)
    }

    password := req.FormValue("password")
    err2 := validate.Var(password, "required,min=6,max=20,alphanum")
    if err2 != nil {
        http.Error(res, "Attempt to signup with invalid password. Password should be alphanumberical and consist between 6 to 20 characters.", http.StatusForbidden)
        log.Warning("Attempt to signup with invalid password - ", err2)
    }

    firstname := req.FormValue("firstname")
    err3 := validate.Var(firstname, "required,min=2,max=30,alphanum")
    if err3 != nil {
        http.Error(res, "Invalid/empty first name, please try again.", http.StatusForbidden)
        log.Warning("Attempt to signup with invalid first name - ", err3)
    }

    lastname := (req.FormValue("lastname"))
    err4 := validate.Var(lastname, "required,min=2,max=30,alphanum")
    if err4 != nil {
        http.Error(res, "Invalid/empty last name, please try again.", http.StatusForbidden)
        log.Warning("Attempt to signup with invalid last name - ", err4)
    }

    if err1 != nil || err2 != nil || err3 != nil || err4 != nil {
        return
    }
```

Example: Third party package “Validator” is used to validate submitted form values. In above example, user inputs are expected to be simple alphanumerical patterns with character count requirements (such as username being alphanumerical and between 3 and 30 characters).



```
Invalid/missing username, please try again.
```

Example of http error response due to input validation failing.

2.2 Input Sanitization

Input sanitization is the process of removing or replacing submitted data, typically after input validation is done to strength data safety. Untrusted data should be encoded before inserting into HTML elements to escape characters that may switch to execution context (such as script, style or event handlers).

In this assignment, third party library bluemonday (https://github.com/microcosm-cc/bluemonday) has been imported to perform the stripping of all tags – it is a HTML sanitizer implemented in Go and takes untrusted user generated content as an input, and returns HTML that has been sanitized against a whitelist of approved HTML elements. Such a practice can help protect sites from potential XSS attacks.

```go
// Do this once for each unique policy, and use the policy for the life of the program
// Policy creation/editing is not safe to use in multiple goroutines
var pol = bluemonday.UGCPolicy() //pol for policy
```

```go
name = pol.Sanitize(name) //pol.Sanitize is used to sanitize inputs

addRegExp := regexp.MustCompile(`^[\w'\-,.][^_!i?÷?¿/\\+=$%ˆ&*(){}|~<>;:[\]]{2,100}$`) ////address regexp to check for address pattern match
add := strings.TrimSpace(req.FormValue("address"))
if !addRegExp.MatchString(add) {
    http.Error(res, "You have entered an invalid address.", http.StatusBadRequest)
    log.Warning("Invalid user input for address field")
    return
}
add = pol.Sanitize(add) //pol.Sanitize is used to sanitize inputs
```

2.3 URL request path

In net\http, ServerMux, an HTTP request multiplexer type, matches incoming request to the registered patterns and calls the handler that most closely matches the URL. It also sanitizes the URL request path, redirecting requests to cleaner URL. However, it does not change URL request path for CONNECT requests, thus making app vulnerable for path traversal attacks.

As such, third party Gorilla Toolkit – MUX (Https://github.com/gorilla/mux) is used in this assignment to limit allowed request methods to 'GET' and 'POST', and also limit schemes to "https".

```go
func main() {
    r := mux.NewRouter()
    r.Handle("/favicon.ico", http.NotFoundHandler()).Methods("POST", "GET").Schemes("https")
    r.HandleFunc("/", ses.Index).Methods("POST", "GET").Schemes("https")
    r.HandleFunc("/signup", ses.Signup).Methods("POST", "GET").Schemes("https")
    r.HandleFunc("/login", ses.Login).Methods("POST", "GET").Schemes("https")
    r.HandleFunc("/logout", ses.Logout).Methods("POST", "GET").Schemes("https")
    r.HandleFunc("/menu", menu).Methods("POST", "GET").Schemes("https")
```

Example of respective handlers being registered using Gorilla MUX to match only allowed HTTP methods or URL schemes.

2.4 Cross Site Scripting

Text/plain and/or the text/template package will not keep our application away from XSS as it does not sanitize user input. By using html/template package in my application, the web application will be able to proceed safely.

All user supplied inputs are also sanitized using bluemonday package (as mentioned in section 2.2) and therefore also provides extra protection against potential cross site scripts.

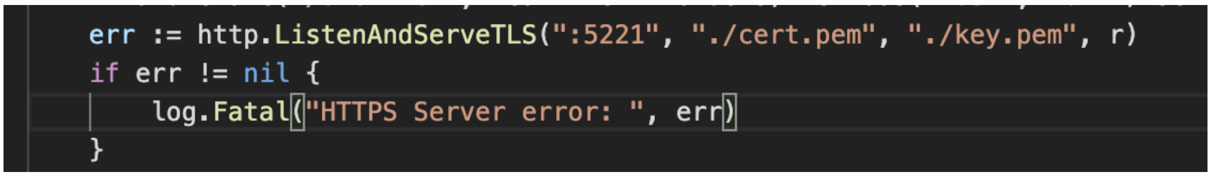
2.5 HTTP/TLS

TLS/SSL is a cryptographic protocol that allows encryption over insecure communication channels. Usage of TSL/SSL is most commonly used to provide secure HTTP communication, also known as HTTPS. The protocol ensures that the communication channel has the following properties:

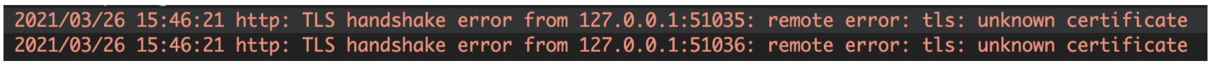
- Privacy
- Authentication
- Data integrity

For development and testing purposes, self-generated certificates and keys through OpenSSL are used for this web application.

Cert.pem file is the SSL certificate and key.pem is the private key for the server.

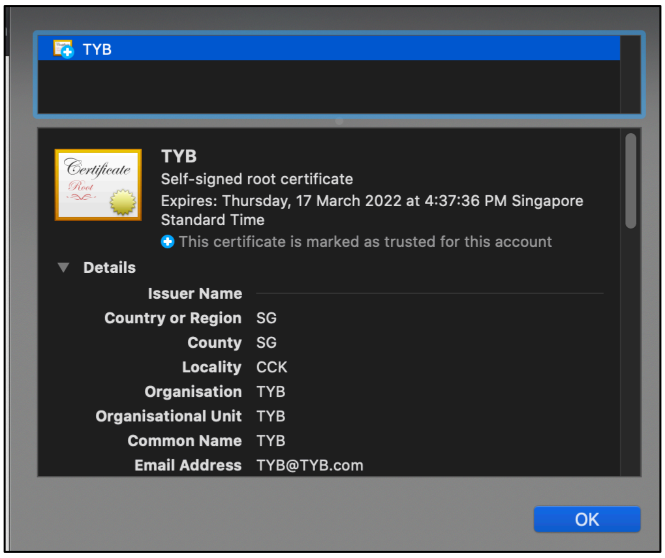


```
    err := http.ListenAndServeTLS(":5221", "./cert.pem", "./key.pem", r)
    if err != nil {
        log.Fatal("HTTPS Server error: ", err)
    }
```



```
2021/03/26 15:46:21 http: TLS handshake error from 127.0.0.1:51035: remote error: tls: unknown certificate
2021/03/26 15:46:21 http: TLS handshake error from 127.0.0.1:51036: remote error: tls: unknown certificate
```

Upon launch of the server, requests sent to the server typing URL as https://localhost:5221 would show a TLS handshake error due to untrusted connection but the fact this error occurs indicates that SSL/TLS is working.



TYB

TYB
Self-signed root certificate
Expires: Thursday, 17 March 2022 at 4:37:36 PM Singapore Standard Time
This certificate is marked as trusted for this account

Details

Issuer Name	
Country or Region	SG
County	SG
Locality	CCK
Organisation	TYB
Organisational Unit	TYB
Common Name	TYB
Email Address	TYB@TYB.com

Self-signed certificate is invalid for browser, changing browser setting to trust this certificate will allow the browser to run the web application.

2.6 Error handling & Logging

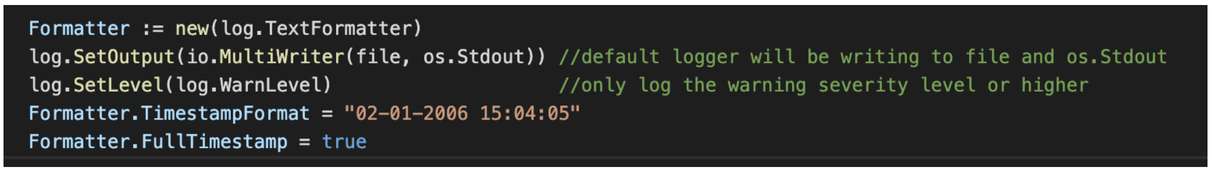
Error handling and logging were incorporated as part of this application's protection.

The log entries are used to log events of severity warning and above (in the case of application, this would be levels "Warning", "Error", "Panic" and "Fatal". The log entries were created to be generic in nature yet provide useful information to administrator without leaking sensitive information.

Some of the important event data that were logged in this assignment are:

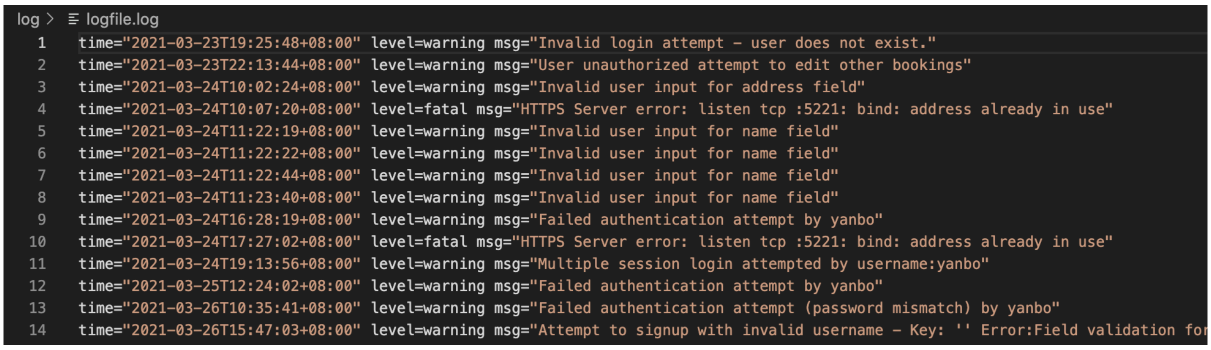
- Input validation failures
- Authentication attempt failures
- Access control failures (eg. Non admin accessing admin content)
- Tampering events (user trying to change order details without authorization)

For this assignment, a third party logger ("github.com/sirupsen/logrus") was used to log entries to log/logfile.log.



```go
Formatter := new(log.TextFormatter)
log.SetOutput(io.MultiWriter(file, os.Stdout)) //default logger will be writing to file and os.Stdout
log.SetLevel(log.WarnLevel)                    //only log the warning severity level or higher
Formatter.TimestampFormat = "02-01-2006 15:04:05"
Formatter.FullTimestamp = true
```

Setting of logger parameters in package main as part of function init().



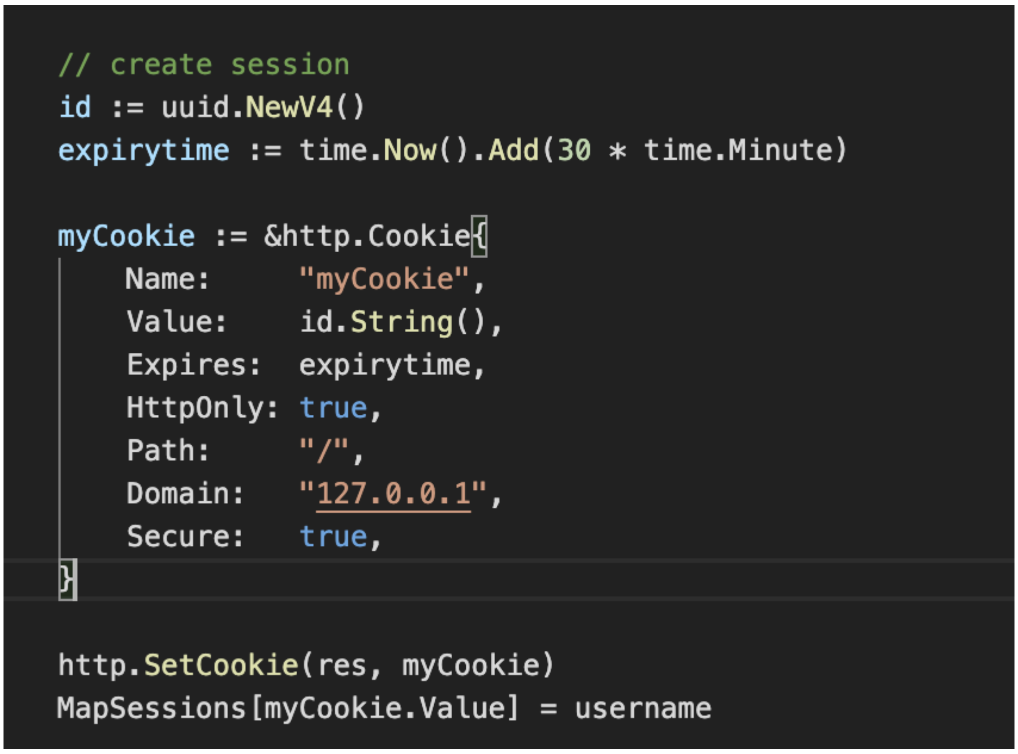
```
time="2021-03-23T19:25:48+08:00" level=warning msg="Invalid login attempt - user does not exist."
time="2021-03-23T22:13:44+08:00" level=warning msg="User unauthorized attempt to edit other bookings"
time="2021-03-24T10:02:24+08:00" level=warning msg="Invalid user input for address field"
time="2021-03-24T10:07:20+08:00" level=fatal msg="HTTPS Server error: listen tcp :5221: bind: address already in use"
time="2021-03-24T11:22:19+08:00" level=warning msg="Invalid user input for name field"
time="2021-03-24T11:22:22+08:00" level=warning msg="Invalid user input for name field"
time="2021-03-24T11:22:44+08:00" level=warning msg="Invalid user input for name field"
time="2021-03-24T11:23:40+08:00" level=warning msg="Invalid user input for name field"
time="2021-03-24T16:28:19+08:00" level=warning msg="Failed authentication attempt by yanbo"
time="2021-03-24T17:27:02+08:00" level=fatal msg="HTTPS Server error: listen tcp :5221: bind: address already in use"
time="2021-03-24T19:13:56+08:00" level=warning msg="Multiple session login attempted by username:yanbo"
time="2021-03-25T12:24:02+08:00" level=warning msg="Failed authentication attempt by yanbo"
time="2021-03-26T10:35:41+08:00" level=warning msg="Failed authentication attempt (password mismatch) by yanbo"
time="2021-03-26T15:47:03+08:00" level=warning msg="Attempt to signup with invalid username - Key: '' Error:Field validation for
```

Snapshot of logged events in logfile.log

2.7. Session Management

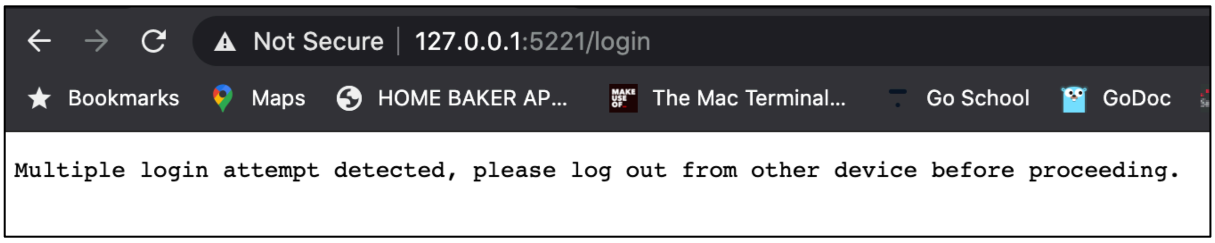
Home baker application's HTML session is set by the server's management controls. UUID V4 tokens are used to generate the session identifier which are sufficiently strong to prevent session brute forcing. Cookie parameters are also set with Domain, Path, Expires, HTTP only and Secure.

A new session is also generated upon sign-in and expire parameter has been set to 30 mins for this application. Also, concurrent login is disallowed and repeated user login attempt will be disallowed. This is done by maintaining a list of logged in users through map sessions stored in server which contains list of logged users against their current cookie token. HTTPS is also used throughout this web application.



```go
// create session
id := uuid.NewV4()
expirytime := time.Now().Add(30 * time.Minute)

myCookie := &http.Cookie{
    Name:     "myCookie",
    Value:    id.String(),
    Expires:  expirytime,
    HttpOnly: true,
    Path:     "/",
    Domain:   "127.0.0.1",
    Secure:   true,
}

http.SetCookie(res, myCookie)
MapSessions[myCookie.Value] = username
```

Snapshot of cookie parameters set by server.



```
Multiple login attempt detected, please log out from other device before proceeding.
```

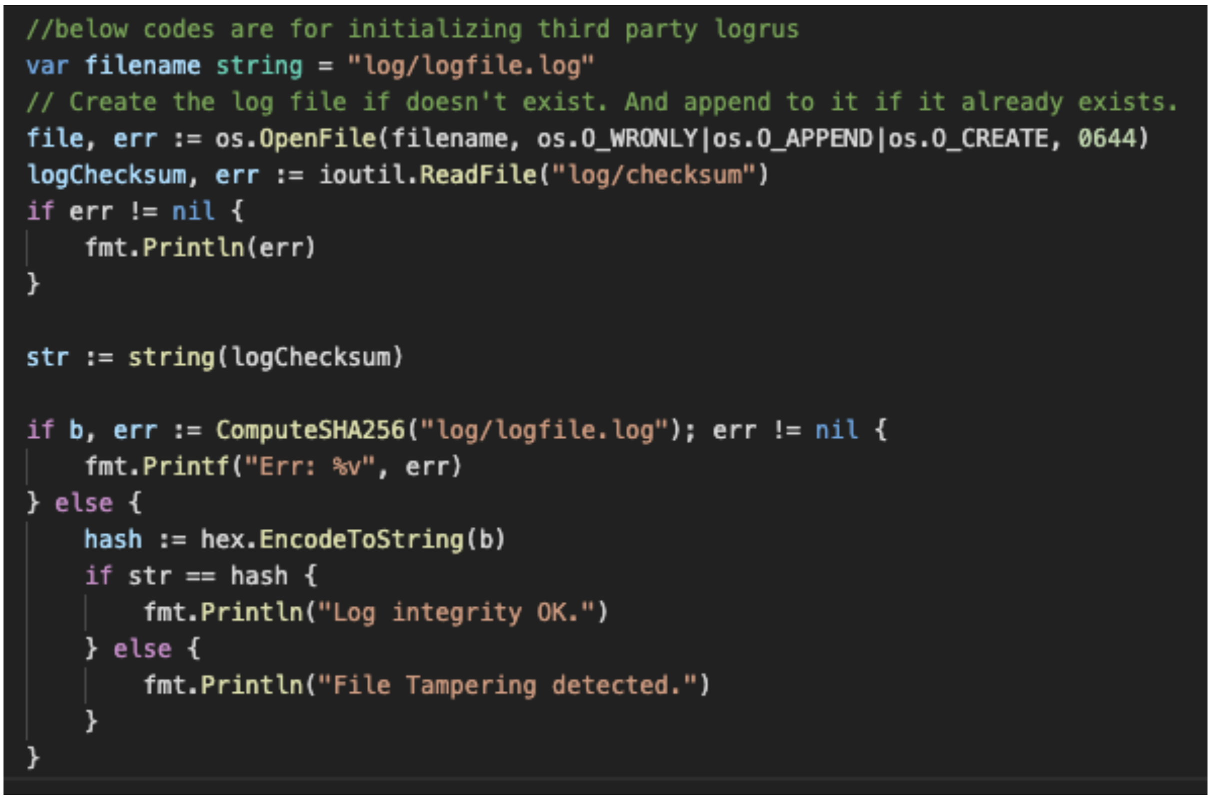
Example of user being rejected from multiple logins

Finally, logout link is provided from all pages of the web application and when logged out, will fully terminate the associated session by deleting the cookie from the client. User session on server similarly is deleted from map sessions.

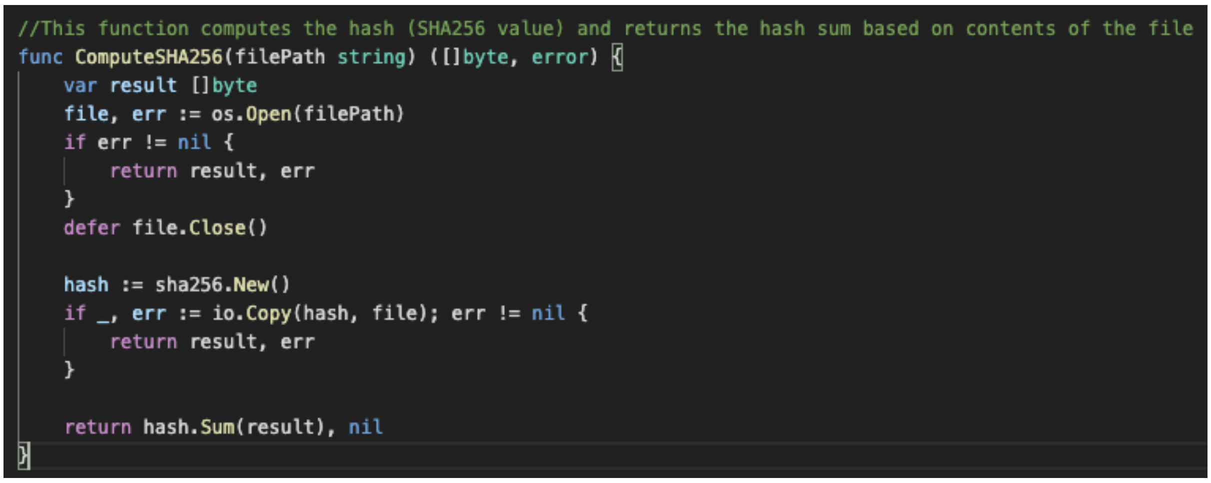
2.8 Cryptographic Practices

Hashing of log-file

This assignment aims to showcase the use of hashing (SHA256) to hash the existing logfile into a checksum. This checksum has been provided in the assignment as file “checksum”. At the start of the program initialize, the logfile’s hash will be computed and compared against the checksum already provided in the assignment folder. If the hash value and checksum values are different then it is indicative that the logfile has been tampered.



```
//below codes are for initializing third party logrus
var filename string = "log/logfile.log"
// Create the log file if doesn't exist. And append to it if it already exists.
file, err := os.OpenFile(filename, os.O_WRONLY|os.O_APPEND|os.O_CREATE, 0644)
logChecksum, err := ioutil.ReadFile("log/checksum")
if err != nil {
    fmt.Println(err)
}

str := string(logChecksum)

if b, err := ComputeSHA256("log/logfile.log"); err != nil {
    fmt.Printf("Err: %v", err)
} else {
    hash := hex.EncodeToString(b)
    if str == hash {
        fmt.Println("Log integrity OK.")
    } else {
        fmt.Println("File Tampering detected.")
    }
}
```



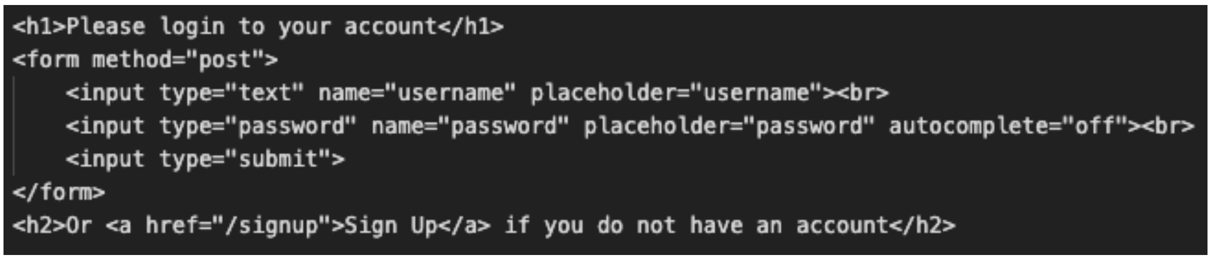
```
//This function computes the hash (SHA256 value) and returns the hash sum based on contents of the file
func ComputeSHA256(filePath string) ([]byte, error) {
    var result []byte
    file, err := os.Open(filePath)
    if err != nil {
        return result, err
    }
    defer file.Close()

    hash := sha256.New()
    if _, err := io.Copy(hash, file); err != nil {
        return result, err
    }

    return hash.Sum(result), nil
}
```

2.9 Communicating Authentication Data and password storage

A list of pre-generated user credentials have been created and stored in users.json that will be unmarshalled into the server during server initialize. For demonstration purpose, the following credentials can be used to navigate the web application.

1. Username : “admin”, password : “password”. This account is the only account with admin access.
2. Username “yanbo”, password: “password”. This is a non-admin account with one order pre-loaded into the application, and thus can be used to test the view/edit order function.

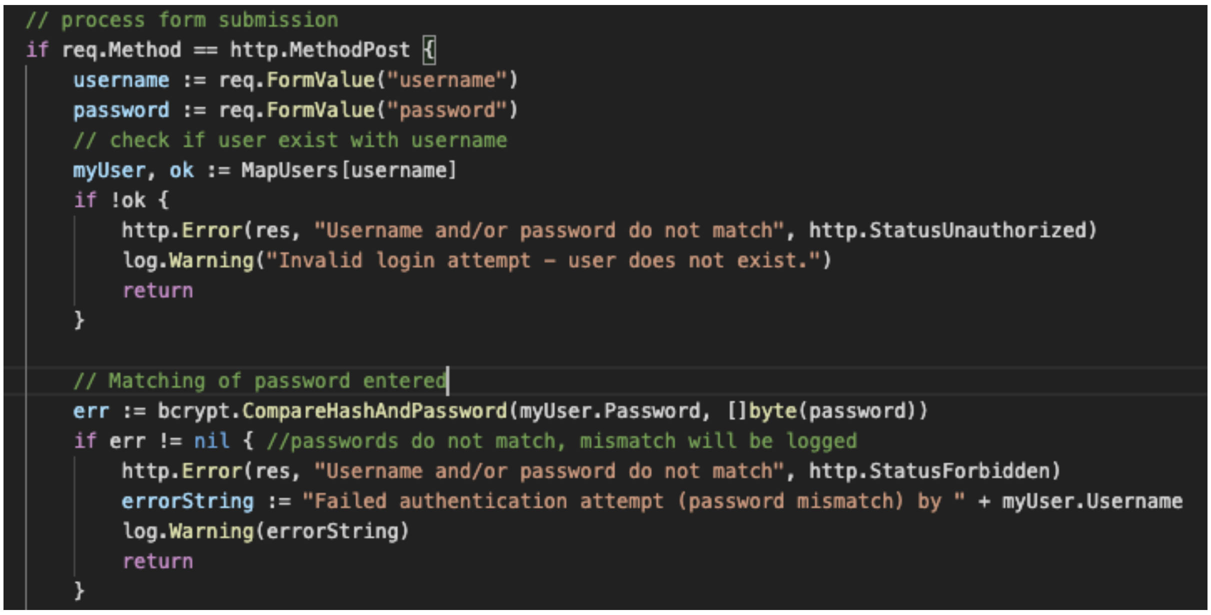
For the web application, password entry is obscured on the user’s screen and the “remember me” functionality is disabled.



```
<h1>Please login to your account</h1>
<form method="post">
    <input type="text" name="username" placeholder="username"><br>
    <input type="password" name="password" placeholder="password" autocomplete="off"><br>
    <input type="submit">
</form>
<h2>Or <a href="/signup">Sign Up</a> if you do not have an account</h2>
```

Authentication credentials are sent through HTTPS connection and data is sent to the server using the HTTP POST method to prevent credentials leakage to HTTP server logs.

For the handling of authentication errors, the application does not disclose which part of the authentication data was incorrect, a generic “Invalid username and/or password” is returned if there is an error.



```
// process form submission
if req.Method == http.MethodPost {
    username := req.FormValue("username")
    password := req.FormValue("password")
    // check if user exist with username
    myUser, ok := MapUsers[username]
    if !ok {
        http.Error(res, "Username and/or password do not match", http.StatusUnauthorized)
        log.Warning("Invalid login attempt - user does not exist.")
        return
    }

    // Matching of password entered
    err := bcrypt.CompareHashAndPassword(myUser.Password, []byte(password))
    if err != nil { //passwords do not match, mismatch will be logged
        http.Error(res, "Username and/or password do not match", http.StatusForbidden)
        errorString := "Failed authentication attempt (password mismatch) by " + myUser.Username
        log.Warning(errorString)
        return
    }
```

In the case of password storage, hashing algorithm bcrypt is used for this assignment’s application. Bcrypt provides a simple and secure way to compare a plaintext password with an already hashed password.

3. Idiomatic Go Techniques

Idiomatic go techniques were applied while writing the source code for this application. Some of the pointers taken note of were:

1. Formatting
2. Usage of line comments (//)
3. Every package having a package comment. And multi-file package only has package comment in one file.
4. Doc comments set up as complete sentences. Every exported name has a doc comment.
5. Package names are given in lower case. Identifiers use mixedCaps or MixedCaps for names.
6. Multiple return values are used in some functions and in most functions where possible error handling is incorporated for defensive coding.

The above are only a fraction of considerations used with the source code for this assignment. As much as possible proper idiomatic go techniques and commenting were applied to the codes for this assignment.

4. Appendix (Go Documentation)

Please see appendix folder attached within submission folder for the pdf printouts of the application's Go Documentation.

Also, as mentioned in section 2.9, please use below credentials to test the web application and its features.

1. Username : "admin", password : "password". This account is the only account with admin access.
2. Username "yanbo", password: "password". This is a non-admin account with one order pre-loaded into the application, and thus can be used to test the view/edit order function.