Go In Action II Assignment

Author: Yeoh Aik Huang Alvin / contact@alvinyeoh.com

1. Comments & Instructions

- The app is configured to be accessible at `https://localhost:5221` by default. The hostname and port can be changed in the web package's config file.
- The app is configured by default to connect to a live MySQL database, loads frontend dependencies (js/css) from CDNs and also fetches data from a web api so an internet connection is required. If desired, the default database connection configurations can be modified to connect to a different database in the ***.env*** file in the root directory.
- The app initializes the system with data fetched from the database by default.
- You can reset the database and re-seed test data by setting `resetAndSeedDB` to `true` in the clinic package's ***config.go*** file and running the app. Seeding of test data is performed in the clinic package's ***helper.go*** file via goroutines. You can also use this to quickly set up a custom database.
- If `resetDB` is set to `true` and `resetAndSeedDB` to `false`, a clean system with no data will be initialized.
- If both `resetDB` & `resetAndSeedDB` are set to false, the app will initialize with the existing data in the database. This is the default setting when the assignment is submitted.
- A staff / admin is a Patient whose Id is inside the Admins slice. Admins slice is loaded via `seedAdmins` function which in turns gets it from the ADMIN_IDS field in the ***.env*** file.
- The test accounts that were seeded automatically are listed below. As mentioned before, they can be re-seeded by setting `resetAndSeedDB` to `true` and running the app. The default password of seeded test accounts are "**12345678**". This is also declared in the ***.env*** file.

Username (NRIC)	Is Admin
S1111111B	No
S2222222C	No
S3333333D	No
S4444444D	No
S5555555E	No
S6666666F	No
S7777777G	No
S0000000A	Yes
S1234567A	Yes
S7654321A	Yes

S8888888A	Yes
S9999999A	Yes

---

2. Secure software development techniques applied

User inputs are all processed by the handler functions inside the ***web*** package.

GET vs POST Requests

All requests that change data on the server (create/update/delete) are sent via POST requests to prevent cross-site request forgery (CSRF).

E.g. If an endpoint like https://localhost:5221/admin/appointment/edit?apptId=1008&action=cancel accepts a GET request to delete an appointment of ID 1008, an authenticated user can be tricked into executing it either directly accessing it accidentally or unknowingly via an iframe embed or javascript request.

Cookie Settings

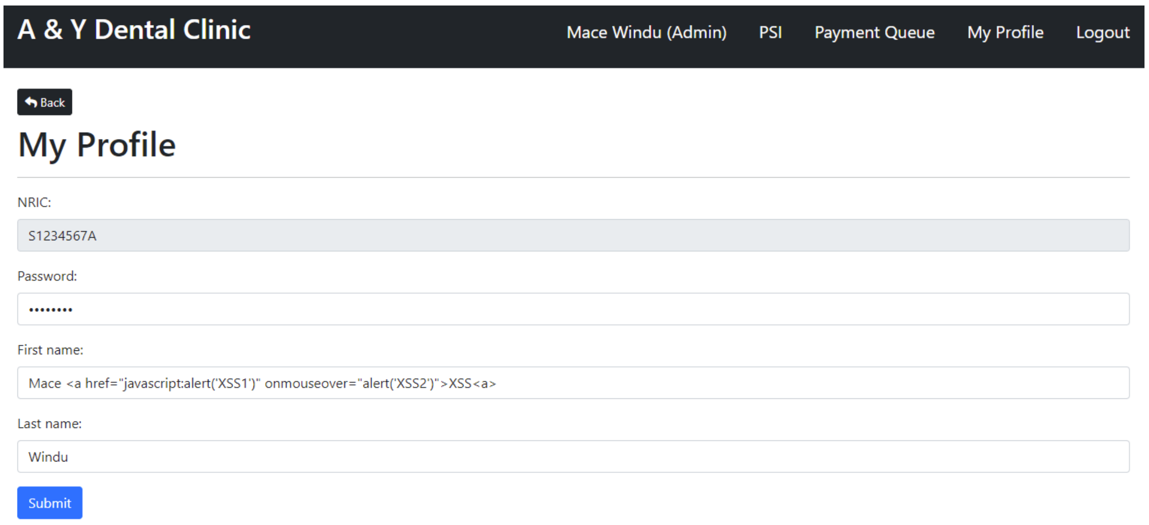
Cookies are created with the **`SameSite`** attribute set to 3 (Strict) and **`Secure`**. SameSite value being strict also helps prevent CSRF as it prevents cross-site requests. However, as cookies are stored client side, the enforcement depends on browser compatibility. The **`HttpOnly`** attribute restricts the cookie from being accessed by client-side APIs, such as JavaScript.

Input Validation & Sanitization

Only user registration and profile update consists of user inputs that are used as output. For the purpose of the assignment, external libraries are used sparingly.

Register / Profile Update / Login

- The **`bluemonday`** library is used to sanitize username (NRIC), firstname and lastname by stripping away tags to protect the application (app) from XSS. The tags shown below under first name will be stripped away and left with spaces which will be trimmed away via further sanitization resulting in the first name being just “Mace XSS”.


A & Y Dental Clinic

Mace Windu (Admin) PSI Payment Queue My Profile Logout

Back

My Profile

NRIC:

S1234567A

Password:

••••••••

First name:

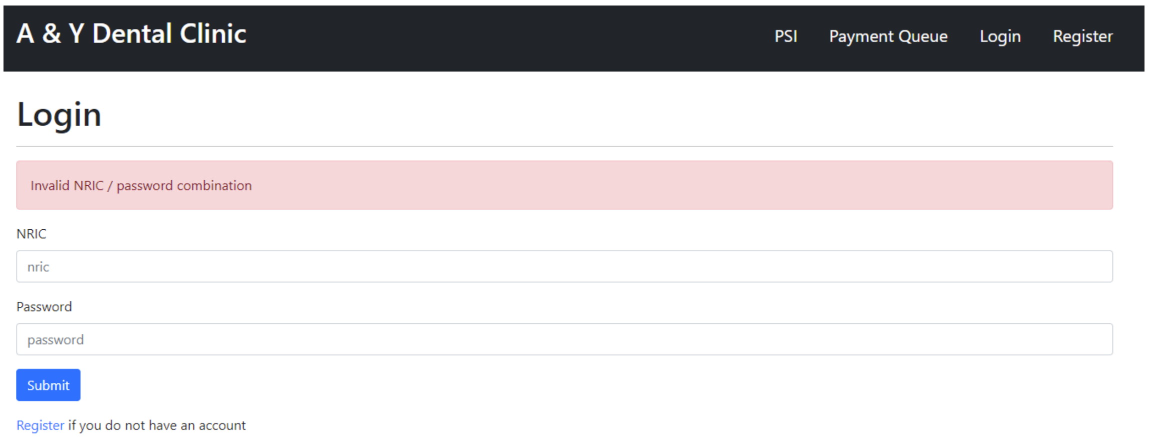
Mace <a href="javascript:alert('XSS1')" onmouseover="alert('XSS2')">XSS<a>

Last name:

Windu

Submit

- After sanitization, I perform trimming of trailing and leading space. Username is transformed to all uppercase to standardize the attribute.
- The validity of user inputs are then checked for the following and returns the appropriate error messages if any fails.
  - Empty string (except for password e.g. space is a character)
  - Password length meeting the MinPasswordLength policy (default: 8)
  - NRIC validity
    - Currently, it only checks for length of 9
    - To enable the actual NRIC validity check which compares the NRIC against a checksum, set the `strictNRIC` attribute in ***`clinic/config.go`*** to true
  - Existing user (NRIC)
- When performing login, if either an NRIC can't be found or Password is incorrect, a generic error message "`invalid NRIC / password combination`" to not alert the user which is the wrong input to help mitigate brute force attacks.


A & Y Dental Clinic

PSI Payment Queue Login Register

Login

Invalid NRIC / password combination

NRIC

nric

Password

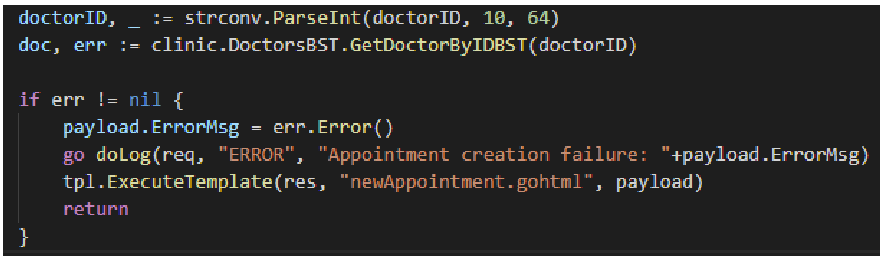
password

Submit

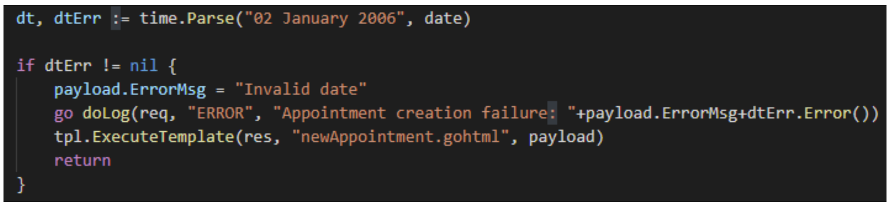
Register if you do not have an account

String parsing

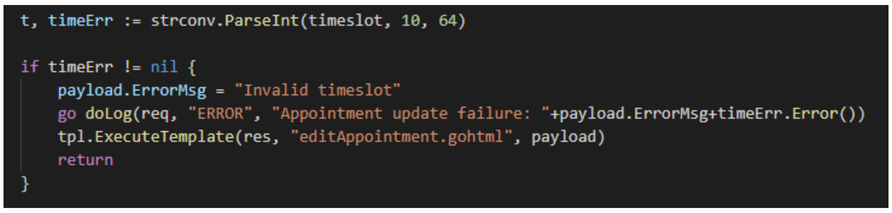
- Most of the app's other handler functions accept inputs which are parsed from string to their appropriate types (e.g. int64 / time) and then checked for errors via Go's strconv functions.
  - E.g. when making a new appointment request, `doctor id`, `date` and `time` are user inputs that are parsed and checked for errors.



```go
doctorID, _ := strconv.ParseInt(doctorID, 10, 64)
doc, err := clinic.DoctorsBST.GetDoctorByIDBST(doctorID)

if err != nil {
    payload.ErrorMsg = err.Error()
    go doLog(req, "ERROR", "Appointment creation failure: "+payload.ErrorMsg)
    tpl.ExecuteTemplate(res, "newAppointment.gohtml", payload)
    return
}
```

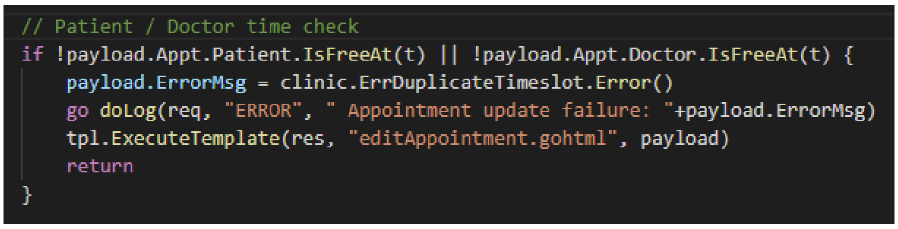


```go
dt, dtErr := time.Parse("02 January 2006", date)

if dtErr != nil {
    payload.ErrorMsg = "Invalid date"
    go doLog(req, "ERROR", "Appointment creation failure: "+payload.ErrorMsg+dtErr.Error())
    tpl.ExecuteTemplate(res, "newAppointment.gohtml", payload)
    return
}
```



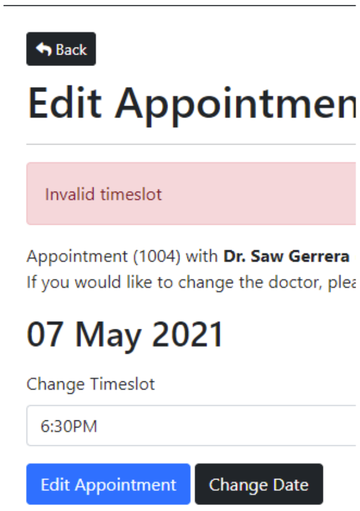
```go
t, timeErr := strconv.ParseInt(timeslot, 10, 64)

if timeErr != nil {
    payload.ErrorMsg = "Invalid timeslot"
    go doLog(req, "ERROR", "Appointment update failure: "+payload.ErrorMsg+timeErr.Error())
    tpl.ExecuteTemplate(res, "editAppointment.gohtml", payload)
    return
}
```

- After type conversion checks have passed successfully, app specific checks are also performed such as checking if the doctor id really exists (shown above from `GetDoctorByIDBST`) and if the date given is an actual date a doctor and patient is actually able to make an appointment for (shown below) as it's trivial to forge the payload of a POST request.



```go
// Patient / Doctor time check
if !payload.Appt.Patient.IsFreeAt(t) || !payload.Appt.Doctor.IsFreeAt(t) {
    payload.ErrorMsg = clinic.ErrDuplicateTimeslot.Error()
    go doLog(req, "ERROR", " Appointment update failure: "+payload.ErrorMsg)
    tpl.ExecuteTemplate(res, "editAppointment.gohtml", payload)
    return
}
```

- As seen below, when I modify the time value client side and submit, I'll get the "`Invalid timeslot`" error.



Back

Edit Appointmen

Invalid timeslot

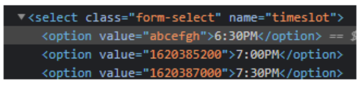
Appointment (1004) with **Dr. Saw Gerrera**
If you would like to change the doctor, plea

07 May 2021

Change Timeslot

6:30PM

Edit Appointment
Change Date



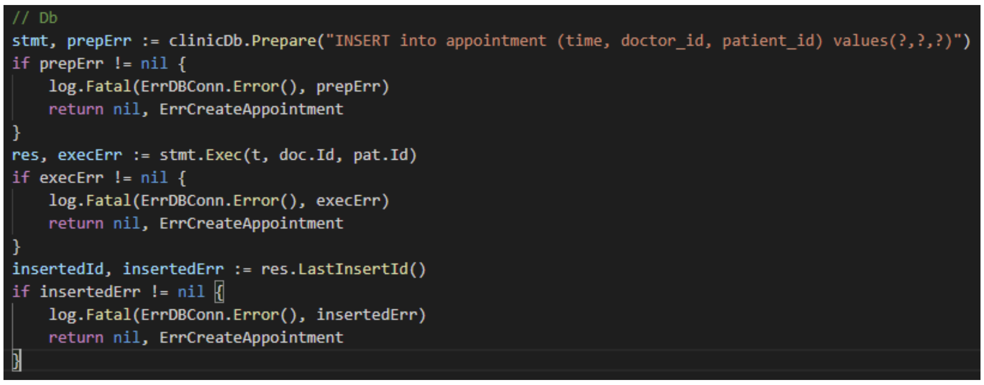
```html
▼<select class="form-select" name="timeslot">
    <option value="abcefgh">6:30PM</option> == $
    <option value="1620385200">7:00PM</option>
    <option value="1620387000">7:30PM</option>
```

Context Awareness

- Go's template engine is used for all pages of the app so it's another layer of protection against XSS via input users that are used as output in the HTML.
    - When I set my firstname as "Mace<script>alert('hi')</script>" without sanitization, it gets output as "Mace&lt;script&gt;alert('hi')&lt;/script&gt;" by the template engine instead and the script is not executed.

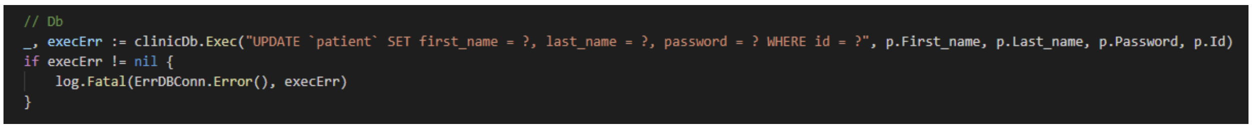
SQL Injection

- Sql statements are mostly found in the ***`clinic`*** package. Any sql statement that contains dynamic parameters uses prepared statements.
- E.g. Appointment creation



```go
// Db
stmt, prepErr := clinicDb.Prepare("INSERT into appointment (time, doctor_id, patient_id) values(?,?,?)")
if prepErr != nil {
    log.Fatal(ErrDBConn.Error(), prepErr)
    return nil, ErrCreateAppointment
}
res, execErr := stmt.Exec(t, doc.Id, pat.Id)
if execErr != nil {
    log.Fatal(ErrDBConn.Error(), execErr)
    return nil, ErrCreateAppointment
}
insertedId, insertedErr := res.LastInsertId()
if insertedErr != nil {
    log.Fatal(ErrDBConn.Error(), insertedErr)
    return nil, ErrCreateAppointment
}
```

- E.g. User profile update
    - password contains the binary data of the hashed password and not the original password in plain text



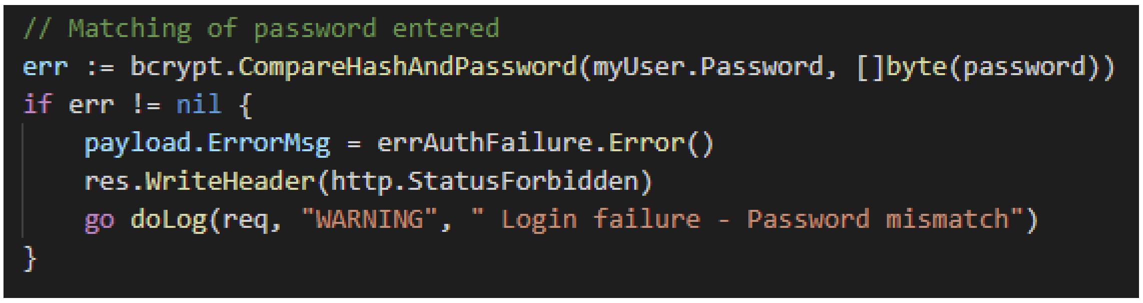
```go
// Db
_, execErr := clinicDb.Exec("UPDATE `patient` SET first_name = ?, last_name = ?, password = ? WHERE id = ?", p.First_name, p.Last_name, p.Password, p.Id)
if execErr != nil {
    log.Fatal(ErrDBConn.Error(), execErr)
}
```

Communication Security

The app uses HTTPS to encrypt and protect communications. However, it uses a self signed key for the purpose of the assignment as opposed to one from a Certificate Authority so we'll be presented with a "the certificate cannot be trusted error message". The certs are located in the ***`.cert`*** hidden directory which is also a directory that has been excluded from being tracked and committed to git repository by editing ***`.gitignore`***.

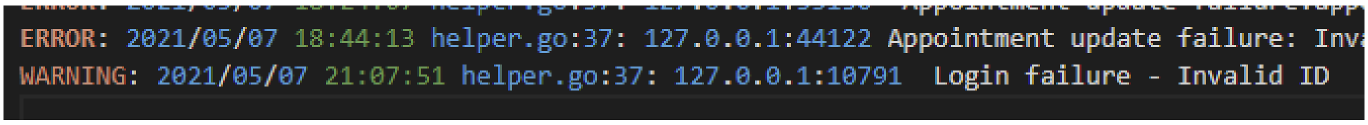
Error Handling & Logging

During any checks, if any error is encountered, the user is directed to either another page or the same page with the appropriate error message as well as having the action logged. The logs contain a more detailed description of the error as compared to the error message presented to the user to assist in debugging or identifying potential malicious actions. No sensitive information like passwords like displayed to a user or stored in logs.



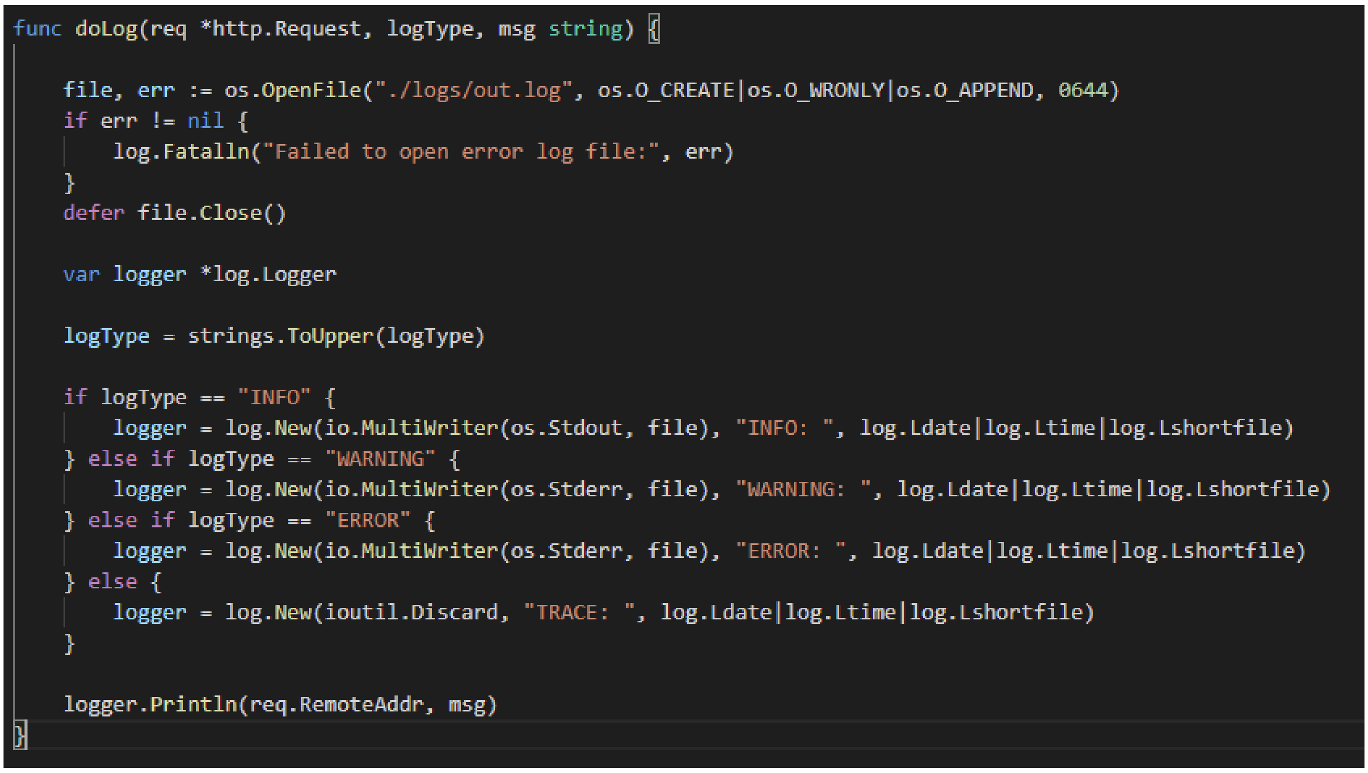
```go
// Matching of password entered
err := bcrypt.CompareHashAndPassword(myUser.Password, []byte(password))
if err != nil {
    payload.ErrorMsg = errAuthFailure.Error()
    res.WriteHeader(http.StatusForbidden)
    go doLog(req, "WARNING", " Login failure - Password mismatch")
}
```

For the example above, if the password entered is different from what the app has in record, the user will be presented with the error message "**`invalid NRIC / password combination`**" and it'll be logged to the server log files in an entry shown below.



```
[illegible]
ERROR: 2021/05/07 18:44:13 helper.go:37: 127.0.0.1:44122 Appointment update failure: Inva
WARNING: 2021/05/07 21:07:51 helper.go:37: 127.0.0.1:10791  Login failure - Invalid ID
```

Logging in the ***web*** package is performed through the custom **`doLog`** function in **`helper.go`**.



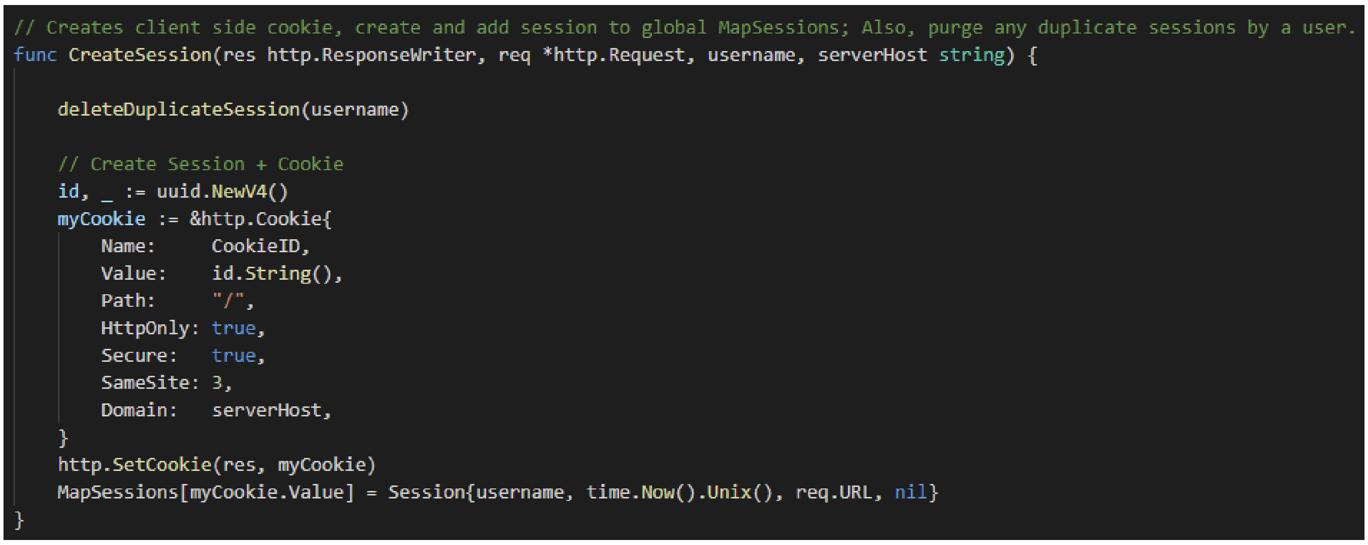
```go
func doLog(req *http.Request, logType, msg string) {

    file, err := os.OpenFile("./logs/out.log", os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0644)
    if err != nil {
        log.Fatalln("Failed to open error log file:", err)
    }
    defer file.Close()

    var logger *log.Logger

    logType = strings.ToUpper(logType)

    if logType == "INFO" {
        logger = log.New(io.MultiWriter(os.Stdout, file), "INFO: ", log.Ldate|log.Ltime|log.Lshortfile)
    } else if logType == "WARNING" {
        logger = log.New(io.MultiWriter(os.Stderr, file), "WARNING: ", log.Ldate|log.Ltime|log.Lshortfile)
    } else if logType == "ERROR" {
        logger = log.New(io.MultiWriter(os.Stderr, file), "ERROR: ", log.Ldate|log.Ltime|log.Lshortfile)
    } else {
        logger = log.New(ioutil.Discard, "TRACE: ", log.Ldate|log.Ltime|log.Lshortfile)
    }

    logger.Println(req.RemoteAddr, msg)
}
```

Session Management

Session is handled by the ***`session`*** package which creates a cookie in the **`CreateSession`** function. The attributes HttpOnly, Secure and SameSite have been covered prior so I'll skip those.

UUID

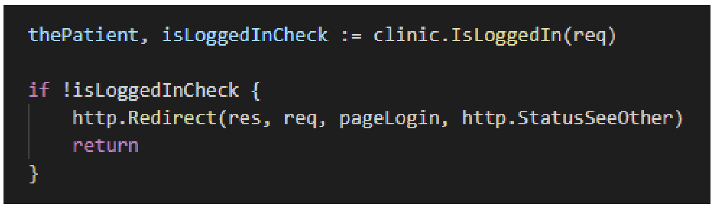
The value that's stored is a **`universally unique identifier`** (UUID) string for the user that's generated from the **`UUID`** package. The UUID value for the user will then be stored server side in a map with the UUID as the key to a Session item containing the user's username (NRIC), the current time and request URL.



```go
// Creates client side cookie, create and add session to global MapSessions; Also, purge any duplicate sessions by a user.
func CreateSession(res http.ResponseWriter, req *http.Request, username, serverHost string) {

    deleteDuplicateSession(username)

    // Create Session + Cookie
    id, _ := uuid.NewV4()
    myCookie := &http.Cookie{
        Name:     CookieID,
        Value:    id.String(),
        Path:     "/",
        HttpOnly: true,
        Secure:   true,
        SameSite: 3,
        Domain:   serverHost,
    }
    http.SetCookie(res, myCookie)
    MapSessions[myCookie.Value] = Session{username, time.Now().Unix(), req.URL, nil}
}
```

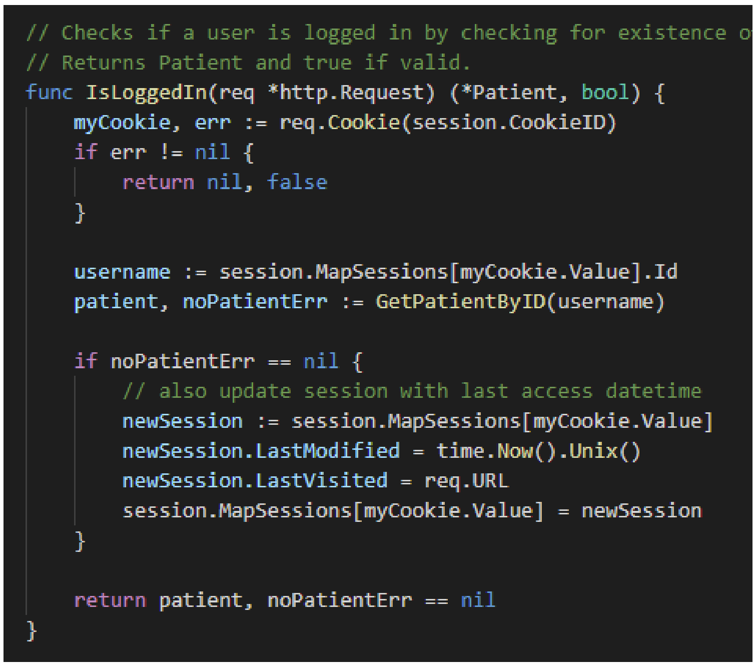
Authentication

The app has 3 different access levels. Public with no requirements, members' section where a user has to be logged in and staff section where the user that is logged in is identified as an admin (staff). All non-public pages will first check if the user is logged in via the IsLoggedIn function which returns both a Patient item and error. If the error isn't nil, the user is redirected back to the public login page.



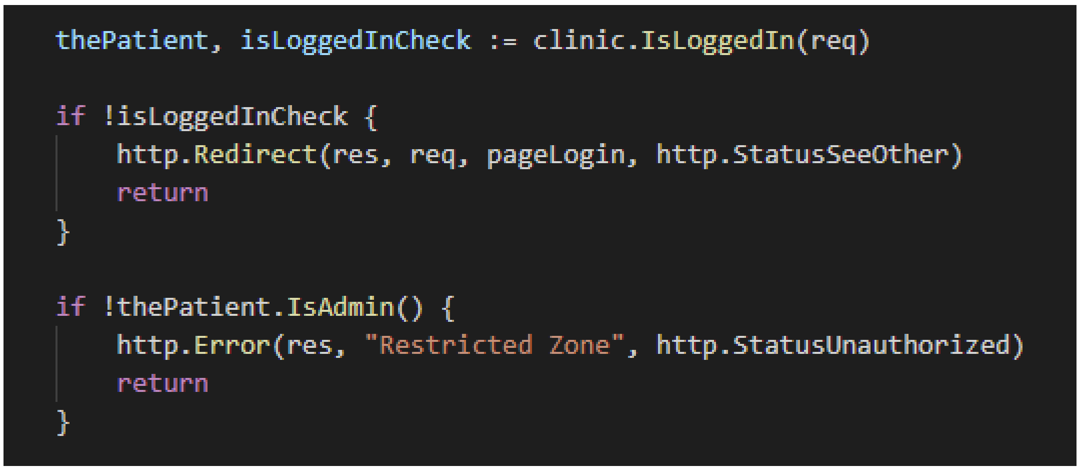
```go
thePatient, isLoggedInCheck := clinic.IsLoggedIn(req)

if !isLoggedInCheck {
    http.Redirect(res, req, pageLogin, http.StatusSeeOther)
    return
}
```

The IsLoggedIn function does this by checking if the client has a cookie, retrieves the value of it and tries to map it to our MapSessions global variable to retrieve the logged in user's identity. It also updates the session item with the user's last request time and page.



```go
// Checks if a user is logged in by checking for existence o
// Returns Patient and true if valid.
func IsLoggedIn(req *http.Request) (*Patient, bool) {
    myCookie, err := req.Cookie(session.CookieID)
    if err != nil {
        return nil, false
    }

    username := session.MapSessions[myCookie.Value].Id
    patient, noPatientErr := GetPatientByID(username)

    if noPatientErr == nil {
        // also update session with last access datetime
        newSession := session.MapSessions[myCookie.Value]
        newSession.LastModified = time.Now().Unix()
        newSession.LastVisited = req.URL
        session.MapSessions[myCookie.Value] = newSession
    }

    return patient, noPatientErr == nil
}
```

Similarly, when a user attempts to visit a staff restricted page, IsLoggedIn is first used to check that the user is a valid logged in then proceeds to check if he / she is an admin via the `IsAdmin` method of Patient and returning 401 error.

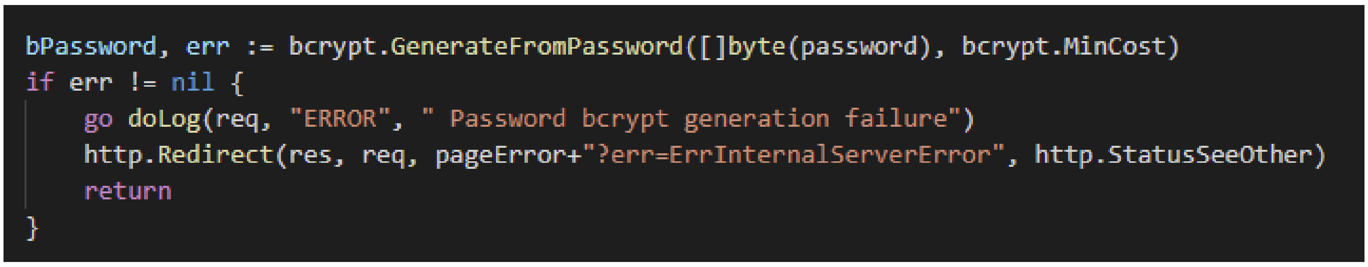


```go
thePatient, isLoggedInCheck := clinic.IsLoggedIn(req)

if !isLoggedInCheck {
    http.Redirect(res, req, pageLogin, http.StatusSeeOther)
    return
}

if !thePatient.IsAdmin() {
    http.Error(res, "Restricted Zone", http.StatusUnauthorized)
    return
}
```

Password Hashing

The life cycle of a password when registering or updating of profile in the app is as follows:

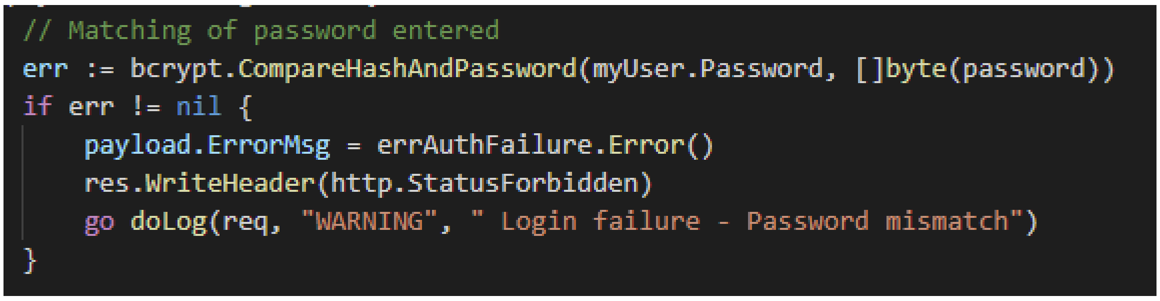
1. Sent encrypted over HTTPS from client to server.
2. Decrypted on the server side to original text.
3. Hashing performed via **bcrypt** library’s GenerateFromPassword function (show below).
4. Hash value of the original password is stored in the user's corresponding Patient item’s password field and also inserted / updated to the database as binary strings. The server does not know what is the original password in plain text from this point onwards nor stores that information anywhere.



```go
bPassword, err := bcrypt.GenerateFromPassword([]byte(password), bcrypt.MinCost)
if err != nil {
    go doLog(req, "ERROR", " Password bcrypt generation failure")
    http.Redirect(res, req, pageError+"?err=ErrInternalServerError", http.StatusSeeOther)
    return
}
```

Login Check

When a user performs login, the password provided is transferred from client to server via the same mechanism as before and a hash of the provided password is once again generated. This time the hashed value of the provided password is compared against the previously stored hashed password belonging to the user and if it matches, the user is logged in.



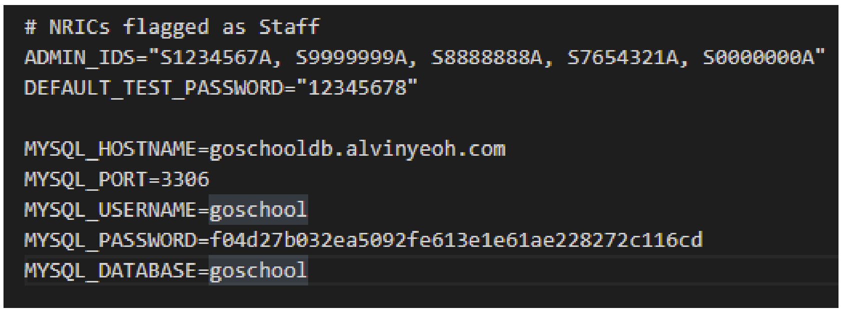
```go
// Matching of password entered
err := bcrypt.CompareHashAndPassword(myUser.Password, []byte(password))
if err != nil {
    payload.ErrorMsg = errAuthFailure.Error()
    res.WriteHeader(http.StatusForbidden)
    go doLog(req, "WARNING", " Login failure - Password mismatch")
}
```

Any password mismatch failures are also logged.

Sensitive App Data

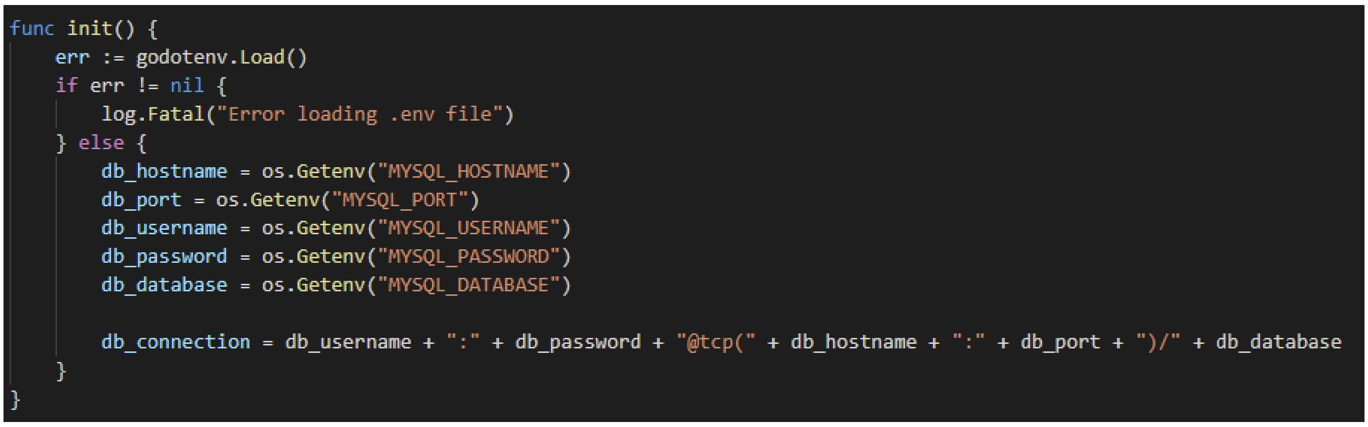
Secrets are stored in a ***.env*** file which is parsed and handled by the [GoDotEnv] library. It allows us to separate sensitive information from the source code. The file itself is hidden by default and also excluded from being committed to repositories by configuring ***.gitignore***.

Information inside my app's .env file includes the csv list of ids recognized as staff, default password for test accounts and database connection information.



```
# NRICs flagged as Staff
ADMIN_IDS="S1234567A, S9999999A, S8888888A, S7654321A, S0000000A"
DEFAULT_TEST_PASSWORD="12345678"

MYSQL_HOSTNAME=goschooldb.alvinyeoh.com
MYSQL_PORT=3306
MYSQL_USERNAME=goschool
MYSQL_PASSWORD=f04d27b032ea5092fe613e1e61ae228272c116cd
MYSQL_DATABASE=goschool
```

They can be accessed via the **os.Getenv** function as seen in the clinic package's config.go file where I get the database connection information.



```go
func init() {
    err := godotenv.Load()
    if err != nil {
        log.Fatal("Error loading .env file")
    } else {
        db_hostname = os.Getenv("MYSQL_HOSTNAME")
        db_port = os.Getenv("MYSQL_PORT")
        db_username = os.Getenv("MYSQL_USERNAME")
        db_password = os.Getenv("MYSQL_PASSWORD")
        db_database = os.Getenv("MYSQL_DATABASE")

        db_connection = db_username + ":" + db_password + "@tcp(" + db_hostname + ":" + db_port + ")/" + db_database
    }
}
```

---

3. Idiomatic Go techniques applied

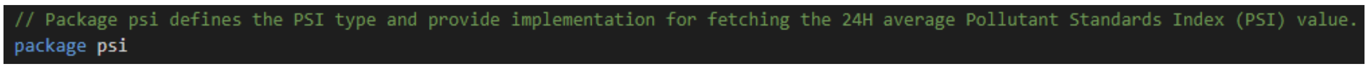
Formatting

Formatting of the app's source code was handled automatically by Visual Studio Code's [gopls] extension by Google instead of [gofmt] or [gofumpt].

Comments

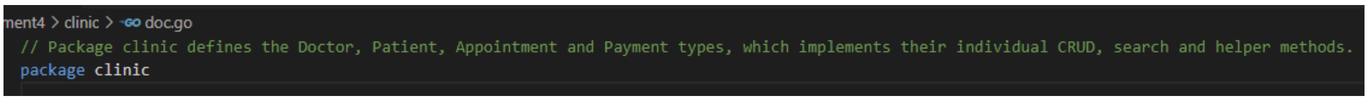
All packages and exported names have comments preceding them. Package names are all lower case single words and are of the same name as their containing directory. No underscore is used anywhere in the app for naming.

For packages with just 1 file like ***psi*** package, the package comment is before the package declaration in the sole file.



```go
// Package psi defines the PSI type and provide implementation for fetching the 24H average Pollutant Standards Index (PSI) value.
package psi
```

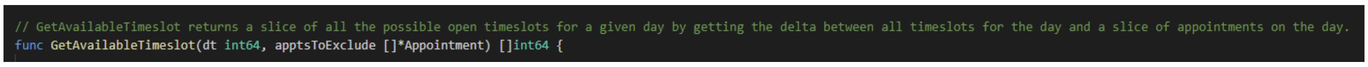
For packages with multiple files like ***web*** or ***clinic*** packages, their package commentary is located inside **doc.go**.



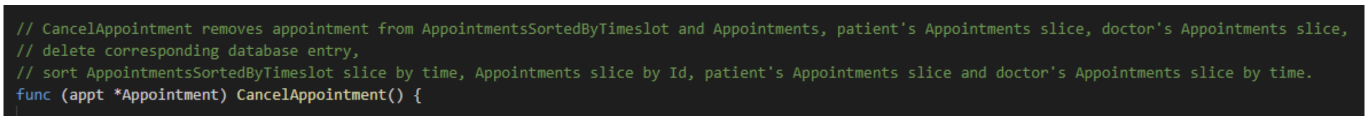
```
ment4 > clinic > doc.go
```

```go
// Package clinic defines the Doctor, Patient, Appointment and Payment types, which implements their individual CRUD, search and helper methods.
package clinic
```

All exported functions begin with the name being declared and describes what they are for.



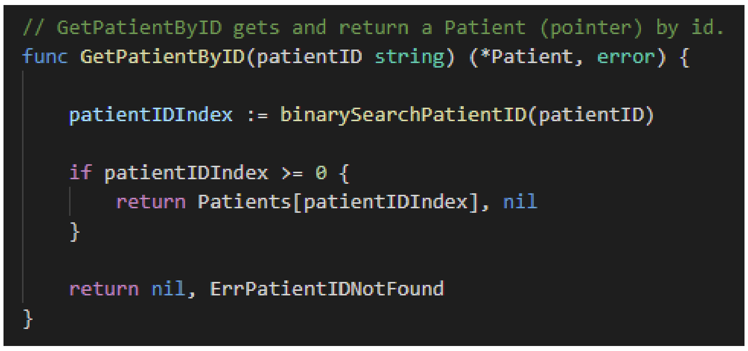
```go
// GetAvailableTimeslot returns a slice of all the possible open timeslots for a given day by getting the delta between all timeslots for the day and a slice of appointments on the day.
func GetAvailableTimeslot(dt int64, apptsToExclude []*Appointment) []int64 {
```



```go
// CancelAppointment removes appointment from AppointmentsSortedByTimeslot and Appointments, patient's Appointments slice, doctor's Appointments slice,
// delete corresponding database entry,
// sort AppointmentsSortedByTimeslot slice by time, Appointments slice by Id, patient's Appointments slice and doctor's Appointments slice by time.
func (appt *Appointment) CancelAppointment() {
```

Function Return

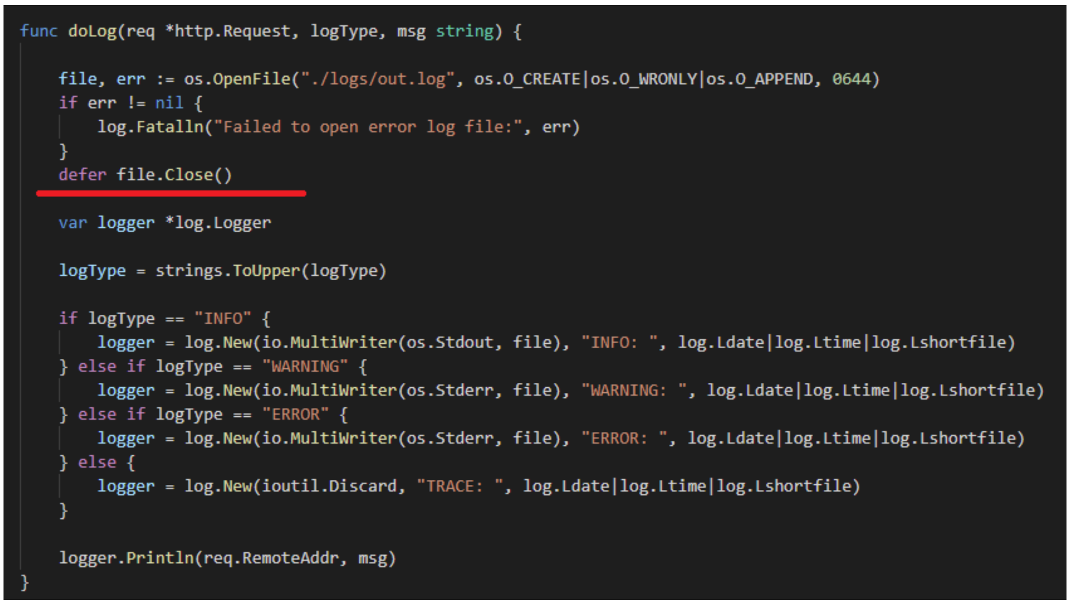
Most functions in the app return both an intended result alongside an error which can be nil if there isn't any error. As seen below, if binarySearchPatientID returns a value bigger than or equal to 0, we return the patient item and nil for the error. However, if it's below 0, an **ErrPatientIDNotFound** error is returned alongside a nil pointer to a patient.



```go
// GetPatientByID gets and return a Patient (pointer) by id.
func GetPatientByID(patientID string) (*Patient, error) {

    patientIDIndex := binarySearchPatientID(patientID)

    if patientIDIndex >= 0 {
        return Patients[patientIDIndex], nil
    }

    return nil, ErrPatientIDNotFound
}
```

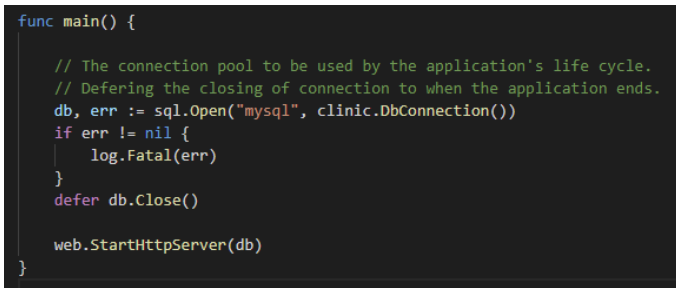
Defer

Defer is used in the **doLog** function to call **file.close** to ensure the file is closed before exiting the function.



```go
func doLog(req *http.Request, logType, msg string) {

    file, err := os.OpenFile("./logs/out.log", os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0644)
    if err != nil {
        log.Fatalln("Failed to open error log file:", err)
    }
    defer file.Close()

    var logger *log.Logger

    logType = strings.ToUpper(logType)

    if logType == "INFO" {
        logger = log.New(io.MultiWriter(os.Stdout, file), "INFO: ", log.Ldate|log.Ltime|log.Lshortfile)
    } else if logType == "WARNING" {
        logger = log.New(io.MultiWriter(os.Stderr, file), "WARNING: ", log.Ldate|log.Ltime|log.Lshortfile)
    } else if logType == "ERROR" {
        logger = log.New(io.MultiWriter(os.Stderr, file), "ERROR: ", log.Ldate|log.Ltime|log.Lshortfile)
    } else {
        logger = log.New(ioutil.Discard, "TRACE: ", log.Ldate|log.Ltime|log.Lshortfile)
    }

    logger.Println(req.RemoteAddr, msg)
}
```

It’s also used in package main right after creating a database connection so if the program is stopped, the database connection will be closed before exiting.



```go
func main() {

    // The connection pool to be used by the application's life cycle.
    // Defering the closing of connection to when the application ends.
    db, err := sql.Open("mysql", clinic.DbConnection())
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    web.StartHttpServer(db)
}
```

4. Godoc Generated Documentation

assignment4

Package main kickstarts the application by calling StartHttpServer function from web package. It also creates a pooled mysql database connection that is passed to StartHttpServer and used in the rest of the program.

Subdirectories

Name	Synopsis
..	
clinic	Package clinic defines the Doctor, Patient, Appointment and Payment types, which implements their individual CRUD, search and helper methods.
psi	Package psi defines the PSI type and provide implementation for fetching the 24H average Pollutant Standards Index (PSI) value.
session	Package session defines the Session and Notification types; It provides implementation for creating and deleting of server side session, client side cookie and use of notification inside session to pass messages between http request.
web	Package web defines route constants and provides the implementation for http handlers and server.

Package clinic

```
import "assignment4/clinic"
```

Overview
Index

Overview ▾

Package clinic defines the Doctor, Patient, Appointment and Payment types, which implements their individual CRUD, search and helper methods.

Index ▾

func (p *PaymentQueue) PrintAllQueueIDs(skipFirst bool) string

Package files

appointment.go config.go doc.go doctor.go errors.go helper.go patient.go payment.go

Constants

Maximum number of days in the future allowed to make an appointment.

```
const MaxAdvanceApptDays = 90
```

Password policy.

```
const MinPasswordLength = 8
```

Variables

Errors and accompanying messages to be output in logs or to users.

```
var (
    // Appointments
    ErrInvalidTimeslot        = errors.New("invalid timeslots entered")
    ErrDoctorNoMoreTimeslot   = errors.New("doctor has no more timeslots
available for today")
    ErrPatientNoMoreTimeslot = errors.New("patient has no more timeslots
available for today")
    ErrNoMoreTimeslot         = errors.New("there are no other timeslots
available with the chosen doctor")
    ErrDuplicateTimeslot      = errors.New("there's already an appointment
scheduled for that timeslot")
    ErrTimeslotExpired        = errors.New("timeslot has already expired")
    ErrAppointmentIDNotFound = errors.New("appointment id not found")

    ErrCreateAppointment = errors.New("unable to create appointment")

    // Doc
    ErrDoctorIDNotFound = errors.New("doctor id not found")
    ErrCreateDoctor     = errors.New("unable to create doctor")

    // Patient
    ErrPatientIDNotFound = errors.New("patient id not found")
    ErrCreatePatient     = errors.New("unable to create patient")

    // Payment
    ErrEmptyPaymentQueue = errors.New("empty payment queue")

    // DB
    ErrDBConn = errors.New("unable to get db connection")
)
```

Admins is a slice containing Ids of clinic staff.

```
var Admins = []string{}
```

Appointments holds all the appointments sorted by Id.

```
var Appointments = []*Appointment{}
```

AppointmentsSortedByTimeslot holds all appointments sorted by time.

```
var AppointmentsSortedByTimeslot = []*Appointment{}
```

Doctors hold all the doctors sorted by incremental id.

```
var Doctors = []*Doctor{}
```

MissedPaymentQ holds the outstanding payments that have been moved over from PaymentQ.

```
var MissedPaymentQ = &PaymentQueue{}
```

Patients holds all the patients sorted by Id (alphanumeric).

```
var Patients = []*Patient{}
```

PaymentQ holds the payments that are pending in a FIFO queue.

```
var PaymentQ = &PaymentQueue{}
var Wg sync.WaitGroup
```

func **BinarySearchApptID**

```
func BinarySearchApptID(apptID int64) int
```

BinarySearchApptID performs binary search for appointment id in Appointments.

func **BinarySearchApptTime**

```
func BinarySearchApptTime(time int64) int
```

BinarySearchApptTime performs binary search for appointment time in Appointments.

func **DbConnection**

```
func DbConnection() string
```

DbConnection returns the database connection string.

func **GetAvailableTimeslot**

```
func GetAvailableTimeslot(dt int64, apptsToExclude []*Appointment) []int64
```

GetAvailableTimeslot returns a slice of all the possible open timeslots for a given day by getting the delta between all timeslots for the day and a slice of appointments on the day.

func **IsApptTimeValid**

```
func IsApptTimeValid(t int64) (bool, error)
```

IsApptTimeValid checks if time of appointment is in the past - e.g. process started at 3:55 PM, user chose 4 PM timeslot but submitted form at 4:05 PM.

func **IsNRICValid**

```
func IsNRICValid(nric string) bool
```

IsNRICValid checks if a given NRIC is valid - Checks for length of 9 if strictNRIC is set to false (default) in clinic config; If true, will perform full NRIC validity check against checksum. Translated from https://gist.github.com/kamerk22/ed5e0778b3723311d8dd074c792834ef

func **IsThereTimeslot**

```
func IsThereTimeslot(dt int64, pat *Patient, doc *Doctor) (bool, error)
```

IsThereTimeslot checks if there's timeslot available for the day by checking both the patient's and doctor's appointments for the day.

func **SeedData**

```
func SeedData()
```

SeedData resets the database, setup the database, seed test data or load clinic globals from database depending on settings in clinic config.

func **SetDb**

```
func SetDb(myDb *sql.DB)
```

SetDb sets the singleton database connection to be used by package.

type **Appointment**

```
type Appointment struct {
    Id      int64 // unique identifier
    Time    int64 // unix time for easy sorting via int value comparison
    Patient *Patient
    Doctor  *Doctor
}
```

func MakeAppointment

```
func MakeAppointment(t int64, pat *Patient, doc *Doctor, wgrp
*sync.WaitGroup) (*Appointment, error)
```

Create Appointment, insert to database, add Appointment to global slice AppointmentsSortedByTimeslot and Appointments, sort global slice AppointmentsSortedByTimeslot, patient's Appointments slice and doctor's Appointments slice by appointment time.

func (*Appointment) CancelAppointment

```
func (appt *Appointment) CancelAppointment()
```

CancelAppointment removes appointment from AppointmentsSortedByTimeslot and Appointments, patient's Appointments slice, doctor's Appointments slice, delete corresponding database entry, sort AppointmentsSortedByTimeslot slice by time, Appointments slice by Id, patient's Appointments slice and doctor's Appointments slice by time.

func (*Appointment) EditAppointment

```
func (appt *Appointment) EditAppointment(t int64, pat *Patient, doc
*Doctor) error
```

EditAppointment updates appointment item, updates corresponding database entry, sort AppointmentsSortedByTimeslot slice by time, Appointments slice by Id, patient's Appointments slice and doctor's Appointments slice by time.

type BST

```
type BST struct {
    // contains filtered or unexported fields
}
```

DoctorsBST is a balanced binary search tree of doctors.

```
var DoctorsBST *BST
```

func (*BST) GetDoctorByIDBST

```
func (bst *BST) GetDoctorByIDBST(docID int64) (*Doctor, error)
```

GetDoctorByIDBST gets a Doctor from the global DoctorsBST by Id. Returns pointer to Doctor if found.

type BinaryNode

```
type BinaryNode struct {
    // contains filtered or unexported fields
}
```

type Doctor

```
type Doctor struct {
    Id           int64
    First_name   string
    Last_name    string
    Appointments []*Appointment
}
```

func (*Doctor) GetAppointmentsByDate

```
func (d *Doctor) GetAppointmentsByDate(dt int64) []*Appointment
```

GetAppointmentsByDate returns a slice of Appointments (pointers) on the given date (unix time). Todo: Can improve by making it binary search instead of sequential since Appointments is sorted by time.

func (*Doctor) IsFreeAt

```
func (d *Doctor) IsFreeAt(t int64) bool
```

type **Patient**

```
type Patient struct {
    Id           string
    First_name   string
    Last_name    string
    Password     []byte
    Appointments []*Appointment
}
```

func CreatePatient

```
func CreatePatient(username, first_name, last_name string, password []byte)
(*Patient, error)
```

CreatePatient is for creating new Patient, inserting to database, add Patient to Patients slice and sort Patients slice via mergesort.

func GetPatientByID

```
func GetPatientByID(patientID string) (*Patient, error)
```

GetPatientByID gets and return a Patient (pointer) by id.

func IsLoggedIn

```
func IsLoggedIn(req *http.Request) (*Patient, bool)
```

IsLoggedIn checks if a user is logged in by checking for existence of client side Cookie and comparing Cookie's value to server side session data in MapSessions to check for validity; Returns Patient and true if valid.

func (*Patient) DeletePatient

```
func (p *Patient) DeletePatient() error
```

DeletePatient is for deleting Patient, Patient's appointments, removing patient from Patients slice and deleting corresponding database entry.

func (*Patient) EditPatient

```
func (p *Patient) EditPatient(username, first_name, last_name string,
password []byte)
```

EditPatient is for updating Patient, update corresponding database entry and sort Patients slice via mergesort.

func (*Patient) GetAppointmentsByDate

```
func (p *Patient) GetAppointmentsByDate(dt int64) []*Appointment
```

GetAppointmentsByDate gets a Patient's appointments (slice of pointers) on a given date (unix time).

func (*Patient) IsAdmin

```
func (p *Patient) IsAdmin() bool
```

IsAdmin returns true if Patient is an admin. Checks recursively against Admins slice.

func (*Patient) IsFreeAt

```
func (p *Patient) IsFreeAt(t int64) bool
```

type Payment

```
type Payment struct {
    Id          int64
    Appointment *Appointment
    Amount      float64
}
```

func (*Payment) ClearPayment

```
func (pmy *Payment) ClearPayment()
```

ClearPayment deletes the payment entry from database.

type PaymentNode

```
type PaymentNode struct {
    Payment *Payment
    Next    *PaymentNode
}
```

type PaymentQueue

```
type PaymentQueue struct {
    Front *PaymentNode
    Back  *PaymentNode
    Size  int
}
```

func CreatePayment

```
func CreatePayment(appt *Appointment, amt float64, wg *sync.WaitGroup)
(*PaymentQueue, error)
```

CreatePayment is for creating payment, adding to database, adding to payment queue and removing the appointment.

func (*PaymentQueue) Dequeue

```
func (p *PaymentQueue) Dequeue() (*Payment, error)
```

Dequeue is for removing a payment from a queue.

func (*PaymentQueue) DequeueToMissedPaymentQueue

```
func (p *PaymentQueue) DequeueToMissedPaymentQueue() (*Payment, error)
```

DequeueToMissedPaymentQueue removes a payment from a queue and move it to MissedPaymentQ.

func (*PaymentQueue) DequeueToPaymentQueue

```
func (p *PaymentQueue) DequeueToPaymentQueue() (*Payment, error)
```

DequeueToPaymentQueue removes a payment from a queue and move it to PaymentQ.

func (*PaymentQueue) Enqueue

```
func (p *PaymentQueue) Enqueue(pmy *Payment) error
```

Enqueue is for adding a payment to a queue.

func (*PaymentQueue) PrintAllQueueIDs

```
func (p *PaymentQueue) PrintAllQueueIDs(skipFirst bool) string
```

PrintAllQueueIDs returns a CSV of appointment ids from payments inside a payment queue.

Package psi

```
import "assignment4/psi"
```

Overview ▾

Package psi defines the PSI type and provide implementation for fetching the 24H average Pollutant Standards Index (PSI) value.

Index ▾

Package files

psi.go

type PSI

```
type PSI struct {
    Value       string
    Description string
}
```

func GetPSI

```
func GetPSI() (*PSI, error)
```

GetPSI returns a PSI item containing the 24H national average pollutant standards index value and description.

Package session

```
import "assignment4/session"
```

Overview
Index

Overview ▾

Package session defines the Session and Notification types; It provides implementation for creating and deleting of server side session, client side cookie and use of notification inside session to pass messages between http request.

Index ▾

Package files

doc.go notification.go session.go

Variables

CookieID is the name of the client side cookie.

```
var CookieID string
```

MapSessions stores all the user session(s) of the app.

```
var MapSessions = make(map[string]Session)
```

func **ClearNotification**

```
func ClearNotification(req *http.Request) error
```

ClearNotification deletes a notification message from user's session.

func **CreateSession**

```
func CreateSession(res http.ResponseWriter, req *http.Request, username,
serverHost string)
```

CreateSession creates client side cookie, create and add session to global MapSessions; Also, purge any duplicate sessions by a user.

func DeleteSession

```
func DeleteSession(res http.ResponseWriter, req *http.Request)
```

DeleteSession expires a user's client side cookie and remove session from global MapSessions.

func SetNotification

```
func SetNotification(req *http.Request, notificationMsg, notificationType
string) error
```

SetNotification sets a notification message to user's session.

type Notification

```
type Notification struct {
    Message string
    Type    string // Types: "Success", "Error"
}
```

func GetNotification

```
func GetNotification(req *http.Request) (*Notification, error)
```

GetNotification gets a notification message from user's session.

type Session

```
type Session struct {
    Id           string
    LastModified int64
    LastVisited  *url.URL
    Notification *Notification
}
```

Package web

```
import "assignment4/web"
```

Overview ▾

Package web defines route constants and provides the implementation for http handlers and server.

Index ▾

Package files

adminPages.go appointment.go config.go doc.go doctor.go errorPages.go errors.go helper.go httpServer.go index.go patient.go payment.go psi.go routes.go

func **StartHttpServer**

```
func StartHttpServer(myDb *sql.DB)
```

StartHttpServer setup routes & handlers and start web server over https. It also calls SeedData from clinic package to perform database setup and/or seeding of test data depending on clinic package's config settings.