“A & Y Dental Clinic” Application Overview

The app listens on a hostname and port for client requests over http. In our case, the hostname and port are **`localhost`** and **`5221`** respectively.

Instead of accessing the app over cli, clients are now able to access the app through http requests. Ideally, over a web browser to view the graphical user interface (GUI) presented through html.

The architecture is such that there’s a single server listening and as many clients as the server machine is able to handle requests over http.

Functionality

The app is intended to be operated by both patients and clinic staff. In the system’s context, all users are considered “patients”. Clinic staff are patients that have been flagged as administrators. Appointments can be made by registered users between 8 am to 10 pm same day at time slots with intervals of 30 mins.

Public access (Non login users)

- Register
- Login
- View payment queue
- View Pollutant Standards Index (PSI)

Patient

- Make appointment
- Edit appointment
- Cancel appointment
- View appointments
- View doctors & their available appointment time slots

Staff

- View clinics’ appointments
- Edit appointment
- Cancel appointment
- Manage sessions
- Manage users
- Move appointment to payment queue
- Move payment in payment queue to missed payment queue (missed)
- Move payment out from payment queue (paid)

Go In Action 1 Concepts

http server, client & json

The app acts as a server and listens for requests at **`localhost`** and port number **`5221`**. It uses a http client in **`psiPage.go`** to perform a get request to gov.data.sg to obtain a json encoded set of data which is then decoded through the unmarshal function for us to obtain the PSI value.

HandleFunc functions are used with custom handler functions for each of the app's webpages. All of the app's URL patterns can be found in **`routes.go`** and the different HandleFunc functions can be found in **`httpServer.go`.**

As the pattern " / " matches all paths not matched by other registered patterns, the page not found handling is coded into indexPage whereby a function notFoundErrorHandler returns http.StatusNotFound status code and presents the app's custom error page through the **`404.gohtml`** template if the request URL is not " / ".

FileServer is used to handle returning of static files such as the style.css file used to power the GUI.

Templates

HTML templates are used on all pages to present users with a HTML powered GUI. They can be found in the templates directory and have the .gohtml extension. Nested templates are used to separate out the header and footer and data is passed into them from the ExecuteTemplate function.

The templates make use of logics and loops to process and present data to the user. Examples include iterating over a slice of appointments and presenting it in a table and conditional checks to see if the user is an admin.

All the templates are parsed in the init function of main.go along with a couple of helper functions from **`helper.go`** via FuncMap for use inside templates.

States & Sessions

Various parts of the app uses URL query parameters to pass data around such as identifying a doctor via his id, an appointment id or the intended action such as edit or delete. Some examples from the app are given below.

- http://localhost:5221/doctors?doctorID=101
- http://localhost:5221/appointment/edit?apptId=1002&action=cancel
- http://localhost:5221/admin/appointment/edit?apptId=1002&action=edit
- http://localhost:5221/admin/appointment/edit?apptId=1002&action=cancel
- http://localhost:5221/admin/payment/enqueue?apptId=1002
- http://localhost:5221/admin/users?action=delete&userId=S0000000A

Form post requests are used in registration and login to pass on the authentication information over to the server. Post is used over Get to obscure and avoid appending the password over plain text in the url. Forms are also used extensively to pass data from the client over to the server when making or editing appointments.

Cookie is created on a user's browser whenever they login which in turns stores a session id on the server that can be mapped to the user. This lets us identify the user on subsequent visits anywhere on the app (website). On the server side, the user's last active time and visited page are also tracked.

The mapping of cookie and session id to user allows us to identify if the user is logged in through the existence of our cookie and whether he / she is an admin. Thus, allowing the app to restrict pages to certain groups. For example if a non logged-in user tries to access appointments page or if a non admin tries to access the manage clinic's appointments page, he / she will be redirected away.

Error Handling

Many of the functions are written to handle errors in Go's idiomatic way by returning an error type or nil. Custom errors can be found in **`errors.go`**.

The logics for the different web pages are coded to always return a page by default and only changes what's presented based on any change of state e.g. from query string values, form post values or related cookies.

Concurrency

Go's net/http which is used to run the http server automatically employs concurrency.

Mutex, WaitGroup and Atomic operations are mechanisms used to handle race condition related issues.

They can be observed right at the start of the app through the **`seedDoctors`** and **`seedPatients`** functions which are called in **`main.go`**'s init. Both of them use goroutines to create their respective patient / doctor items and append them to their global slice variables. The appending to the global slice is when race condition issues might arise.

WaitGroup's counter is used to let the app know when a set of goroutines, like the 7 createPatient goroutines in seedPatients have finished running (in whatever order) and to wait till then.

Inside createPatient and addDoctor functions, the mutex locks and ensures only 1 goroutine is able to append to their respective slice at any point in time.

Atomic AddInt64 function is used to increment the int64 id field for new doctor, payment and appointment items such that only 1 increment can be done at any point in time to avoid getting duplicate ids.

Similarly, all of the struct's write related functions (create / update / delete) use mutex to protect against race conditions in the app runtime coming from concurrent web requests. They are listed below with their file names and functions.

Mutex Usage

- doctor.go
    - addDoctor
- appointment.go
    - makeAppointment, editAppointment and cancelAppointment
- patient.go
    - createPatient, editPatient, deletePatient
- payment.go
    - createPayment, enqueue, dequeue

Comments & Instructions

Initial test accounts are shown below. Their passwords have all been set to "**12345678**". For more info, refer to point # 6 below.

NRIC	Is Admin?
S0000000A	Yes
S1111111B	No
S2222222C	No
S3333333D	No
S4444444D	No
S1234567A	Yes
S9999999C	Yes

1. The app is coded to provide same day appointment booking between 8 am to 10 pm. You are only able to make appointments for the current day. This is changeable in **`config.go.`**
2. The available appointment time slots are calculated based on current time in 30 minute blocks so if accessed at 9:01 pm, only 9:30 pm and 10 pm slots will be available.
    a. If testing the app between 10 pm and 12 midnight, there will not be any available time slots.
    b. You can "spoof" the app by setting a fake current time. This is done inside **`config.go`** by setting **`testFakeTime`** to **`true`**, **`testHour`** and **`testMinute`** to your desired hour and minute
3. NRIC field checks only for length of 9. There is an actual NRIC check that's coded in but not used. It can be toggled on by setting **`strictNRIC`** to **`true`** in **`config.go`**.
4. When the app is started, it seeds the app with a number of test data. Doctors and patients are hardcoded while appointments and payment queues are randomly generated. They can be found in **`helper.go`**. You can disable seeding by commenting them out in the init function of main.go.

5. Admin IDs are hardcoded inside the **`helper.go`** file and in the **`seedAdmins`** function. All seeded users have their password set as "**`12345678`**". Admin IDs are **`S1234567A`**, **`S0000000A`** and **`S9999999C`**.
6. Payment queues follow FIFO queue data structure instead of the real life queue system where numbers are usually not called in sequence.
7. Internet access is recommended as the app uses a couple of frontend dependencies that are loaded from content delivery networks.
8. PSI value is obtained via api call to data.gov.sg and that would also require internet access.