Go In Action 1 - Rachel Tan

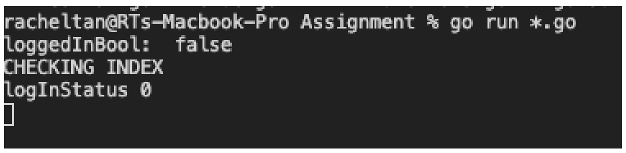
Description

What's In the Zip File:

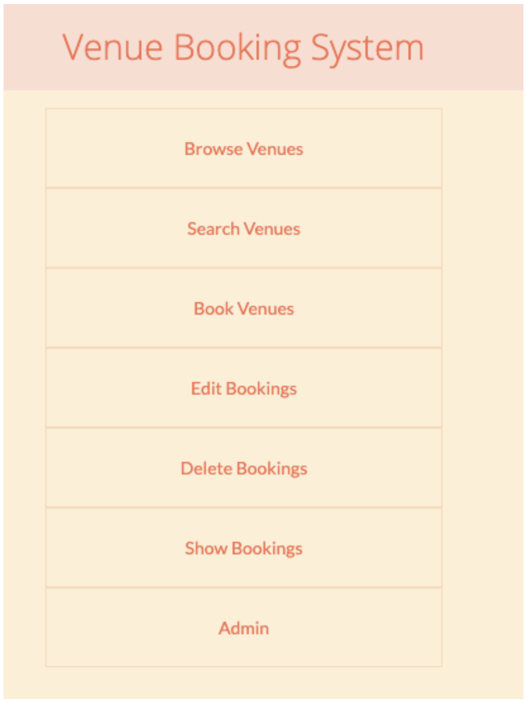
- Main.go - handler that spins up and handles the response and requests
- functionHelper.go - helper functions to perform processing within handlers
- Initializer.go - seed some test data (seeds venues and bookings)
- vApp - holds code from previous assignment
  - LinkedList.go - implementation of LinkedList to hold bookings nodes
  - Tree.go - implementation of AVLTree
  - Venue.go - holds methods to traverse through LinkedList
  - BookingNodes.go - holds struct for booking information
  - Functions - new methods added in this assignment
- Css - contains app.css for formatting
- Templates - gohtml files

Running Application:

1. Go run *.go
2. Login → **username: "admin", pw: "1234"** to access the entire application
3. Signup → for any other username



```
racheltan@RTs-Macbook-Pro Assignment % go run *.go
loggedInBool:  false
CHECKING INDEX
logInStatus 0
```



Description of Application:

- Header: Venue Booking System : link to homepage /
    - Conditional Header
        - SignUp Login: if the user doesn’t have a cookie it’ll prompt login
        - UserName Logout: if user is logged in, the userName will be passed into header to indicate the user is logged in and can access certain features
- Menu:
    - Browse Venues: Open to everyone to view all the venues
    - Search Venues: Open to everyone to view, uses a form to capture what form of search choice:
        - Type
        - Capacity
        - Date Availability
    - Book Venues: have to be logged in, the booking will be using the username
    - Edit Bookings: have to be logged in, it will search for the venue and booking details and update with new booking information
    - Delete Bookings: have to be logged in, will ask for venue and booking information to be deleted
    - Show Bookings: open to everyone, it will run through the entire database to search for bookings
    - Admin: only available for admin
        - Add Venues: enter new information for a new venue
        - Delete Sessions: the server will run through the session data and display the users attached to the session to be deleted
        - Delete Users: the server will run through the user map and display the username to be picked
            - “admin” is protected from user deletion

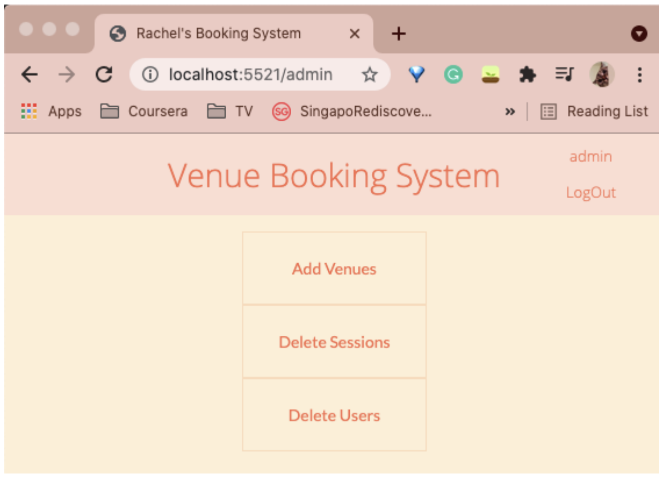
Global Variables:

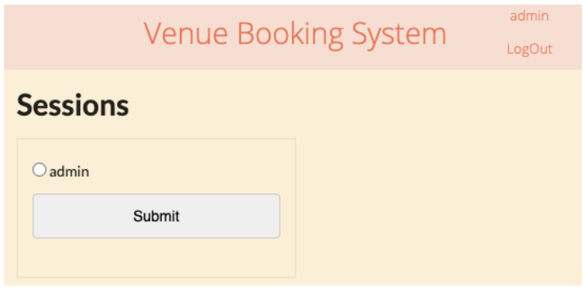
- myMap map[string]*AVLTree - holds the entire structure for venues and bookings
- capacityMap map[int]bool - a quick map to hold the available capacity for easier searching when going through the trees to look for a Venue
- mapUsers map[string]user{} - map of username to User{} that holds the encrypted password
- mapSession map[string]string{} - map of sessions to usernames that will tell if the user is logged in or not

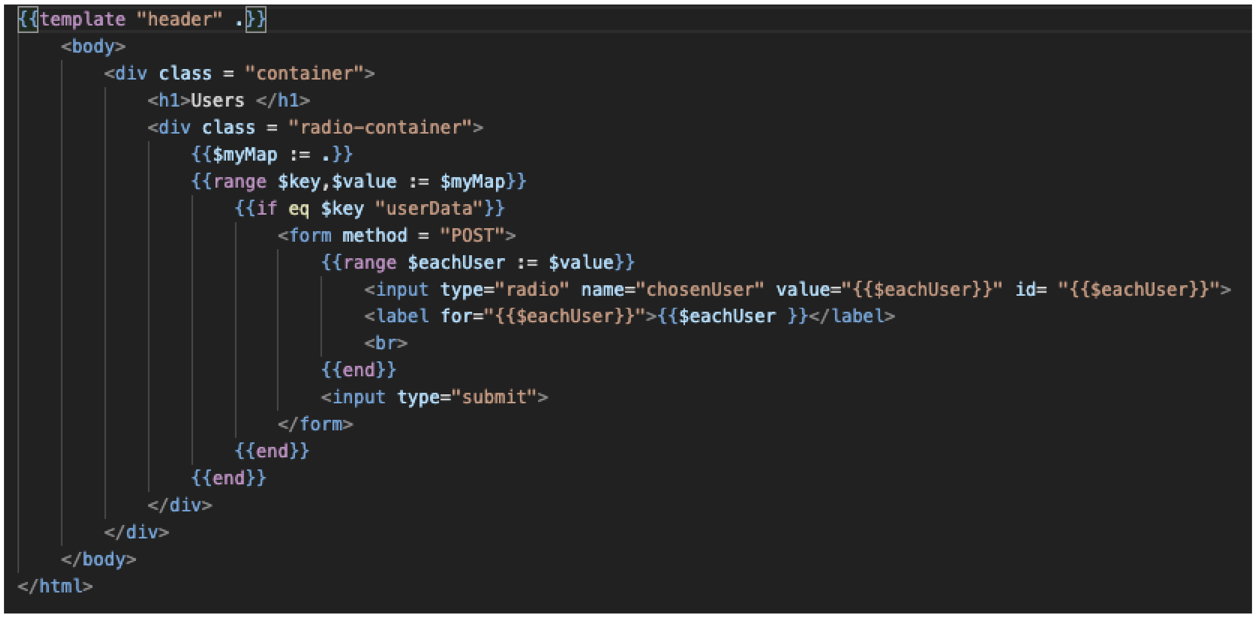
3 Concept in Go In Action 1

Templates

1. On restricted pages, there's a check on whether if the user if logged in, and if so, the user's information is passed into the HTML to be shown to indicate that the user is logged in.
2. Deleting Sessions/Users passes in the data to populate the form. For example, deleting users, the user data is saved with a map[username string]userStruct{}. The key is passed into the HTML template, with the username as the value so that the chosen username can be passed back in the request.







```
{{template "header" .}}
    <body>
        <div class = "container">
            <h1>Users </h1>
            <div class = "radio-container">
                {{$myMap := .}}
                {{range $key,$value := $myMap}}
                    {{if eq $key "userData"}}
                        <form method = "POST">
                            {{range $eachUser := $value}}
                                <input type="radio" name="chosenUser" value="{{$eachUser}}" id= "{{$eachUser}}">
                                <label for="{{$eachUser}}">{{$eachUser }}</label>
                                <br>
                            {{end}}
                            <input type="submit">
                        </form>
                    {{end}}
                {{end}}
            </div>
        </div>
    </body>
</html>
```

Cookies/Sessions Authentication

1. At signup, a cookie is given to the user with sessionID attached to it
    a. The session is added to mapSessions
    b. The password is encrypted and passed into a user struct
    c. The user is added into mapUsers for storage - this is locked by mutex incase multiple users are signing up and might overlap
2. At login, the password is checked with bcrypt
    a. If valid, a new sessionID is generated and the a cookie is given to the user and stored into mapSession
3. At logout, the cookie is grabbed from the user
    a. It's deleted from mapSession and the cookie is deleted from the user side as well

Modules/Packages

Most of the base code from previous assignment is put into vApp folder and imported into main.go

```
import (
   vApp "AssignmentV2/vApp"
   "bufio"
   "fmt"
   "html/template"
   "log"
   "net/http"
   "os"
   "strconv"
   "strings"
   "sync"
   "time"

   uuid "github.com/satori/go.uuid"
   bcrypt "golang.org/x/crypto/bcrypt"
)
```

Since manual seeding of test data is done outside the package, certain functions and fields were changed to upper case to allow access from main.go.

3 Error Handling/Concurrency

Error Handling

- Form inputs are checked prior to proceeding with data processing, and if there is error, it'll be passed in a map to feed into the HTML to be displayed to inform the user of the error

```
if len(choice2) > 0 && len(choice3) > 0 {
       //search for date availability
       i, err1 := strconv.Atoi(choice2[0])
       j, err2 := strconv.Atoi(choice3[0])
       if err1 != nil || err2 != nil {
```

```
err := printByTypeCheck(a[0], typeSlice)
               if err != nil {
                   m["error"] = "There was an error! Please try again!"
```

```
{{if eq $key "notLoggedInError"}}
                   <p class = "error" >{{$value}}</p>
               {{end}}
               {{if eq $key "successAction"}}
                   <p class = "successAction" >{{$value}}</p>
```

Concurrency

Areas where it requires to run through all venues for searching and browsing, concurrency is used to help process. Since the structure of the data is stored with type → tree, we can pass each tree into a goroutine to process.

```
typeSlice := []string{}
for key, _ := range myMap {
    typeSlice = append(typeSlice, key)
}
data := []*vApp.Venue{}
taskLoad := len(typeSlice)
tasks := make(chan string, taskLoad)
wg.Add(4)
```

```
            for gr := 1; gr <= 4; gr++ {
                go searchForAvailabilityConc(tasks, i, j, &data)

            }
            for _, venueType := range typeSlice {
                tasks <- venueType
            }
            close(tasks)
            wg.Wait()
```

Mutex Lock/Unlock

In the above example, we see that multiple goroutines are used for function searchForAvailabilityConc. This function takes in an array and writes its results into the array. The function that writes to the array is wrapped with mutex so that when multiple goroutines are writing to the result array, it wouldn't run into the problem of multiple goroutines writing to it at once.

```
func arrayReader(finalList *[]*Venue, input *Venue) {
   mutex.Lock()
   *finalList = append(*finalList, input)
   mutex.Unlock()
}
```

Areas of the server where it touches the data such as mapUsers, mapSessions or myMap (where the entire venue data is stored) is wrapped with mutex in case when multiple users connecting to the server is making changes.

```
   mutex.Lock()
   delete(mapSessions, c.Value)
   mutex.Unlock()
```