T.C. MARMARA UNIVERSITY
FACULTY OF ENGINEERING COMPUTER ENGINEERING DEPARTMENT

CSE 4074 – Programming Assignment

Socket Programming – HTTP-based Room Reservation

Group Members

Yusuf Taha Atalay - 150119040
Ahmet Emre Sağcan 150119042

For this project we asked to implement a room reservation API with 3 different servers with their own databases.

Some functionalities were common amongst these servers.

Common Functionalities

All 3 servers need to produce their return messages in the form of an HTML format. For that we've created a useful function called CreateHTTPResponse, this function is located at each server's helper package.

```
func CreateHTTPResponse(servername string, statuscode int, status string, title string, body
string) string {
    response := strings.Builder{}
    afterheader := strings.Builder{}
    afterheader.WriteString("<!DOCTYPE html>\n")
    afterheader.WriteString("<html lang=\"en\">\n")
    afterheader.WriteString("<head>\n")
    afterheader.WriteString("<meta charset=\"UTF-8\">\n")
    afterheader.WriteString("<meta name=\"viewport\" content=\"
width=device-width, initial-scale=1.0\">\n")
    afterheader.WriteString("<title>")
    afterheader.WriteString(servername)
    afterheader.WriteString("</title>\n")
    afterheader.WriteString("</head>\n")
    afterheader.WriteString("<body>\n")
    afterheader.WriteString("<h1>")
    afterheader.WriteString(title)
    afterheader.WriteRune('\n')
    afterheader.WriteString("</h1>\n")
    afterheader.WriteString("<p>")
    afterheader.WriteString(body)
    afterheader.WriteRune('\n')
    afterheader.WriteString("</p>\n")
    afterheader.WriteString("</body>\n")

    response.WriteString("HTTP/1.0 ")
    response.WriteString(strconv.Itoa(statuscode))
    response.WriteString(" ")
    response.WriteString(status)
    response.WriteString("\r\n")
    response.WriteString("Content-Type: text/html")
    response.WriteString("\r\n")
    response.WriteString("Content-Length: ")
    response.WriteString(strconv.Itoa(len([]byte(afterheader.String()))))
    response.WriteString("\r\n")
    response.WriteString("\r\n")
    response.WriteString(afterheader.String())

    return response.String()
}
```

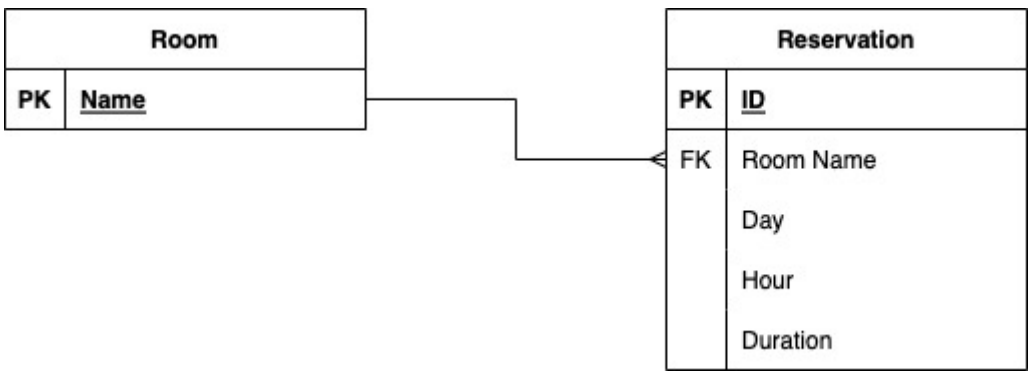
In order for reservation server to locate the two other servers (room, activity);
Each backend server needs to log their port number in a shared environment file.
Both of the room and activity servers have this code on their initializing sequence.

```
// open the config file to write this serve's information
    err := godotenv.Load("../.env")
    if err != nil {
        log.Fatalf("cannot found project's dotenv file: %v\n", err)
    }
    if len(os.Args) != 2 {
        panic(errors.New("Should only provide a port number."))
    }

    // insert this server's port number to config file
    godotenv.Write(map[string]string{
        "ROOMSERVERPORT":     os.Args[1],
        "ACTIVITYSERVERPORT": os.Getenv("ACTIVITYSERVERPORT"),
    }, "../.env")
```

regardless of which server has initiated first they both register their port number to both the environment file and the OS's environment variable table.

Room Server

Database design for the room server is illustrated below.



We have declared both room and reservation as different models in our code. They have their own validators and database hooks (beforecreate).

Here is the room's validation and beforecreate method.

```go
// There is no constraint for room creation, only error would be the situation where user
// tries to create a room with an existing name, and we won't consider that case in here.
func (room *Room) Validate() (err error) {

    if room.Name == "" {
        err = errors.New("room name cannot be empty")
    }
    return
}

func (room *Room) BeforeCreate(tx *gorm.DB) (err error) {

    err = room.Validate()

    // check whether if same room already exists
    var exists bool

    err = database.DBConn.Model(Room{}).Select("count(*) > 0").Where("name = ?", room.Name).Find(&
exists).Error
    if err != nil {
        log.Printf("Error: %+v", err)
        return err
    }

    if exists {
        return errors.New("Room already exists")
    }

    return
}
```

Here is the reservation’s validation method.

```go
func (reservation *Reservation) Validate() error {

    // check if room exists in database
    var exists bool
    err := database.DBConn.Model(Room{}).Select("count(*) > 0").Where("name = ?"
, reservation.RoomName).Find(&exists).Error
    if err != nil {
        log.Printf("Error: %+v", err)
        return err
    }
    if !exists {
        return errors.New("Room does not exists")
    }

    if reservation.Day < 1 || reservation.Day > 7 {
        return errors.New("day value of reservation should be 1 to 7")
    }

    if reservation.Hour < 9 || reservation.Hour > 17 {
        return errors.New("hour value of reservation should be 9 to 17")
    }

    if reservation.Hour+reservation.Duration > 17 {
        return errors.New("Invalid reservation time slice")
    }
    return nil
}
```

Here is the creation of the tcp server for room server.

```go
// create a tcp socket that listens localhost:PORT
    ln, err := net.Listen("tcp", "localhost:"+os.Getenv("ROOMSERVERPORT"))
    if err != nil {
        log.Fatal(err)
    }
    // deferred close the connection
    defer ln.Close()

    //  program loop
    for {
        // accept connection
        conn, err := ln.Accept()
        if err != nil {
            log.Fatal(err)
        }
        // create new thread for each connection request to handle concurrency
        go handleConnection(conn)
    }
```

In this part of the code we are listening to the localhost's user provided port for tcp connections. The defer part is for closing the connection **after** the program finishes its execution. In an endless loop we are accepting any incoming connection request. For each request we are creating a new **thread** with the "go" keyword.

The handleConnection function is responsible for calling the appropriate handler function according to the url.

```
func handleConnection(conn net.Conn) {
    // Use bufio to read the request
    buf := make([]byte, 1024)
    _, err := conn.Read(buf)
    if err != nil {

        return
    }

    // Parse the request

    reqStr := string(buf)
    reqParts := strings.Split(reqStr, "\n")
    url := strings.TrimSpace(strings.Split(reqParts[0], " ")[1])

    // use appropriate handler for the url
    if strings.HasPrefix(url, "/health") {
        health(&conn)
    } else if strings.HasPrefix(url, "/add") {
        HandleAdd(&conn, reqStr)

    } else if strings.HasPrefix(url, "/remove") {
        HandleRemove(&conn, reqStr)

    } else if strings.HasPrefix(url, "/reserve") {
        HandleReserve(&conn, reqStr)
    } else if strings.HasPrefix(url, "/checkavailability") {
        HandleCheckAvailability(&conn, reqStr)
    } else if strings.HasPrefix(url, "/checkweeklyavailability") {
        HandleCheckWeeklyAvailability(&conn, reqStr)
    }

}
```

The health endpoint was not specified in the project document but we have thought that it would be useful for the reservation server to understand whether the calling server is healthy or not.

```go
// health is for letting the controller server to know that this server is active.
func health(conn *net.Conn) {
    // just return 200 OK to let the caller know this server is active

    response := "HTTP/1.0 200 OK\r\n"
    response += "Content-Type: text/plain\r\n"
    response += "\r\n"
    response += "Room Server is healty at port -> " + os.Getenv("ROOMSERVERPORT")
    response += "\r\n"
    _, err := (*conn).Write([]byte(response))
    if err != nil {
        log.Fatal(err)
    }

    (*conn).Close()
}
```

It just sends a “200 OK” message alongside with the healthy message to the caller.

All the handler functions also support POST request, if the user chooses to use GET request; we are parsing the URL, if POST request was used, then we are parsing the body (after carriage return line feed), we assume that the body will be in JSON format.
For example, here is the reservation endpoint.

```go
func HandleReserve(conn *net.Conn, req string) {
    reqParts := strings.Split(req, "\n")
    method := strings.TrimSpace(strings.Split(reqParts[0], " ")[0])
    url := strings.TrimSpace(strings.Split(reqParts[0], " ")[1])
    response := ""
    defer func() {
        _, err := (*conn).Write([]byte(response))
        if err != nil {
            log.Fatal(err)
        }
        (*conn).Close()
    }()
```

In this part we are parsing the request string to extract the method and url.
The defer func is an anonymous function that will be executed when the parent function has returned, its only purpose is to write the later created response string to the given connection.

Using the extracted method name , if it's a GET request we are again extracting the query from the URL and parse that query to get the key and value pairs. After getting the key and value pairs we are running a pre-validation part, here is an example.

```
switch method {
    case "GET":
        // GET request has been made, read the query params from the URL
        query := strings.Split(url, "?")[1]
        params := strings.Split(query, "&")
        paramsMap := make(map[string]string)
        for _, param := range params {
            temp := strings.Split(param, "=")
            paramsMap[temp[0]] = temp[1]
        }
        // return error if roomname has not given
        if _, ok := paramsMap["name"]; !ok {
            response = helper.CreateHTTPResponse("Room", 400, "Bad Request",
                "Empty Parameter", "name parameter is empty")

            return
        }
```

After all validations have passed, we are finally attempting to create the reservation with given parameters, in case of any error we are notifying the user with document provided HTTP response codes.

```
err = models.CreateReservation(&models.Reservation{
            RoomName: paramsMap["name"],
            Day:      dayint,
            Hour:     hourint,
            Duration: durationint,
        })
        if err != nil {
            if err.Error() == "Room does not exists" {
                response = helper.CreateHTTPResponse("Room", 403, "Forbidden",
                    "Room does not exists", "There is no room exists with the given name")

                return
            } else if err.Error() == "Already Reserved" {
                response = helper.CreateHTTPResponse("Room", 403, "Forbidden",
                    "Room already reserved", "Room already reserved in given time slice")

                return
            } else {
                response = helper.CreateHTTPResponse("Room", 400, "Bad Request",
                    "Database Error", err.Error())

                return
            }
        }

        response = helper.CreateHTTPResponse("Room", 200, "OK",
            "Succesfull", "Reservation created successfully")

        return
```

For the POST request; the only difference from the GET request is the way we parse the request string, here is the parsing part. Since the user has provide a JSON, which starts and ends with curly braces, we are parsing it according to that.

```
case "POST":
        // POST request has been made, read the request body

        reqBody := strings.Split(req, "{")[1]
        reqBody = "{" + reqBody
        reqBodyByte := bytes.Trim([]byte(reqBody), "\x00")
//       // unmarshall the json body to a struct
        var body models.Reservation

        err := json.Unmarshal(reqBodyByte, &body)
        if err != nil {
            response = helper.CreateHTTPResponse("Room", 400, "Bad Request",
                "Parser Error", err.Error())

            return
        }
```

Activity Server

Activity server only contains a single table as a database, which can be illustrated like this.

Activity	
PK	Name

Its validation and handler methods are pretty similar with the Room Server, to keep this document at a reasonable length we won't be pasting code screenshots for this server.

Reservation Server

Reservation server has a single table as a database.

	RoomReservation
PK	ID
	ActivityName
	RoomName
	Day
	Hour
	Duration

Reservation server has additional helper functions to create HTTP request strings and parsing the HTML response.

```go
func CreateHTTPRequest(url string) string {
    request := strings.Builder{}
    request.WriteString("GET ")
    request.WriteString(url)
    request.WriteString(" HTTP/1.0\r\n")

    return request.String()
}

func GetHTMLPBody(s string) string {
    start := strings.Index(s, "<p>")
    end := strings.Index(s, "</p>")
    return s[start+3 : end]
}

```

When Reservation server is initiated, it first fetches the other two server's port numbers from the environment variable. Then by using the /health endpoint that the other servers provide, it checks whether or not the servers are alive.

```
// open the config file to write this serve's information
    err := godotenv.Load("../.env")
    if err != nil {
        log.Fatalf("cannot found project's dotenv file: %v\n", err)
    }

    ROOMSERVERPORT = os.Getenv("ROOMSERVERPORT")
    ACTIVITYSERVERPORT = os.Getenv("ACTIVITYSERVERPORT")

    // check if room server is active using health endpoint using the socket connection
    // if not, exit the program
    RoomServerConn, err := net.Dial("tcp", "localhost:"+ROOMSERVERPORT)
    if err != nil {
        log.Fatalf("Cannor connect to Room Server %s", err.Error())
    }

    _, err = RoomServerConn.Write([]byte("GET /health HTTP/1.0"))
    if err != nil {
        log.Fatal(err)
    }
    buf := make([]byte, 1024)
    _, err = RoomServerConn.Read(buf)
    if err != nil {
        log.Fatal(err)
    }
    if !strings.Contains(string(buf), "200 OK") {
        fmt.Println("Room server is not active.")

        return
    }
    RoomServerConn.Close()
    // check if activity server is active using health endpoint using the socket connection
    // if not, exit the program
    ActivityServerConn, err := net.Dial("tcp", "localhost:"+ACTIVITYSERVERPORT)
    if err != nil {
        log.Fatalf("Cannot connect Activity Server %s", err.Error())
    }
    _, err = ActivityServerConn.Write([]byte("GET /health HTTP/1.0"))
    if err != nil {
        log.Fatal(err)
    }

    buf = make([]byte, 1024)
    _, err = ActivityServerConn.Read(buf)
    if err != nil {

        log.Fatal(err)
    }
    if !strings.Contains(string(buf), "200 OK") {
        fmt.Println("Activity server is not active.")

        return
    }
    ActivityServerConn.Close()
```

It has a similar way of handling incoming requests concurrently like the other servers.

Here is an example of a Reservation server talking to another server and parsing it's response.

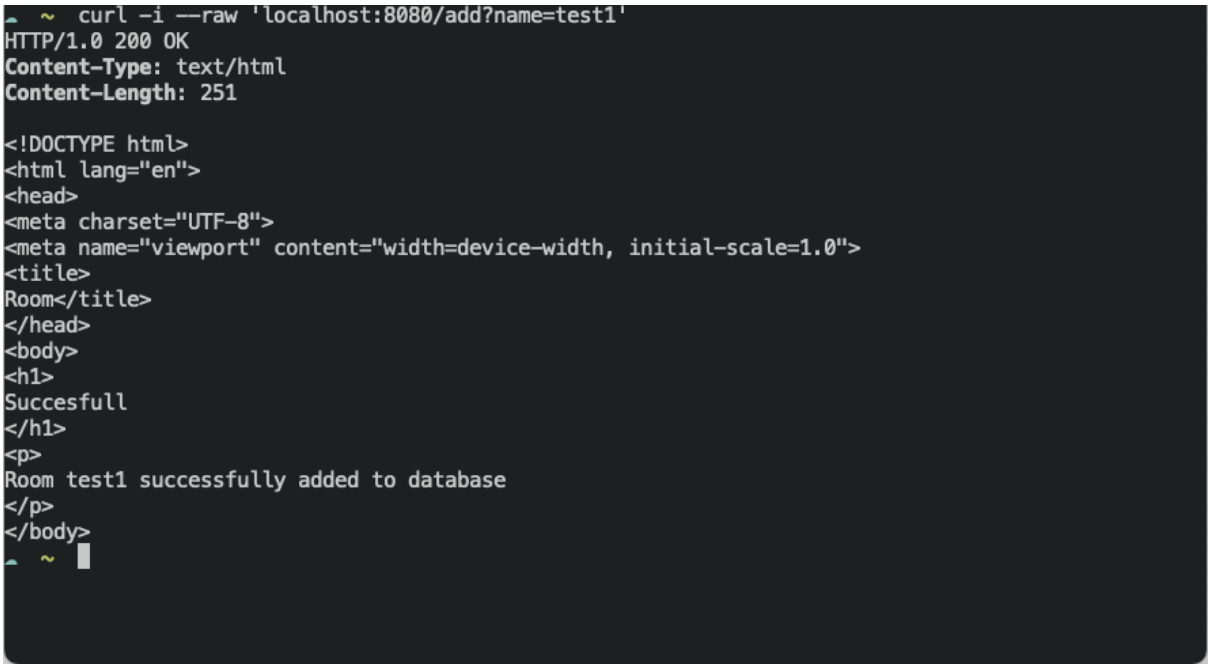
```
// GET request has been made, read the query params from the URL
        query := strings.Split(url, "?")[1]
        params := strings.Split(query, "&")
        paramsMap := make(map[string]string)
        for _, param := range params {
            temp := strings.Split(param, "=")
            paramsMap[temp[0]] = temp[1]
        }
        // check activity server if the activity exists make a socket request
        req := helper.CreateHTTPRequest(fmt.Sprintf("/check?name=%s", paramsMap["activity"]))
        _, err := ActivityServerConn.Write([]byte(req))
        if err != nil {
            log.Fatal(err)
        }
        buf := make([]byte, 1024)
        _, err = ActivityServerConn.Read(buf)
        if err != nil {
            if err.Error() != "EOF" {
                log.Fatal(err)
            }
        }
        if !strings.Contains(string(buf), "200 OK") {
            response = helper.CreateHTTPResponse("Reservation", 404, "Not Found",
                "Database error.", "Activity not found.")

            return
        }
        // check room server if the input is valid
        req = helper.CreateHTTPRequest(fmt.Sprintf("/reserve?name=%s&day=%s&hour=%s&duration=%s",
            paramsMap["room"], paramsMap["day"], paramsMap["hour"], paramsMap["duration"]))

        _, err = RoomServerConn.Write([]byte(req))
        if err != nil {
            if err.Error() != "EOF" {
                log.Fatal(err)
            }
        }
```

This is from the /reserve endpoint's GET request handler. It first creates a HTTP request to the activity server to see if that activity exists, if so then creates another HTTP request for the room server's /reserve endpoint, that endpoint creates a reservation if everything went successful.

Sample Executions

Room Server

Adding a non-existed room with GET request:



Adding non-existed room with POST request:

Trying to add already existed room:

```
~  curl -i --raw 'localhost:8080/add?name=test1'
HTTP/1.0 403 Forbidden
Content-Type: text/html
Content-Length: 268

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>
Room</title>
</head>
<body>
<h1>
Room already exists
</h1>
<p>
There is already a room exists with the same name
</p>
</body>
~
```

Deleting a existing room

```
~  curl -i --raw 'localhost:8080/remove?name=test1'
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 255

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>
Room</title>
</head>
<body>
<h1>
Succesfull
</h1>
<p>
Room test1 successfully removed from database
</p>
</body>
~
```

Deleting a non-existing room

```
~ curl -i --raw 'localhost:8080/remove?name=test1'
HTTP/1.0 403 Forbidden
Content-Type: text/html
Content-Length: 263

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>
Room</title>
</head>
<body>
<h1>
Room does not exists
</h1>
<p>
There is no room exists with the given name
</p>
</body>
~
```

Reserving an available room

```
~ curl -i --raw 'localhost:8080/reserve?name=test2&day=3&hour=12&duration=4'
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 242

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>
Room</title>
</head>
<body>
<h1>
Succesfull
</h1>
<p>
Reservation created successfully
</p>
</body>
~
```

Reserving non-available room

```
~  curl  -i --raw 'localhost:8080/reserve?name=test2&day=3&hour=12&duration=4'
HTTP/1.0 403 Forbidden
Content-Type: text/html
Content-Length: 262

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>
Room</title>
</head>
<body>
<h1>
Room already reserved
</h1>
<p>
Room already reserved in given time slice
</p>
</body>
~
```

Since Functionalities of Activity server is highly similar to Room server we will not add screenshots of it in order to keep this document short.

Reservation Server

Reservation of a room for an activity

```
~  curl -i --raw 'localhost:8082/reserve?room=test2&activity=testactivity1&day=5&hour=12&duration=3'
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 339

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Reservation</title>
</head>
<body>
<h1>Reservation successful.
</h1>
<p>Reservation Details:
Reservation ID: 1
Room: test2
Activity: testactivity1
Day: 5
Hour: 12
Duration: 3


</p>
</body>
~
```

Trying to reserve an unavailable room

```
~  curl -i --raw 'localhost:8082/reserve?room=test2&activity=testactivity1&day=5&hour=12&duration=3'
HTTP/1.0 403 Forbidden
Content-Type: text/html
Content-Length: 238

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Reservation</title>
</head>
<body>
<h1>Database error.
</h1>
<p>Room not available.
</p>
</body>
~
```