docker-compose.yml

```yaml
version: "3.9"
services:
  auth-service:
    build:
      context: .
      dockerfile: Auth/Dockerfile
    ports:
      - "8000:8000"
  roster-service:
    build:
      context: .
      dockerfile: Roster/Dockerfile
    ports:
      - "8001:8000"
  directions-service:
    build:
      context: .
      dockerfile: Directions/Dockerfile
    env_file:
      - Directions/.env
    ports:
      - "8002:8000"
  journey-service:
      build:
        context: .
        dockerfile: Journey/Dockerfile
      ports:
        - "8003:8000"
```

Auth

Dockerfile

```
FROM golang:1.15

WORKDIR /app/
COPY Auth ./Auth
RUN go get github.com/gorilla/mux golang.org/x/crypto/bcrypt
github.com/dgrijalva/jwt-go

EXPOSE 8000
CMD ["go", "run", "/app/Auth/auth.go"]
```

auth.go

```
package main

import (
    "encoding/json"
    "log"
    "net/http"
    "time"

    "github.com/dgrijalva/jwt-go"
    "github.com/gorilla/mux"
    "golang.org/x/crypto/bcrypt"
)

var jwtKey = []byte("secret_jwt_key")

type User struct {
    Username string `json:"username"`
    Name string `json:"name"`
    PasswordHash string `json:"-"`
}

var accounts = map[string] User{}

type Claims struct {
    Username string `json:"username"`
    Name string `json:"name"`
    jwt.StandardClaims
}

func signIn(w http.ResponseWriter, r *http.Request) {
    // Get given username and password values from request
    r.ParseForm()

    // Set response type to JSON
    w.Header().Set("Content-Type", "application/json")
```

```
    username := r.FormValue("username")
    password := r.FormValue("password")

    // Lookup user in accounts map. Fetch the hashed password
    user := accounts[username]
    hashedPassword := user.PasswordHash

    // If the password does not match the hash, return 401.
    if !verifyPassword(hashedPassword, password) {
        log.Printf("Sign-in of user %s failed.", username)
        w.WriteHeader(http.StatusUnauthorized)
        w.Write([]byte("{\"error\": \"Incorrect credentials
provided\"}"))
        return
    }

    // Calculate an expiration time 5 minutes from now
    expirationTime := time.Now().Add(5 * time.Minute)

    // Create a claims struct that includes the username and expiration
time.
    claims := &Claims{
        Username: username,
        StandardClaims: jwt.StandardClaims{
            ExpiresAt: expirationTime.Unix(),
        },
    }

    // Create a token with the HS256 hash method and the claims created
above
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    tokenString, err := token.SignedString(jwtKey)

    if err != nil {
        log.Printf("Error: Could not create JWT for user %s : %s",
username, err)
        w.WriteHeader(http.StatusInternalServerError)
    }

    userInfo := struct {
        Token string `json:"token"`
    }{
        Token: tokenString,
    }

    // Return JSON with user token encoded
    json.NewEncoder(w).Encode(userInfo)
    log.Printf("JWT Token successfully created for user %s.", username )
}
```

```
func validateToken(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    rawToken := vars["token"]

    // Surprisingly hard to find documentation for the function below.
    // https://github.com/dgrijalva/jwt-
go/blob/master/MIGRATION_GUIDE.md
    token, err := jwt.ParseWithClaims(rawToken, &Claims{}, func(token
*jwt.Token) (interface{}, error) {
        return jwtKey, nil
    })

    if err != nil || !token.Valid {
        log.Printf("Invalid or incorrect JWT token received : %s", err)
        w.WriteHeader(http.StatusUnauthorized)
        w.Write([]byte("{\"error\": \"Invalid or incorrect JWT token
received.\"}"))
        return
    }

    claims := token.Claims.(*Claims)

    // Since account deletion is not required in spec, we cannot have a
valid JWT for an account that does not exist.
    // Ok to ignore error value below.
    user, _ := accounts[claims.Username]

    log.Printf("User %s JWT token successfully validated",
user.Username)
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(user)
}

func hashSaltPassword(pwd string) (string, error) {
    hash, err := bcrypt.GenerateFromPassword([]byte(pwd),
bcrypt.MinCost)
    // If there is an error, pass it on.
    if err != nil {
        return "", err
    }
    return string(hash), nil
}

func verifyPassword(hash, password string) bool {
    return bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
== nil;
}
```

```
func initialiseAccounts() {
    // Controlled case should not have error, safe to ignore here.
    password1, _ := hashSaltPassword("astonmartin")
    password2, _ := hashSaltPassword("edgarwright")

    accounts["sebvet"] = User {
        Username: "sebvet",
        Name: "Sebastian Vettel",
        PasswordHash: password1,
    }

    accounts["babydriver"] = User {
        Username: "babydriver",
        Name: "Ansel Elgort",
        PasswordHash: password2,
    }

}

func handleRequests() {
    router := mux.NewRouter().StrictSlash(true)
    router.HandleFunc("/login", signIn).Methods("POST")
    router.HandleFunc("/validate/{token}", validateToken).Methods("GET")
    log.Fatal(http.ListenAndServe(":8000", router))
}

func main() {
    log.Println("Starting Auth Service")
    initialiseAccounts()
    handleRequests()
}
```

Directions

.env

```
MAPS_API_KEY=YOUR_KEY_HERE
```

Dockerfile

```
FROM golang:1.15

WORKDIR /app/
COPY Directions ./Directions
RUN go get github.com/gorilla/mux github.com/kr/pretty
googlemaps.github.io/maps

EXPOSE 8000
CMD ["go", "run", "/app/Directions/directions.go"]
```

directions.go

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "os"
    "regexp"

    "github.com/gorilla/mux"
    "googlemaps.github.io/maps"
)

type Route struct {
    TotalDistance int `json:"totaldistance`
    ARoadDistance int `json:"aroaddistance`
}

// Calculate the distance of A roads within a journey
func calcARoadDistance(s []maps.Route) int {
    dist := 0
    for _, step := range s[0].Legs[0].Steps {

        regexA, _ := regexp.Compile("A([0-9]+)")

        // Find steps that have A roads appearing in the instructions
        if regexA.MatchString(step.HTMLInstructions) == true {
            dist = dist + step.Distance.Meters
        }
    }
    return dist
}
```

```
func getRouteDistanceHelper(origin, destination string) (distance,
aRoadDistance int, err error) {
    // Make sure you insert your API Key to access the Google Directions
API
    c, err := maps.NewClient(maps.WithAPIKey(os.Getenv("MAPS_API_KEY")))
    if err != nil {
        log.Fatalf("fatal error: %s", err)
    }
    r := &maps.DirectionsRequest{
        Region:      "UK",
        Origin:      origin,
        Destination: destination,
    }
    route, _, err := c.Directions(context.Background(), r)
    if err != nil {
        log.Fatalf("fatal error: %s", err)
        return 0, 0, err
    }

    // Distance made on A road
    distA := calcARoadDistance(route)

    // Total distance of the journey in meters
    distTotal := route[0].Legs[0].Distance.Meters

    return distTotal, distA, nil
}

func getRouteDistance(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    origin := vars["from"]
    destination := vars["to"]

    totalDistance, aRoadDistance, err := getRouteDistanceHelper(origin,
destination)

    if err != nil {
        log.Printf("Error: Could not find route between %s and %s : %s",
origin, destination, err)
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte(fmt.Sprintf("{\"error\": \"Could not find route
between %s and %s\"}", origin, destination)))
        return
    }

    route := Route{
        TotalDistance: totalDistance,
        ARoadDistance: aRoadDistance,
    }
```

```
    log.Printf("Finding distance between %s and %s", origin,
destination)
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(route)
}

func handleRequests() {
    router := mux.NewRouter().StrictSlash(true)
    router.HandleFunc("/directions/{from}/{to}",
getRouteDistance).Methods("GET")
    log.Fatal(http.ListenAndServe(":8000", router))
}

func main() {
    log.Println("Starting Directions Service")
    handleRequests()
}
```

Journey

Dockerfile

```
FROM golang:1.15

WORKDIR /app/
COPY Journey ./Journey
RUN go get github.com/gorilla/mux

EXPOSE 8000
CMD ["go", "run", "/app/Journey/journey.go"]
```

journey.go

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "time"

    "github.com/gorilla/mux"
)
```

```
type driver struct {
    Username string `json:"username"`
    Name string `json:"name"`
    Rate int `json:"rate"`
}

type route struct {
    TotalDistance int `json:"totaldistance`
    ARoadDistance int `json:"aroaddistance`
}

type journey struct {
    StartPoint string `json:"start_point"`
    EndPoint string `json:"end_point"`
    TotalDistance int `json:"total_distance"`
    ARoadDistance int `json:"a_road_distance"`
    BestDriver driver `json:"best_driver"`
    Cost int `json:"cost"`
}

func getJourney(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    vars := mux.Vars(r)
    origin := vars["from"]
    destination := vars["to"]

    // Get route distance
    resp, err := http.Get(fmt.Sprintf("http://directions-
service:8000/directions/%s/%s", origin, destination))

    if err != nil {
        log.Printf("Error: Could not fetch route between %s and %s :
%s", origin, destination, err)
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte(fmt.Sprintf("{\"error\": \"Could not fetch route
between %s and %s.\"}", origin, destination)))
        return
    }

    var distances route
    json.NewDecoder(resp.Body).Decode(&distances)

    // Get cheapest driver
    resp, err = http.Get("http://roster-service:8000/roster")
    if err != nil {
        log.Printf("Error fecthing roster: %s", err)
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte("{\"error\": \"Could not fetch roster data\"}"))
        return
```

```
    }

    var fetchedDrivers []driver
    json.NewDecoder(resp.Body).Decode(&fetchedDrivers)

    cheapestDriver := getCheapestDriver(fetchedDrivers)

    cost := calculateCost(distances, fetchedDrivers)

    response := journey {
        StartPoint: origin,
        EndPoint: destination,
        TotalDistance: distances.TotalDistance,
        ARoadDistance: distances.ARoadDistance,
        BestDriver: cheapestDriver,
        Cost: cost,
    }

    log.Println(fmt.Sprintf("Journey between %s and %s calculated at %dp
with driver %s", origin, destination, cost,

 response.BestDriver.Username))

    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(response)
}

func calculateCost(routeDetails route, availableDrivers []driver) int {
    cheapestDriver := getCheapestDriver(availableDrivers)
    noOfDrivers := len(availableDrivers)

    cost := cheapestDriver.Rate * (routeDetails.TotalDistance / 1000)

    // Check if over half of distance is on A-Roads.
    // Integer division is ideal here. No need to convert types.
    if routeDetails.ARoadDistance > (routeDetails.TotalDistance / 2) {
        cost *= 2
    }

    if noOfDrivers < 5 {
        cost *= 2
    }

    currentHour, _, _ := time.Now().Clock()

    if currentHour >= 23 || currentHour <= 6 {
        cost *= 2
    }
```

```
    return cost
}

func getCheapestDriver(drivers []driver) driver{

    lowestRate := -1
    var lowestDriver driver

    // Find the driver with the lowest rate. This will always be the
best driver for the route.
    for _, currentDriver := range drivers {
        // If first pass, initialise. Lowest rate can never naturally be
-1
        if lowestRate == -1 {
            lowestRate = currentDriver.Rate
            lowestDriver = currentDriver
            continue
        }

        if currentDriver.Rate < lowestRate {
            lowestDriver = currentDriver
            lowestRate = currentDriver.Rate
        }
    }
    return lowestDriver
}

func handleRequests() {
    router := mux.NewRouter().StrictSlash(true)
    router.HandleFunc("/journey/{from}/{to}", getJourney).Methods("GET")

    log.Fatal(http.ListenAndServe(":8000", router))
}

func main() {
    log.Println("Starting Journey Service")
    handleRequests()
}
```

Roster

Dockerfile

```
FROM golang:1.15

WORKDIR /app/
COPY Roster ./Roster
RUN go get github.com/gorilla/mux

EXPOSE 8000
CMD ["go", "run", "/app/Roster/roster.go"]
```

roster.go

```
package main

import (
    "encoding/json"
    "errors"
    "io/ioutil"
    "log"
    "net/http"

    "github.com/gorilla/mux"
)

type driver struct {
    Username string `json:"username"`
    Name string `json:"name"`
    Rate int `json:"rate"`
}

type driverRequest struct {
    Token string `json:"token"`
}

type driverRateRequest struct {
    driverRequest
    Rate int `json:"rate"`
}

var Roster = map[string]driver{}


func authenticateUser(token string) (*driver, error) {

    r, err := http.Get("http://auth-service:8000/validate/"+token)
```

```
    if err != nil || r.StatusCode != http.StatusOK {
        return nil, errors.New("unauthorised jwt")
    }

    var authenticatedDriver driver
    json.NewDecoder(r.Body).Decode(&authenticatedDriver)

    // Note that just because driver is authenticated, doesn't mean they
are in roster
    // Catch on other side
    return &authenticatedDriver, nil
}

// Requires authentication
func joinRoster(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")

    body, err := ioutil.ReadAll(r.Body)

    if err != nil {
        log.Printf("Error: Parsing request to join roster failed: %s",
err)
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte("{\"error\": \"Parsing request to join roster
failed\"}"))
        return
    }

    var requestData driverRateRequest
    err = json.Unmarshal(body, &requestData)

    if err != nil {
        log.Printf("Error: Request is missing JWT token or rate: %s",
err)
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("{\"error\": \"Request is missing JWT token or
rate\"}"))
        return
    }

    user, err := authenticateUser(requestData.Token)

    if err != nil {
        log.Printf("Error: Invalid JWT token: %s", err)
        w.WriteHeader(http.StatusUnauthorized)
        w.Write([]byte("{\"error\": \"Invalid JWT token\"}"))
        return
    }
```

```
    _, ok := Roster[user.Username]

    // Check if driver is already in roster.
    if ok {
        log.Println("Error: User is already in roster.")
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("{\"error\": \"User is already in roster\"}"))
        return
    }

    // Cannot have a rate of less than or equal to 0p.
    if requestData.Rate <= 0 {
        log.Println("Error: Invalid rate value supplied.")
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("{\"error\": \"Invalid rate value supplied\"}"))
        return
    }

    // At this point, we can safely add driver to the roster with the
given rate
    // Note we do not ask for username or name in this endpoint.
    // By the time they have a token, they have already given this
information.

    user.Rate = requestData.Rate
    Roster[user.Username] = *user

    log.Printf("User %s added to roster with rate %dp", user.Username,
user.Rate)
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(user)
}

// Requires authentication
func leaveRoster(w http.ResponseWriter, r *http.Request) {
    body, err := ioutil.ReadAll(r.Body)

    if err != nil {
        log.Printf("Error: Parsing request to leave roster failed: %s",
err)
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte("{\"error\": \"Parsing request to leave roster
failed\"}"))
        return
    }

    var requestData driverRequest
    err = json.Unmarshal(body, &requestData)
```

```
    if err != nil {
        log.Printf("Error: Request is missing JWT token: %s", err)
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("{\"error\": \"Request is missing JWT token\"}"))
        return
    }

    user, err := authenticateUser(requestData.Token)

    if err != nil {
        log.Printf("Error: Invalid JWT token: %s", err)
        w.WriteHeader(http.StatusUnauthorized)
        w.Write([]byte("{\"error\": \"Invalid JWT token\"}"))
        return
    }

    _, ok := Roster[user.Username]

    // Check if driver is already in roster.
    if !ok {
        log.Println("Error: User is not in roster.")
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("{\"error\": \"User is not in roster\"}"))
        return
    }

    delete(Roster, user.Username)
    log.Printf("User %s removed from roster.", user.Username)

    w.WriteHeader(http.StatusOK)
}

// Requires authentication
func changeRate(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")

    body, err := ioutil.ReadAll(r.Body)

    if err != nil {
        log.Printf("Error: Parsing request to update rate failed : %s",
err)
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte("{\"error\": \"Parsing request to join roster
failed\"}"))
        return
    }

    var requestData driverRateRequest
    err = json.Unmarshal(body, &requestData)
```

```
    if err != nil {
        log.Printf("Error: Request is missing JWT token or rate : %s",
err)
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("{\"error\": \"Request is missing JWT token or
rate\"}"))
        return
    }

    user, err := authenticateUser(requestData.Token)

    if err != nil {
        log.Printf("Error: Invalid JWT token : %s", err)
        w.WriteHeader(http.StatusUnauthorized)
        w.Write([]byte("{\"error\": \"Invalid JWT token\"}"))
        return
    }

    rosterUser, ok := Roster[user.Username]

    // Check if driver is already in roster.
    if !ok {
        log.Println("Error: User is not in roster.")
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("{\"error\": \"User is not in roster\"}"))
        return
    }

    // Cannot have a rate of less than or equal to 0p.
    if requestData.Rate <= 0 {
        log.Println("Error: Invalid rate value supplied.")
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("{\"error\": \"Invalid rate value supplied\"}"))
        return
    }

    rosterUser.Rate = requestData.Rate

    // Replace record in map with updated rate
    Roster[rosterUser.Username] = rosterUser

    log.Printf("Rate updated to %dp for User %s", rosterUser.Rate,
rosterUser.Username)
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(rosterUser)
}

func getDrivers(w http.ResponseWriter, r *http.Request) {
```

```
    w.Header().Set("Content-Type", "application/json")
    var returnList []driver
    for _, value := range Roster {
        returnList = append(returnList, value)
    }
    log.Println("Requesting driver roster info.")
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(returnList)
}

func handleRequests() {
    router := mux.NewRouter().StrictSlash(true)
    router.HandleFunc("/roster", joinRoster).Methods("POST")
    router.HandleFunc("/roster", leaveRoster).Methods("DELETE")
    router.HandleFunc("/roster", changeRate).Methods("PUT")
    router.HandleFunc("/roster", getDrivers).Methods("GET")
    log.Fatal(http.ListenAndServe(":8000", router))
}

func main() {
    log.Println("Starting Roster Service")
    handleRequests()
}
```