ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

«ΕΦΑΡΜΟΓΗ ANDROID ΓΙΑ ΚΑΤΑΓΡΑΦΗ ΟΔΗΓΙΚΩΝ ΣΥΜΠΕΡΙΦΟΡΩΝ ΜΕ ΛΕΙΤΟΥΡΓΙΕΣ GPS ΚΑΙ SPRING BOOT BACKEND»

«ANDROID APP FOR RECORDING DRIVING BEHAVIOUR WITH GPS SERVICES AND SPRING BOOT BACKEND»

Φοιτητής Ιωάννης Κανελλόπουλος – Π17037

Επιβλέπων
Δρ. Ευθύμιος Αλέπης
Καθηγητής Πανεπιστήμιου Πειραιά

Ιούλιος 2022

Ευχαριστίες

Πρώτα από όλα θα ήθελα να ευχαριστήσω την οικογένεια μου, την μητέρα μου Κωνσταντίνα Καψαλού, τον πατέρα μου Παναγιώτη Κανελλόπουλο και τον αδερφό μου Αναστάσιο Κανελλόπουλο για την υποστήριξη που μου έδειξαν καθ' όλη την διάρκεια των μαθητικών και φοιτητικών μου χρόνων.

Στην συνέχεια, θα ήθελα να εκφράσω την ευγνωμοσύνη μου σε όλους τους καθηγητές του τμήματος που μέσα από τα μαθήματα τους και τις προκλήσεις που έβαλαν να αντιμετωπίσω, με βοήθησαν να εξελιχθώ στο αντικείμενο του προγραμματισμού και της επιστήμης των υπολογιστών. Τέλος, θα ήθελα να ευχαριστήσω για ακόμη μία φορά τον επιβλέποντα καθηγητή μου κ. Ευθύμιο Αλέπη για την υποστήριξη και την κατανόηση που μου έδειξε, ώστε να με βοηθήσει να ολοκληρώσω την πτυχιακή μου εργασία.

Περίληψη-Σκοπός της εργασίας

Ο σκοπός της πτυχιακής μου εργασίας ήταν να φτιάξω μια εφαρμογή Android στην οποία θα καταγράφω τις οδηγικές συμπεριφορές των χρηστών της εφαρμογής, ενώ ταυτόχρονα θα τους δίνω την δυνατότητα να αξιοποιήσουν λειτουργίες GPS που παρέχονται από την Google. Οι χρήστες θα μπορούν σε πραγματικό χρόνο να δουν στατιστικά δεδομένα σχετικά με τον τρόπο που οδηγούν, όπως την ταχύτητα του οχήματος, την επιτάχυνση κ.λπ. Επίσης, θα μπορούν να εξετάσουν και κάποιες παρατηρήσεις σχετικά με τον τρόπο οδήγησης τους όπως για παράδειγμα τις απότομες επιταχύνσεις και τα απότομα φρεναρίσματα που κάνουν σύμφωνα με τους υπολογισμούς της εφαρμογής. Τέλος, θα έχουν την δυνατότητα να χρησιμοποιήσουν λειτουργίες όπως αυτές παρέχονται από το Google Maps π.χ. αναζήτηση τοποθεσίας, προσδιορισμός της βέλτιστης διαδρομής προς κάποιον προορισμό κ.λπ.

Η διακίνηση των δεδομένων των χρηστών θα γίνεται μέσω του Spring Boot Backend, με το οποίο εκτελώντας CRUD(Create, Read, Update, Delete) operations θα αποθηκεύονται πληροφορίες για τους εγγεγραμμένους στην εφαρμογή χρήστες, καθώς και στατιστικά στοιχεία που συλλέγονται μετά από κάθε κούρσα τους. Τα δεδομένα αυτά μπορούν να χρησιμοποιηθούν για να βγάλουμε χρήσιμα συμπεράσματα όπως ποιοι χρήστες χρίζουν βελτίωσης ως προς την οδηγική τους συμπεριφορά, να εντοπιστούν ποιοι δρόμοι είναι ενδεχομένως ποιο επικίνδυνοι για οδήγηση κ.α.

Για την υλοποίηση της Android εφαρμογής χρησιμοποιήθηκε το Android Studio σε γλώσσα προγραμματισμού Java, με το οποίο έχουμε δημιουργήσει το επίπεδο παρουσίασης της εφαρμογής και το Sprint Boot το οποίο χειρίζεται το Backend.

Περιεχόμενα

Spring Boot Backend

Τί είναι το Spring Boot

Το Spring Boot είναι ένα Java framework το οποίο μας διευκολύνει στην δημιουργία εφαρμογών/υπηρεσιών που τρέχουν στο διαδίκτυο με την αρχιτεκτονική των Micro Services(Διαχωρισμός της παραγωγής σε επίπεδα/layers το καθένα από τα οποία εκτελεί μια συγκεκριμένη λειτουργία). Ο βασικός του στόχος είναι να μειώσουμε τον χρόνο παραγωγής καθιστώντας ταχύτερη την διαδικασία αποθήκευσης και μετάδοσης των δεδομένων της εφαρμογής. Χρησιμοποιεί ένα ενσωματωμένο Tomcat Java Servlet Container το οποίο μας μας επιτρέπει να τρέχουμε web applications όπως μια οποιαδήποτε εφαρμογή σε Java.

Dependency Injection

Το Dependency Injection χρησιμοποιείται για να προσθέσουμε όλες τις απαραίτητες άδειες/dependencies ώστε να λειτουργήσει το Spring Boot πρόγραμμα μας όπως εμείς το επιθυμούμε. Κάνοντας μετάβαση στον ιστότοπο https://start.spring.io/ μπορούμε να δημιουργήσουμε ένα νέο Spring Boot Project και στο πεδίο Dependencies προσθέτουμε όλες τις άδειες που θα χρησιμοποιήσουμε. Τα dependencies που έχουμε προσθέσει φαίνονται στο pom.xml του Spring προγράμματος μας. Συγκεκριμένα, κάποια από τα σημαντικά dependencies που έχω χρησιμοποιήσει είναι:

1) ***Spring Starter Test,*** περιέχει μια H2 memory database για να κάνουμε test χωρίς χρήση κάποιας άλλης βάσης δεδομένων.
2) ***PostgreSQL Driver***, που επιτρέπει να συνδεθώ με μια PostgreSQL database.
3) ***MS SQL Server Driver*****,** ομοίως μου επιτρέπει να συνδεθώ σε SQL Server database.
4) ***Spring Data JPA***, το χρησιμοποιώ για να κάνω σύνδεση και για να επιδρώ στην βάση δεδομένων μου.
5) ***Spring Web***, το οποίο μου επιτρέπει να τρέχω web applications και συνοδεύεται με ένα ενσωματωμένο Tomcat για να κάνω με εύκολα testing στις εφαρμογές μας.
6) ***Lombok***, που μέσω των κατάλληλων annotation '@' δημιουργεί Getters, Setters, Constructors κ.λπ.
7) ***Spring-Boot-Maven-Plugin,*** μου επιτρέπει να δημιουργήσω το εκτελέσιμο αρχείο του Spring Boot Project σε μορφή .jar file, χρησιμοποιώντας την εντολή: ***mvnw package*** στο Command Prompt για Windows. Στην συνέχεια μπορούμε να τρέξουμε αυτό το εκτελέσιμο αρχείο, είτε κάνοντας διπλό κλικ στο .jar αρχείο είτε με την εντολή: ***java -jar filename.jar***

Αρχιτεκτονική του Spring Boot

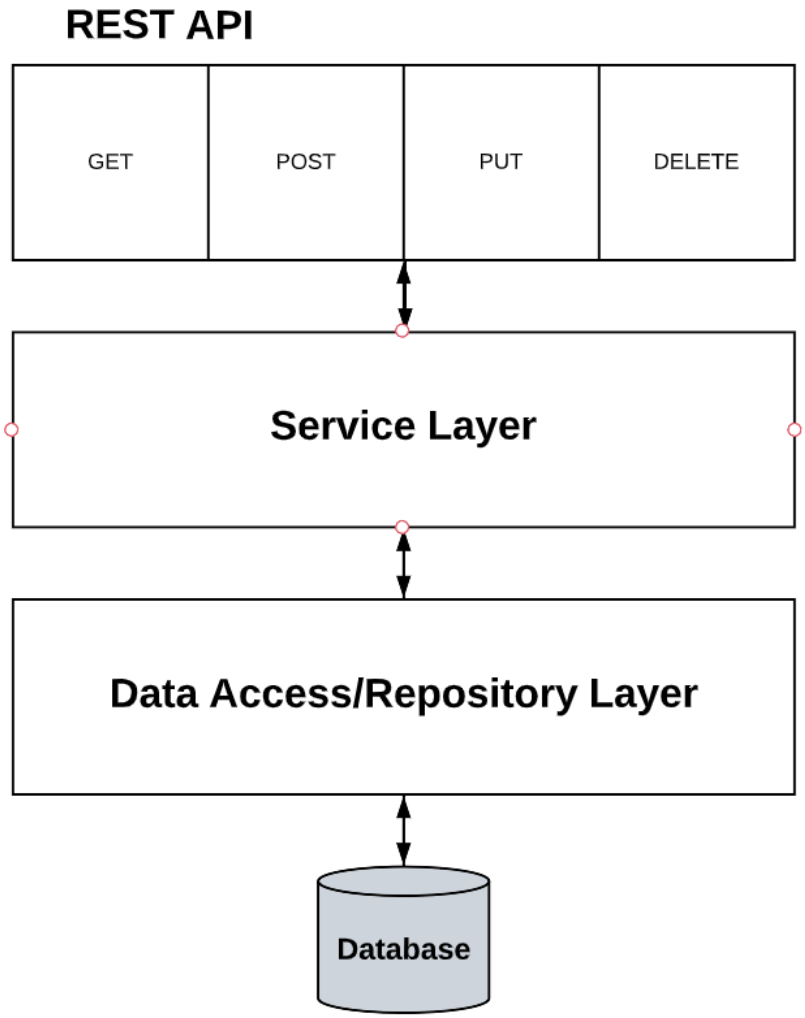
Το Spring Boot χωρίζεται σε επίπεδα/layers τα οποία μας βοηθούν να βελτιστοποιήσουμε την απόδοση του κώδικα, να επιταχύνουμε την διαδικασία παραγωγής, να έχουμε καλύτερη οργάνωση στα προγράμματα και γενικότερα να κάνουμε την «ζωή» των προγραμματιστών πιο εύκολη. Τα επίπεδα χωρίζονται σε:

1. ***Controller Layer*****:** Είναι το επίπεδο στο οποίο χειριζόμαστε όλα τα requests που παίρνουμε από το Front-End της εφαρμογής, δηλαδή από το Android Studio αλλά είναι ταυτόχρονα και ο δρομολογητής που θα επιστρέφει όλες τις πληροφορίες πίσω στο Front End δηλαδή στους χρήστες.

2. ***Service/Business Layer*****:** Το επίπεδο αυτό περιέχει όλη την λογική με την οποία θα διακινούμε τα δεδομένα από το Front End στο Back End της εφαρμογής. Είναι ο «μεσάζοντας» ανάμεσα στο Controller Layer και στο τρίτο επίπεδο το Data Access/Repository Layer.

3. ***Data Access/Repository Layer*:** Αυτό το επίπεδο θα είναι υπεύθυνο για την επικοινωνία με την βάση δεδομένων. Θα εκτελεί όλες τις εντολές που προσθέτουν ή συλλέγουν δεδομένα από την βάση δεδομένων.

Έχω χωρίσει το Spring Boot project μου σε δύο βασικά μέρη: Σέ έναν φάκελο που περιέχει ότι κώδικα είναι σε άμεση σχέση με τους χρήστες της εφαρμογής και σε ένα φάκελο που περιέχει την λογική με την οποία καλώ το WeatherAPI https://www.weatherapi.com/ το οποίο είναι ένα API που χρησιμοποιώ στην εφαρμογή μου για να εμφανίζω την τωρινή θερμοκρασία.



User-Controller Layer

Το Controller Layer για τους χρήστες περιέχει μια κλάση την UserController που είναι annotated ως @RestController. Με αυτό θα έχουμε την δυνατότητα να φτιάξουμε restful web services (Post, Get, Put, Delete operations), μπορούμε δηλαδή να χειριστούμε όλα τα http requests που γίνονται από την εφαρμογή. Κάθε εντολή θα πρέπει να έχει το δικό της url για να εξυπηρετηθεί. Όλες οι εντολές θα ακολουθούν μια επέκταση του βασικού/base url που κανονικά είναι το domain name του

ιστότοπου αλλά εδώ είναι το localhost:8080 επειδή βρισκόμαστε σε τοπικό επίπεδο. Μερικά http requests που μπορεί να ζητηθεί να εκτελεστούν είναι

1) **Δημιουργία καινούργιου χρήστη της εφαρμογής**. Στην προκειμένη περίπτωση μιλάμε για ένα POST request με το οποίο θέλουμε να προσθέσουμε ένα χρήστη στην database. Για τον σκοπό αυτό θα μεταβούμε στο localhost/users ως εξής: **@PostMapping**(/users) οπού εκεί εκτελείτε η συνάρτηση saveUser που με την σειρά της από το Controller layer θα περάσει τον έλεγχο στο Service layer η λειτουργία του οποίου θα συζητηθεί αναλυτικότερα αργότερα. Εδώ θεωρώ ότι είναι σημαντικό να επικεντρωθώ στη χρήση του **@Autowired** annotation το οποίο μας επιτρέπει να κάνουμε αρχικοποίηση του αντικειμένου της κλάσης UserServiceImplementation χωρίς να χρειάζεται να τρέξουμε το new operator που δημιουργεί ένα instance της UserServiceImplementation. Βασική προϋπόθεση για να μπορούμε να χρησιμοποιήσουμε το **@Autowired** είναι ότι η κλάση του αντικειμένου είναι annotated ως **@Service** δηλαδή είναι ένα Spring Bean.
2) **Εμφάνιση του μέσου όρου των μετρήσεων για τους χρήστες της εφαρμογής.** Μπορούμε να πάρουμε το επιθυμητό αποτέλεσμα κάνοντας ένα GET request ως εξής: **@GetMapping**(users/getGlobalAverageInformation). Στα GET request δεν είναι δυνατή η χρήση κάποιας παραμέτρου στο σώμα της συνάρτησης, αλλά υπάρχει τέτοια ανάγκη.

Αντίστοιχα εκτελούνται οι λειτουργίες για PUT και DELETE Requests. Περισσότερα παραδείγματα εφαρμογών CRUD request θα δούμε στο Service Layer.

User-Service Layer

Όπως είπαμε, στο Service Layer είναι εκεί που περιέχεται όλη η λογική με την οποία διαχειριζόμαστε τα δεδομένα της εφαρμογής. Για τους χρήστες έχω χωρίσει το Service Layer σε δύο κλάσεις:

1) Το **UserService**, το οποίο είναι ένα Interface στο οποίο επονομάζω όλες τις λειτουργίες που θα πρέπει να εξυπηρετηθούν από το Service Layer.

2) Το **UserServiceImplementation**, το οποίο εκτελεί τις λειτουργίες που αναφέρει το UserService.

Ανέφερα προηγουμένως ότι με το Post Request και την εκτέλεση της συνάρτησης saveUser, θα περάσουμε τον έλεγχο από το Controller Layer στο Service Layer. Εδώ είναι που θα φτιάξουμε την λειτουργία αποθήκευσης του χρήστη στην database, ως εξής:

```java
@Override
public User saveUser(User user) {

    List<User> users = userRepository.findAll();

    for(User currUser : users)
    {
        if(currUser == user) {
            System.out.println("User Already exists!");
            return null;
        }
        else if(currUser.getEmail().equals(user.getEmail()))
        {
            System.out.println("A user with email already exists");
            return null;
        }
    }

    return userRepository.save(user);
}
```

Κάνουμε τον έλεγχο ότι δεν υπάρχει άλλος χρήστης στην εφαρμογή με τα ίδια στοιχεία προτού των προσθέσουμε στην βάση.

Μερικές ακόμη εργασίες που καταγράφονται από το Service Layer για τους Χρήστες είναι :

- Το ιστορικό που έχει ο κάθε χρήστης για κάθε κούρσα που έχει κάνει.
- Ο υπολογισμός των κατά μέσο όρο στατιστικών του κάθε χρήστη για όλες του τις κούρσες.
- Η μέτρηση συνολικού μέσου όρου των στατιστικών όλων των χρηστών της εφαρμογής.
- Η καταγραφή «Επικίνδυνων Τοποθεσιών» στις οποίες έχει οδηγήσει ο χρήστης

User-Repository Layer

Φτιάχνω ένα Interface για κάθε table που υπάρχει στο database. Κάθε τέτοιο interface θα είναι annotated με το **@Repository** το οποίο σηματοδοτεί ότι η συγκριμένη κλάση θα χρησιμοποιηθεί για διακίνηση δεδομένων σε και από μια βάση δεδομένων. Κάνω extend το JpaRepository το οποίο μου επιτρέπει να εκτελέσω CRUD Operations πάνω στους πίνακες της βάσης μου. Το JpaRepository παίρνει σαν όρισμα μια κλάση τύπου T και τον τύπο της μεταβλητής που είναι το primary key του αντίστοιχου πίνακα.

Για παράδειγμα, υπάρχει η κλάση User η οποία περιέχει της πληροφορίες για τον κάθε χρήστη (Ονοματεπώνυμο, ημερομηνία γέννησης, email και κωδικός πρόσβασης). Έχω χρησιμοποιήσει τα εξής annotation:

1) Το **@Data**, **@EqualsAndHashCode**, **@NoArgsConstructor** του Lombok dependency το οποίο δημιουργεί αυτόματα όλα τα Getters, Setters το Constructor με όλες τις παραμέτρους της κλάσης, το Constructor χωρίς παραμέτρους, την String class κ.λπ.

2) Το **@Entity** annotation το οποίο επισημαίνει ότι η κλάση είναι ένα entity και έχει χαρτογραφηθεί σε ένα πίνακα στην βάση δεδομένων.

3) Το **@Table** annotation που μου επιτρέπει να καθορίσω λεπτομέρειες για τον πίνακα που θα είναι αποθηκευμένος στην βάση, όπως π.χ. το όνομα του πίνακα στην βάση(name = “user_table”)

4) Μέσα στην κλάση πρέπει να καθορίσω την σχέση που θα υπάρχει ανάμεσα στα δεδομένα. Σε αυτό το table θα ορίσω ως πρωτεύον κλειδί ένα self-incrementing id τύπου Long, το userId για να μπορώ να ξεχωρίσω τον κάθε χρήστη στον πίνακα. Για να πετύχω την λειτουργία αυτή, χρησιμοποιώ το **@Id** annotation το οποίο ορίζει την μεταβλητή userId ως primary key του user_table.

Για την κλάση User υπάρχει και το αντίστοιχο interface και συγκεκριμένα το UserRepository. Έχει την δομή που περιγράψαμε προηγουμένως και περνάει σαν παράμετρο στο JpaRepository το User για την κλάση και το Long το οποίο αντιστοιχεί στο userId που είναι το primary key του πίνακα των User.

Weather API

Για το Weather API απαιτείται πολύ λιγότερη ανάλυση αφού το μόνο που χρειάζεται είναι ένας Controller ο οποίος θα κάνει ένα POST request για να παίρνουμε την θερμοκρασία με βάση την τοποθεσία που έχει περάσει από το στίγμα του χρήστη. Το στίγμα θα περαστεί σαν παράμετρος στο url που θα κάνουμε το POST request και έχει την εξής δομή:

```
String url =
"http://api.weatherapi.com/v1/forecast.json?key=6ab7944a41754c46b90105507221903&q="+

latAndLong.getLatitude()+","+latAndLong.getLongitude()+"&days=1&aqi=no&alerts=no";
```

Οπού latAndLong είναι αντικείμενο της helper κλάσης LatAndLong με την οποία τραβώ το γεωμετρικό πλάτος και μήκος του χρήστη.

Μερικές ακόμη συναρτήσεις που υλοποιώ στο Spring Boot

- ***@GetMapping("/users")***
  List<User> fetchUserList() : Επιστρέφει μια λίστα με τις πληροφορίες για όλους τους χρήστες της εφαρμογής. Χρησιμοποιεί την συνάρτηση την συνάρτηση fetchUserList() του User Service.

- ***@GetMapping("/users/{id}")***
  User fetchUserById(@PathVariable("id") Long userId) : Με την οποία μπορώ να αναζητήσω την ύπαρξη ενός χρήστη στην βάση δεδομένων. Το **@PathVariable** annotation μας δίνει πρόσβαση στο πεδίο {id} που περιγράφει το userId του χρήστη. Χρησιμοποιεί την συνάρτηση userService.fetchUserById(Long userId).

- ***@PostMapping("/users/drive")***
  UserStatus saveDriveDetais(@Valid @RequestBody DriveResults driveResults): Αποθηκεύει τα αποτελέσματα μιας κούρσας του χρήστη. Το αποτέλεσμα που επιστρέφει είναι ένα Enum τύπου UserStatus το οποίο επιστρέφει “Success” για επιτυχή αποθήκευση, “Failure” για αποτυχία αποθήκευσης.

- ***@DeleteMapping("/users/{id}")***
  UserStatus deleteUserById(@PathVariable("id") Long userId) : Με αυτήν την συνάρτηση μπορώ να διαγράψω τον χρήστη με userId = {id} από την βάση δεδομένων.

- ***@PostMapping("users/getAverageInformation")***
  AverageDriveResults getUserAverageInformation(@Valid @RequestBody int id): Με την συνάρτηση αυτή επιστρέφω ένα object της κλάσης AverageDriveResults. Το object αυτό θα περιέχει τον στατιστικό μέσο όρο ενός χρήστη για όλες τις κούρσες που έχει κάνει όπως την μέση ταχύτητα, μέση απόσταση που διανύει σε κάθε κούρσα κ.λπ.

- ***@GetMapping("users/getGlobalAverageInformation")***
  GlobalAverageDriveResults getGlobalAverageInformation(): Με αυτήν την συνάρτηση επιστρέφω ένα object της κλάσης GlobalAverageDriveResults. Το αποτέλεσμα αυτό το “τραβάω” από έναν πίνακα στον Server που περιέχει μια και μοναδική εγγραφή. Αυτή η

εγγραφή είναι ο στατιστικός μέσος όρος για όλους τους χρήστες από όλες τους τις κούρσες.

Σύνδεση με βάση δεδομένων σε Server

Για να συνδεθούμε με την βάση δεδομένων που θα περιέχει τους πίνακες του προγράμματος μου, πρέπει να επιλέξω πρώτα έναν Server στον οποίο θα βρίσκεται η βάση μου. Υπάρχουν διάφοροι Server που θα μπορούσα να φτιάξω την βάση μου, όπως SQL Server, MySQL Server Postgres SQL Server κ.λπ. Για τις ανάγκες της δικής μου εφαρμογής δημιούργησα την σύνδεση δοκιμάζοντας τους SQL Server και Postgres SQL Server. Για να εγκατασταθεί η σύνδεση με την βάση ακολούθησα την παρακάτω διαδικασία:

1) Στο application.properties που βρίσκεται στο μονοπάτι /src/main/resources/ application.properties του project πρέπει να θέσουμε τα κατάλληλα configuration ανάλογα με τον server που θέλουμε να συνδεθούμε.

   Για να συνδεθούμε με **Postgres SQL Server**: χρησιμοποιούμε το παρακάτω configuration:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/driving_behaviour
spring.datasource.username=postgres
spring.datasource.password=johnt
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.format_sql=true
server.error.include-message=always
```

   Χρησιμοποιούμε το port 5432 για να συνδεθούμε στο local Postgres SQL Server που θα περιέχει την βάση μας με όνομα driving_behaviour. Σαν username το postgres είναι default value και το password το καθορίζουμε εμείς και πρέπει να είναι ίδιο με τον κωδικό που έχουμε εμείς ορίσει για τον χρήστη postgres. Για να αλλάξουμε τον κωδικό, μπορούμε μέσω του λογισμικού PgAdmin4(ή οποιουδήποτε άλλου management tool της Postgres) στην καρτέλα SQL να τρέξουμε την ακόλουθη εντολή:

   ALTER USER postgres WITH PASSWORD 'johnt';

   Το ***spring.jpa.hibernate.ddl-auto=create-drop*** υποδηλώνει ότι οι πίνακες θα γίνουν πάλι initialize μόλις κάνουμε μια καινούργια εκτέλεση του Spring Boot Project.

Για να συνδεθούμε με **SQL Server** η διαδικασία είναι ανάλογη, πρέπει να ορίσουμε τα εξής στο application.properties:

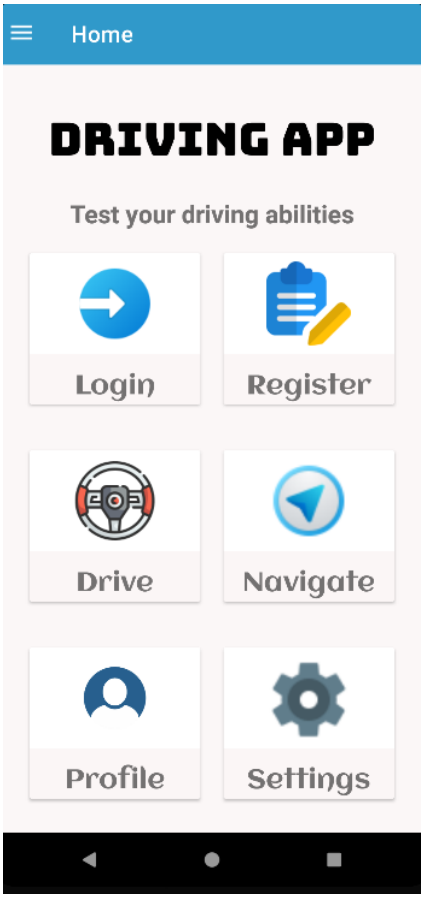
```
spring.datasource.url=jdbc:sqlserver://localhost;databaseName=driving_behaviour
createDatabaseIfNotExist=true
spring.datasource.username=sa
spring.datasource.password=johnt
spring.datasource.driverClassName=com.microsoft.sqlserver.jdbc.SQLServerDriver
spring.jpa.show-sql=true
spring.jpa.hibernate.dialect=org.hibernate.dialect.SQLServer2012Dialect
spring.jpa.hibernate.ddl-auto = create-drop
```

Εδώ χρησιμοποιούμε το url για να συνδεθούμε σε SQL Server. Το username **sa** ορίζεται από default στον SQL Server και password πρέπει να είναι ίδιος με τον κωδικό που επιλέξαμε κατά την εγκατάσταση του Server.

2) Για να συνδεθούμε με την βάση μας όποια και αν είναι, πρέπει πρώτα να δημιουργήσουμε την βάση στον αντίστοιχο server επειδή, το Spring Boot δεν υποστηρίζει αυτόματη δημιουργία για την βάση κατά την εκτέλεση του προγράμματος.

Εφαρμογή Android

Αρχική οθόνη:



Εδώ είναι το Front End της εφαρμογής, δηλαδή το τελικό αποτέλεσμα που θα βλέπει ο χρήστης. Θα χωρίσω την παρουσίαση αυτού του κομματιού της εργασίας σε 4 σημεία:

1) ***Σύνδεση με Spring Boot – Retrofit,*** που θα αποκαλύψω πώς μεταφέρονται τα δεδομένα από τον Client-Android Studio, δηλαδή το Frontend, στον Server δηλαδή στο Spring Boot Backend.

2) ***Οδηγική Συμπεριφορά***, στο οποίο θα γίνει αναφορά στους υπολογισμούς που διεξάγονται κατά την εκτέλεση της εφαρμογής για τον τρόπο οδήγησης του εκάστοτε χρήστη.

3) ***GPS λειτουργίες,*** εδώ θα μιλήσω για τις λειτουργίες GPS που χρησιμοποιούνται στην εφαρμογή καθώς και για τα διάφορα API της Google που καλούνται για την υλοποίηση των παραπάνω λειτουργιών.

4) ***Υπόλοιπες λειτουργίες,*** οπού θα παρουσιάσω υλικό κυρίως σχεδιαστικού χαρακτήρα UI της εφαρμογής κ.λπ.

Σύνδεση με Spring Boot - Retrofit

Πριν μπω σε λεπτομέρειες για το σημαντικό κομμάτι της εργασίας, θα ήθελα να αναφέρω πρώτα πώς μεταφέρω τα δεδομένα από το συσκευή που χρησιμοποιεί την εφαρμογή στην βάση δεδομένων και στους υπόλοιπους server στους οποίους γίνονται κλήσεις κατά την διάρκεια της εφαρμογής. Για να το καταφέρω αυτό χρησιμοποιώ το ***Retrofit*** που είναι ένα HTTP Client για να κάνω POST/GET/PUT/DELETE requests. Για να το χρησιμοποιήσουμε πρέπει να θέσουμε ένα base url που θα είναι το μονοπάτι στο οποίο όλες οι κλήσεις θα πρέπει να ακολουθήσουν για να φτάσουν στην λειτουργία που θέλουμε να εκτελεστεί π.χ. θα πρέπει να δηλώσουμε το **http://www.foo.com/** σαν base url για να μπορούμε να φτάσουμε στο **http://www.foo.com/something** κ.λπ.

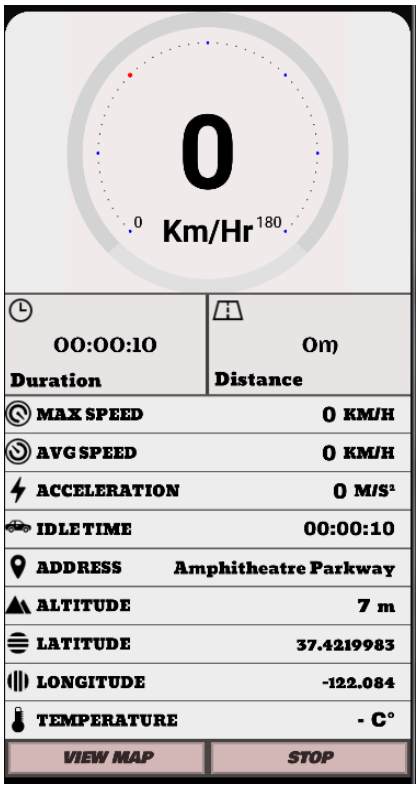
Εδώ είναι σημαντικό να αναφέρω ότι για να μπορέσω να εγκαταστήσω την σύνδεση ανάμεσα στην βάση δεδομένων μου με την Android εφαρμογή πρέπει να έχω πρόσβαση στην IP διεύθυνση του τοπικού δικτύου της συσκευής που έχει ενεργοποιήσει το Spring Boot project. Δηλαδή, για να κάνω κλήση στο *url_path/users/login* που είναι το url path για ένα χρήστη που θέλει να κάνει login στην εφαρμογή, πρέπει να ορίσω σαν base url στο retrofit client το *url_path.* Το *url_path* θα είναι η IPv4 διεύθυνση εκείνης της συσκευής. Πρέπει επίσης το κινητό που τρέχει την εφαρμογή να είναι στο ίδιο δίκτυο με την συσκευή που τρέχει το Spring Boot κώδικα. Χωρίς τα παραπάνω δεν μπορεί να υπάρξει επικοινωνία ανάμεσα στο server που βρίσκεται η βάση δεδομένων και την εφαρμογή. Υπενθυμίζω ότι την IPv4 μπορούμε να την βρούμε σε περιβάλλον Windows τρέχοντας την εντολή ipconfig.

Οδηγική Συμπεριφορά

Περιγραφή

Βασική προϋπόθεση για να έχουν νόημα τα αποτελέσματα των υπολογισμών της εφαρμογής είναι ότι ο χρήστης βρίσκεται σε περιβάλλον οδήγησης και ότι το κινητό θα είναι σε κάποιο

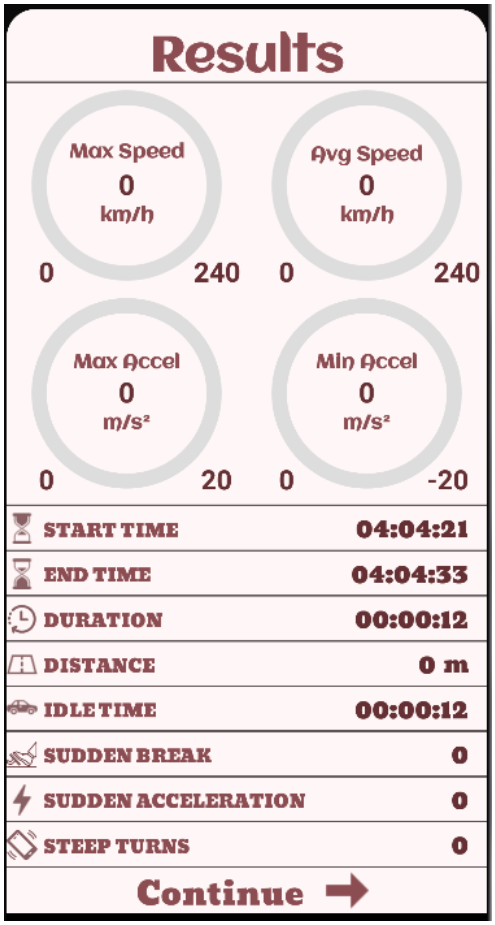
σταθερό σημείο όπως μια βάση κινητού αυτοκινήτου. Η βασική φόρμα στην οποία διεξάγονται όλοι οι υπολογισμοί για την οδηγική συμπεριφορά των χρηστών είναι η παρακάτω:



Όπως βλέπουμε γίνονται διάφορες μετρήσεις:

1) ***Ταχύτητα***, το αποτέλεσμα της οποίας μετριέται σε χιλιόμετρα ανά ώρα
2) ***Διάρκεια κούρσας***
3) ***Απόσταση*** που έχει διανυθεί, αυξάνεται σε πραγματικό χρόνο
4) ***Μέγιστη ταχύτητα***
5) ***Μέση ταχύτητα***
6) ***Επιτάχυνση***, θετική για αυξανομένη ταχύτητα, αρνητική για μείωση της ταχύτητας του οχήματος
7) ***Στασιμότητα***, πόση ώρα έμεινε στάσιμος ο χρήστης
8) ***Διεύθυνση***, απαιτεί πρόσβαση στο διαδίκτυο για να εμφανιστεί
9) ***Υψόμετρο***, υπολογίζεται από το Elevation API της Google
10) ***Γεωμετρικό Πλάτος*** και ***Γεωμετρικό Μήκος***
11) ***Θερμοκρασία***, μας τι προσφέρει το Weather API. Το request στο API γίνεται από το Spring Boot

Για να δει πιο ακριβείς μετρήσεις ο χρήστης πρέπει με τον τερματισμό της κούρσας του να πατήσει το κουμπί Stop με το οποίο θα μεταφερθεί στην φόρμα των αποτελεσμάτων:



Εκεί θα έρθει αντιμέτωπος με κάποιους ακόμη υπολογισμούς:

1) ***Απότομα φρεναρίσματα*** που έχει κάνει, δηλαδή οι φορές που η επιτάχυνση του έχει πέσει ποιο χαμηλά από -10m/s^2

2) ***Απότομες επιταχύνσεις***, οι φορές που η επιτάχυνση του χρήστη έχει φτάσει υψηλότερα από 10m/s^2

3) ***Απότομες στροφές / Κραδασμοί***, όταν το accelerometer εντοπίζει έντονους κραδασμούς. Πολύ σημαντικό εδώ είναι ο χρήστης να προσπαθήσει να κρατήσει το κινητό του σε κάποιο σταθερό σημείο όπως μια βάση κινητού, αλλιώς οι μετρήσεις θα επηρεαστούν δραστικά.

Υπολογισμοί μέσω της τοποθεσίας

Είναι σημαντικό να αναφέρω ότι για να λειτουργήσει η εφαρμογή πρέπει να αποδοθεί η άδεια πρόσβασης στην τοποθεσία της συσκευής, χωρίς αυτή η εφαρμογή δεν μπορεί να

λειτουργήσει. Για τον υπολογισμό της τοποθεσίας χρησιμοποιούνται 2 εργαλεία, ένα είναι το ***Location Listener*** interface το οποίο εντοπίζει αλλαγές στην τοποθεσία του χρήστη, και ομοίως το ***FusedLocationProviderClient***. Για το Location Listener:

1) Ορίζω ένα αντικείμενο της κλάσης ***LocationManager*** η οποία μου επιτρέπει να παίρνω περιοδικές ειδοποιήσεις για την τοποθεσία του χρήστη.

2) Αρχικοποιώ αυτό το αντικείμενο θέτοντας στην συνάρτηση requestLocationUpdates της κλάσης τις τιμές 0 για τον ελάχιστο χρόνο απόκρισης και 0 για την ελάχιστη απόσταση απόκρισης.

3) Στην συνάρτηση onLocationChanged του Location Listener είναι εκεί οπού θα γίνονται οι περισσότεροι υπολογισμοί.

Η ταχύτητα προκύπτει πολλαπλασιάζοντας το αποτέλεσμα της συνάρτησης Location.getSpeed()*3.6 επειδή το αποτέλεσμα που παίρνουμε μετριέται σε μέτρα/δευτερόλεπτο και γι’ αυτό το κάνουμε convert σε km/h.

Η μέση ταχύτητα υπολογίζεται διαιρώντας την απόσταση που έχουμε διανύσει με τον χρόνο που έχει διαρκέσει η κούρσα.

Η απόσταση υπολογίζεται προσθέτοντας κάθε φορά στην συνολική απόσταση, την ταχύτητα επί τον χρόνο που πέρασε για να πάρουμε το επόμενο Location Update.

Η μέγιστη ταχύτητα υπολογίζεται απλά με το να κρατάμε την μέγιστη ταχύτητα που έχει εμφανιστεί.

Η ταχύτητα χρησιμοποιείται επίσης και για τον υπολογισμό της επιτάχυνσης που παρατηρείται στο όχημα. Συγκεκριμένα, έχω δημιουργήσει έναν μετρητή μέσω της κλάσης Timer με τον οποίο μπορώ να προγραμματίσω την δημιουργία thread. Έχω δημιουργήσει ένα thread λοιπόν, το οποίο τρέχει ανά 1 δευτερόλεπτο και εκτελεί τις εξής λειτουργείες:

1) Κρατάει τον λογαριασμό της ώρας που διαρκεί η κούρσα και της στασιμότητας στην οδήγηση του χρήστη. Τα αποτελέσματα εμφανίζονται με την συνάρτηση ConvertSecondsToTime που αλλάζει το format από έναν ακέραιο σε Hours:Minutes:Seconds.

2) Χρησιμοποιείται για τον υπολογισμό της επιτάχυνσης. Ανά 1 δευτερόλεπτο, καλεί την συνάρτηση meausreAcceleration() η οποία παίρνει την διαφορά της ταχύτητας που έχει

τώρα το όχημα, με αυτήν που είχε πριν ένα δευτερόλεπτο. Έτσι, μπορούμε εύκολα να παρατηρήσουμε τις φορές που ο χρήστης επιτάχυνε ή φρέναρε απότομα.

Το FusedLocationProviderClient είναι μία κλάση που έχει δημιουργηθεί για τον ίδιο λόγο με το LocationListener interface. Στην προσπάθεια να καταλάβω ποια από τις δύο μεθόδους ήταν πιο ακριβείς στις μετρήσεις της, έβγαλα το συμπέρασμα ότι το FusedLocationProviderClient έχει χαμηλότερη ακρίβεια. Για αυτόν τον λόγο, δεν χρησιμοποίησα το FusedLocationProviderClient για τον υπολογισμό των παραπάνω(ταχύτητα, μέση ταχύτητα, επιτάχυνση). Από την άλλη όμως, είδα ότι είναι πιο γρήγορο ως προς τους χρόνους απόκρισης της από την onLocationChanged και για αυτό την χρησιμοποίησα για άλλους σκοπούς. Για να χρησιμοποιήσουμε την κλάση πρέπει πρώτα να ορίσουμε τα εξής:

1) Το FusedLocationProviderClient object μέσω της συνάρτησης *getFusedLocationProviderClient της κλάσης Location Services.*

2) Το Location Request που χρησιμοποιείται για να καθορίσουμε αν θα δώσουμε προτεραιότητα σε μεγαλύτερη ακρίβεια ή μεγαλύτερη εξοικονόμηση ενέργειας. Του δίνουμε default και fastest interval να είναι τα 2 δευτερόλεπτα

3) Το LocationCallback που είναι η αντίστοιχη onLocationChanged για τον FusedLocationProviderClient.

Χρησιμοποιούμε το LocationCallback για τις εξής λειτουργίες:

- Υπολογίζουμε το γεωμετρικό πλάτος και μήκος
- Βάσει το γεωμετρικό πλάτος και ύψος που υπολογίστηκε, καλούμε ανά 10 δευτερόλεπτα τις calculateAltitude() και calculateTemperature() από τις οποίες αντλούμε το υψόμετρο και την θερμοκρασία αντίστοιχα. Για το calculateAltitude όπως ανέφερα χρησιμοποιείται ειδικό API της Google το ElevationAPI, η λειτουργία του οποίου θα αναλύσω όταν μιλήσουμε για τις GPS λειτουργίες της εφαρμογής.
- Επίσης εμφανίζουμε και την διεύθυνση στην οποία βρίσκεται η συσκευή την συγκεκριμένη χρονική στιγμή.

Accelerometer

Για τους υπολογισμούς των κραδασμών/απότομων στροφών γίνεται χρήση ενός accelerometer και συγκεκριμένα του Sensor Manager του Android. Αφού ορίσουμε ένα object τύπου SensorManager που μου δίνει πρόσβαση στους αισθητήρες του κινητού και ένα object τύπου Sensor δίνοντας του ως τύπο το accelerometer, τρέχουμε τον SensorEventListener ο οποίος χρησιμοποιείται για τον εντοπισμό αλλαγών στο sensor. Σε αυτόν, υπολογίζουμε την κίνηση στους άξονες x,y,z. Έπειτα, παίρνουμε το άθροισμα των τετραγώνων αυτών των τιμών και υπολογίζουμε την τετραγωνική ρίζα του αθροίσματος. Η διαφορά του τωρινού αποτελέσματος με αυτό που είχα στο προηγούμενο interval είναι η αλλαγή που έχουμε στην επιτάχυνση. Με βάση αυτήν την τιμή εντοπίζουμε ακραίες αλλαγές στον αισθητήρα κίνησης του κινητού και συγκεκριμένα, αν αυτή η τιμή είναι >=12 σημαίνει ότι μιλάμε για ένα δυνατό κραδασμό.

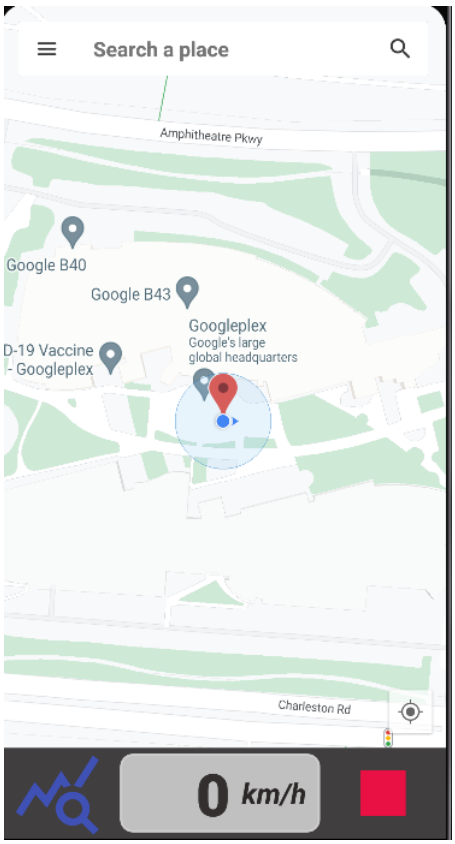
Τις τοποθεσίες στις οποίες βρήκαμε κραδασμούς, τις στέλνουμε στον server μας οπού εκεί μπορούμε να τις αναλύσουμε για να δούμε αν είναι τυχαίο ότι υπήρξε μεγάλη αλλαγή στην κίνηση της συσκευής ή εφόσον υπάρχει μεγάλο δείγμα από κραδασμούς στην συγκεκριμένα, φταίει η σχεδίαση του δρόμου(Είναι πολύ απότομη η στροφή ή ο δρόμος χρειάζεται επιδιόρθωση κ.λπ.).

GPS λειτουργίες

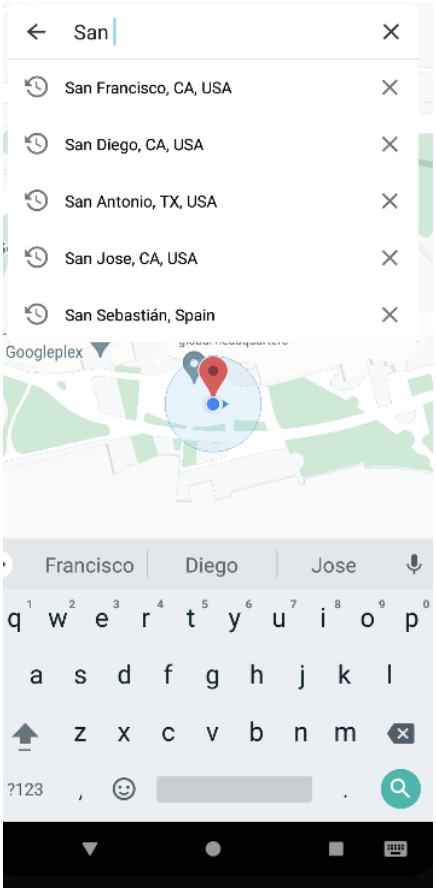
Όπως ανέφερα και προηγουμένως, η εφαρμογή υποστηρίζει και μερικές λειτουργίες GPS. Οι λειτουργίες αυτές προσφέρονται από διάφορα API του Google Cloud platform όπως το DirectionsAPI για τον υπολογισμό βέλτιστης διαδρομής σε χάρτη κ.λπ. Πρέπει να επισημάνω ότι για να μπορώ να κάνω κλήση στα παραπάνω API πρέπει να έχω δημιουργήσει billing account, το οποίο έρχεται με μια ελάχιστη μηνιαία χρέωση μετά από ένα μεγάλο αριθμό κλήσεων σε αυτά τα API, αν δηλαδή υπήρχαν πολύ χρήστες που χρησιμοποιήσαν την εφαρμογή. Τα Google API's που καλούνται είναι:

- ***Directions API***, για τον υπολογισμό της βέλτιστης διαδρομής σε χάρτη
- ***Distance Matrix API,*** για τον υπολογισμό ώρα άφιξης σε προορισμό
- ***Places API,*** για να βρούμε μέρη βάσει της τοποθεσίας μας
- ***Elevation API,*** που επιστρέφει το υψόμετρο στο οποίο βρισκόμαστε

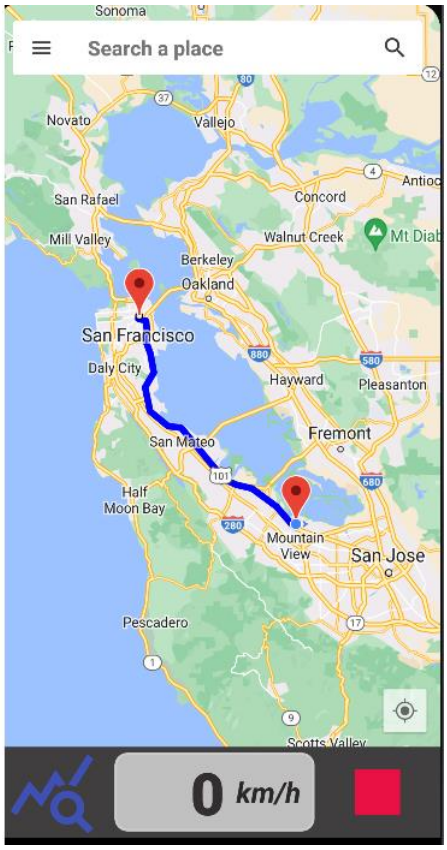
Αρχική οθόνη:



Αναζήτηση τοποθεσίας μέσω του PlacesAPI(Από τον emulator πάντα η τοποθεσία μας θεωρείται πως είναι στα γραφεία της Google):



Υπολογισμός βέλτιστης διαδρομής με το DirectionsAPI για οδήγηση:



Η κλήση σε όλα τα API της Google γίνονται απευθείας από ένα άλλο instance της κλάσης Retrofit στην οποία έχω θέσει σαν base url https://maps.googleapis.com/ το οποίο επεκτείνω για να κάνω κλήση σε όλα τα API. Δηλαδή δεν στέλνω πρώτα στο Spring Boot για να κάνει εκείνο τα request.

Τέλος κάποια πράγματα που πρέπει να επισημάνω είναι ότι το κάτω αριστερά μπλε κουμπί χρησιμοποιείται για να κάνουμε toggle ανάμεσα στην φόρμα των υπολογισμών και της ίδιας της φόρμας. Οι υπολογισμοί για την οδηγική συμπεριφορά του χρήστη συνεχίζονται και σε αυτήν την φόρμα στο background. Το κόκκινο κουμπί κάτω δεξιά είναι για να σταματήσουμε την κούρσα και να δούμε τα αποτελέσματα. Τέλος, το κουμπί για την τοποθεσία που είναι το κουμπί πάνω από το κόκκινο κουμπί του τερματισμού χρησιμοποιείται για να κάνουμε zoom in στην τοποθεσία του χρήστη αλλά και για να κλειδώσουμε την κάμερα ώστε να ακολουθεί τον χρήστη. Πατώντας πάλι το ίδιο κουμπί θα αποδεσμεύσει την κάμερα από την τοποθεσία του χρήστη και θα κάνει zoom out.

Directions API

Ο τρόπος με τον οποίο γίνεται κλήση σε όλα αυτά τα API είναι παρόμοιος και είναι μια διαδικασία στην οποία αρχικά «χτίζω» ένα url με κατάλληλες παραμέτρους. Το url αυτό το χρησιμοποιώ σαν Base Url στην κλάση Retrofit Client και κάνω request στο Server του Google Cloud. Ο Server τελικά μου επιστρέφει ένα Json object το οποίο πρέπει να «καταναλώσω» δημιουργώντας τις Java κλάσεις ώστε να αναπαραστήσω κατάλληλα την δομή του Json που μου δόθηκε.

Στην περίπτωση του DirectionsAPI πρέπει να κάνω request σε ένα url της μορφής ***http://maps.googleapis.com/maps/api/directions/json?destinations=MyDestination&origins=My Location&key=MyAPIkey***. Στο παραπάνω μπορώ να θέσω και άλλες παραμέτρους όπως το mode να είναι μόνο για driving/οδήγηση κ.λπ. Για να καταναλώσω αυτό το Json και οποιοδήποτε άλλο Json Object στην εφαρμογή χρησιμοποιώ την εξής τακτική:

1) Δημιουργώ μια κλάση για κάθε πεδίο του Json
2) Αν ένα πεδίο είναι της μορφής “Class” : “value” απλά δημιουργώ ένα αντικείμενο αυτής της Class
3) Αν είναι πεδίο της μορφής “Class” : […] τότε ορίζω μια λίστα της Class
4) Κάνω annotate κάθε μεταβλητή της κλάσης ως JsonObject

Με τα παραπάνω παίρνω όλη τη χρήσιμη πληροφορία που μου δίνεται από το Json Object. Στην συγκεκριμένη περίπτωση είναι μια λίστα της κλάσης Routes που περιέχει μια λίστα της κλάσης Legs, που με την σειρά της περιέχει λίστα της κλάσης Steps και τέλος αυτή περιέχει την συνάρτηση getPolylines(). Η συνάρτηση αυτή επιστρέφει ένα αντικείμενο τύπου Polyline που θα το κάνω decode μέσω της κλάσης DecodePolyline() και τελικά θα έχω ότι χρειάζομαι για να

ζωγραφίσω στον χάρτη την βέλτιστη διαδρομή στον προορισμό από την τοποθεσία που βρίσκεται εκείνη την στιγμή ο χρήστης.

Distance Matrix API

Το Distance Matrix API χρησιμοποιείται για τον υπολογισμό της απόστασης και της ώρα που χρειάζονται για να φτάσουμε από μια αρχική τοποθεσία/origin σε ένα προορισμό/destination. Επίσης έχει την ικανότητα να επιλέξει την βέλτιστη διαδρομή με βάση την κίνηση που υπάρχει σε πραγματικό χρόνο.

Ο τρόπος με τον οποίο κάνουμε κλήση στο συγκριμένο API έχει την εξής μορφή:

`https://maps.googleapis.com/maps/api/distancematrix/`*`outputFormat`*`?`*`parameters`*

Οπού ως outputFormat θα επιλέγουμε πάντα το Json format και σαν παραμέτρους ορίζουμε το source δηλαδή την τοποθεσία μας την συγκεκριμένη χρονική στιγμή και το destination δηλαδή την επιλογή που κάναμε στο search bar.

Η χρήσιμη πληροφορία που παίρνουμε από το Distance Matrix API είναι η προβλεπόμενη ώρα η οποία θα μας πάρει για να φτάσουμε στον προορισμό μας. Το αποτέλεσμα αυτό το παίρνουμε αφού πρώτα:

1) Δημιουργήσουμε μια κλάση για να αναπαραστήσουμε το rows field του Json object

2) Μέσα στο rows θα ορίσουμε μια κλάση που θα αναπαριστά το πεδίο elements

3) Τέλος, μέσα στο πεδίο elements θα υπάρχει μια κλάση Duration η οποία περιέχει τα πεδία text που είναι η ώρα ταξιδιού σε μορφή String και το πεδίο value που είναι η ίδια ώρα αλλά σε μορφή Integer.

Όταν ο χρήστης επιλέξει μια τοποθεσία αυτομάτως θα γίνει κλήση στο Distance Matrix API που επιστρέψει στον χρήστη της εφαρμογής την προβλεπόμενη ώρα ταξιδιού.

Places API

Το Places API έχει διάφορες χρήσιμες δυνατότητες όπως προτάσεις για τοποθεσίες με βάση την τοποθεσία του χρήστη, πληροφορίες για μια συγκεκριμένη τοποθεσία αν υπάρχουν όπως το

τηλέφωνο ή email αν μιλάμε για επιχείρηση κ.λπ. Το μόνο που χρησιμοποιώ από τις πολλές λειτουργίες που προσφέρει είναι το place autocomplete δηλαδή εμφάνιση προτάσεων για τοποθεσίες με βάση το τι πληκτρολογεί ο χρήστης στην μπάρα αναζήτησης. Για να κάνουμε χρήση του Places Api και συγκεκριμένα του autocomplete feature πρέπει να κάνουμε εγκυροποίηση του Google API key μας με την συνάρτηση Places.Initialize() και να δημιουργήσουμε ένα object τύπου PlacesClient με την συνάρτηση Places.createClient(Context activity). Τέλος πρέπει να δημιουργήσουμε ένα token της κλάσης AutocompleteSessionToken με την συνάρτηση AutocompleteSessionToken.newInstance(). Η σημαντικές λειτουργίες του AutocompleteSessionToken αξιοποιούνται στο MaterialSearchBar.

Για να αναπαραστήσω το search κατέβασα ένα χρήσιμο tool το material search bar. Περιέχει διάφορες χρήσιμες λειτουργίες για να κάνει την μπάρα αναζήτησης πιο διαδραστική χωρίς ιδιαίτερο κόπο. Περιέχει διάφορους action listener που ο καθένας πραγματοποιεί μια διαφορετική λειτουργία. Την λειτουργία της αναζήτησης την πραγματοποιώ συνδυάζοντας το Places API με το material search bar και συγκεκριμένα την συνάρτηση beforeTextChanged του addTextChangedListener. Κάθε φορά που πληκτρολογώ κάτι στην μπάρα αναζήτησης δημιουργώ ένα καινούργιο αντικείμενο της κλάσης FindAutocompletePredictionsRequest και μέσω του builder αυτής της κλάσης περνάω σαν όρισμα στη συνάρτηση setSessionToken το αντικείμενο της κλάσης AutocompleteSessionToken και στην συνάρτηση set Query() το περιεχόμενο της μέχρι τώρα αναζήτησης του χρήστη.

Το αν υπήρξε επιτυχία στην αναζήτηση θα το δούμε με την κλάση FindAutocompletePredictions του PlacesClient περνώντας σαν όρισμα το FindAutocompletePredictionsRequest object. Τα αποτελέσματα θα τα επικαρπωθούμε από την συνάρτηση FindAutocompletePredictionsResponse και συγκεκριμένα με την συνάρτηση getAutoCompletePredictions(). Το τελικό αποτέλεσμα της εκάστοτε αναζήτησης θα το εμφανίσουμε στον χρήστη μέσω της συνάρτησης updateLastSuggestions() της material search bar περνώντας σαν όρισμα την λίστα με τα προτεινόμενα μέρη.

Η δεύτερη σημαντική λειτουργία της material search bar έρχεται με τον onClickListener. Σε αυτό όταν επιλέγουμε μια τοποθεσία από αυτές που έχουν προταθεί ξεκινάμε μια αλυσίδα από ενέργειες:

1) Παίρνουμε την επιλεγμένη τοποθεσία από την λίστα predictionsList
2) Βρίσκουμε το id της τοποθεσίας με την συνάρτηση getPlaceId()
3) Περνάμε το id που βρήκαμε στον builder της κλάσης FetchPlaceRequest

4) Τέλος, ενεργοποιούμε το PlacesClient.fetchPlace(FetchPlaceRequest fetchPlaceRequest).addOnSuccessListener στο οποίο σε περίπτωση επιτυχής αναζήτησης τοποθεσίας γίνονται μια πληθώρα από ενέργειες.

Μερικές από τις λειτουργίες που εκτελούνται είναι:

- Τοποθέτηση δεικτών στην τοποθεσία του χρήστη και στην επιλεγμένη τοποθεσία.
- Zoom out για να μπορεί ο χρήστης να δει που βρίσκεται η επιλεγμένη τοποθεσία σε σχέση με την δική του
- Κλήση του DirectionsAPI που με τις λειτουργίες που περιγράψαμε προηγουμένως, θα ζωγραφίσουμε την βέλτιστη διαδρομή προς τον επιλεγμένο προορισμό
- Κλήση του DistanceMatrixAPI που μας δίνει μια εκτίμηση για την ώρα άφιξης στον προορισμό

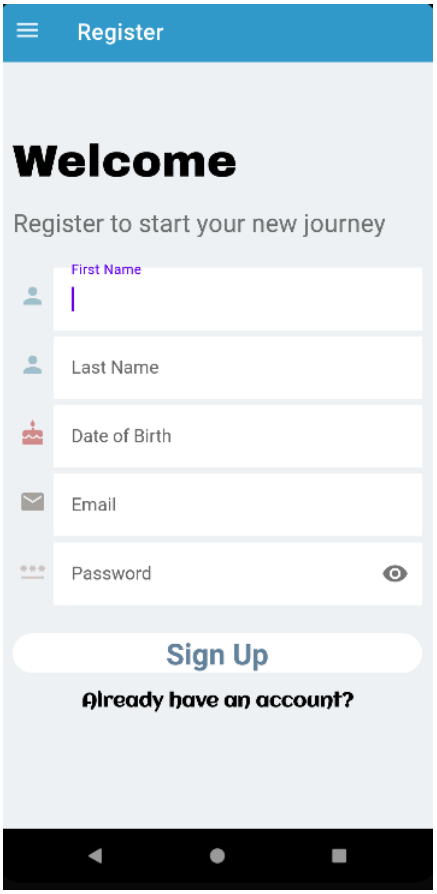
Elevation API

Ο τρόπος κλήσης αυτού του API ομοιάζει με αυτόν όλων των υπόλοιπων API που έχω δείξει μέχρι τώρα. Σε ένα interval των 10 δευτερολέπτων κάνω κλήση στο API χτίζοντας ένα url που θα έχει την εξής μορφή: *https://maps.googleapis.com/maps/api/elevation/json?locations=my_latitude%2C-my_longitude &key=MY_API_KEY*. Το μοναδικό αποτέλεσμα που παίρνουμε από το Json είναι ένα πεδίο με όνομα elevation και είναι εκείνο από το οποίο αντλούμε το επιθυμητό αποτέλεσμα. Η μοναδική φόρμα στην οποία ο χρήστης βλέπει το συγκεκριμένο αποτέλεσμα είναι σε αυτήν που μετρά τα στατιστικά οδήγησης του χρήστη.

Δημιουργία λογαριασμού για χρήστες της εφαρμογής

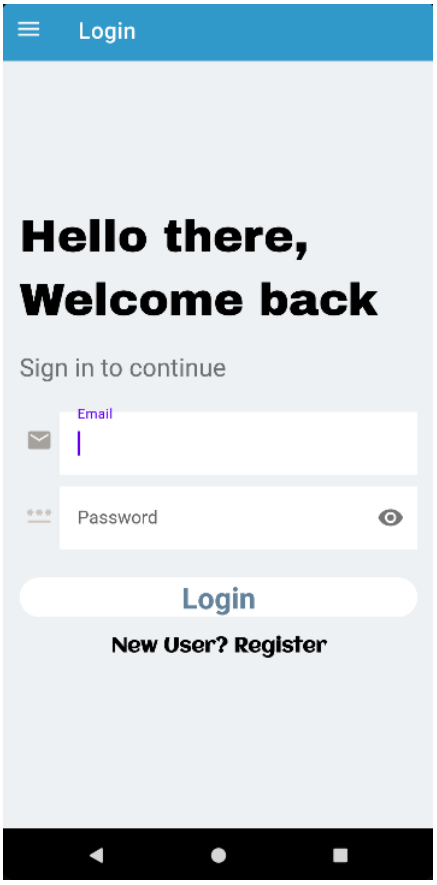
Εδώ πρέπει να επισημάνω ότι υπάρχει μια διαφορά ανάμεσα στους χρήστες της εφαρμογής που έχουν δημιουργήσει λογαριασμό με αυτούς που δεν έχουν. Αυτή είναι ότι μόνο εκείνοι που έχουν λογαριασμό μπορούν να αποθηκεύσουν τα δεδομένα τους στον server για να μπορούν να δουν στατιστικά όπως το συνολικό ιστορικό τους σε κάθε κούρσα, τον μέσο όρο των κουρσών τους ή τον συνολικό μέσο όρο όλων των χρηστών της εφαρμογής.

Για να συνδεθεί ένα χρήστης στην εφαρμογή πρέπει να περάσει πρώτα από την φόρμα δημιουργίας λογαριασμού:



Σε αυτήν συμβαίνουν διάφοροι έλεγχοι για να ξέρουμε με περισσότερη σιγουριά ότι ο χρήστης έχει ακολουθήσει σωστά την διαδικασία δημιουργίας λογαριασμού. Με επιτυχία στην παραπάνω διαδικασία, ο χρήστης στέλνει τα δεδομένα του στον server με το Retrofit. Το αποτέλεσμα θα περάσει στο Controller Layer του Spring boot μέσω του url: localhost/users που είναι ένα POST request. Έπειτα το Controller layer θα δώσει τον έλεγχο στο Service layer οπού θα γίνουν περαιτέρω έλεγχοι ορθής πληροφορίας από τον χρήστη. Τέλος η μεταφορά δεδομένων θα τελειώσει όταν το Service Layer περάσει τον έλεγχο στο Data Link Layer οπού εκεί θα εκτελεστεί η διαδικασία αποθήκευσης στην βάση δεδομένων και συγκεκριμένα στο table users στο οποίο βρίσκονται αποθηκευμένοι όλοι οι συνδεδεμένοι χρήστες.

Ένας χρήστης που έχει ήδη λογαριασμό πρέπει να επικυρώσει τα στοιχεία του στην φόρμα σύνδεσης:



Το μόνο που απαιτείται είναι να αποδώσει τον σωστό συνδυασμό για το email και τον κωδικό πρόσβασης του. Πάλι η επικοινωνία με τον server επιτυγχάνεται μέσω του Retrofit και συγκεκριμένα κάνοντας request στο url: …/localhost/users/login το οποίο πάλι είναι ένα POST request που έχουμε περάσει σαν body το συνδυασμό email/password. Ο έλεγχος γίνεται στο spring boot , το οποίο ελέγχει την βάση δεδομένων για την ύπαρξη του συγκεκριμένου συνδυασμού.

Επιπλέον λειτουργίες για τους συνδεδεμένους χρήστες

Όπως έχω αναφέρει και προηγουμένως οι συνδεδεμένοι στην εφαρμογή χρήστες κερδίζουν και κάποιες επιπλέον λειτουργίες. Αυτές είναι:

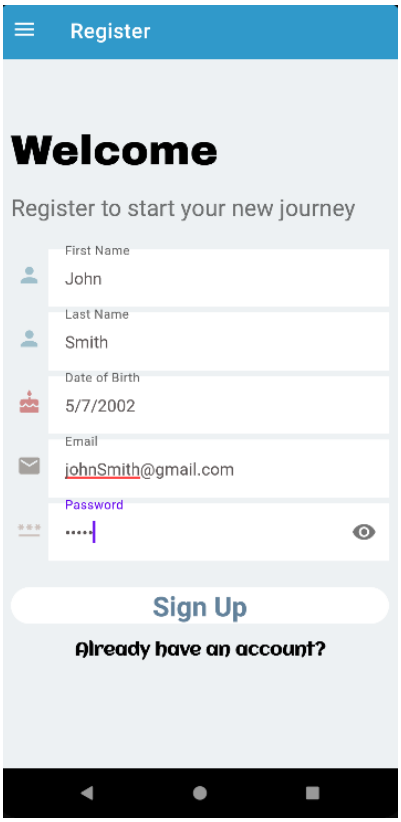
- Η δυνατότητα δημιουργίας προφίλ
- Θα μπορούν να δουν όλες τις κούρσες που έχουν καταγράψει στην εφαρμογή
- Θα έχουν την δυνατότητα να εξετάσουν τον μέσο όρο των μετρήσεων τους
- Θα είναι σε θέση να κάνουν σύγκριση του δικού τους μέσου όρου με αυτού του μέσου χρήστη
- Θα μπορούν να κάνουν επεξεργασία στο προφίλ όπως το να προσθέσουν κάποια εικόνα προφίλ ή να αλλάξουν τα στοιχεία πρόσβασης τους στην εφαρμογή

Η δημιουργία προφίλ γίνεται όπως ανέφερα προηγουμένως στην δημιουργία λογαριασμού για τους χρήστες. Όσον αφορά την αποθήκευση των στατιστικών μετρήσεων τους στην βάση δεδομένων έχουμε τα εξής:

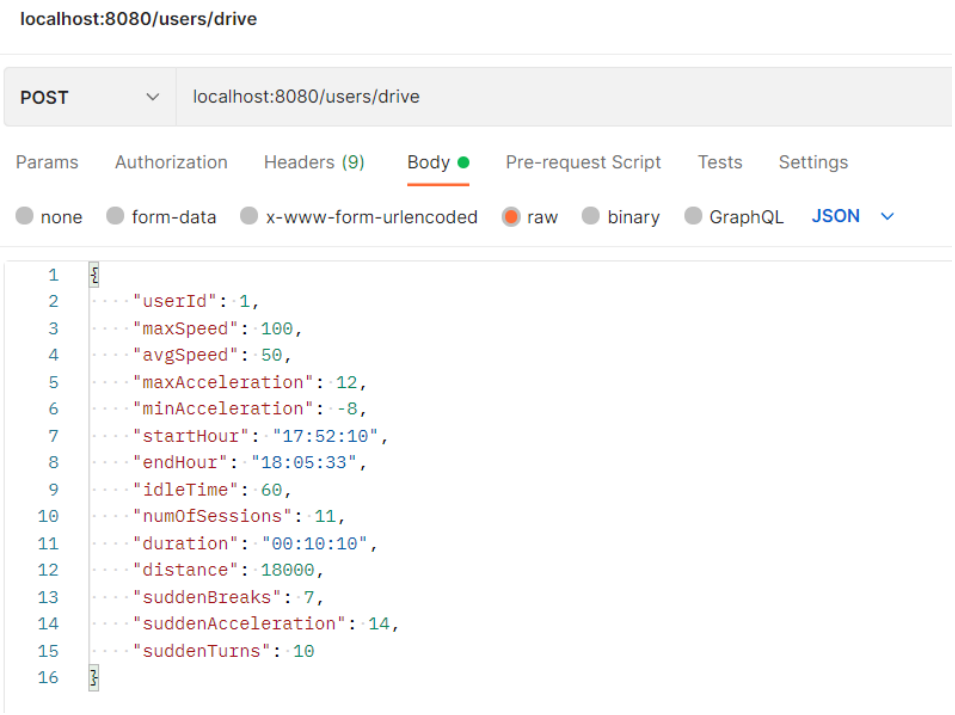
- Στο τέλος κάθε κούρσας, όταν δηλαδή ο χρήστης πατάει το κουμπί τερματισμού, αυτομάτως δημιουργούνται δύο αντικείμενα: DriveResults που περιέχει όλα τα δεδομένα της συγκεκριμένης κούρσας και της κλάσης AverageDriveResults. Την κλάση DriveResults την χρησιμοποιώ για να αποθηκεύσω όλες τις κούρσες που έχει κάνει ο χρήστης, ώστε να μπορεί να τις δει μετά στο προφίλ του. Μόλις στείλω αυτό το αντικείμενο στον server με το Retrofit μέσω του url localhost/users/drive το μόνο που έχει να κάνει το Spring Boot είναι απλά να αποθηκεύσει τα δεδομένα χωρίς περαιτέρω υπολογισμούς. Τη δε AverageDriveResults την χρησιμοποιώ για να αποθηκεύσω τον μέσο όρο του χρήστη σε όλες του τις κούρσες. Εκεί απαιτούνται περαιτέρω υπολογισμοί για να βρούμε τον καινούργιο μέσο όρο και τελικά γίνεται ανανέωση της βάσης δεδομένων. Το ίδιο Object της κλάσης AverageDriveResults χρησιμοποιείται και για τον υπολογισμού του μέσου όρου όλων των χρηστών.

Όπως έχω αναφέρει, τα αποτελέσματα από τις κούρσες του, ο χρήστης μπορεί να τα δει στο προφίλ του:

Παράδειγμα εισαγωγής χρήστη στο server για να δημιουργήσω προφίλ:

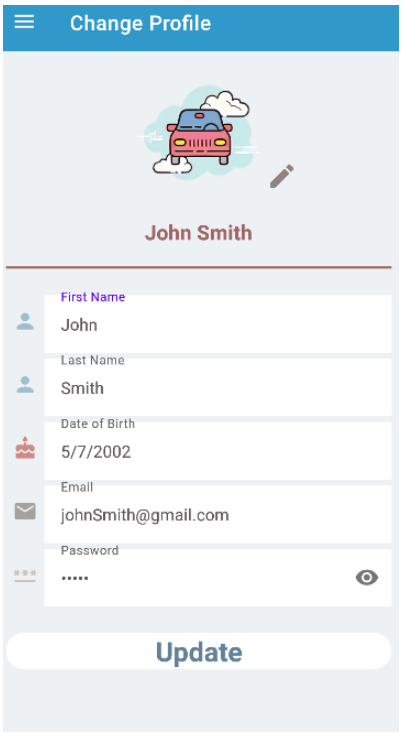


Αποστολή κάποιων τυχαίων δεδομένων για μια κούρσα του νέου μας χρήστη με userId=1. Για την αποστολή των δεδομένων χρησιμοποιώ το Postman, που είναι ένα χρήσιμο εργαλείο με το οποίο στέλνουμε εύκολα HTTP requests. Στο συγκεκριμένο παράδειγμα στέλνομε τα δεδομένα σε Postgresql database.

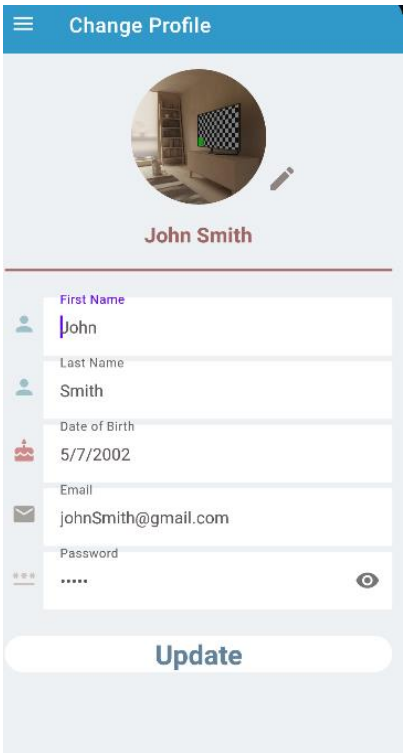


Μετάβαση του χρήστη στο προφίλ του:

Εδώ ο χρήστης μπορεί να επεξεργαστεί το προφίλ αν το επιθυμεί:



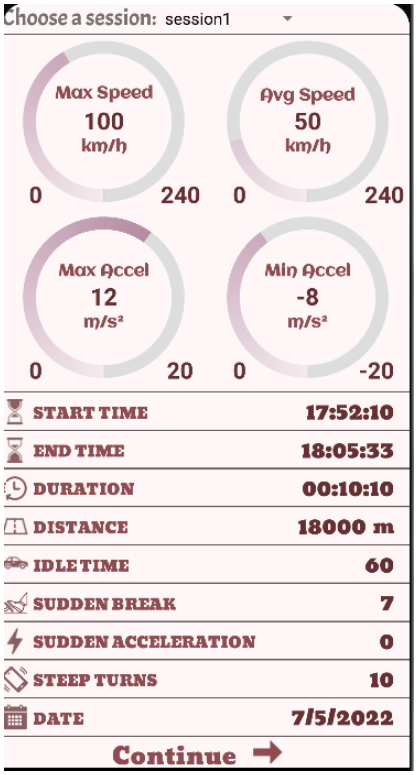
Προσθήκη εικόνας στο προφίλ του:



Η αποθήκευση των εικόνων γίνεται σε Firebase storage. Αυτό επειδή διαπίστωσα ότι δεν ήταν αποδοτική η αποθήκευση των εικόνων στον server, αφού χρειάζονταν πολλές μετατροπές για να κάνω convert το image σε byte array και αντίστροφα. Η Firebase λειτούργησε ευκολότερα και αποδοτικότερα. Περισσότερα για την Firebase θα τα αναφέρω στην συνέχεια.

Μετάβαση στην προηγούμενη φόρμα για να δει ο χρήστης το ιστορικό του:

Επιλογή της διαδρομής που βάλαμε εμείς μέσω του Postman:

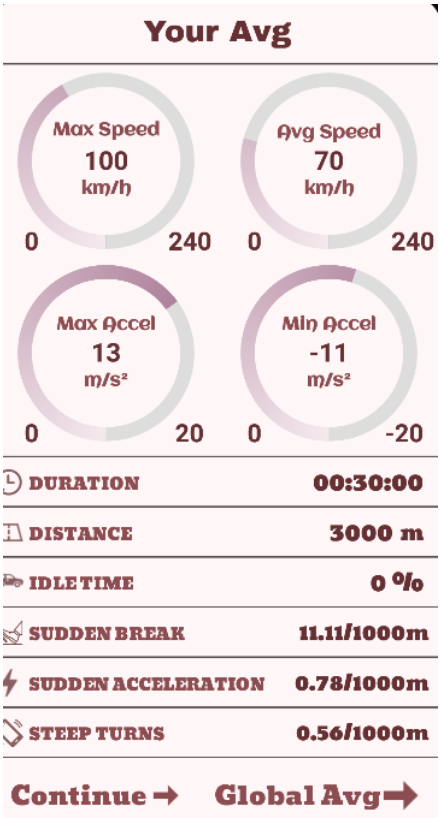


Όπως ανέφερα και προηγουμένως, με το τέλος της κούρσας δημιουργείται ένα AverageDriveResults object το οποίο στέλνεται στον server με το Retrofit για να υπολογιστεί ο μέσος όρος του χρήστη και του συνόλου. Το url που χρησιμοποιείται είναι το localhost/users/averageDriveResults. Το παραπάνω object δέχεται επεξεργασία για να ανανεώσει με

ορθό τρόπο την βάση δεδομένων. Συγκεκριμένα, για τον μέσο όρο του χρήστη γίνονται οι ακόλουθες μετρήσεις:

1) Την *μέση ταχύτητα* την αναβαθμίζουμε ως εξής: *Νέα μέση ταχύτητα = (μέση ταχύτητα κούρσας * διάρκεια κούρσας + μέση ταχύτητα * μέση διάρκεια)/(συνολική διάρκεια* +διάρκεια κούρσας). Έτσι, αν για παράδειγμα είχαμε την πρώτη κούρσα με διάρκεια 1h και μέση ταχύτητα 70km/h και μετά μια δεύτερη κούρσα με διάρκεια 30 λεπτά και μέση ταχύτατα 80km/h, τότε η νέα μέση ταχύτητα θα ήταν ίση με (80*30 + 70*60)/(70 = (2400+4200)/(60+30) = 6600/90 = 73. Μαζί με την μέση ταχύτητα θα ανανεωθεί και η μέση διάρκεια καθώς και η συνολική διάρκεια.
2) Με παρόμοιο τρόπο ανανεώνονται και η μέση απόσταση, διάρκεια και στασιμότητα.
3) Τα απότομα φρεναρίσματα, επιταχύνσεις και στροφές υπολογίζονται σε μέση παραβίαση ανά χιλιόμετρο, ως εξής:
   *παραβίαση ανά χιλιόμετρο= (αριθμός παραβιάσεων*1000/συνολική απόσταση)* .
4) Παρόμοιοι είναι και οι υπολογισμοί για το συνολικό μέσο όρο των χρηστών.

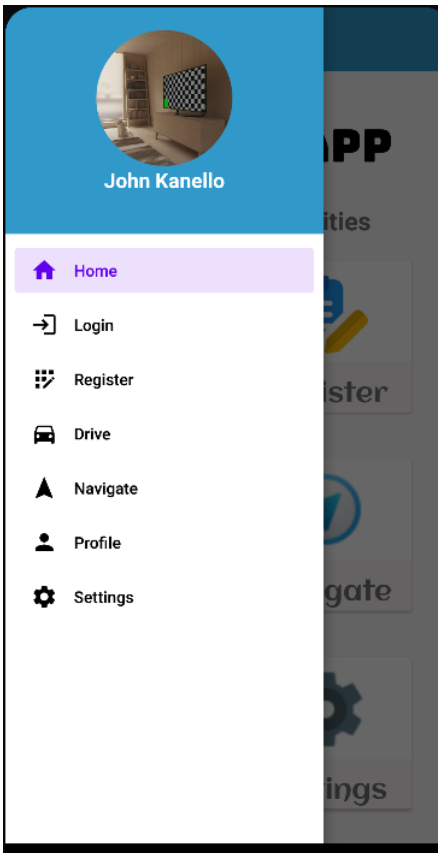
Τα δεδομένα αυτά οι χρήστες μπορούν να τα δουν στο προφίλ τους στην καρτέλα View Statistics:



Υπόλοιπες λειτουργίες

Εδώ θα μιλήσω για μερικές από τις σχεδιαστικές αποφάσεις που πήρα για το UI ώστε να κάνω την εφαρμογή πιο διαδραστική, όπως το navigation drawer για να γίνει γρηγορότερη η πλοήγηση στην εφαρμογή, την χρήση του constraint layout για να προσαρμόσω την εφαρμογή σε διαφορετικά μεγέθη οθονών, το firebase κ.λπ.

Navigation Drawer



Στις φόρμες που θέλω να προσθέσω το Navigation drawer πρέπει να ακολουθήσω τα εξής βήματα :

1) Να βάλω σαν γονέα το Drawer Layout για να μπορούμε να σύρουμε το navigation drawer με το δάχτυλο μας από την αριστερή μεριά της οθόνης προς τη δεξιά. Πρέπει να αναφέρω ότι το Drawer Layout δέχεται μέχρι δύο παιδιά. Το ένα παιδί θα είναι το Navigation View δηλαδή ο Navigation Drawer και το άλλο ένα Linear Layout με vertical orientation στο οποίο θα περιέχεται ένα toolbar με το οποίο μπορώ να ανοίξω τον drawer μέσω ενός κουμπιού και ένα ScrollView με το οποίο ο χρήστης θα μπορεί να δει όλα τα στοιχείο στην οθόνη κάνοντας σύρσιμο προς τα κάτω.

2) Να φτιάξω ένα menu drawer header xml αρχείο το οποίο θα είναι ο “σκελετός” του navigation drawer. Θα καθορίζει το χρώμα, τον τίτλο και την εικόνα που θα βλέπουμε στο drawer. Το θέτουμε στο πεδίο app:headerLayout του Navigation View.

3) Να γεμίσω το παραπάνω header με menu items δημιουργώντας ένα menu xml αρχείο το οποίο θα απαριθμεί τις κατηγορίες που μπορούμε να επιλέξουμε όπως το Home, Login, Register, Profile etc. Το θέτουμε στο πεδίο app:menu του Navigation View.

4) Στο αρχείο με τον πηγαίο κώδικα της κλάσης πρέπει να κάνω implement το interface NavigationView.OnNavigationItemSelectedListener το οποίο αυτομάτως απαιτεί την δημιουργία της συνάρτησης onNavigationItemSelected το οποίο είναι το action listener με το οποίο ελέγχουμε πότε ο χρήστης χρησιμοποιεί το drawer. Επίσης πρέπει να κάνω την αρχικοποίηση των drawerLayout, navigationView και toolbar με τον εξής τρόπο:

```
drawerLayout = findViewById(R.id.drawer_layout);
navigationView = findViewById(R.id.nav_view);
toolbar = findViewById(R.id.toolbar);

setSupportActionBar(toolbar);

navigationView.bringToFront();
ActionBarDrawerToggle toggle = new ActionBarDrawerToggle( activity: this, drawerLayout, toolbar,"Drawer Open", "Drawer Close");
drawerLayout.addDrawerListener(toggle);
toggle.syncState();
```

Firebase Storage

Είχα αναφέρει προηγουμένως ότι οι εγγεγραμμένοι χρήστες της εφαρμογής μπορούν να αποθηκεύσουν μια εικόνα για το προφίλ τους. Επίσης, είπα ότι αυτές οι εικόνες θα αποθηκεύονται στο storage ενός online Firebase. Αυτό το έκανα, επειδή αφού πρώτα προσπάθησα να αποθηκεύσω τις εικόνες των χρηστών στην βάση δεδομένων στον Server παρατήρησα ότι ήταν μια μη αποδοτική διαδικασία στην οποία έπρεπε συνεχώς να γράφω κώδικα για να κάνω convert τις εικόνες σε ένα πίνακα από byte για να μπορέσει να αποθηκευτεί στο Server και να κάνω πάλι convert τις εικόνες από byte array σε Bitmap image ώστε να μπορώ να τις χρησιμοποιήσω στην εφαρμογή.

Το Firebase ήταν ένας πιο ευλύγιστος τρόπος για να πραγματοποιήσω την ίδια λειτουργία. Το μόνο που χρειάζεται για να αποθηκεύσω μια εικόνα στο Firebase Storage της Firebase είναι να δημιουργήσω και να αρχικοποιήσω από ένα αντικείμενο για τις κλάσεις FirebaseStorage και StorageReference . Στην συνέχεια στην συνάρτηση onActivityResult τραβάω την επιλεγμένη εικόνα που επέλεξε ο χρήστης και την προωθώ στην Firebase. Αυτό το κάνω δημιουργώντας ένα παιδί του root node StorageReference και προσθέτοντας στο παιδί την επέκταση images/users/{id}, οπού id είναι το userId του εκάστοτε χρήστη και είναι μοναδικό για τον κάθε χρήστη. Στην

συνέχεια χρησιμοποιώ την συνάρτηση putFile της StorageReference κλάσης με την οποία τοποθετώ την εικόνα στο μονοπάτι που αναφέραμε.

Για να τραβήξω ένα image από την Firebase η διαδικασία είναι επίσης απλή. Απλά χρειάζομαι πρόσβαση στο παιδί του root StorageReference που βρίσκεται στο μονοπάτι images/user/{id}. Στην συνέχεια χρησιμοποιώ την συνάρτηση getFile της κλάσης StorageReference και τοποθετώ το αποτέλεσμα αυτής της συνάρτησης σε ένα αντικείμενο της κλάσης File. Στο αντικείμενο αυτό θα βρίσκεται η εικόνα προφίλ του χρήστη.

Δεν υπάρχει κάποια άλλη χρήση για την Firebase καθώς όλα τα άλλα δεδομένα αποθηκεύονται στον Server όπως έχω ήδη δείξει.

Προσαρμογή εφαρμογής στο μέγεθος οθόνης του χρήστη

Ένα θέμα ακόμη που κλήθηκα να αντιμετωπίσω κατά την εκπόνηση της εργασίας ήταν αυτό της προσαρμογής των UI elements στην εικόνα του χρήστη. Για να μπορέσω να λύσω αυτό το πρόβλημα χρησιμοποίησα δύο εργαλεία:

1) Το πρώτο είναι το ScrollView το οποίο μας δίνει την βεβαιότητα ότι στο πλάτος της οθόνης δεν θα χαθεί κάποιο στοιχείο UI από την εικόνα, αλλά θα μπορεί ο χρήστης να κάνει scroll προς τα κάτω ή προς τα πάνω. Οι λειτουργία του ScrollView χρησιμοποιείται σε φόρμες όπως το home menu της εφαρμογής και το login/register.

2) Σε κάποιες φόρμες της εφαρμογής όπως σε εκείνη της φόρμας οδήγησης πιστεύω ήταν σημαντικό να μπορέσω να μεταφέρω όλη την πληροφορία άμεσα στον χρήστη χωρίς να απαιτείται από εκείνον να κάνει scroll. Για να το πετύχω αυτό, χρησιμοποίησα την λειτουργία των Guidelines του ConstraintLayout.

Τα Guidelines είναι ένα πολύ εύχρηστο εργαλείο που μας προσφέρονται από το ConstraintLayout. Με αυτά πετυχαίνουμε το να χωρίσουμε την οθόνη σε ποσοστά, δηλαδή μπορούμε για παράδειγμα να θέσουμε την λειτουργία **app:layout_constraintGuide_percent** στο 0.50 και το **orientation** να είναι vertical κάτι που θα μας δώσει ένα Guideline το οποίο χωρίζει την οθόνη σε δύο ίσα μέρη.

Όπως έχω ήδη αναφέρει, τα Guidelines χρησιμοποιήθηκαν στην φόρμα οδήγησης αλλά και γενικότερα σε όλες τις φόρμες στις οποίες παρουσιάζω στατιστικά στον χρήστη καθώς θεώρησα ότι θα ήταν πιο διαδραστικό από το να πρέπει να κάνει scroll.

Αλλαγή γλώσσας της εφαρμογής από Αγγλικά στα Ελληνικά και αντίστροφα

Μια ακόμη λειτουργία που υποστηρίζεται στην εφαρμογή είναι η αλλαγή γλώσσας. Μέχρι τώρα έχω παρουσιάσει την εφαρμογή από την οπτική ενός χρήστη που έχει σαν default language στην συσκευή του τα Αγγλικά. Ένας χρήστης που έχει σαν default language τα Ελληνικά θα δει την εφαρμογή στα Ελληνικά. Η αλλαγή γλώσσας επιτυγχάνεται σε συγκεκριμένη φόρμα:

Στα Αγγλικά:

Στα Ελληνικά:

Για να μπορέσουμε να αλλάξουμε τι γλώσσα από αγγλικά σε ελληνικά και αντίστροφα, πρέπει να έχουμε φτιάξει ένα strings.xml για την κάθε γλώσσα. Στα .xml αυτά θα υπάρχουν πεδία <string name=”…” > foo </string>. Το string name θα είναι ίδιο σε κάθε string αρχείο, αλλά το περιεχόμενο αυτού του string θα είναι ανάλογο της γλώσσας.

Για να μπορέσουμε να δώσουμε στο πρόγραμμα να καταλάβει σε ποια γλώσσα θέλουμε να είμαστε, χρησιμοποιώ την παρακάτω συνάρτηση:

```
private void setLocale(String language) {

    Resources resources = getResources();
    DisplayMetrics metrics = resources.getDisplayMetrics();
    Configuration configuration = resources.getConfiguration();
    configuration.locale = new Locale(language);
    resources.updateConfiguration(configuration,metrics);

    onConfigurationChanged(configuration);
}
```

Σε αυτήν αλλάζουμε το Locale της εφαρμογής ανάλογα με την γλώσσα που έχουμε επιλέξει:

- language == “el” , για τα Ελληνικά
- language == “en”, για τα Αγγλικά

Συμπεράσματα

Μερικά από τα συμπεράσματα που έβγαλα από την εργασία τα εξής:

Για το Spring Boot:

1) Είναι πολύ χρήσιμο αν θέλουμε να φτιάξουμε Web Applications σε γλώσσα Java.
2) Εξυπηρετούμε HTTP requests γρήγορα και αποδοτικά.
3) Εύκολη η σύνδεση σε βάση δεδομένων για να εκτελέσουμε CRUD operations.
4) Με την χρήση ενσωματωμένου Tomcat Server, το testing της εφαρμογής γίνεται ευκολότερο
5) Κερδίζουμε πολύ χρόνο στην δημιουργία των προγραμμάτων, αφού με όλες τις έτοιμες βιβλιοθήκες/ dependencies που μας προσφέρει, δεν χρειάζεται να χάνουμε χρόνο για να ρυθμίσουμε εμείς τα configurations.
6) Με την αρχιτεκτονική των Microservices κρατάμε μια ομοιογένεια στο πώς δομούμε τα προγράμματα και τον κώδικα, με αποτέλεσμα να υπάρχει καλύτερη επικοινωνία ανάμεσα στους προγραμματιστές.

Για την Android Εφαρμογή:

1) Γίνονται διάφοροι υπολογισμοί έτσι ώστε να καταγράψουμε την οδηγική συμπεριφορά των χρηστών της εφαρμογής.
2) Οι υπολογισμοί αυτοί γίνονται με ακρίβεια από τα Location Services που μας προφέρει το Android όπως το LocationListener και FusedLocationProviderClient.
3) Με τα στοιχεία αυτά μπορούμε να δούμε ποιοι χρήστες υστερούν στην οδήγηση τους.
4) Μπορούμε επίσης να δούμε αν υπάρχουν επικίνδυνοι δρόμοι στους οποίος πολύ χρήστες παρατηρήθηκαν να κάνουν παραβιάσεις, όπως απότομες στροφές.
5) Όλοι αυτοί οι υπολογισμοί φαίνονται σε real time αλλά και αποθηκεύονται στον server.

6) Το κομμάτι της οδηγικής συμπεριφοράς το συνδέσαμε με τις λειτουργίες GPS. Οι λειτουργίες αυτές όπως ο υπολογισμός βέλτιστης διαδρομής σε προορισμό και ο υπολογισμός της ώρας μέχρι να φτάσουμε στον προορισμό κάνουν την εφαρμογή πιο διαδραστική και χρήσιμη.
7) Είδα πόσο εύκολη είναι η επικοινωνία ανάμεσα στο Android Studio και έναν server μέσω του Retrofit.
8) Έμαθα να κάνω κλήση σε διάφορα API όπως αυτά της Google και το WeatherAPI.
9) Βελτιώθηκα στην δημιουργία UI κάνοντας χρήση διάφορων τεχνικών όπως π.χ. Guidelines, animations και Navigation Drawer για να κάνω την εμπειρία του χρήστη περισσότερο διαδραστική.
10) Περιέχει 2 γλώσσες Ελληνικά/Αγγλικά.
11) Οι Online cloud databases όπως το Firebase είναι πιο αποδοτικές για την αποθήκευση εικόνων από αυτήν σε server

Πηγές και Βιβλιογραφία

1) https://developer.android.com/
2) https://stackoverflow.com/
3) https://www.youtube.com/c/PhilippLackner - ***Constraint Layout Guidelines***
4) https://developers.google.com/maps/documentation/directions - ***Directions API***
5) https://developers.google.com/maps/documentation/distance-matrix - ***Distance Matrix API***
6) https://www.weatherapi.com/ - ***Weather API***
7) https://www.youtube.com/user/shadsluiter - ***Accelerometer***
8) https://www.youtube.com/c/amigoscode ***- Spring Boot Course***
9) https://www.youtube.com/c/DailyCodeBuffer ***- Spring Boot Course***
10) https://www.youtube.com/c/CodingWithTea ***- Android UI elements (Animations, Navigation Drawer)***
11) https://developers.google.com/maps/documentation/places/android-sdk/autocomplete - ***Places API***

12) https://www.youtube.com/c/CodinginFlow - ***Android fundamentals***

13) https://square.github.io/retrofit/ - ***Retrofit***

ΤΕΛΟΣ ΕΡΓΑΣΙΑΣ