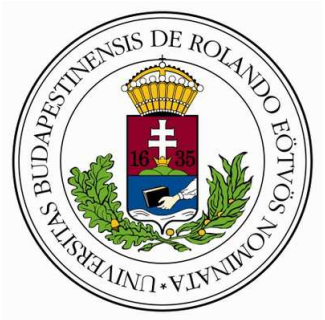


Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

BikeUp!

Dr. Kozsik Tamás
Egyetemi docens

Lombosi Balázs
Programtervező Informatikus BSc

Budapest, 2018

Tartalomjegyzék

I. rész

Bevezetés

Napjainkban nagyon elterjedt közlekedési eszköz lett a bicikli. Budapesten is egyre elterjedtebb, ennek hatására megjelent a BUBI szolgáltatás is. A mindennapi légszennyezést is csökkenteni lehet ezzel a közlekedési eszközzel. Több teszt is kimutatta, hogy egyes esetekben a kerékpár akár gyorsabb is lehet, mint az autó.

Az autósok számára nagyon sok navigációs alkalmazás elérhető, amik segítik a mindennapjaikban az út megtervezését akár a reggeli akár az esti forgalmas órákban. Ezek az alkalmazások a különböző kiegészítő funkcióikkal barátságosabbá és gyorsabbá teszik annak használatát. A Google által szolgáltatott térkép is ilyen alkalmazás.

Annak ellenére, hogy Budapesten folyamatosan egyre több bicikli utat építenek, ebben az alkalmazásban még nem érhető el az a funkció, hogy a megtervezett útvonal a kerékpár utakat részesítse előnyben.

Egyre több iparág kezd el foglalkozni ezzel a témával, többek között jelentek meg a kerékpárra szerelhető telefontartók, és alkalmazások is. De ezek az alkalmazások legtöbb esetben csak webes felületen érhetők el, ami több szempontból sem a legelőnyösebb.

Emiatt egy olyan natív Android-os alkalmazás létrehozása a cél, ami a kerékpárosokat fogja megcélozni. Az alkalmazás elérhető lesz számukra a legtöbb városban. Fő funkciója, hogy olyan útvonalat tervezzen mely ahol lehet kerékpár úton vezeti a felhasználót és csak végső esetben viszi ki őket az autók mellé az utakra. Azokban a városokban is működni fog az útvonal tervezés ahol nincs kerékpár út, csak azok hiánya miatt a megszokott utakon fog tervezni az alkalmazás.

Több kényelmi funkció is megtalálható lesz az alkalmazásban, hogy barátságosabb legyen annak a használata.

Az alkalmazás elérhető lesz mindenki számára a Google Play áruházban és remélem, hogy ezzel segítem és megkönnyítem a kerékpáros közösség közlekedését.

II. rész

Felhasználói Dokumentáció

1. Szükséges környezet

1.1. Android alkalmazás

A telefonos alkalmazás futtatásához egy okostelefon szükséges, amelyen az Android operációs rendszer legalább 5.0-ás(LOLLIPOP), vagy magasabb verziója fut.

Az alkalmazás számos funkciójához szükséges internet elérés, ezért ezt biztosítani kell az eszköz számára.

1.2. Szerver alkalmazás

A szerver alkalmazás futtatásához az operációs rendszerre telepítve kell lennie a Java legalább 8-as verziójának.

Ahhoz, hogy az alkalmazás csatlakozni tudjon az adatbázishoz, egy adatbázis szerver telepítése is elvárt. Ehhez a PostgreSQL 9.4-es verzióját ajánlom.

2. Android alkalmazás

2.1. Telepítés

Az alkalmazás telepítéséhez a mellékleten található apk-t a telefonra másolva és indítva telepíthető. Ehhez egyes eszközökön engedélyezni kell a telepítés USB-ről opciót a beállításokban.

Az alkalmazás elérhető a Google Play áruházban is, az áruházat megnyitva a BikeUp alkalmazásra rákeresve le tudjuk tölteni egyszerűen az alkalmazást.

Az Androidos alkalmazáshoz a folyamatos frissítések a Play áruházon keresztül folyamatosan elérhetőek lesznek. Ezért ezt a fajta telepítési módszert ajánlom.

2.2. Használat

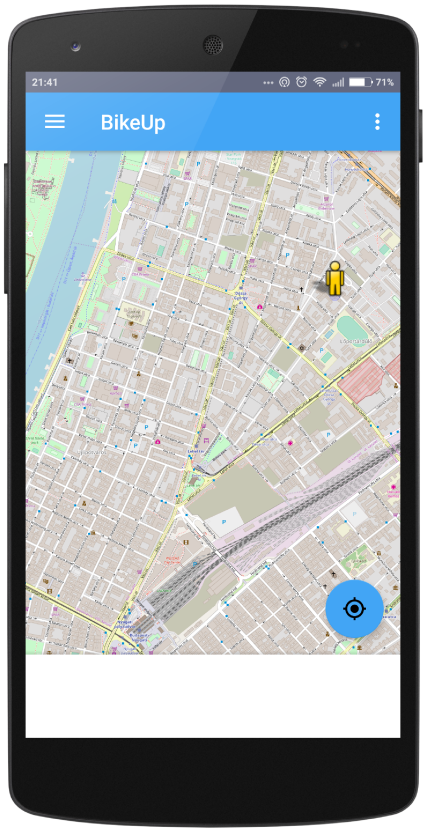
Az alkalmazás első indításkor engedélyeket fog kérni, hogy hozzáférhessen a telefon hely adataihoz és, hogy írhasson-olvashasson a telefonról. Ezek az engedélyek azért szükségesek, hogy az alkalmazás minden funkcióját ki tudja használni. Az engedélyeket megadhatjuk az *ENGEDÉLYEZÉS* gombra kattintva (1.ábra). Ezeknek az engedélyeknek a megadására csak egyszer van szükség, első indításkor, utána az alkalmazás emlékezni fog rá.

1. ábra.

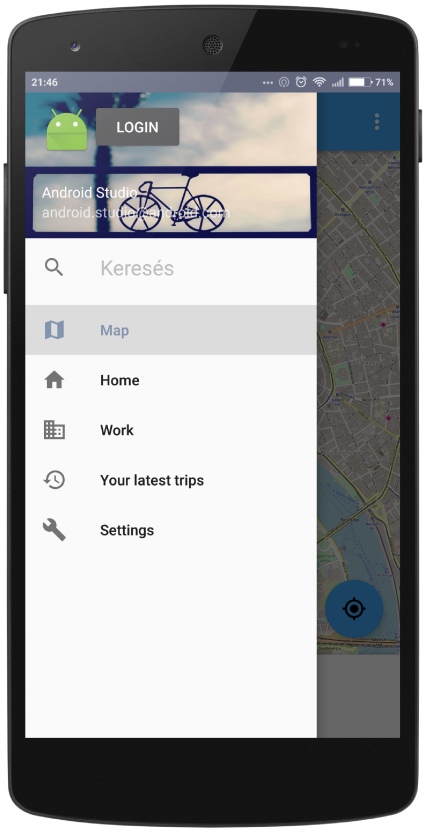
Amennyiben valamelyik engedélyt elutasítjuk, az alkalmazás nem fog elindulni. Ilyenkor vagy indítsuk újra az alkalmazást, vagy a felugró figyelmeztető ablakban az Ok gomb lenyomása után adjuk meg az engedélyeket.

Az alkalmazás elindítása után egy térképet fogunk látni, ahol meg van jelölve a mi pozíciónk (2.ábra). A térképen megjelenő kék célkereszt ikonnal a jelenlegi pozíciónkra tudunk ugrani. Ezt megnyomva az alkalmazás oda közelít, ahol mi vagyunk.

A térképet egyujjas gesztusokkal tudjuk mozgatni különböző helyszínekre. Ha a térképen nagyítani vagy kicsinyíteni szeretnénk, azt a megszokott kétujjas gesztussal megtehetjük.



2. ábra.



3. ábra.

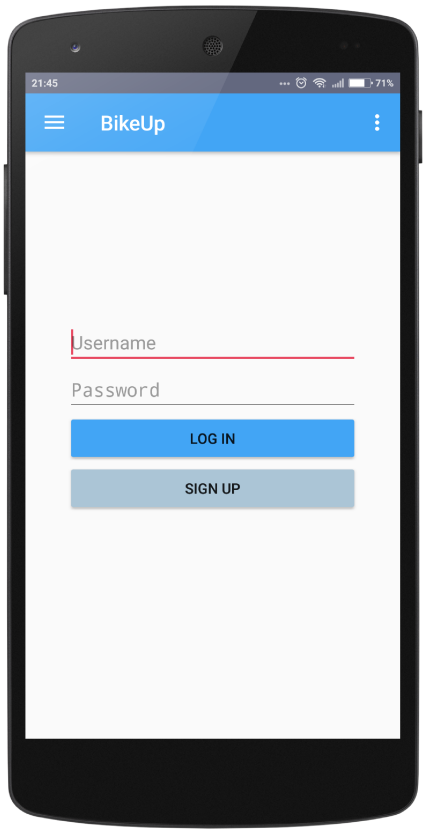
Az alkalmazás tetején megjelenő sávban található három vonal, amivel megnyitható a menü (3.ábra).

A menüből érhető el az alkalmazás több funkciója. A *LOGIN* gombra kattintva a bejelentkező felület jön be. A *MAP* menüpont a térképet hozza elő. A *HOME* és *WORK* menüpont a térképes felületet nyitja meg, majd útvonalat tervez az otthoni illetve munkahelyi címhez. Bejelentkezés után a *Your latests trips*-re tapintva az útvonal történeteket láthatjuk, a *Settings*-re tapintva pedig módosíthatjuk az elmentett helyeket.

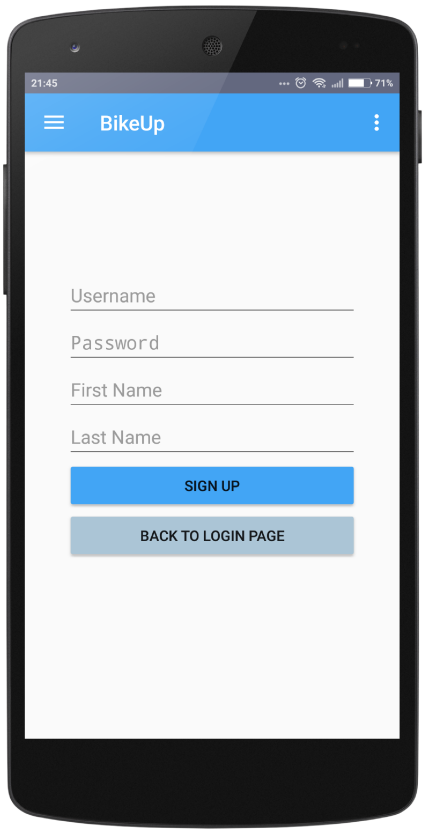
2.2.1. Bejelentkezés és Regisztráció

A Bejelentkező felületen (4.ábra) átválthatunk a regisztrációs felületre (5.ábra). Ezen a felületen, ha megadjuk a felhasználó név, jelszó, keresztnév, vezetéknév mezők mindegyikét, akkor tudunk regisztrálni az alkalmazásba.

Regisztráció után a *Back To Login Page*-re kattintva visszajutunk a bejelentkező felületre. Ezen a felületen a korábbi felhasználó nevünket és jelszavunkat megadva bejelentkezhetünk az alkalmazásba. Bejelentkezés után a menüben a *LOGIN* gomb megváltozik *LOGOUT* gombra, ezzel tudunk kijelentkezni az alkalmazásból, ha esetleg másik felhasználóval szeretnénk belépni.



4. ábra.

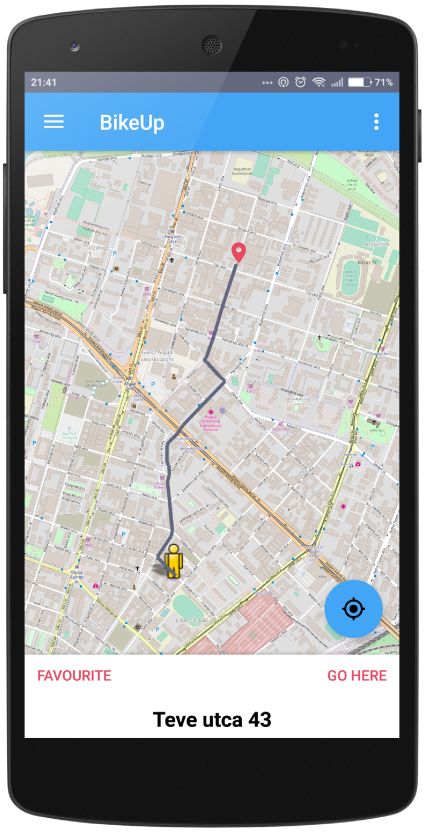


5. ábra.

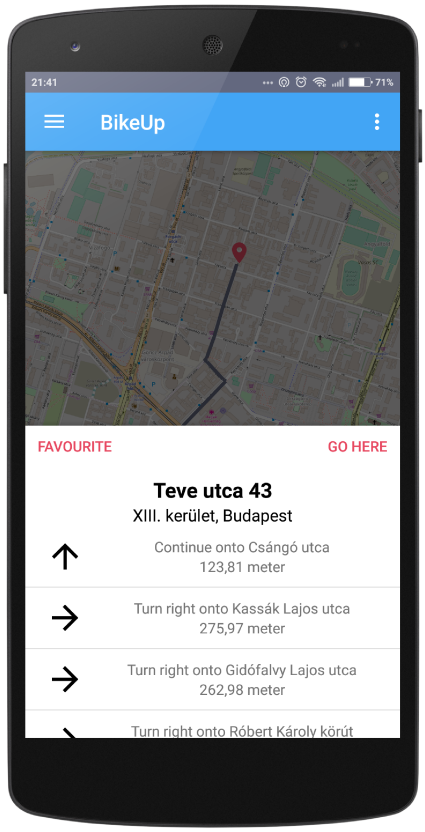
2.2.2. Útvonal tervezés

Célállomásnak tudunk jelölni a térképen egy pontot, ha az adott pontra hosszan tapintunk. Ekkor ezt a pontot egy piros ikon fogja jelölni. Ilyenkor egyből megtörténik az útvonal tervezés is a cél állomáshoz, ezt láthatjuk a térképen (6. ábra).

Az alsó fehér dobozban megjelenik a kijelölt pont pontos címe. Ha ezt az alsó dobozt az ujjunkkal felfelé húzzuk, akkor láthatjuk, hogy az útvonal milyen kanyarokat érint és milyen utcákon fog végig vezetni bennünket az alkalmazás (7. ábra). Ezt a listát az okos eszközökön megszokott gesztusokkal tudjuk lapozni fel-le. Ha



6. ábra.



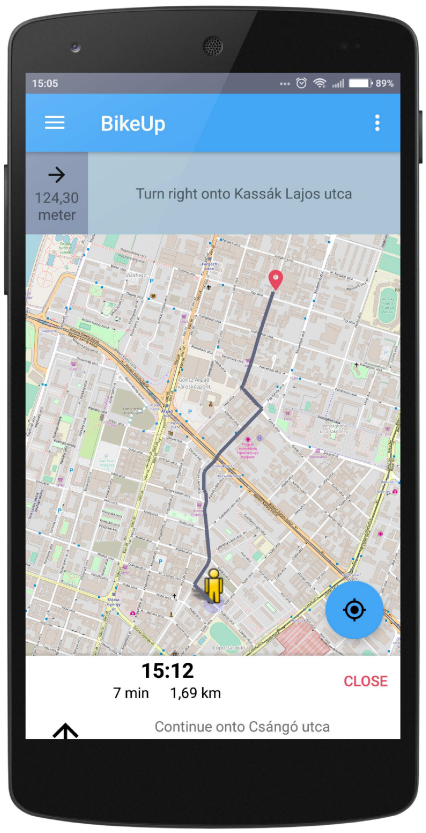
7. ábra.

a *GO HERE* gombra tapintunk, az alkalmazás átvált a követő üzemmódra. Ebben az üzemmódban láthatjuk a fehér szövegdobozban, hogy várhatóan mikor érünk célba, a visszalévő időt és távolságot. Ebben az üzemmódban is megnézhetjük a megtervezett út érintett kanyarjait.

Továbbá megjelenik a felső sorban egy új információs sáv. Ebben a sávban fogjuk megtalálni a következő kanyarhoz tartozó információkat, hogy melyik utcára kell kanyarodnunk és addig hány méter van vissza (8.ábra).

Ennek az üzemmódnak a bekapcsolását ajánlom indulást előtt megtenni, így mindig a saját pozíciónk lesz a középpontban és mindig látni fogjuk, hogy merre kell haladnunk.

Ha menet közben valahol szeretnénk előre megnézni a térképen az útvonalat, eltekerhetjük a térképet, ekkor a kék célkereszt ikonnal visszatudjuk kapcsolni a követő funkciót.

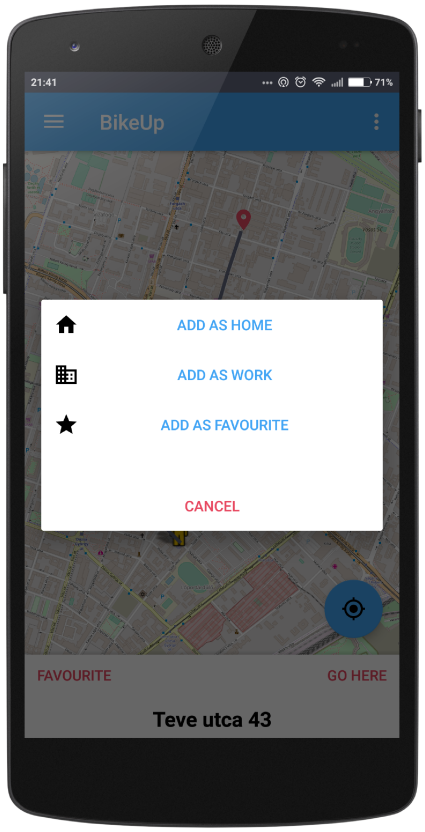


8. ábra.

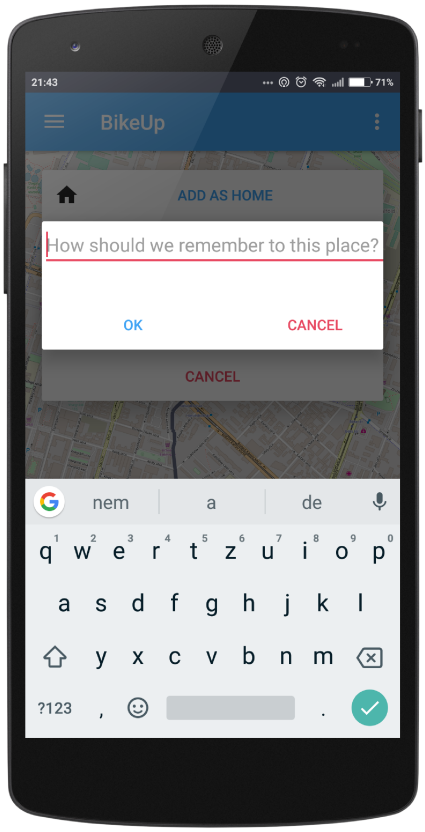
2.2.3. Hely elmentése

A térképen egy cél kiválasztása után megjelenik a *FAVOURITE* gomb. Erre kattintva megjelenik egy dialógus ablak (9.ábra), ahol választhatunk, hogy a kiválasztott helyet Otthoni címként, munkahelyi címként vagy kedvenc helyként szeretnénk elmenteni. A legutóbbit csak akkor fogja engedni az alkalmazás, ha bejelentkeztünk.

Amennyiben kedvenc helyként szeretnénk elmenteni, meg fogja kérdezni tőlünk az alkalmazás egy új ablakban, hogy milyen néven emlékezzen erre a címre (10.ábra). Itt tetszőlegesen elnevezhetjük az adott helyet Például: Egyetem, Mama otthon, stb.



9. ábra.



10. ábra.

Azoknak a helyeknek a felvételét ajánlom, amelyeket gyakran látogat. Ezáltal nem lesz szükség mindig odatekerni a térképen és kijelölni a helyet. Az elmentett helyekhez könnyen útvonalat lehet tervezni.

2.2.4. Elmentett helyek megtekintése

Az elmentett helyeket megtekinthetjük, ha a Menüben a *Settings* gombra tapintunk. Ebben a listában láthatjuk, hogy milyen néven milyen címeket mentettünk el korábban az alkalmazásba (11.ábra).

Amennyiben valamelyik cím már nem valóságos vagy nincs rá szükségünk, a kuka ikonra tapintva törölhetjük. Figyeljünk oda, hogy ha valamelyik címet töröltük az alkalmazásból azt visszaállítani csak úgy tudjuk, ha újra felvesszük.

11. ábra.

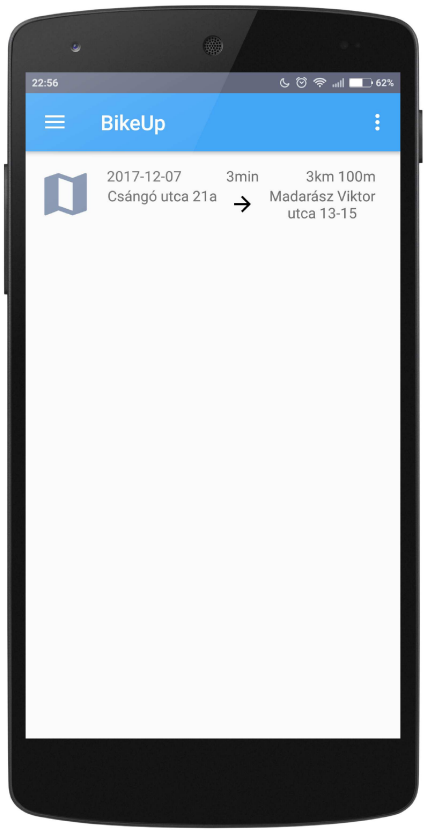
Ezen a felületen keresztül fogunk tudni útvonalat tervezni az elmentett helyeinkhez. Ha egy elmentett helyhez szeretnénk menni, tapintsuk meg a kiválasztott címhez tartozó lista elemet és az alkalmazás automatikusan megfogja tervezni számunkra az útvonalat.

2.2.5. Útvonal történet megtekintése

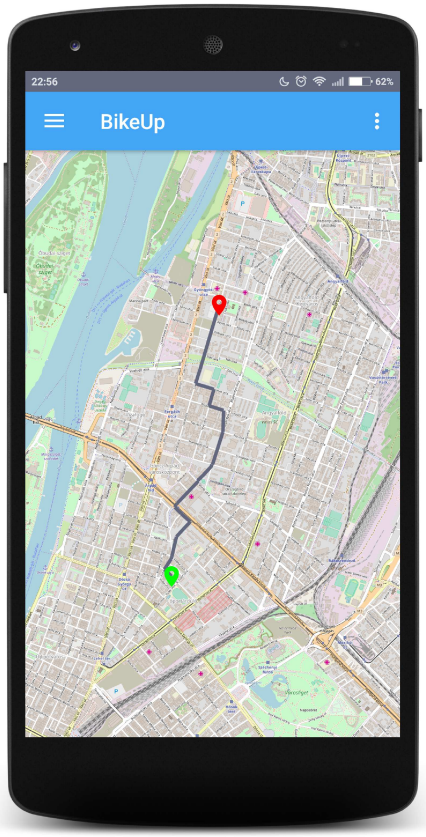
Az alkalmazás minden bejelentkezett felhasználó számára, megjegyzi a megtett utakat. Ezt a célba éréskor tárolja el a telefon, ezért a célba érés előtt nem ajánljuk az alkalmazás kikapcsolását.

A *Your trip history* menüpont kiválasztása után megtekintheti ezeket az elmentett utakat (12.ábra). A listában egy elemen láthatja, hogy mikor, honnan, hova és mennyi idő alatt tette meg az adott utat és, hogy az mekkora távolság volt.

Ha szeretné megnézni, hogy pontosan milyen útvonalon haladt, tapintson a kiválasztott útvonal történetre. Ekkor az alkalmazás be fogja tölteni azt az útvonalat, amin haladt. Zöld pont fogja jelölni a kiindulási címet és piros az érkezési címet (13.ábra).



12. ábra.



13. ábra.

3. Szerver alkalmazás

3.1. Konfigurálás

Az alkalmazás indítása előtt szükséges bekonfigurálni az alkalmazásban az adatbázis elérhetőségét, enélkül nem fog működni az alkalmazás. Ezt az *src/main/resources/application.properties* fájlban tehetjük.

A *spring.datasource.url*-ben kell megadni az adatbázis elérhetőségét.

A *spring.datasource.username*, *spring.datasource.password* pedig az adatbázis eléréséhez szükséges felhasználónevet és jelszót.

A konfiguráció után a szervert szükséges újra fordítani.

Ha nem akarjuk konfiguráció miatt újra fordítani a szervert, akkor a Jar fájlt kitömörítve (Windows-on Total Commander vagy WinRar segítségével) a *BOOT-INF/classes* mappában megtaláljuk ugyan ezt a property fájlt.

3.2. Futtatás

A szerver alkalmazás előtt indítanunk kell egy tetszőleges SQL szervert is. Ahogy már említettem a PostgreSQL 9.4es verzióját ajánlom.

Windows operációs rendszeren telepítés után az adatbázis szerver elfog indulni. A telepítéskor megadott felhasználó név és jelszó szükséges lesz a szerver konfigurációjához ezért ezeket jegyezzük meg. Az adatbázis szerveren alapméretezett beállítás, hogy létezik egy postgres nevű adatbázis. Ha nem szeretnénk más adatbázist létrehozni a tábláknak ezt használhatjuk a konfigurációban.

A szerver alkalmazás indításához Java-ra van szükség. Nyissunk egy új parancs ablakot lépjünk be a projekten belül a target mappában ahol a *bikeup-0.0.1-SNAPSHOT.jar* található. A következő parancsot kiadva az alkalmazás szerver elindul egy beágyazott Tomcat servlet [2] konténerben.

```
java -jar bikeup-0.0.1-SNAPSHOT.jar
```

III. rész

Fejlesztői Dokumentáció

1. Specifikáció

A cél egy olyan navigációs alkalmazás létrehozása, ami az útvonal tervezésekor a bicikli utakat részesíti előnyben, hogy a kerékpárosok számára kedvező útvonalat határozzon meg. Ehhez el kell készíteni egy telefonos alkalmazást, ami most Android operációs rendszeren fog futni, és a felhasználói élmény javítása miatt egy szerver alkalmazást is, melyek egymással kommunikálnak.

1.1. Android

Az Android alkalmazáson belül több funkciót kell megvalósítani. Ezek közül a legfontosabb egy olyan nézet megvalósítása, mely megjelenít egy térképet, ahol a felhasználó jelenlegi pozícióját megjelöljük. Ezen kívül lehetőséget adunk számára, hogy kijelöljön a térképen egy célt, amihez útvonalat tervezünk, figyelembe véve a kerékpár útvonalakat. A térképes nézeten megjelenítjük a felhasználó számára a kiválasztott címet és az útvonal részletes adatait.

A felhasználó átválthatja a nézetet egy követő üzemmódra. Ebben az üzemmódban mindig jelezzük az aktuálisan következő kanyart és, hogy az milyen messze van, továbbá a kiválasztott cél távolságát, és az érkezés idő pontját is.

Az alkalmazáson belül a felhasználónak lehetősége lesz egy saját profilt létrehozni, mellyel a regisztráció után bejelentkezhet és további funkciókat érhet el. A felhasználók hozzáadhatnak, két fix helyet a saját profiljukhoz, az otthonukat, és a munkahelyüket. Amennyiben megadják ezeket a címeket, könnyen tervezhetnek új útvonalat ezekhez a címekhez.

Ha a felhasználó bejelentkezett, felvehet magának tetszőleges számú kedvenc helyet, amelyeknek saját nevet is adhat. Ezeket a kedvenc helyeket illetve, otthoni és munkahelyi címeket tetszőlegesen törölheti és veheti fel újra. Az alkalmazás automatikusan menti a felhasználó által megtett utakat, melyeket a felhasználó később visszatölthet és megnézheti a megtett út adatait.

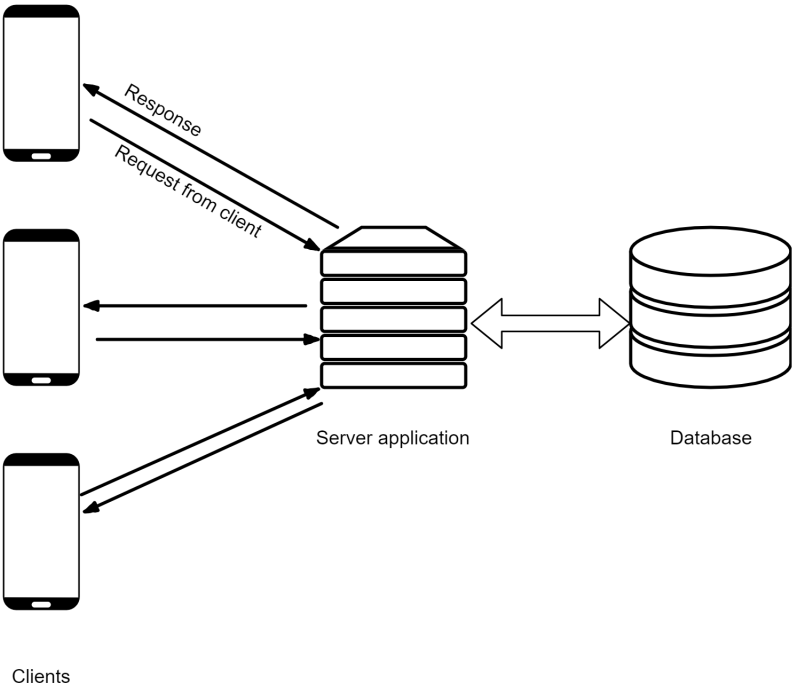
1.2. Szerver Alkalmazás

Az Androidos alkalmazásban elmentett adatokat egy szerveren keresztül egy adatbázisba fogunk menteni, ezért létre kell hoznunk egy szerver alkalmazást is, ami REST végpontokon keresztül kommunikál az Android alkalmazással.

2. Tervezés

Az alkalmazás megvalósításához felépítünk egy kliens-szerver kapcsolatot, ahol a szervert Java-ban írjuk Spring [1] keretrendszer segítségével, a kliens pedig egy Android operációs rendszeren futó alkalmazás lesz.

A szerver és a kliens REST végpontokon keresztül fognak egymással kommunikálni előre meghatározott interface-eken keresztül. A kliens által küldött adatokat a szerveren hitelesíteni fogjuk és elvégezzük rajta a megfelelő transzformációkat, majd egy adatbázisba mentjük. Az adatbázisba írás és az abból való olvasás közvetlenül a szerver alkalmazás feladata lesz, az innen olvasott adatokat szintén átalakítjuk és úgy küldjük el a kliens alkalmazásnak.



1. ábra.

Az 1. ábrán látható, hogy az alkalmazás különböző rétegei hogyan helyezkednek el egymáshoz képest, és hogyan kommunikálnak egymással. Látható hogy mind az adatbázisból, mind a szerver alkalmazásból egyet-egyet üzemelünk be, de kliensből bármennyi csatlakozhat a szerver alkalmazáshoz.

A 2. ábrán láthatjuk az alkalmazás használati eset (Use case) diagramját. Ez a diagram szemléletesen megmutatja, hogy a kétfajta felhasználó (Actorok) az alkalmazás mely funkcióit tudják használni, és azoknak a funkcióknak milyen további működései vannak.

Kedvenc cím felvétele
<<include>>
Kedvenc cím nevének megadása
Elmentett címek módosítása
Útvonal történet
<<extend>>
Útvonal történet megtekintése
<<extend>>
Útvonal történet betöltése
Regisztrált Felhasználó
Kijelentkezés
Munkahelyi cím felvétele
Otthoni cím felvétele
Útvonal tervezés
Bejelentkezés
Új Felhasználó
Regisztráció

2. ábra.

2.1. Android alkalmazás

2.1.1. Architektúra

Az Android alkalmazásokban megszokott az MVC architektúra így mi is ezt fogjuk követni. Elkülönítjük a *Modell* és a *View* réteget egymástól, a modell réteg lesz felelős minden adat tárolásáért, és ha egy adat megváltozik, akkor egy eseményt (event-et, Androidban intent-et) küld a nézet réteg felé, ami ennek az eseménynek hatására megváltoztatja a kiírt adatokat. Felhasználói input hatására a nézet réteg a modell rétegben változtatja meg az adatokat, külön nem tárolódnak el az adatok a nézet rétegen belül. A két réteg között egy kontrollert helyezünk el, ami gyakorlatilag egy interface lesz a két réteg kommunikációjához.

2.1.2. Modell réteg

Az Android alkalmazáson belül a modell réteg lesz az a réteg, ahol az alkalmazás logikai részét valósítjuk meg. Ebben a rétegben fogjuk megvalósítani az adatok kezelését, tárolását és a kommunikációt a szerver alkalmazással.

A modell rétegen belül a logikailag elkülöníthető feladatokat külön csomagokba szervezzük. Ahhoz, hogy ezeket a csomagokat ne kelljen ismernie a nézet rétegnek, létrehozunk egy átfogó osztályt. A nézet réteg ennek az osztálynak a metódusait fogja hívni és ez az osztály fogja meghívni a különböző csomagokból a megfelelő függvényeket.

Kommunikáció a szerverrel

Az Android alkalmazásnak meg kell valósítania egy kommunikációt a szerver alkalmazásunkkal. Ehhez a kommunikációhoz létre fogunk hozni egy olyan osztályt, ahol előre definiáljuk, hogy a szerveren milyen REST végpontokhoz indíthatunk kéréseket. Ezekben a definíciókban szerepelnie kell, a webservice pontos elérési címének. Szerepelnie kell továbbá, hogy az adott webservice milyen HTTP metódussal hívható (legtöbbször *GET*, *POST*, *PUT* vagy *DELETE*). Illetve ha a felhasználó már autentikálta magát az alkalmazásban, akkor be kell állítanunk a HTTP header-ben az autentikációs adatokat.

Ahhoz, hogy a szerveren a webservice-eket megfelelően hívhassuk meg, egyes végpontoknál szükséges a kérésben adatokat elküldenünk. Ezek a kérések főleg a *POST* metódusú kérések, amikor a szervernek el kell mentenie az Android kliens által küldött adatokat. Ezeket a kéréseket *JSON* formátumban fogjuk elküldeni a szervernek. Ahhoz, hogy a megfelelő kérések előálljanak minden kéréshez létre fogunk hozni egy Java osztályt, ami egy DTO lesz.

Miután felépítettük a megfelelő kérést és azt elküldtük a szervernek, várni fogunk a szervertől egy válaszra, ami szintén egy *JSON* formátumú szöveges objektum lesz.

Ezeknek a válaszoknak a könnyű kezeléséhez, ugyanúgy mint a kéréseknél DTO-kat fogunk létrehozni. A szerver válaszának nem csak a törzse lesz fontos a számunkra, hanem a státusz kód is, amivel visszatér, mert ezek fogják megmutatni, hogy sikeres vagy sikertelen volt a szerveren elvégzett művelet.

Ha már minden kész van ezek közül, akkor egy olyan általános osztályt kell létrehoznunk, ami magát a kommunikációt hajtja végre. Ennek az osztálynak a feladata lesz továbbá a kapott kérés törzsét át alakítani *JSON* formátumra, és a szerver válaszában található *JSON* szöveget át alakítani az általunk korábban definiált Java objektummá. Fontos, hogy ez az osztály maga a HTTP kommunikációt fogja megvalósítani, tehát nem számíthatunk arra, hogy rögtön végre hajtódik, a szerver is gondolkozhat a kérésen, és ez az idő alatt is szükséges, hogy az alkalmazás használható legyen. Ezek az okok miatt ennek az osztálynak a kommunikációs műveletének a háttérben kell futnia, ezért célszerű lesz egy aszinkron folyamatot megvalósítania. Ha egy háttérben futó folyamatot valósít meg, akkor szükséges lesz egy *callback* függvény, ami akkor fut le, ha a kommunikáció megtörtént és a szerver alkalmazás válaszolt.

A callback függvényt azért, hogy a nézetet egyszerűen tudjuk frissíteni, ketté választjuk, és attól függően, hogy a szerver alkalmazás mit adott válaszban, egy siker vagy egy sikertelen callback függvényt fogunk meghívni.

Felhasználó kezelés

Az alkalmazásban megkülönböztetjük a bejelentkezett és a nem bejelentkezett felhasználókat, ez a megkülönböztetés a modell feladata lesz.

Meg kell valósítanunk a modellben a bejelentkezés folyamatát, amit ha végrehajt a felhasználó, eltároljuk a nevét és jelszavát egy külön objektumba, hogy, amikor egy kérést indítunk a szerver felé, akkor tudjuk küldeni az autentikációs adatokat a szervernek. Bejelentkezéskor le fogjuk kérdezni a szervertől a felhasználó profilját is. A profilt szintén eltároljuk a modellben, hogy ha szükségünk van valamilyen adatára a felhasználónak, ne kelljen újra elkérnünk a szervertől. Hibás bejelentkezés esetén tájékoztatjuk a felületen a felhasználót.

Kijelentkezéskor a modellből minden adatot kitörlünk, ami a felhasználóra vonatkozik, és a külön objektumba elmentett felhasználónév-jelszó párost is kitöröljük, mert mostantól nem küldjük ezeket az adatokat a szervernek.

A felhasználó lehetőségeit kiemeljük egy külön interfacebe, amit a későbbiekben implementálnia kell minden osztálynak, ami felhasználói műveleteket valósít meg. A felhasználó osztálynak a következő műveleteket kell megvalósítania:

- Otthoni cím hozzáadása és törlése
- Munkahelyi cím hozzáadása és törlése
- Kedvenc hely hozzáadása és törlése

A kétfajta felhasználót (Anonim és Bejelentkezett) el kell különítenünk egymástól, de a két felhasználó logikailag nagyon hasonló, ezért célszerű egy közös ősosztályt létrehoznunk számukra és mindkettőt abból származtatni.

Az anonim felhasználó megvalósításakor implementálnunk kell minden metódust, amit a fentebb definiált interface megkövetel. Az otthoni és munkahelyi cím esetében az elvárt működést kell implementálni. A Kedvenc hely esetében viszont egy kivételt kell dobnunk, mert az anonim felhasználó nem élhet az alkalmazásnak ezzel a funkciójával.

A bejelentkezett felhasználó esetében az otthoni cím és munkahelyi cím működése meg fog egyezni az anonim felhasználóban definiált működéssel azzal a kiegészítéssel, hogy ezeket az adatokat már a szerver felé is el kell küldenünk. A kedvenc hely hozzáadása és törlése funkciót viszont a bejelentkezett felhasználó esetében már implementálnunk kell, mert számára elérhető ez a funkció. Ezek értelmében a bejelentkezett felhasználó tovább származhat akár az Anonim felhasználóból így elkerüljük, hogy többször kelljen implementálnunk a közös metódusokat, és elég csak felül definiálnunk azokat hivatkozva az ősosztályban definiáltra.

Pozicionálás és Útvonal kezelés

Az alkalmazás fő funkciója a navigációs modul, amihez a nézet rétegen meg jelenítünk egy térképet, és az ott megjelenített adatokat elő kell állítanunk a modellben.

Ennek a csomagnak a legmeghatározóbb feladata a pozíció meghatározása lesz és ezt átadni a felület felé. A pozíciót az Android keretrendszer által meghatározott interface-n keresztül fogjuk megkapni a eszköztől.

Amikor visszakapjuk az interface-n keresztül a pillanatnyi pozíciót, le kell futtatni bizonyos algoritmusokat azért, hogy a felületen a legfrissebb adatokat tudjuk kijelezni. A specifikációban meghatározottak alapján a következőket kell meg vizsgálnunk:

Az útvonal kirajzolásnak frissülnie kell folyamatosan, ahogy haladunk a kiszámolt útvonalon. Ennek érdekében meg kell határoznunk, hogy az útvonal mely pontjait hagytuk már el. Amennyiben elhagytunk egy pontot, akkor azt törölnünk

kell és erről egy eseményt küldeni a nézet rétegnek, hogy frissítse az útvonal kirajzolását.

A felületen megjelenítjük a következő kanyart és az ehhez tartozó információkat, pl.: milyen messze van ez a kanyar, melyik útra kell kanyarodni. Ezeket az adatokat is a modell fogja átadni a felületnek. Minden alkalommal, amikor új pozíció érkezik az eszköztől kiszámoljuk a jelenlegi pozíció és a következő kanyar távolságát és frissítjük ezt az adatot. Amennyiben ez a távolság egy paraméterben meghatározott számnál kisebb, frissítjük a kanyarok listáját és ugrunk a következő kanyarra. Minden esetben eseményt küldünk a felületnek, hogy frissültek az információk.

Amennyiben a felhasználó eltér a meghatározott útvonaltól, akkor az útvonalat újra kell számolnunk és eseményt küldeni a felület felé, hogy megváltozott az útvonal, frissítse annak a kirajzolását. Ahhoz, hogy megtudjuk mikor tér el a felhasználó az útvonaltól, kiszámoljuk az útvonal és a pozíció távolságát, és ha ez nagyobb mint egy paraméterben meghatározott szám, akkor frissítjük az útvonalat.

Amikor a felhasználó elért a célhoz, akkor azt észlelnünk kell, ezt szintén távolság számítással fogjuk meghatározni. Amennyiben ez megtörtént, akkor le kell zárnunk az útvonal történet mentését, és szólni a nézet rétegnek, hogy véget ért a navigálás, frissítse a felületet.

A felhasználó útvonal történetének a meghatározása egy érdekes kérdés, mert nem mondhatjuk egy megtervezett útvonalra előre, hogy az útvonal történet, mert a felhasználó időközben eltérhet ettől. Ha az eszköztől kapott minden egyes pozíciót elmentünk mint történet hamar kifuthatunk a memóriából, vagy pedig hatalmas adatokat kell küldenünk a szerver felé. Ahhoz, hogy ezeket elkerüljük azokat a pontokat fogjuk elmenteni a történetben, ami szerepel az útvonalon és a felhasználó elhaladt már azon a ponton.

A navigálás lezárásakor a történetben kiszámoljuk, hogy az elmentett pontok alapján mekkora távolságot tett meg a felhasználó és mennyi idő alatt.

2.1.3. View réteg

A view azaz nézet rétegen belül létre kell hoznunk az összes specifikációban meghatározott funkcióhoz tartozó felületet. Ezeket úgy fogjuk megtenni, hogy minden felülethez tartozni fog egy leíró *XML* alapú fájl, ami magát a felületet határozza meg, a felületen elhelyezkedő elemek (gombok, szövegek) helyét, színét. A statikus szövegeket is az *XML* fájlban írjuk le.

Az XML fájlokban meghatározott felületekhez tartozni fog egy Java osztály, ami a felület működését fogja kontrollálni, a felületen elhelyezkedő gombok működését fogja irányítani és a dinamikus szövegeket változtatni.

Menü

Az alkalmazásban létrehozunk egy menü nézetet. Ebben a nézetben fog tudni a felhasználó váltani a különböző felületek között.

A menü nézetben megfogjuk jeleníteni a felhasználó nevét, és ezen a felületen lesz elérhető a bejelentkezés vagy kijelentkezés gomb is.

Elhelyezünk erre a felületre egy kereső mezőt, amibe a felhasználó beírhat egy tetszőleges címet vagy egy helynek a nevét, és ha rá tapint a helyre, akkor beállítjuk azt a helyet célállomásnak.

Térkép felület

Az egyik nézet osztályunk a navigációs funkciót fogja megvalósítani. Ennek a felületnek meg kell jelenítenie egy térképet. A térkép alatt egy dobozt fogunk megjeleníteni, amin további információkat tudunk kijelezni a felhasználó számára.

A térképen ha egy pontot hosszan tapint a felhasználó, akkor útvonalat tervezünk ahhoz a ponthoz a felhasználó pozíciójától, ezt az útvonalat elmentjük a modell rétegben, és kirajzoljuk a felületre az útvonalat. Ezután az információs dobozt frissítjük a kijelölt célállomás adataival (utca, házszám, kerület, város), és lehetőséget adunk a dobozon belül a felhasználónak hogy felvegye a helyet a kedvencei közé vagy elindítsa a követő üzemmódot.

Ha hozzá akarja adni a kiválasztott helyszínt a kedvencekhez, szükséges megjeleníteni egy *Dialog* ablakot, amiben hozzáadhatja otthoni, munkahelyi vagy kedvenc címnek. Ha a kedvenc címet választja, egy újabb *Dialog* ablakot jelenítünk meg, ahol megadhatja, hogy milyen néven legyen elmentve a cím.

A felhasználó változtatását átadjuk a modell rétegnek, ami elvégzi rajta a szükséges műveleteket, és hiba esetén kijelezzük azt a felhasználó számára.

Ezekhez a *Dialog* ablakokhoz is tartozik egy fentebb említett XML fájl és egy Java osztály, ami a felületi működést biztosítja.

A követő üzemmódot bekapcsolva mindig a felhasználó lesz a térkép középpontjában, és megjelenítünk egy felső sávot is, amelybe kiírjuk dinamikusan a következő kanyart és az ahhoz tartozó információkat. Ezeket az információkat a modell réteg egyik osztálya fogja nekünk biztosítani. Ebből az üzemmódból ki is léphetünk a normál nézetes üzemmódra.

Követő üzemmódban az alsó információs dobozban megváltoztatjuk az eddig kijelzett információkat, és kiírjuk a célba érés időpontját, távolságát, és megjelenítjük a teljes útvonalra a kanyarok listáját.

Egy gombot helyezünk el ezen a nézeten, ami visszanavigálja a felhasználót a térképen a jelenlegi pozíciójához.

Útvonal történet felület

Ebben a nézetben egy listát jelenítünk meg a felhasználó korábban megtett útvonalaival. Ezek az adatok a modellből érkeznek, amikor megnyitjuk ezt a nézetet. Egy lista elemen megjelenítjük, hogy az adott útvonalat a felhasználó melyik nap tette meg, honnan, hova navigált, mennyi idő alatt és mekkora távot tett meg.

Ezek a lista elemek kattinthatóak. Kattintásra betöltünk egy térképet, ahol kirajzoljuk, hogy pontosan milyen útvonalon haladt a felhasználó a kiválasztott történetben.

Amennyiben a felhasználónak nincsen korábbi útvonal története, vagy a modelltől ezek az adatok nem érkeznek meg, tájékoztatjuk erről a felhasználót.

Elmentett helyek felület

Ebben a nézetben szintén egy listát jelenítünk meg, a felhasználó által elmentett helyekkel. Ide tartozik az otthoni és munkahelyi cím, illetve ha van, akkor a további kedvencként elmentett címek is.

Lehetőséget adunk a felhasználónak, hogy ezeket a címeket ezen a felületen keresztül törölje. Ha törlés input érkezik a felhasználó részéről, akkor a modellben az ehhez tartozó műveleteket elvégezzük.

Ha a felhasználó megtapint egy kedvenc címet, akkor átváltunk a térképes nézetre, és az adott címhez navigáljuk a felhasználót, mintha oda tapintott volna.

Bejelentkező felület

Ezen a felületen a felhasználó beütheti a felhasználó nevét és jelszavát, ezeket az információkat továbbítjuk a modell felé, ami elvégzi a megfelelő műveleteket és sikert vagy hibát ad vissza. Ha sikert adott vissza, akkor a felületen frissítjük a felhasználói információt és átváltunk térképes nézetre, ha hibát kaptunk vissza, akkor egy hiba üzenettel jelezzük a felhasználónak, hogy nem sikerült a belépés.

Regisztrációs felület

Ezen a felületen a felhasználó regisztrálhat magának új profilt a megfelelő adatok megadásával (felhasználónév, jelszó, keresztnév, vezetéknév).

Ezeket az adatokat gomb nyomásra tovább küldjük a modellnek, ami vagy sikerrel vagy hibával válaszol a felületnek, és a bejelentkezéshez hasonló módon járunk el itt is.

2.2. Adatbázis

Az adatok tárolásához egy relációs adatbázist építünk fel, melynek a központi táblája a *user_table* tábla lesz, ami a felhasználó adatait fogja tárolni.

Ahhoz, hogy megkülönböztessük a felhasználókat egymástól, szükséges egy *username* adattag, ami a táblában egyedi lesz. Ebben a táblában fogjuk tárolni a felhasználók jelszavait is, természetesen titkosítva. Tárolni fogjuk továbbá a felhasználó vezetéknevét, keresztnevét és egy-egy hivatkozást az elmentett otthoni címére és munkahelyi címére.

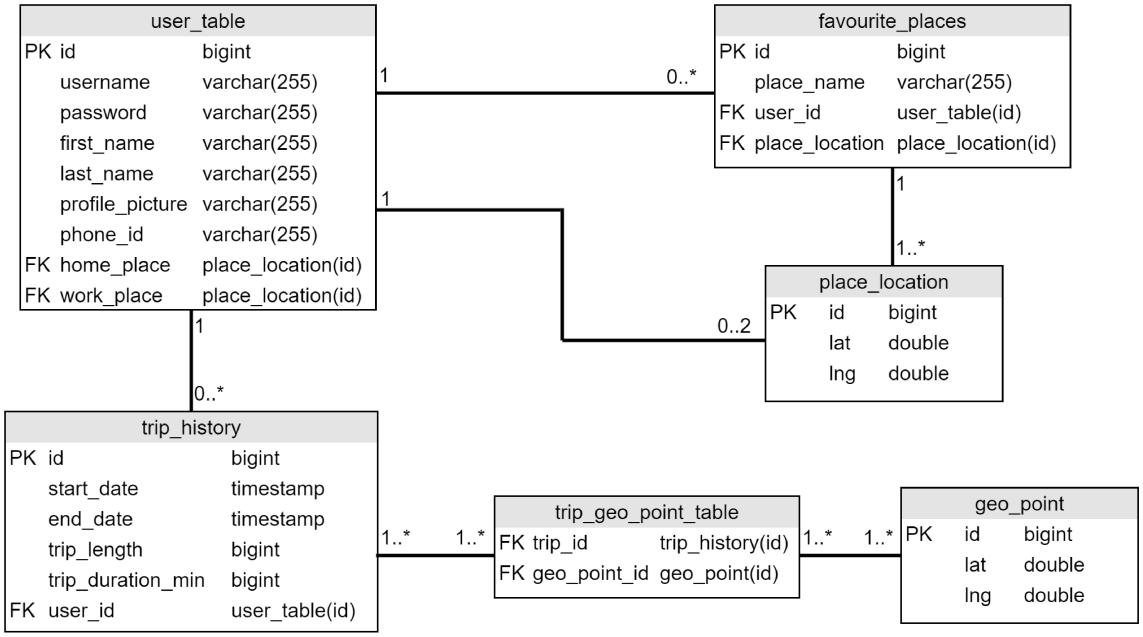
Ezeket a címeket egy külön táblában fogjuk tárolni (*place_location*) melynek két adattagja lesz. Az egyik fogja reprezentálni a szélességi pontot *(lat)* és a másik a hosszúsági pontot *(lng)*, ami a gyakorlatban egy pozíciót fog jelölni a térképen. Az itt elmentett pozíciók ID-jára hivatkozunk a *user_table* táblában.

A specifikációban szerepel, hogy további kedvenc helyek menthetők el a felhasználó számára, erre egy *favourite_places* táblát hozunk létre. Ez a tábla tárolni fogja az elmentett kedvenc hely nevét, egy hivatkozást, hogy melyik felhasználóhoz tartozik (*user_table* tábla ID oszlopára) és egy hivatkozást a korábban létrehozott *place_location* tábla ID-jára ahova elmentjük a kedvenc hely pozícióját.

A felhasználók útvonal történetét és a hozzá tartozó adatokat is mentenünk kell az adatbázisban. Erre a *trip_history* táblát fogjuk létrehozni, ami tárolni fog egy hivatkozást a felhasználóra, egy indulási és egy célba érési időt. Elmentjük a táblába az útvonal hosszát méterben és az útvonal időtartamát percben.

Az útvonal történethez hozzátartozik egy másik tábla, ez lesz a *geo_point*. Ami az előző *place_location* táblához hasonlóan pozíciókat fog tárolni. Azért különítjük el ezt a két táblát egymástól annak ellenére, hogy a két tábla szerkezete megegyezik, mert az utóbbi táblába sokkal több sorra számítunk. Ez azért probléma, mert nagyon meglassíthatja az adatok lekérdezését a táblából, ami nem lenne előnyös a kedvenc helyek pozíciójának lekérdezésekor.

Amiatt, hogy a *geo_point* tábla nagy mennyiségű pozíciók tárolására lett kitalálva, közte és a *trip_history* tábla között *Many To Many* kapcsolatot fogunk kialakítani. Így egy konkrét pozíció nem kerül többször bele ebbe a táblába. Ehhez a *Many To Many* kapcsolathoz a *trip_geo_point_table* kapcsoló táblát fogjuk használni.



3. ábra.

A 3. ábrán láthatjuk az adatbázis táblák pontos definícióját és a köztük lévő kapcsolatot szemléletesen, UML diagram segítségével.

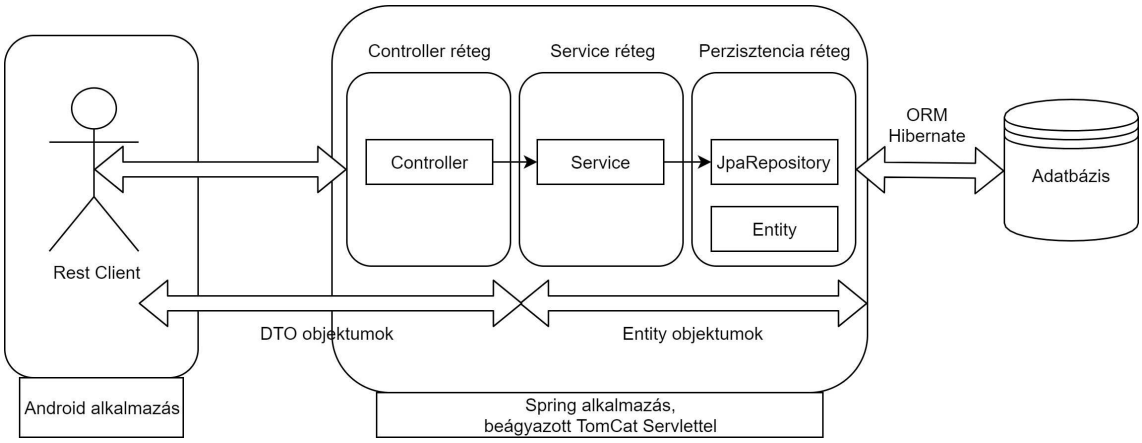
2.3. Szerver alkalmazás

A szerver alkalmazásunk egy Java szerver alkalmazás lesz. Ennek az elkészítéséhez a Spring keretrendszert fogjuk használni. Az alkalmazás könnyű telepíthetősége és futtathatósága miatt Spring boot-ot fogunk használni. A Spring boot alkalmazások indításkor egy alkalmazásba ágyazott Tomcat [2] szervert fognak indítani, így lesz elérhető a szerverünk a külvilág számára.

2.3.1. Architektúra

A Spring-es az alkalmazásoknál elterjedt az MVC architektúra. A mi alkalmazásunk is hasonlóan fog felépülni, de a mi alkalmazásunkban a megjelenítő (View) réteg az Androidos kliens alkalmazás lesz.

A szerver alkalmazás 3 rétegből fog állni. A legalsó réteg lesz a perzisztencia réteg, ami a kommunikációt fogja végezni az adatbázissal. Felette lesz a service réteg, ahol az alkalmazás üzleti logikáját valósítjuk meg. Közvetlenül a service réteg fog kommunikálni a perzisztencia réteggel. A legfelső szinten a kontrollerek fognak állni, amik a HTTP kéréseket fogják fogadni és hívni a service réteg megfelelő folyamatait.



4. ábra.

2.3.2. Perzisztencia réteg

Az adatbázis tervezése során megtervezett táblákhoz szükséges hozzáférnünk, ezért a táblákat Java osztályokként is megfogjuk valósítani. Ezt a technikát ORM-nek (Object-relational mapping) nevezik, amikor át konvertáljuk az adatbázis tábláit osztályokká, a mi esetünkben Java osztályokká. Ezeket az osztályokat entity-nek vagy entitásnak fogjuk nevezni a továbbiakban. A különböző entity-k kapcsolatban állhatnak egymással, de ezekben az entitásokban logikát nem fogunk megvalósítani.

Az egyes entity-khez tartozni fog egy repository interface is, ami a konkrét adatbázis műveleteket fogja végezni, mint például a lekérdezés vagy írás, ezek a JpaRepository (Java Persistence API) leszármazottai lesznek.

A JPA egy interface, ami leírja, hogy milyen műveletekkel tudunk hozzáférni a perzisztencia réteghez, mi esetünkben az adatbázis réteghez. Az alkalmazásunkban a Hibernate-t, a JPA egyik megvalósítását fogjuk használni az adatok hozzáféréshez.

Felhasználók

Az első entitásunk a *user_table* táblához fog kapcsolódni. A korábban megtervezett táblához létre hozzunk egy User Java osztályt, aminek az adattagjai az adatbázisban létrehozott adattagok lesznek. Ebben a táblában korábban két külső hivatkozást is megterveztünk. A *home_place* és *work_place* adattag típusa egy másik entitás lesz majd.

A User osztályhoz tartozik a UserRepository osztály, aminek a korábban említett JpaRepository lesz az ősosztálya. Az ősosztálytól meg fog örökölni több számunkra fontos metódust, mint például az adatok kiírása az adatbázisba vagy az adatbázis tábla kiolvasása. A későbbiekben szükségünk lesz majd arra, hogy a táblából lekérdezhessünk egyetlen sort is felhasználó név alapján, ezért ezt a metódust itt kell majd megvalósítanunk.

Kedvenc helyek

Ehhez a funkciókhoz az adatbázisban két tábla is tartozik, a *favourite_places* és a *place_location* tábla. Mindkét táblához létre kell hoznunk a saját entitásukat, melyek hasonlóan a korábbiakhoz az adatbázisban szereplő adattagokat fogják tartalmazni üzleti logika nélkül.

A PlaceLocation entitás lesz az egyik, amire korábban már hivatkoztunk a User entitásból. A másik a FavouritePlace lesz, ami szintén tartalmazni fog két hivatkozást. Az egyik a korábbi User entity-re a másik pedig a PlaceLocation entity-re.

Ezekhez az entitásokhoz is tartozni fognak repository osztályok. A FavouritePlaceRepository osztályban a JpaRepositoryból megöröklött metódusokon túl szükségünk lesz több fajta lekérdezésre. Az egyik lekérdezés a felhasználó alapján kell, hogy történjen és egy listát várunk a felhasználó kedvenc helyeivel.

A másik lekérdezésünk során input paraméter lesz egy felhasználó és egy kedvenc hely neve, ekkor egy konkrét FavouritePlace-t várunk az adatbázisból.

A PlaceLocationRepository osztályban csak a megörökölt metódusok lesznek, mert nem szükséges kedvenc helyet lekérdeznünk pozíció alapján.

Útvonal történet

Az útvonal történet felhasználói funkcióhoz három táblát terveztünk korábban. A három táblából a két fő tábla a *trip_history* és a *geo_point* tábla, amikhez saját entity osztályokat fogunk létrehozni a megtervezett adattagokkal. A harmadik tábla egy kapcsoló tábla, amihez külön entity nem fog tartozni, mert ebben a táblában csak hivatkozás lesz a másik két táblára. Ezt a táblát a JPA által nyújtott annotációkon keresztül fogjuk elérni, ezt bővebben a megvalósítás fejezetben fejtem ki.

A TripHistory és a GeoPoint entity-hez repository osztályok is fognak tartozni. A TripHistoryRepository osztályban az ősosztály metódusain kívül egy felhasználó szerinti lekérdezésre lesz szükségünk, ami egy listát fog vissza adni.

A GeoPointRepository osztályban szükségünk lesz egy lekérdezésre, annak érdekében, hogy a korábban taglalt ManyToMany kapcsolatot megvalósítsuk. Csak akkor mentünk el egy rekordot az adatbázisba, ha az adott lat, lng páros még nem szerepel benne.

2.3.3. Service réteg

A service réteg fogja az alkalmazásban elvégezni az üzleti logikát. Ez a réteg lesz felelős azért, hogy a repositorykon keresztül kiolvasott adatokat átadja a kontrollereknek. Továbbá ebben a rétegben fogjuk elvégezni a kontrollerből kapott adatokon a hitelesítést.

Általánosságban, amikor a kontrollerből meghívunk egy service valamely függvényét, akkor a kért paramétert először levalidáljuk majd a repository

osztályokon keresztül elmentjük az adatbázisba. Ha a kontroller rétegnek valamilyen információra van szüksége az adatbázisból, akkor a service réteg lesz az, amelyik elkéri a repository-n keresztül az adatokat az adatbázisból és átadja a kontrollernek.

Felhasználó service

A felhasználókkal kapcsolatban két fő feladatot kell megvalósítania ennek a rétegnek. Az egyik egy lekérdezés, egy felhasználónév alapján. Ezt a lekérdezést korábban már megterveztük a felhasználók repository osztályában. Ezt a függvényt kell meghívnunk a service osztályból. Ügyelnünk kell arra, hogy a kontrollertől bármilyen felhasználónevet kaphatunk ezért, ha a repositoryból nem kapunk vissza felhasználót a kérésre egy kivételt kell dobnunk.

A másik feladata a service osztálynak az új felhasználó létrehozása és adatbázisba való mentése. Az adatbázis mentését a perzisztencia rétegben már megtárgyaltuk. A felhasználó elmentése előtt le kell ellenőriznünk, hogy adott User az adott felhasználó névvel létezik-e már, ha igen kivételt kell dobnunk. Amennyiben a felhasználónak az adatai helyesen lettek kitöltve az általa megadott jelszót titkosítanunk kell.

Kedvenc helyek service

A kedvenc helyek service osztálya nem csak a FavouritePlace táblát fogja módosítani hanem a User táblát is. Logikailag ebben a service osztályban helyezkedik el a felhasználók otthoni és munkahelyi címének kezelése is, ezért ezeket itt valósítjuk meg.

Az otthoni és munkahelyi cím elmentésének semmilyen különösebb üzleti logikát nem kell megvalósítania. Ezeknek a függvényeknek a feladata az lesz, hogy a kapott címet beállítják az felhasználónak aki küldte a kérést, és elmentik az adatokat az adatbázisba. Ezeknek a címeknek a törlésekor a felhasználó entity-ről levesszük a hivatkozást és elmentjük újra a felhasználót, így törlődnek az adatok.

Amikor egy új kedvenc helyet mentünk el a cím mellett egy nevet is társítunk a kedvenc helyhez. Mielőtt elmentjük a helyet az adatbázisba meg kell vizsgálnunk, hogy ilyen névvel szerepel-e már kedvenc helye az adott felhasználónak az adatbázisban, ha igen, akkor kivételt fogunk dobni.

Törléskor a kontroller rétegtől egy kedvenc helynek a nevét várjuk. A kapott név alapján és az aktuális felhasználó alapján a repository-n keresztül az adatbázisból kiolvassuk, hogy létezik-e ilyen elmentett hely. Ha nem létezik, akkor kivételt dobunk egyéb esetben pedig a repository rétegen keresztül kitöröljük ezt az adatot.

Útvonal történet service

Az útvonal történet funkció működéséhez a service rétegnek két fő feladata lesz. Az egyik a kontrollertől kapott felhasználó útvonal történetét lekérdezni a repository rétegen keresztül. A másik feladata a lekérdezett adatok visszaadása.

Itt nem kell vizsgálni semmilyen kivételes esetet, mert ha a felhasználónak nincs útvonal története vagy a felhasználó nem létezik, akkor üres listát adunk vissza.

2.3.4. Controller réteg

A controller réteg fogja a bejövő HTTP kéréseket fogadni. Amennyiben a HTTP kérés útvonala nem illeszkedik egy controllerre se, a Spring keretrendszer 404-es (oldal nem található) státusz kóddal fog válaszolni.

A controllerek metódusainak a feladata lesz még az adat konverzió. Mivel REST webservice-eket írunk, így a kontrollerbe a kérések JSON formában fognak érkezni. Ezt a JSON objektumot kell egy entity-vé átkonvertálnunk és ezzel az entity-vel hívnunk a megfelelő service metódusát.

Ahhoz hogy ez a konverzió gördülékenyen működjön DTO-kat (Data Transfer Object) hozunk létre amik megfeleltethetőek az egyes Entityk-nek. A DTO tulajdonsága, hogy csak adattagokat és a hozzájuk tartozó *Getter* és *Setter* metódusokat tartalmazza, de nem szükséges az entity összes adattagját tartalmaznia. Így ha lekérdez a kliens egy User objektumot az adatbázisból, el tudjuk érni a DTO segítségével, hogy az adatbázisban szereplő kényes információt (például jelszó) ne küldjük vissza a kliens számára.

A controller rétegben kell leimplementálnunk a HTTP kérések válaszát is. Amikor egy service metódustól kapunk egy adatbázis objektumot, azt át kell konvertálnunk egy DTO-ra, ami majd a tovább konvertálódik JSON-né a válaszban. Ha a service réteg egy kivételt dobott a végrehajtás során, azt a controllerben elkapjuk és beállítunk a válasz státusz kódjába, egy olyan kódot, ami hibára utal (Pl.: 400 - Bad request, 401 - Unauthorized, 501 - Iternal server error).

A hiba kód mellett egy error kódot is vissza adunk a HTTP válasz törzsében, így a kliensen megfelelő hiba üzeneteket tudunk megjeleníteni.

Három controller osztályt fogok létrehozni, mert ezek működése logikailag elkülöníthető.

User controller

A user controllerbe a felhasználó kezeléshez szükséges webserviceket kell definiálnunk. Ezek közé tartozik, hogy új felhasználót tudjunk regisztrálni, ez egy olyan HTTP kérés, aminek a tartalmában kapjuk meg az új felhasználó adatait. A REST szabvány szerint ennek egy POST kérésnek kell lennie.

A másik webservice, amit ebben az osztályban kell majd megvalósítani, az a felhasználó profiljának letöltése. Itt adatokat kérünk az adatbázistól és ezt küldjük vissza a kliens számára JSON-ben. Ennek a HTTP kérésnek a metódusa GET lesz. Az hogy melyik felhasználó profilját akarjuk letölteni, a kérés URL-jéből tudjuk meg.

FavouritePlace controller

Ebben a controllerben fogjuk elhelyezni az összes olyan webservice-t, ami a felhasználó valamennyi címének a módosítását okozzák.

Az első ilyen webservice-ünk az otthoni cím felvétele lesz. A HTTP kérés törzsében kapjuk meg az adatokat a kliens alkalmazástól, POST metóduson keresztül.

Egy ugyanilyen metódust fogunk felvenni a munkahelyi cím megadásához is, természetesen a két metódus a service réteg különböző folyamatait hívják.

A kedvenc hely felvételekor is egy HTTP POST metódusú kérést várunk. A kérés törzsében szerepelnie kell a hely címének és nevének is.

A kedvenc hely törlésekor egy olyan webservice-t kell megvalósítanunk, aminek következményében az adatbázisból törölni fogunk. Ezért ennek a webservice metódusa DELETE lesz.

TripHistory controller

A TripHistory controllerben az útvonal történethez fűződő webservice-eket valósítjuk majd meg.

Az egyik ilyen webservice at új történet mentése lesz. Ennek a webservice-nek a HTTP metódusa POST lesz.

A másik webservice az, ahol lekérdezhetjük az adatokat a szervertől. Ekkor az adatbázisból csak olvasás történik ezért a HTTP metódus GET lesz. Későbbiekre tekintettel az útvonal történet lekérdezésekor meg kell adni a felhasználó nevet, hogy melyik felhasználó útvonalát akarjuk lekérni. Ezt az adatot ezúttal nem a kérés törzsében várjuk hanem az URL-ben, URL paraméterként.

2.3.5. Autentikáció

A szerveren egy autentikációs folyamatot is meg kell valósítani. Az autentikáció egy HTTP Basic autentikáció lesz, aminek a folyamán a kérések header részében kell küldeni a felhasználónév-jelszó párost. Azt hogy a megadott adatok valósak-e, ellenőrizni fogjuk adatbázisból olvasás segítségével.

Az autentikációval fogunk védeni minden REST webservice-t, kivéve a regisztrációs API végpontot, hogy azt, akkor is hívhassák a felhasználók, ha még nincsenek bejelentkezve. Ha egy védett végpontot próbál meghívni a felhasználó autentikáció nélkül, vagy rossz bejelentkezési adatokkal, akkor 401 státusz kódot fogunk a szerver válaszában elküldeni.

3. Megvalósítás

3.1. Szerver alkalmazás

Ahogy a tervezés során említettem az alkalmazást Spring keretrendszerrel fogom elkészíteni azon belül is a Spring-boot-ot fogom használni. Az projekthez Maven [3] projekt menedzsment és building toolt fogok használni.

A tervezés során felmerültek az alkalmazásnak különböző függőségei (dependecy). Ezeket a függőségeket maven segítségével hozzá kell adjuk a projektünkhöz, hogy használni tudjuk őket. A *pom.xml* egy leíró fájl, amiben az összes dependency és projekt információ megtalálható. A legfontosabb függőségek, amiket hozzá kell adnunk a projekthez:

- spring-boot-starter-data-jpa, spring-boot-starter-web, spring-boot-starter-security. Ezek a függőségek magát a Spring keretrendszert fogják betölteni számunkra.
- postgresql, liquibase-core. Az adatbázishoz tartozó függőségek.
- spring-boot-starter-test, spring-security-test, com.h2database. Ahhoz, hogy az alkalmazásunkat tesztelni tudjuk további függőségeket húzunk be a projektbe.

Egy példa, hogy hogyan tudunk felvenni egy függőséget a *pom.xml*-be

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

A Spring-boot alkalmazásokhoz tartozik egy konfigurációs fájl. Ez a fájl az *src/-main/resources/application.properties*. Ebben a fájlban kulcs-érték párokban kell megadni az alkalmazáshoz tartozó konfigurációkat. Itt konfigurálom be az adatbázis elérését.

Magának a JPA implementációjának, a hibernatenek meg kell adnunk, hogy hogyan kezelje az adatbázis. Ezt a *spring.jpa.hibernate.ddl-auto* kulcs értékének a beállításával tehetjük meg. A mi alkalmazásunkban a hibernate-nek nem kell létrehoznia az adatbázist és frissítenie sem kell. Az adatbázis módosításait mi fogjuk egy új tool-al megtenni. Ezért ezt a beállítást *validate*-re állítjuk, hogy a létrehozott entity-ket és annak adattagjait hasonlítsa össze az adatbázisban szereplőkkel.

3.1.1. Adatbázis kezelése

Az adatbázist a szerver alkalmazás fogja kezelni. Az adatbázis karbantartásához a *liquidbase* nevű tool-t fogjuk használni. Ennek a toolnak a segítségével könnyen

tudjuk nyomon követni az adatbázis változásokat és könnyen tudjuk verziókezelni az adatbázist. Az alkalmazás properties fájljában bekonfiguráltok a *liquidbase* scriptünket, ami felelős lesz az adattáblák létrehozásáért. Ez a script a *src/main/resources/db/changelog/db-changes.xml* fájl lesz. SQL parancsok helyett XML formában fogjuk leírni az adatbázisunkat. Liquidbase-ben az adatbázis módosításához úgy nevezett *<changeSet>* -eket kell létrehoznunk. A changeSet-nek meg kell adnia az ID-ját és a szerzőjét (author). Az ID tagnak egyedinek kell lennie. Minden changeSet csak egyszer fog lefutni és a lefutott changeSet IDja el lesz tárolva az adatbázis egy táblájában. Innen fogja tudni a script, hogy melyik changeSet-ek az újak és melyeket kell még lefuttatni. Nézzünk meg egy példát egy táblának a létrehozására liquidbase segítségével. A következő changeSet-et vesszük fel:

```
<changeSet id="1" author="balazs">
   <createTable tableName="user_table">
      <column name="id" type="bigint" autoIncrement="true">
         <constraints primaryKey="true" nullable="false"/>
      </column>
      <column name="username" type="varchar(255)">
         <constraints nullable="false"/>
      </column>
      ...
      <column name="home_place_id" type="varchar(255)" />
      <column name="work_place_id" type="varchar(255)" />
   </createTable>
</changeSet>
```

Ebben a changeSet-ben a *user_table* táblát hozzuk létre. Láthatjuk, hogy kell megadni a tábla nevét és oszlopait és azoknak típusait. Ha jól megnézzük ezt a changeSet-et láthatjuk, hogy a tervezéssel ellentétben itt még az otthoni cím és munkahelyi cím nem egy idegen kulcs egy másik táblára, hanem csak egy szöveges mező. Ezt egy későbbi changeSetben javítottam:

```
<changeSet id="21" author="balazs">
   <dropColumn tableName="user_table" columnName="home_place_id"/>
   <addColumn tableName="user_table">
   <column name="home_place" type="bigint">
      <constraints foreignKeyName="fk_home_place_location_id"     ↩
         references="place_location(id)"/>
   </column>
   </addColumn>
</changeSet>
```

Ebből a példából látható miért fontos a liquidbase használata, hogy lássuk hogyan fejlődött és változott az adatbázisunk.

3.1.2. Perzisztencia réteg

Az alkalmazásban az entitásokat a *com.balazslombosi.bikeup.entity* Java csomagban fogjuk elhelyezni. A következő entitásokat fogjuk létrehozni:

- User - A user_table táblához tartozó entitás.
- FavouritePlace - A favourite_place táblához tartozó entitás.
- PlaceLocation - A place_location táblához tartozó entitás.
- TripHistory - A trip_history táblához tartozó entitás.
- GeoPoint - A geo_point táblához tartozó entitás.

Ezeket az osztályokat különböző annotációkkal kell ellátnunk, hogy a Hibernate összetudja kapcsolni a Java osztályokat az adattáblákkal. A legfontosabb annotációk:

- @Entity - Ezzel az annotációval el kell látnunk az összes Java osztályt. Ez fogja megmondani a rendszernek, hogy ezek nem sima osztályok hanem adatbázisban szereplő táblához tartoznak.
- @Table(name="...") Ezzel az annotációval adjuk meg, hogy az adott osztály melyik táblának feleltethető meg.
- @Column(name="...") Ezzel az annotációval adjuk meg, hogy az adott adattag melyik oszlopnak feleltethető meg.
- @OneToOne, @OneToMany, @ManyToMany Ezekkel az annotációkkal fűzünk össze két entitást. Két paramétert használunk ezeknél az annotációknál.

  A *fetch = FetchType.LAZY* arra szolgál, hogy, amikor egy adatot kiolvasunk az adatbázisból, akkor ha a LAZY fetch type rá van annotálva, akkor azokat az adatokat nem tölti be az alkalmazás a memóriába. Ez különösen hasznos, amikor nagyon mennyiségű pontot kéne betöltenünk egy útvonal vagy a kedvenc helyek betöltésekor.

  A másik paraméter a *cascade = CascadeType.ALL*, ez a paraméter arra szolgál, hogy ha például a User java objektumnak beállítjuk a homeLocationjét és elmentjük a User objektumot az adatbázisba, akkor mivel rá annotáltuk a homeLocation-re a cascadetype.all -t ezért azt az objektumot is elfogja menteni az adatbázisba.

Tervezés során TripHistory és GeoPoint entitások között lévő kapcsolatot az adatbázisban egy kapcsolótáblával tettük meg. Implementáció során ezt a kapcsolótáblát nem hoztuk létre külön entitásként. Kettejük közötti kapcsolatot a TripHistory entitásban egy annotáción keresztül fogjuk megadni.

```
@JoinTable(
   name="trip_geopoint_table",
   joinColumns=@JoinColumn(name="trip_id",referencedColumnName="id"), ↩
      inverseJoinColumns=@JoinColumn(name="geo_point_id", ↩
      referencedColumnName="id"))
```

Az annotáció első paramétere a kapcsoló tábla neve, második paramétere egy újabb annotáció, hogy a TripHistory entitásnak az ID-ja a kapcsoló tábla melyik mezőjére fog hivatkozni. A harmadik paraméter ismét egy annotáció, hogy a másik entitás (GeoPoint) ID-ja melyik mezőre fog hivatkozni. A kapcsolat másik entitásában a GeoPoint-ban megadjuk, hogy melyik mezőn van megadva ez az annotáció a *@ManyToMany* annotáció *mappedBy* paraméterén keresztül.

Az entitásokhoz tartozó repository interface-k a *com.balazslombosi.bikeup.repository* csomagban fogjuk elhelyezni. Ebben a csomagban megtalálunk minden entitáshoz tartozó repository interface-t. Ezeknek az interface-knek közös tulajdonságuk, hogy mindegyik elvan látva *@Repository* annotációval és a JpaRepository-ból származik, hogy megörökölje a JpaRepository metódusait. Interface-k tehát csak metódus deklarációkat tartalmaznak implementációt nem. A metódusok alá az implementációt a Hibernate fogja adni. Az interface-ben létrehozott metódusok nevei egy szintakszis szabályból adódnak, ami alapján letudja generálni a Hibernate az SQL Query-ket. Minden lekérdezés a findBy kulcsószval kezdődik. A findBy után az adattag elnevezése következik, ami alapján szeretnénk a lekérdezésünket elvégezni például a UserRepository-ban szereplő findBy a felhasználó név alapján kérdez le az adatbázisból:

```
   User findByUsername(String username);
```

Amennyiben több feltételt is meg szeretnénk adni a szűrés paraméterében azokat az adattagokat az And vagy Or kulcsszóval kell elválasztani például:

```
   FavouritePlace findByUserAndPlaceLocation(User user, ↩
      PlaceLocation placeLocation);
```

3.1.3. Service réteg

Tervezés során 3 service osztályról beszéltünk ezek a *UserService*, *FavouritePlaceService* és a *TripHistoryService*. Ezeket az osztályokat a *com.balazslombosi.bikeup.service* csomagba fogjuk elhelyezni. Közös tulajdonsága ezeknek az osztályoknak, hogy az osztályok a @Service annotációval vannak ellátva. Az annotáció azért fontos, mert így jelezzük a keretrendszer felé, hogy ez az osztály nem egy sima osztály hanem egy Spring komponens (bean). Ezeket a komponenseket az alkalmazásunkba sehol nem példányosíthatjuk ezt a keretrendszer megteszi helyettünk. Minden bean csak egyszer lesz példányosítva, és ahol használnunk kell őket ott az @Autowired metódussal fogjuk be inject-elni az adott osztályba, ezt a technikát dependency inejction-nek nevezzük.

```
@Autowired
private UserRepository userRepository;
```

Ugyanilyen bean-ek a repository osztályaink. A service osztályok adattagjai a megfelelő repository osztályok lesznek dependency injectionnel elhelyezve az osztályba.

A service osztályokban megtalálható egy metódus a *getCurrentUser()*, ami a security contextből kiolvassa az éppen autentikált felhasználó nevét és ezzel a névvel visszaadja a User objektumot miután lekérdezte az adatbázisból. Erre azért van szükség, mert több metódus is a jelenleg autentikált felhasználó adatait módosítja.

A service osztályok több folyamata is hibára futhat ilyenkor kivételt dobunk a service osztályból. A kivételek könnyű kezeléséhez létrehozunk a *com.balazslombosi.bikeup.exception* csomagba egy saját **ServiceException** kivétel típust. Ennek a kivétel osztálynak lesz egy típus adattagja, ami *ServiceExceptionType* típusú. A ServiceExceptionType egy felsoroló típus, amiben definiáljuk az összes lehetséges service hibát.

3.1.4. Controller réteg

A kontrollereket a *com.balazslombosi.bikeup.controller* csomagban találjuk meg. Három kontroller osztályunk lesz, ezek a *UserController*, a *FavouritePlaceController* és a *TripHistoryController*.

A keretrendszer 2 fajta kontroller annotációt biztosít számunkra, az egyik a sima @Controller, amit MVC alkalmazásokban szoktak használni, a másik pedig a @RestController annotáció. Mi a kontrollere osztályainkat az utóbbival fogjuk felannotálni. Ez az annotáció fogja biztosítani, hogy az osztály metódusainak a visszatérési értéke a HTTP válasz törzsébe kerüljön bele.

A kontroller osztályainkra további annotációt rakunk ez a *@RequestMapping("/api")*, ezzel definiáljuk, hogy a REST Api végpontjai a */api* URL elérés alatt

lesznek. A service réteghez hasonlóan a kontrollerkhez be injektáljuk a megfelelő service osztályokat.

A kontrollerek metódusait ha fel annotáljuk a *@RequestMapping* annotációval akkor azok webservice-ként fognak működni. Az annotáció *value* paraméterében megadhatjuk a webservice elérését a *method* paraméterében pedig a HTTP metódust, amin keresztül hívjuk.

A webservice metódusok paraméter listájába felveszünk egy *HttpServletResponse* típusú paramétert. Ezen a paraméteren keresztül fogjuk beállítani a HTTP válasz státusz kódját például:

```
httpResponse.setStatus(HttpServletResponse.SC_BAD_REQUEST);
```

Szemléltetés céljából nézzük meg a következő kontroller osztályt:
UserController.java - ebben a kontoller osztályban két webservicet valósítunk meg.

Az új felhasználó létrehozása a egy POST metódusú webservice, ami a /api/users útvonalon érhető el. A metódus első paramétere a @RequestBody annotációval van ellátva, ami azt jelenti, hogy a HTTP kérés törzsében egy adott DTO típusú JSON objektumot várunk. Második paraméter pedig a korábban említett HTTP válasz objektum.

```
@RequestMapping(value = "/users", method = RequestMethod.POST)
public BaseResponseDTO createNewUser(@RequestBody ↩
    WebRegistrationRequestDTO userRequest,
HttpServletResponse httpResponse)
```

A felhasználó profiljának lekérdezése a másik webservice, ami ebben az osztályban szerepel. Ennek első paramétere szintén annotált de ezúttal egy @PathVariable annotációt használunk, ami azt jelenti hogy a webservice elérésének címében egy változót helyezünk el. Ebben az esetben a felhasználó név került átadásra a URL címben.

```
@RequestMapping(value = "/users/{username:.+}", method = ↩
    RequestMethod.GET)
public BaseResponseDTO getUserByUsername(@PathVariable String ↩
    username, HttpServletResponse httpResponse)
```

Az alkalmazásnak egy érdekes része az útvonal történetek a ManyToMany kapcsolata a GeoPointal. Amikor beérkezik a TripHistoryControllerbe a kérés a kliens felől az új történet elmentéséhez a kontroller metódusában megvizsgálom a beérkezett GeoPoint-okat. Ha egy GeoPoint-ot megtalálok az adatbázisban, akkor a létrehozott TripHistory entitást azzal a GeoPoint-al fogom összekapcsolni, egyéb esetben pedig új GeoPoint entitást hozok létre.

```
historyRequest.getLocationHistoryList().forEach(point -> {
   GeoPoint p = geoPointRepository.findByLatAndLng(point.getLat(), ↩
       point.getLng());
   if (p != null) {
      List<TripHistory> trips = p.getTripHistory();
      trips.add(tripHistory);
      geoPointRepository.findByLatAndLng(point.getLat(), ↩
          point.getLng()).setTripHistory(trips);
   } else {
      p = new GeoPoint();
      p.setLat(point.getLat());
      p.setLng(point.getLng());
      List<TripHistory> trips = new ArrayList<>();
      trips.add(tripHistory);
      p.setTripHistory(trips);
   }
locationList.add(p);
});
```

3.1.5. DTO osztályok

A DTO osztályokat a *com.balazslombosi.bikeup.dto* csomagban találhatjuk meg. A csomagon belül megtalálhatjuk az entitásoknak megfeleltethető DTO osztályokat. A csomagban további két csomagot helyezünk el a *dto.request* és *dto.response* csomagokat. Ezekben a HTTP kérésekben és annak a válasz törzsében szereplő DTO osztályokat helyezzük el. A DTO és a JSON közötti konverziókat nem mi fogjuk elvégezni kézzel, hanem egy *Jackson* nevű könyvtárat fogunk használni. Ez a könyvtár a Spring keretrendszerrel együtt működik, így nekünk külön kódot nem kell írnunk, hogy ezek a konverziók megtörténjenek.

RequestDTO osztályok

Ezek az osztályok a HTTP kérések törzsében szereplő JSON objektumokat fogják leírni Java osztályként. Minden adattagjuk privát és mindegyikhez létezik getter és setter metódus, emellett kell, hogy legyen paraméter nélküli konstruktora.

ResponseDTO osztályok

Minden webservice metódus visszatérési értéke egy ResponseDTO lesz. Ahhoz, hogy ezeket a válasz DTO-kat könnyen tudjuk kezelni létrehozunk egy *BaseResponseDTO*

osztályt. Ennek az osztálynak két adattagja lesz.

```
private ServiceExceptionType error;
private String message;
```

Ha egy kontroller osztályban elkapunk egy ServiceExceptiont akkor a BaseResponseba beállítjuk annak a típusát az error adattagba, és további üzenetet küldhetünk a kliens felé.

Mivel nem tudjuk előre, hogy a kérés sikerre vagy hibára fog futni ezért minden kontroller metódusnak a BaseResponseDTO lesz a visszatérési típusa. Ezért minden további ResponseDTO a BaseResponseDTO típusból fog származni. Ha nem fut hibára a webservice metódus akkor egy konkrét DTO típussal térünk vissza egyébként pedig a BaseResponseDTO-val.

3.1.6. API konfiguráció és security

A szerver egyes végpontjait csak autentikált felhasználók érhetik el. Azokat a java osztályokat, amik ezeket a konfigurációkat tartalmazzák, a *com.balazslombosi.bikeup.config* csomagban találjuk meg. A WebSecurityConfig.java osztályban az API végpontok konfigurációján kívül az autentikációt is megvalósítjuk a *configureGlobal* metódusban. Az autentikációnál megadott felhasználónév alapján lekérdezzük a felhasználót az adatbázisból. Ha a User object nem null akkor a *BCryptPasswordEncoder* bean segítségével összehasonlítjuk az elkódolt jelszót a felhasználó által megadott jelszóval. Az autentikáció konfigurációjához hozzátartozik a *MyBasicAuthenticationEntryPoint* osztály is, ami a HTTP Basic autentikációt fogja konfigurálni.

Ebben az osztályban elhelyezkedő osztályokon az @Configuration annotációt használjuk. A keretrendszer ezzel az annotációval ellátott osztályokat induláskor betölti és alkalmazza a beállításokat.

Az autentikáció a HTTP protokoll mellett, a biztonságosabb HTTPS protokollon keresztül is működik.

3.2. Android alkalmazás

Az Android alkalmazás Android Studio fejlesztői környezet alatt készült. Az IDE-ben beépített Gradle projekt és dependency menedzsment tool található, ezért ezt használtam az alkalmazáshoz.

Az Android alkalmazásban a tervezés során elkülönítettük a modell és a view réteget egymástól. Ez a megvalósítás során az Android projectben két teljesen külön modul lesz. A modell réteg a bikeupservice, a view réteg pedig az app modul lesz.

Minden modulnak megvan a saját *build.gradle* fájlja. Ezekben a fájlokban beállítjuk a modulok dependency-jeit. Mindkét modulnak betöltjük az OpenStreetMap térkép könyvtárat és az ehhez tartozó bonuspack-et. Az app modulba még pluszba felveszünk a megjelenítéshez a google által szolgáltatott könyvtárból design elemeket. A dependency felvételére egy példa:

```
compile 'org.osmdroid:osmdroid-android:5.6.5'
compile 'com.github.MKergall:osmbonuspack:6.3'
```

3.2.1. BikeUpService modul

A modell réteg központi osztálya a *BikeService* lesz. Ez az osztály az Android *Service* osztályából fog származni. Számunkra ez azt jelenti, hogy ez egy háttérben futó osztály lesz, azok a műveletek, amik itt mennek végbe nem fogják blokkolni a UI threadet. Ennek az osztálynak a feladata lesz, hogy példányosítsa az adattagjaiba a modell további osztályait. A BikeService osztálynak saját metódusai nem lesznek, inkább egy interface lesz a UI és a tovább modell osztályok között. A modellben további három package-et hozunk létre

User package

A User package-t a *com.balazslombosi.bikeupservice.user* alatt találjuk meg. Ebben a csomagban valósítjuk meg a felhasználók kezeléséhez tartozó műveleteket. A felhasználók műveleteinek leírására létrehozunk egy public interface-t a *IUserOptions*-t. Ebben az osztályban definiáljuk, hogy a User osztályoknak milyen metódusokat kell mindenképp implementálniuk.

A Userek egy abstract ősosztályból fog származni a *AUser*-ből. Ebben az osztályban létrehozzuk a Userhez tartozó összes adattagot és azok getter és setter metódusait. Már az ősosztályban implementáljuk a *IUserOptions* interface-t hogy a leszármazottakba ne felejtsük el megvalósítani.

Az AUser absztrakt osztály első leszármazottja az *AnonymUser* lesz. Az interface metódusait megvalósítja, kettő kivételével. Az *addFavouritePlace* és a *removeFavouritePlace* metódus hívásakor kivételt dobunk. Erre a célra létrehozunk egy *UserOptionException* kivételt.

A többi metódus megvalósításkor beállítja a megfelelő adatokat, majd meghívja a *SaveUserToPreferences* folyamatot. Ennek a folyamatnak a feladata a User objektum elmentése lesz a telefonra, hogy az adatokat visszatölthessük, hiszen anonim felhasználó esetén nem beszélünk szerver oldali támogatásról.

A Regisztrált User osztályunk az AnonymUser-ből fog származni így meg tudja örökölni az ott megvalósított metódusokat. Ezért nem kell ebben az osztályban is leimplementálnunk, például a User telefonra mentését. Az interface-ben szereplő

metódusokat felüldefiniáljuk és az ősosztály metódusának hívása után a megfelelő webservice-t meghívjuk az egyes metódusokban. Ezeket a webservice hívásokat a *UserService* osztályba külön kiemeltem. Az *addFavouritePlace* és *removeFavouritePlace* metódust ebben az osztályban exception nélkül implementáljuk.

A *UserManager* osztály lesz felelős magáért a felhasználó kezelésért. Ebben az osztályban valósítjuk meg a bejelentkezést, kijelentkezést és szinkronizációt.

A *loginUser* metódus valósítja meg a bejelentkezést. A két paramétert (felhasználónév és jelszó) a nézet réteg adja át számunkra, ezeket eltároljuk, majd egy webservice híváson keresztül lekérjük az adott felhasználónévvel a User-t a szervertől. Ha sikeres volt a kérés, akkor ezt átalakítjuk RegistredUser típusú objektummá és elmentjük a telefonba.

A *syncUserFromSavedCredential* metódus az alkalmazás indításakor fog lefutni. Ez a metódus egyből bejelentkezteti a felhasználót. Ha a bejelentkezés valamilyen okból sikertelen (nem érhető el a szerver vagy nincs internetkapcsolat a telefonon) akkor a telefonban eltárolt User object-et fogjuk betölteni. Ezért a betöltésért a *setUserFromSharedPreferences* metódus fog felelni. Ez a metódus lefutása végén küld egy eseményt a nézet rétegnek, hogy megváltozott a felhasználó, frissítsük a nézetet.

A kijelentkezés a *logoutUser* metódus. Ebben a metódusban létrehozunk egy új AnonymUser objektumot és beállítjuk azt a jelenlegi felhasználónak. Ezzel az objektummal felülírjuk a korábban elmentett RegistredUser objektumot így a felhasználót teljesen kiléptettük az alkalmazásból.

Restservice package

Ebben a csomagban (*com.balazslombosi.bikeupservice.restservice*) fogjuk megtalálni az összes olyan implementációt, ami a szerverrel való kommunikációhoz szükséges.

A DTO csomagban ugyanazokat az osztályokat fogjuk megtalálni mint a szerver alkalmazás DTO csomagjában. Ez fontos, hogy ezek az osztályok megegyezzenek mert ez az interface a kliens és szerver alkalmazás között. Ez a dolog annyival kiegészül, hogy létrehozunk egy üres BikeRequestDTO osztályt is és a Request DTO osztályokat ebből származtatjuk le. Ez a későbbiekben fontos lesz számunkra.

Létrehozunk egy RestApiEndPoints osztályt, aminek a fő tulajdonsága, hogy itt lesznek definiálva a webservice elérhetőségek. A konstruktor paraméterül kapja a BikeService kontextusát azért, mert ezen az osztályon keresztül fogjuk elmenteni az autentikációs adatokat a *SharedPreferences*-be. Ebben az osztályban a metódusok static metódusok lesznek, hogy ne legyen szükséges példányosítanunk mindenhol ezt az osztályt ahol szükségünk van egy webservice elérésére. A metódusok közös jellemzői, hogy *HttpUrlConnection* a visszatérési típusuk, ami maga az *UrlConnection* objektum lesz. Ezt az objektumot úgy hozzuk létre, hogy példányosítunk egy *Url*

objektumot, aminek megadjuk a webservice elérési címét. Ezen az URL objektumon keresztül létrehozzuk a *HttpUrlConnection* objektumot, majd beállítjuk neki az autentikációs header adatokat. A *urlConnection.setRequestMethod* -on keresztül pedig beállítjuk, hogy milyen HTTP metódussal legyen hívva az adott webservice.

Az autentikációs adatok beállítását kiemeltem a *setAuthentication* függvénybe, ahol, a SharedPreferences-ből kiolvasott felhasználónevet és jelszót *Base64*-ben kódoljuk és az így kapott stringet rakjuk bele headerbe. A szerver alkalmazás így fogja várni tőlünk ezeket az adatokat.

Ennek a csomagnak a fő osztálya a *RestApiCallerTask* lesz, ami magát a HTTP kérést fogja lebonyolítani. Ezt az osztályt az *AsyncTask* osztályból származtatjuk, hogy aszinkron történjen a HTTP kérés, ezt egyébként az Android operációs rendszer el is várja tőlünk.

Az osztálynak lesz egy típus paramétere. Ez a paraméter a kérés, válasz objektumának a típusa lesz, ezért ennek a BaseResponseDTO-ból kell származnia. A konstruktorba paraméterül átadjuk többek között magát az UrlConnection és RequestBody objektumokat. Továbbá átadunk egy Class<T> típust, ami a típus paraméternek a futás idejű osztálya. Negyedik paraméter egy callback függvény lesz, ami akkor fog lefutni ha az aszinkron folyamat befejezte a futást.

Ezek után a *doInBackground* metódusban megvalósítjuk magát a HTTP kérést. A szerver alkalmazásból ismert Jackson könyvtárat fogjuk használni a Java objektumok JSON objektummá való konvertálásához, ám ez itt nem történik automatikusan. Az ObjectMapper osztály *writeValue* metódusával tudjuk átalakítani a Java objektumainkat JSON-né és a *readValue*-val vissza felé. A szerver válaszát egy stream objektumból tudjuk kiolvasni egy stringbe ez lesz a metódus visszatérési értéke. Ha kivételt kapunk a kiolvasás közben akkor egy error-t rakunk bele ebbe a strinbe.

Az *onPostExecute* paraméterében megkapjuk a szerver válaszát. Most lesz fontos számunkra, hogy a konstruktor megkapta a válasz DTO class típusát mert az ObjectMapper readValue metódusának szüksége van erre. Miután át alakítottuk Java objektummá a szerver válaszát, megnézzük, hogy az error adattag, ami a BaseResponseDTOban szerepel üres-e. Ha üres akkor az *onRequestSuccess* callback függvényt hívjuk meg a response object-el, egyébként pedig az *onRequestError*-t.

Egy példa ennek az osztálynak a használatára:

```
public void saveHome(PlaceLocationDTO requestDTO) {
   RestApiCallerTask<BaseResponseDTO> apiCaller = new ↩
       RestApiCallerTask<> (RestApiEndPoints.postHomePlace(), ↩
       requestDTO, BaseResponseDTO.class, new ↩
       RequestCompleteListener() {
      @Override
      public void onRequestSuccsess(BaseResponseDTO responseDTO) {
         ...
      }

      @Override
      public void onRequestError(BaseResponseDTO responseDTO) {
         ...
      }
   });
   apiCaller.execute();
}
```

Location package

A (*com.balazslombosi.bikeupservice.location*) csomagban fogjuk megvalósítani az összes olyan funkciót, ami a pozíció meghatározásához köthető. Így ebbe a csomagba a következő osztályok lesznek: *MyGpsLocationProvider*, *RoutingService*, *TripHistory*.

A **RoutingService** osztályban fogjuk eltárolni az útvonalat és a kiválasztott címet. Az útvonal egy *Road* típusú objektum, aminek a fő adattagjai:

- Az *mRouteHigh* egy *ArrayList<GeoPoint>* típusú adattag. Ebben az adattagban lesznek eltárolva azok az útvonal pontok, amik szükségesek a megjelenítéshez. Ezek a pontok olyan helyeken helyezkednek el ahol egy kanyar van, vagy az út megtörik, így az egyenes vonalat is meg kell törni.

- Az *mNodes* adattag szintén egy lista lesz, ami *RoadNode* típusú objektumokat fog tárolni. Ezek az objektumok a kanyarokat leíró osztályok. Bennük megtalálható a kanyarhoz tartozó információs szöveg, a kanyar iránya, pozíciója és hogy a következő kanyarig mekkora távot kell megtenni km-ben.

- Az *mDuration* az útvonal időtartamát tárolja másodpercben megadva.

- Az *mLength* pedig az útvonal hosszát km-ben megadva fogja tárolni.

Az útvonal történet logikailag ide tartozik, így ennek az osztálynak az adattagja lesz a *TripHistory* is.

A *finishTrip* metódus meg fogja hívni a *TripHistory* osztály finishTrip metódusát, majd elküldjük az útvonal történetet a szerver felé.

A **TripHistory** osztályban egy listát fogunk tárolni azokról a *GeoPoint*-okról, amiket már elhagytunk.

Amikor meghívódik az osztály *finishTrip* metódusa kiszámoljuk a megtett út hosszát és időtartamát.

A **MyGpsLocationProvider** osztály lesz felelős a telefon GPS hardware-e által szolgáltatott pozíció feldolgozásáért. Ez az osztály a *GpsMyLocationProvider* ősosztályból fog származni, ami az OSM térkép szolgáltató könyvtárban található meg. Az ősosztályban található a pozicionálás konfigurálása, indítása, a pozíció feldolgozása és továbbítása a nézet modul felé. A mi osztályunkban a kapott pozíció által szükséges adatokat fogjuk kiszámolni és elmenteni.

Az osztályban a következő segédfüggvényeket fogjuk implementálni:

- A *distance* metódus a két paraméterül kapott pont távolságát fogja kiszámolni. Ezek a pontok GeoPoint típusúak egy latitude és egy longitude koordinátát tartalmaznak. A függvény visszaadja a két GeoPoint távolságát méterben.

- A *getDistanceFromActualRoute* paraméterül várja az útvonal első két pontját, és a jelenlegi pozíciónkat. Ebből a pontból kiszámítja, hogy hány méterre vagyok jelenleg az útvonaltól. Ezt úgy teszi meg, hogy a három pont által kijelölt háromszög jelenlegi pozíciónk csúcsához tartozó magasságot számolja ki.

- Három darab segédfüggvényt találunk még ebben az osztályban, amiknek a feladata, hogy egy eseményt váltsanak ki, amire az *App modul* feliratkozik és a megfelelő felületi elemet frissíteni fogja.

Ebben az osztályban az *onLocationChanged* metódus fog a középpontban állni, ami minden egyes alkalommal le fog futni, amikor az operációs rendszer jelet küld az alkalmazásnak, hogy megváltozott a pozíció. A paraméterben kapott Location típusú objektumban fogunk megtalálni minden információt, amit a telefon átad számunkra. Ilyen adatok lesznek többek között a hosszúsági és szélességi pont, a pozíció pontossága és a pillanatnyi sebesség. Ebben a függvényben több dolgot is ki kell számolnunk, hogy a felület friss információkat tudjon kijelezni.

Először is fontos, hogy ezek az információk mind az útvonal tervezéshez illetve magához az útvonalhoz köthetők így ha nincs megtervezett útvonal nem számolunk semmit.

A metódusban először megvizsgáljuk a pozíciónk és az útvonal távolságát. Erre hoztuk létre a *getDistanceFromActualRoute* segédfüggvényt. Ez arra lesz alkalmas

számunkra, hogy kiderítsük, hogy a felhasználó az útvonalon halad-e vagy eltért tőle. Ha ez a távolság nagyobb mint egy paraméterben megadott szám, akkor az útvonalat újra kell terveznünk. Ehhez a *ACTION_NEW_ROUTE_CALC_NEEDED* eseményt váltjuk ki.

A következő vizsgálatunkban azt ellenőrizzük, hogy a pillanatnyi pozíció és az útvonalon a második pont távolsága kisebb-e, mint egy paraméterben megadott szám. Ez a paraméter most a location objektum pontossága lesz. Ha kisebb akkor meghívjuk a *refreshRemainingRodeNodes* függvényt, ami kitörli az első pontot, hiszen a másodikhoz már elég közel vagyunk. Ez a metódus az *ACTION_NEW_ROUTE* eseményt küldi el, hogy a felület újra tudjon rajzolni. Ezek mellett pedig a kitörölt pontot feldolgozzuk és beletesszük az útvonal történetbe, amit majd a szerver felé elfogunk küldeni.

A kanyarok frissítéséhez megnézzük, hogy a kanyar listában szereplő 1. indexű, azaz második kanyarnak a paraméterben megadott környezetébe értünk-e. Azért a második elemet fogjuk vizsgálni mert az a következő kanyar, de az első elemben van megadva a távolsága. Ha elég közel értünk ehhez a ponthoz, akkor frissítjük a hátra lévő kanyarok listáját *refreshRemainingRodeNodes* metódussal. Egyéb esetben kiszámoljuk a következő kanyar távolságát, ezt a 0. indexü elemben fogjuk tárolni. A *updateRemaningTimeAndKmInformations* metódusban frissítjük, hogy a teljes útból hány km van hátra és, hogy mikor fogunk megérkezni a célhoz.

Az utolsó vizsgálatban azt fogjuk ellenőrizni, hogy az utolsó pont paraméterben megadott közelségébe értünk-e. Ha igen akkor a még nem feldolgozott útvonal pontokat hozzáadjuk az útvonal történethez. Ezután eseményt küldünk, hogy az útvonal történet itt befejeződött.

3.2.2. App modul

Az *src/main/AndroidManifest.xml* fájl lesz az Android modul konfigurációs fájlja. Ebben a fájlban határozzuk meg, hogy a felhasználónak milyen engedélyeket kell megadnia az alkalmazásnak, hogy az működhessen. Ebben a fájlban továbbá definiálunk két Activity-t. Az Androidos alkalmazások ilyen activityk-ből állnak, ezek jelennek meg.

A megjelenítéshez tartozó Java osztályokat a *src/main/java*, a megjelenítést leíró XML osztályokat az *src/main/res/layout* könyvtár alatt találjuk. Ebben a resource mappában még több az alkalmazás megjelenítéséhez szükséges fájl található, mint például képek, illetve statikus szövegek *(src/main/res/values/strings.xml)*.

Az Android keretrendszerből származtatott megjelenítő osztályoknak (Activity, Fragment) a következő metódusai vannak, amiket a legtöbb esetben felül kell írnunk.

- Az *onCreate* az adott osztály inicializálásakor fut le. Ebben a metódusban vesszük ki az XML objektumból a megjelenítendő objektumokat és társítjuk hozzá egy Java objektumhoz. Itt fűzzük hozzá a megjelenített gombokra az *onClickListener* metódust, ami akkor fog lefutni, amikor az adott gombra rátapint a felhasználó.

- Az *onResume* függvény fut le, amikor egy view osztály betöltődik az alkalmazásba. Ebben azokra az eseményekre iratkozunk fel, amelyikre az osztályban figyelni kell.

- Az *onPause* függvény akkor fut le, amikor egy view osztály a háttérbe kerül és nem látható az alkalmazásban. Ekkor ideiglenesen leiratkozunk az eseményekről, mert ha nincs előtérben a felület, nem szükséges, hogy műveleteket végezzen.

SplashActivity

Az első activity-nk a **SplashActivity**, ami az alkalmazás betöltésére, jogosultságok ellenőrzésére szolgál. Ebben az osztályban fogunk futási időben engedélyt kérni a felhasználótól a Manifestben megjelölt engedélyekre. Ezt az engedélykérést az *askForPermissions* metódus fogja csinálni. Amikor a felhasználó megadja az engedélyeket, az *onRequestPermissionsResult* callback metódusba érkezik be a válasz. Ha minden engedélyt megadott a felhasználó betöltjük a *MainActivity*-t, ha pedig valamilyen engedélyt nem adott meg egy *Error dialog* ablakot nyitunk meg neki.

MainActivity

Az alkalmazás fő megjelenítését a **MainActivity** végzi. Ezt az activity-t leíró XML fájl az *activity_main.xml*. Ebben az XML fájlban lesz definiálva az alkalmazás oldalsó menüje, ami *NavigationView* típusú lesz. Közvetetten itt fogjuk beilleszteni a *content_main.xml* tartalmát is az activity megjelenítéséhez. Ezek a leíró fájlok adják az egész alkalmazás megjelenítéséhez a keretet és minden további részletes tartalom pedig egy Fragment osztályból fog származni, amit a keretben definiált fragment konténer layout-ba fogunk beilleszteni.

Ez az osztály lesz az alkalmazásban a kontroller. A különböző felületek további osztályokban lesznek megvalósítva, és ha a modellt szeretnék elérni akkor ezen az activity osztályon keresztül tudnak behívni a *BikeService* modell osztályba.

A *MainActivity*-be létre fogunk hozni egy *BroadcastReceiver* típusú adattagot. Ezen az adattagon keresztül fogunk feliratkozni az eseményekre. Ha egy olyan intent-et érzékel az alkalmazás, amit hozzáadtunk a szűrőhöz az *onResume* metódusban akkor beesik ennek az adattagnak az *onRecive* függvényébe. A következő eseményekre fogunk feliratkozni a MainAc-

tivityben: *ACTION_USER_UPDATED*, *ACTION_NEW_USER_LOGIN*, *ACTION_USER_LOGOUT*, *ACTION_NEW_TRIP_HISTORY_SELECTED*. Ezeknek az eseményeknek a hatására frissítjük a felület megfelelő megjelenítő elemeit.

Ebben az osztályban található a *NavigationView*-nak az implementációja is. Azon túl hogy a layout xml-ben létrehoztuk a megjelenítendő elemet az *src/main/res/menu/activity_main_drawer.xml* fájlban felvesszük a menüpontokat. Ha egy menüpontot kiválaszt a felhasználó akkor az *onNavigationItemSelected* metódusba fog érkezni a program. Ebben a metódusban elágazássokkal lekezeljük, hogy melyik menüpontot választotta majd elvégezzük a megfelelő műveletet. Legtöbbször ez azzal jár, hogy kicseréljük a fragment konténerben az aktuális fragment-et. Egy példa arra, hogyan tudjuk a map fragment-et betölteni:

```
getSupportFragmentManager().beginTransaction()
   .replace(R.id.fragment_container, mMapFragment)
   .addToBackStack(null)
   .commit();
```

Ez az osztály implementálja az egyes fragmentek *InteractionListener* interfaceeit. Ha valamelyik fragment meghívja a listener metódusát az ebben az activity osztályban leimplementált metódust fogja hívni.

Fragment megjelenítő osztályok

A **SavedPlacesListView** fogja megjeleníteni az elmentett helyeket egy lista nézetben. Ehhez a megjelenítéshez két leíró XML tartozik a lista megjelenítése *(fragment_saved_places_list.xml)* és a listában megjelenő egyes elemek megjelenítése *(saved_place_row_layout)*. Az elemeket a listában a modellből fogjuk venni. Az elemekért felelős osztály a *CustomListAdapter* lesz. Ebben az osztályban állítjuk össze egy elemnek a megjelenítését.

A **TripHistoryListView** és *TripHistoryCustomListAdapter* hasonlóan fog működni mint az előző elmentett helyek megjelenítése. Az útvonal történet megjelenítéséhez fog tartozni egy plusz *TripHistoryMapFragment*. Ezt a térkép megjelenítést a lista *onItemClick* listener metódusán keresztül töltjük be. Ez a metódus egy eseményt fog kiváltani, amit a MainActivityben lekezelünk.

A **LoginFragment**-hez és **RegistrationFragment**-hez tartozó XML-ek a *fragment_login* és a *fragment_registration*. Működésük nagyon hasonló. Mind két osztályban, a felhasználó által megadott inputokat adjuk át a MainActivity-n keresztül a modellnek, ami egy webservice hívást fog indítani.

A **MapFragment** osztályban valósítjuk meg a térképes megjelenést. Ezt az *OSMDroid* és *OSMDroidBonusPack* nevű könyvtárak segítségével fogjuk tudni megvalósítani. Az osztályhoz tartozó XML fájlok a követ-

kezőek: *fragment_map, fargment_favouritedialog*, *fragment_favouritenamedialog*, *fragment_navigation_on_row_layout*, *route_details_row_layout*, *sliding_up_panel_content*, *sliding_up_panel_header*.

A felületre elhelyezünk egy SlidingUpPanel típusú elemet. Erre az elemre helyezzük el az *addToFavouriteButton*-t és a *routeHereButton* -t. Az utóbbi a GPS mód be-és kikapcsolásáért lesz felelős. A GPS módot ki és bekapcsoló függvények a *setGpsModeOn* és a *setGpsModeOff*, az első gomb pedig a *FavouriteDialog* ablakot fogja megnyitni, amiben a kedvenc helyek funkciói vannak leimplementálva.

Az osztály legfontosabb adattagja a MapView, ami maga a térkép lesz. Az osztály *onCreate* metódusában meghívjuk az *initMapView* függvényünket. Ebben a függvényben konfiguráljuk be a térkép elemet. Beállítjuk a közelítési paramétereket és a térkép kinézetét. Hozzárendeljük a térkép hosszú tapintása eseményhez (*longPressHelper*) a *newGeoPointSelected* metódust. Ennek a metódusnak a paramétere az a GeoPoint lesz, ahova a felhasználó tapintott. Ezt a GeoPointot a *GeoCoder* objektum segítségével visszafejtjük és a *Google Api* segítségével megkeressük a hozzá tartozó *Address* objektumot.

Ugyanezt a metódust fogja használni az *ACTION_NEW_DESTINATION* intenthez tartozó metódus is. Ebben az intent-ben extra paraméterként megkapjuk a menü autocomplete keresője által visszaadott *GooglePlaceId* stringet. Ehhez az idhoz a *GoogleDataApi* segítségével meg kell keresünk a helyet, ami hozzá tartozik. Sajnos ez a hely nem egyből *Address* hanem *Place* típusú lesz. Ebből a place-ből kinyerjük a *GeoPoint*-ot és ezzel fogjuk hívni a *newGeoPointSelected* metódusunkat.

Az *initMyLocationProvider*-ben hozzákapcsoljuk a modellből származó helymeghatározó osztályunkat a térkép elemhez. Hozzákapcsolás után ezt az overlay-t elhelyezzük a térképen és innentől automatikusan fogja kijelezni a térkép a jelenlegi pozíciónkat.

Az *updateSelectedLocationInformations, putDestinationMarker, updateRouteInformations, drawRoute, updateSelectedLocationInformationRouteDetails* metódusok a térképen megjelenő információk folyamatos frissítéséhez szolgáló függvények.

A *getRoute* metódus fogja meghívni a *GetRouteAsync* folyamatunkat, ami az útvonal lekérdezéséért lesz felelős. Ennek a metódusnak bemenő paramétere a cél állomás pozíciója. Ezt a pozíciót és a modellben tárolt jelenlegi pozíciónkat átadjuk a *GetRouteAsync* folyamatnak és elindítjuk ezt az aszinkron folyamatot. Ha nincs jelenlegi pozíciónk, mert a telefon még nem találta meg azt akkor addig hibaüzenetet írunk ki a felületre.

A *GetRouteAsync* az AsynTask osztályból fog származni, hogy aszinkron tudjon futni. A *doInBackground* metódusában fogjuk megvalósítani az útvonal lekérdezését. Az útvonalat a GrapHopper publikus szolgáltatótól fogjuk lekérdezni. A lekérdezéshez beállítjuk az option-be hogy bicikli útvonalakat szeretnénk, és

átadjuk a mi pozíciónkat és a cél pozíciót a kérésnek. Ha a kérés sikeres volt eltároljuk a modellben az útvonalat és eseményt küldünk róla, hogy van új útvonal.

4. Fordítás

4.1. Szerver alkalmazás

A szerver alkalmazás fordításához maven szükséges. Annak érdekében, hogy bárhonnan lehessen fordítani a szervert egy maven wrapper futtatható állományt helyeztem el a szerver gyökér könyvtárába, ez az *mvnw.cmd* fájl. A fordítást egy shell ablakban a következő paranccsal tudjuk megtenni:

```
./mvnw package
```

A package maven task elfogja végezni a java osztályok fordítását, a Unit tesztek lefuttatását és a végén egy Jar futtatható állományt készít az alkalmazásból. A fordítás közben létrejött fájlokat és a futtatható fájlt a target mappában találjuk.

A maven számára internetelérést kell biztosítanunk, hogy a megfelelő függőségeket letudja tölteni a fordításhoz. Így sajnos offline környezetben nem fordítható az alkalmazás.

Az elkészült szerver alkalmazást elérhetővé tettem a *https://bikeup.herokuapp.com/* címen, későbbiekben az android alkalmazásban ezt a címet konfiguráltam be. Ezen a címen keresztül leteszteltem a HTTPS protokoll működését is.

4.2. Android alkalmazás

Az Android alkalmazás fordításához gradle program szükséges. A szerverhez hasonlóan a projekt mappájában itt is elhelyeztem egy gradle wrapper fájlt a *gradlew.cmd*-t. Fontos, hogy mielőtt lefordítjuk az alkalmazást és publikáljuk a RestApiEndPoints osztályban a megfelelő szerver elérhetőséget megadjuk. Ezt a baseURL változó megfelelő módosításával tudjuk megtenni. A fordítást a következő paranccsal indíthatjuk el:

```
./gradlew build
```

Ez a gradle task először lefogja tölteni a dependency-ket majd lefordítja a Java osztályokat és a végén Android eszközre telepíthető *.apk* kiterjesztésű fájlokat készít. A release és a debug apk-t az *app/build/outputs/apk* mappába fogjuk megtalálni.

5. Tesztelés

5.1. Szerver alkalmazás

A szerver alkalmazás működését Unit tesztekkel fogjuk leellenőrizni. A teszteket az *src/test/java* mappába fogjuk elhelyezni. Az alkalmazás rétegei szerint a teszteket is három csomagba osztjuk, a *controller*, a *repository* és a *service* csomagba.

A teszteket egy parancsablakból következő paranccsal tudjuk futtatni a szerver gyökér könyvtárából:

```
./mvnw test
```

Erre a parancsra az alkalmazásban található 40 unit teszt fog lefutni.

A teszt osztályainkban két fajta metódust fogunk megkülönböztetni egymástól. A @Before annotációval ellátott függvények azok a metódusok lesznek, amik osztályonként a Unit testek előtt lefutnak. Ezekben a metódusokban fogjuk inicializálni az adatokat, amiken a teszteket fogjuk végezni.

A másik a @Test annotációval ellátott metódusok lesznek. Ezek lesznek a unit tesztjeink, amin belül assert-eket fogunk használni a teszteléshez.

Egyes teszteknél szükségünk lesz más rétegek működésére is. Ezekhez a mockito [4] könyvtárat fogjuk használni. Mock-olásnak nevezzük azt a technikát, amikor megmondjuk, hogy egy adott függvénynek az adott paraméterre mit kell visszaadnia. Ezt a technikát olyankor alkalmazzuk, amikor egy függvényt szeretnénk unit tesztelni és az a függvény egy másik függvényt hív.

Repository osztályok tesztelésénél azt kell letesztelnünk, hogy ha az adatbázisban van adat akkor azt a repository-jaink helyesen és hiánytalanul visszaadják a megadott entity object-ben. Ezeket az osztályokat a @DataJpaTest annotációval látjuk el, így jelezzük, hogy az adatbázis réteget fogjuk tesztelni. Ez azért lesz különösen fontos, mert a teszt során elmentett adatok és adat módosítások nem szeretnénk, hogy az éles adatbázison meglátszódjanak. Ezzel az annotációval egy teszt adatbázis fog létrejönni ezekhez az osztályokhoz.

A **UserRepositoryTest** osztálynak a setup metódusában felveszünk két teszt felhasználót akiknek beállítjuk az adatait. Az első tesztjeink a felhasználó lekérdésére fognak vonatkozni. A *whenFindByUsernameThenReturnUser* teszt arra fog vonatkozni, amikor létezik a User az adatbázisban, a *whenFindByUsernameThenReturnNull* pedig, amikor nem.

A *whenUsernameNotUnique* tesztesetben azt fogjuk tesztelni, hogy az adatbázis megengedi-e, hogy két ugyanolyan felhasználónevet elmentsünk. Ehhez a balazs2 felhasználónevet megpróbáljuk elmenteni újra az adatbázisba.

Mivel most arra számítunk, hogy ezt nem engedi meg az adatbázis, ezért a *@Test(expected=DataIntegrityViolationException.class)* annotációban megadtuk paraméterben, hogy milyen kivételre számítunk.

A **FavouritePlaceRepositoryTest** osztályban a setup metódus után két tesztet írtunk meg. Mindkét tesztünk az általunk definiált findBy metódusokat tesztelik, hogy valóban azokat az adatokat kapjuk meg, amiről a lekérdezések szólnak (*whenFindByUsernameThenReturnFavouritePlace*, *whenFindByUsernameAndNameThenReturnFavouritePlace*).

A **TripHistoryRepositoryTest**-ben az útvonal történetek esetén le fogjuk tesztelni, hogy a lekérdezéskor visszakapott entitások megfelelően tartalmazzák-e a hozzájuk tartozó entitást is. Konkrét példa erre a *whenFindTripHistoryListByUserTest1GeoPointList* ahol a GeoPoint listát fogjuk ellenőrizni.

A **PlaceLocationRepositoryTest** osztályban arra láthatunk példát, hogy miután elmentettünk az adatbázisba egy objektumot az valóban elmentésre került és vissza is tudjuk kérdezni *whenSaveAndFlush*.

Service osztályok tesztelésekor már alkalmazni fogjuk a mock-olás technikát. Adattagok között fogunk találni egy *@TestConfiguration* annotációval ellátott static osztályt, ami magát a service osztályt mint bean fogja definiálni. Ezek után be injectáljuk a service bean-t a testbe. @MockBean annotációval felvesszük az adattagok közé a service-hez tartozó repository osztályt is. Ezeknek a mock-olt osztályoknak meg kell mondanunk, hogy a metódusaik milyen bemenő paraméterre milyen értékkel térjenek vissza.

A **UserServiceTest** osztályban a userRepository osztályt fogjuk mock-olni. Ebben az osztályban leteszteljük, a felhasználó lekérdezését, amihez előre megmondjuk a repository osztálynak, hogy milyen felhasználót adjon vissza. Az új felhasználó létrehozására két tesztünk van. Az egyik, amikor gond nélkül létrejön az új felhasználó, a másik, amikor már létezik a felhasználó az adatbázisban, ezért *ServiceException*-t várunk a service rétegtől.

Controller osztályok teszteléséhez már bonyolultabb környeztet kell összeállítanunk. Az osztályokat két új annotációval fogjuk ellátni az egyik a @WebMvcTest, aminek segítségével be tudjuk injectálni a MockMvc objektumot. A másik annotáció a @WithMockUser(username="balazs"), ami azért szükséges, mert a legtöbb REST végpont autentikációval van védve. Ezzel az annotációval megmondjuk, hogy milyen felhasználónévvel autentikálva tesztelje a webserviceket. Ezekben az osztályokban a a service metódusokat fogjuk mock-olni, a repository-kat felesleges mivel a service már mock-olva lesz így a repository metódusait a service már nem fogja meghívni.

A post metódusok teszteléséhez a DTO objektumainkat JSON-né kell alakítanunk, a webservice hívása előtt ehhez a Jackson könyvtár ObjectMapper osztályát fogjuk használni.

A **UserControllerTest** osztályban találunk példát mind GET, mind pedig POST kérésre. Mindkét esetben a *mockMvc perform* metódusát fogjuk meghívni, aminek a paraméterében megadjuk a tesztelni kívánt webservice elérési útvonalát és metódusát. A *content* metódus paraméterében átadjuk a postolni kívánt json objektumot és az *andExcept* metódusban ellenőrizzük a válaszokat.

Az *andExpect*-ben először a státusz kódot fogjuk ellenőrizni, hogy jól adta-e vissza a webservice, hiba esetén hibát jelző státusz kódot adott-e. Továbbiakban a jsonPath függvény segítségével ellenőrizzük a válasz JSON tartalmát. Ennek a függvénynek első paramétere a JSON objektum adattagja a második pedig a várt érték. Ha a json objektumban egy tömböt várunk annak is ki tudjuk szedni az egyes értékeit. Ezekre egy-egy példa:

```
.andExpect(jsonPath("$.username", is("test1")))
.andExpect(jsonPath("$.favouritePlaces[0].placeLocation.lng", ↩
    is(2.0D)))
```

5.2. Android alkalmazás

Az Android alkalmazás esetében felületi tesztelés szükséges. Ehhez kézzel teszteltem végig az egyes funkciókat. Sorrendben a következőket vizsgáltam meg:
Jogosultság vizsgálat:

- Az alkalmazás első indításakor megjelenik az engedélykérő ablak.
- Ha elfogadjuk az engedélyeket akkor betölt a térkép. Ha nem fogadjuk el őket akkor a figyelmeztető ablak jön be.
- Ha az Android beállításokban kézzel letiltjuk a megadott engedélyeket akkor az alkalmazás következő indításakor újra feljön az engedélykérő ablak.

Útvonaltervezés:

- Az alkalmazásban, ha nincs megadva otthoni cím akkor a *Home* menüre tapintva figyelmeztető szöveg jön be, hogy ez nincs megadva. A munkahelyi cím esetén hasonlóan működik az alkalmazás.
- Hosszan tapintottam a térképen egy pontra, és azt megjelölte az alkalmazás egy célállomásként.

- Ha volt GPS koordinátám akkor útvonalat tervezett oda az alkalmazás egyből, ha nem volt akkor pedig kiírta, hogy nem tud útvonalat tervezni pozíció nélkül.
- A kiválasztott címnél az alsó dobozban megjelent a címnek az adatai.
- Ha felhúztam ezt az információs dobozt akkor megjelent a részletes útvonal terv a címhez.

Menü tesztelése:

- Az információs dobozban a *Favoruite* feliratra kattintva megjelent, a dialog ablak, amiben az *Add as Home*-ra kattintva hozzáadta az otthoni címemhez.
- Újra hosszan tapintottam egy másik pontra a térképen és hozzáadtam munkahelyi címként.
- A menüből útvonalat terveztem felváltva a két címhez és folyamatosan ellenőriztem, hogy a részletes útvonal kiírás változik.
- Ha megvan adva otthoni vagy munkahelyi cím, akkor a menüpont kiválasztása után a térképen a megadott címre ugrunk. Ha van GPS koordináta egyből megtervezzük az útvonalat is. Ha nincs GPS koordináta jelezzük a felhasználónak, hogy nincs.
- Anonim felhasználóként az útvonal történet és elmentett helyek menüpont megnyitása esetén figyelmeztető szöveg jelenik meg, hogy bejelentkezés után érhető el ez a funkció.
- A menüből a login gombra tapintva a bejelentkező felület jön be. Ezen a felületen helyes adatok megadásával az alkalmazás bejelentkeztet minket és átvált a térképes felületre. Helytelen adatok megadásakor az alkalmazás ezt jelzi.
- Bejelentkezett felhasználóként, ha újra indítjuk az alkalmazást, az egyből beléptet minket a profilunkba.
- A login felületen a Sign Up gombra tapintva a regisztrációs felület jön be.

Regisztráció:

- A regisztrációs felületen, ha nem töltöttem ki minden adatot és úgy próbáltam regisztrálni, figyelmeztető szöveget ír ki a felület.
- Ha minden adat helyes akkor kiírja a felület, hogy sikerült a regisztráció.
- Ha foglalt felhasználónévvel próbáltam regisztrálni ezt is kiírta a felület.

Bejelentkezés és kijelentkezés:

- A bejelentkező felületen helytelen adatokat megadva hibát ír ki az alkalmazás.
- Helyes adatok megadása után átirányít a térkép felületre.
- Bejelentkezés után ha újraindítom az alkalmazást az automatikusan beléptet.
- Bejelentkezett felhasználóként elérhető az útvonal történet és elmentett helyek menüpont.
- Kijelentkezéskor az alkalmazás töröl minden adatot, újra indításkor nem léptet be automatikusan.

Kedvenc helyek:

- Újra bejelentkeztem az alkalmazásba.
- Kiválasztottam egy helyet a térképen és a kedvenc dialogba az *Add as Favourite*-re tapintottam. Megjelent a dialog, ahol beírtam a hely nevét: *ELTE IK*.
- A menüben elnavigáltam az elmentett helyekhez ahol kilistázta az otthoni, munkahelyi címet és az *ELTE IK* nevű kedvenc helyet.
- A törlés ikonra kattintva törölni lehet elmentett helyet. Az alkalmazást újraindítva se jelennek meg újra a listában.

Útvonal történet:

- Ennek a funkciónak a tesztelését a Lockito nevű Android alkalmazás segítségével végeztem. Miután letöltöttem ezt az alkalmazást az Android fejlesztői beállítások alatt megadtam helyimitáló alkalmazásnak. Az alkalmazáson belül létrehoztam egy útvonalat, amit mock-olni fog az alkalmazásom számára.
- A megtervezett útvonal végét kijelöltem a saját alkalmazásomba cél állomásnak majd elindítottam a GPS jelek mock-olását.
- Amikor elért a célállomáshoz a pozícióm, az alkalmazás kiírta hogy az útvonal történet mentése sikeres volt.
- A menüből az útvonal történet menüt kiválasztva le ellenőriztem, hogy elmentődött-e az útvonal.
- Ugyanerre a célállomásra navigáltam újra, de ezúttal egy másik útvonalt mock-oltam meg. Ezen az új útvonalon a programom többször is újratervezte az útvonalat.
- Miután elmentette az útvonaltörténetet összehasonlítottam, hogy azt az útvonalat mentette-e el, amit a mock programban beállítottam.

6. Hivatkozások

[1] Phillip Webb et al. *Spring Boot Reference Guide.*
https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/

[2] Apache Software Foundation: *Apache Tomcat.*
http://tomcat.apache.org/

[3] Apache Software Foundation: *Apache Maven.*
https://maven.apache.org/

[4] *Mockito.*
http://site.mockito.org/