Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Community Parking

Közösségi parkolást segítő portál

Szoftverarchitektúrák házi feladat

Készítették: Fodor Árpád, Gyönki Bendegúz

KONZULENS
Gazdi László

BUDAPEST, 2020

Tartalom

1. Követelményspecifikáció

1.1. Feladatkiírás: Közösségi parkolást segítő portál

A hallgatók feladata egy olyan térkép alapú online rendszer kifejlesztése, amelyik lehetővé teszi felhasználóinak, hogy megosszák az általuk ismert ingyenes/előnyös parkolási lehetőségeket a többi felhasználóval. A rendszer legyen elérhető mobil készülékekről. Az alkalmazás tegye lehetővé fényképek feltöltését és a cím alapú keresést.

1.2. A fejlesztői csapat

Csapattag neve	**Neptun-kód**	**E-mail cím**
Gyönki Bendegúz	IZZT5E	gyonkibendeguz@gmail.com
Fodor Árpád	S4AZIE	arpadfodor01@gmail.com

A csapatban a következőképpen osztottuk fel a feladatokat:

- **Bendegúz** a backend (szerveralkalmazás) fejlesztésével foglalkozik
- **Árpád** a frontend (Android kliens) fejlesztését végzi

1.3. Részletes feladatleírás

Célunk a projekt keretében egy olyan rendszer készítése, amellyel a felhasználók bárhol képesek parkolási lehetőségek bejelentésére és megtekintésére. Ezáltal adott területek monitorozására nyílik lehetőség, ami megkönnyítheti a városban autóval járók mindennapjait.

Az alábbiakban ismertetjük az Android operációs rendszerre készített kliens és a szerveroldali alkalmazás tervezett funkcióit.

1.3.1. Android alkalmazás

Egy bejelentéshez a GPS koordináták és a szükséges adatok megadásán felül (pl. fizetős/ingyenes parkolás) lehetőség van képek feltöltésére is, amelyek révén a többi felhasználó könnyebben találhatja meg az adott helyet. A képek a gyors bejelentés céljából az alkalmazásban készülhetnek, de lehetőség van eszközön tárolt fotók feltöltésére is.

A felhasználók többféle módon böngészhetnek a bejelentések között. Egyrészt egy interaktív térképen vizuálisan láthatják a bejelentéseket, illetve lehetőség van cím alapján történő keresésre is. A kiválasztott parkolóhely részletes adatai (pozíció, típus, bejelentés ideje, kép) megtekinthetők, illetve navigáció indítása is lehetséges az adott helyre.

Egy felhasználó az alábbi műveleteket hajthatja végre egy potenciális parkolóhely esetén:

- Megadja a tulajdonságait (pl. fizetős/ingyenes)
- Képet készít hozzá
- Bejelenti

Egy már bejelentett parkolóhelyen a felhasználók az alábbi műveleteket végezhetik:

- Lefoglalás (egy felhasználó jelzi, hogy az adott helyre tart, oda fog parkolni)
- Foglalás levétele
- Módosítás (pl. egy felhasználó jelzi, hogy az adott parkolóhelyen már áll valaki; vagy képet fűz hozzá)

1.3.2. Szerveralkalmazás

A szerveroldali alkalmazás az alábbi, parkolóhelyekkel kapcsolatos műveleteket támogatja:

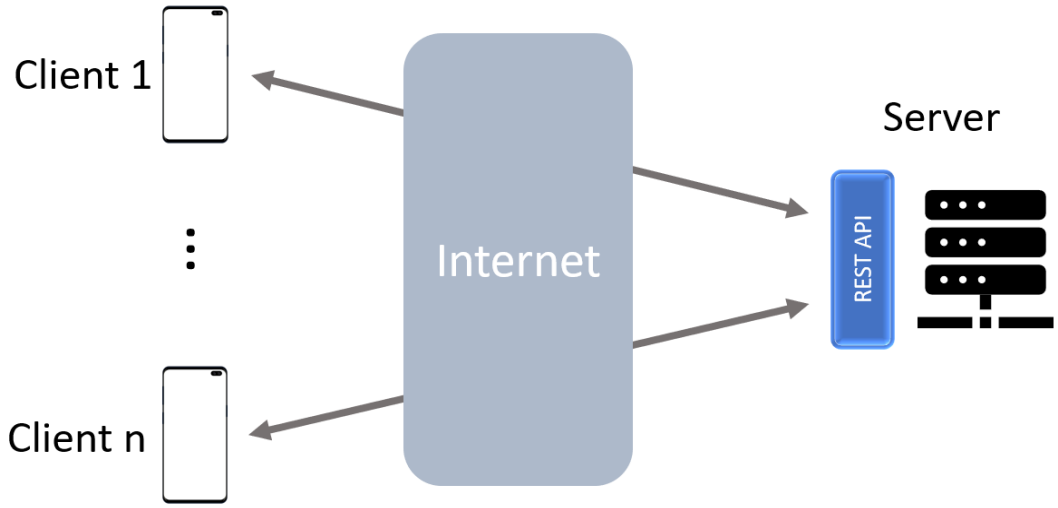
- Új parkolóhely hozzáadása az adatbázishoz
- Meglévő parkolóhely adatainak módosítása az adatbázisban
    - Parkolóhely lefoglalása
    - Foglalás levétele
- Meglévő parkolóhely törlése az adatbázisból
- Parkolóhely keresése

- GPS koordináták alapján és a kliens által megadott sugarú körben

A szerveroldali alkalmazás felel a felhasználók nyilvántartásáért is. A szerver a felhasználók alábbi adatait tárolja: felhasználónév, email cím, jelszó (titkosított formában). A szerver lehetővé teszi új felhasználó regisztrálását, illetve a felhasználói adatok módosítását.

1.4. Architektúra

A rendszert kliens-szerver architektúra alapján tervezzük megvalósítani. A kliensek Android alkalmazások, a szerver pedig egy REST API-t biztosít a kliensek számára. A bejelentett parkolóhelyek, képek, koordináták a szerveren tárolódnak, a kliensek ide tudnak bejelenteni új dolgokat, vagy lekérdezni az aktuális állapotot. A kommunikáció HTTPS protokollon keresztül történik, a felhasználók azonosítása pedig HTTP Basic authentikációval történik.



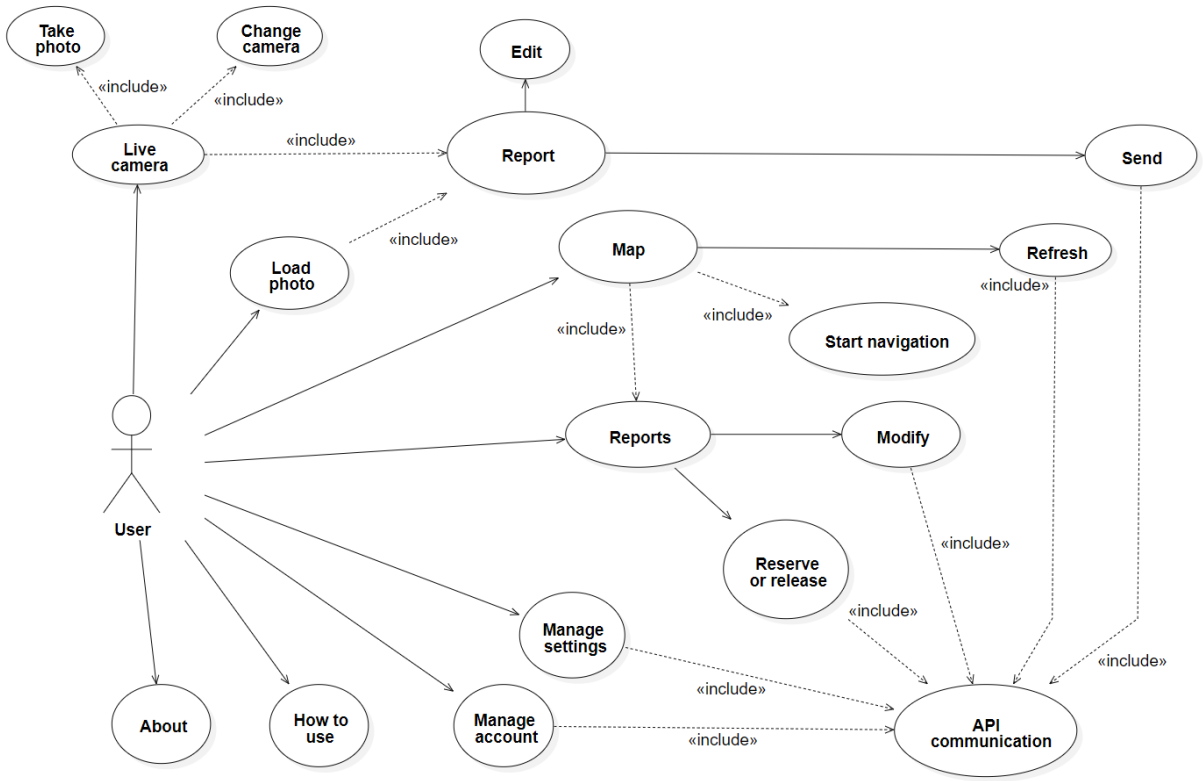
ábra 1: Kliens-szerver architektúra

1.5. Technikai paraméterek

A rendszer kliensoldali része Android platformra készül. Az okostelefonok napjainkra széles körben elterjedtek, a legdominánsabb operációs rendszer pedig jelenleg az Android, ezért ezt a platformot célozva potenciálisan széles lehet a felhasználók köre. A fejlesztés Kotlin nyelven történik. Az alkalmazás specifikus perzisztens adatok SQLite alapú Room adatbázisban kerülnek tárolásra. A kamerakezelés a CameraX API segítségével kerül megvalósításra. A térkép megjelenítéséhez a Google Maps API segít, míg a szerverrel történő hálózati kommunikációhoz a Retrofit kerül felhasználásra.

A szerveralkalmazás fejlesztése is Kotlin nyelven történik, a Ktor keretrendszer felhasználásával. A Ktor egy aszinkron keretrendszer mikroszolgáltatások és webalkalmazások fejlesztéséhez. A szerver az adattároláshoz egy PostgreSQL adatbázist használ, az adatbázisműveletek pedig a Jetbrains által készített Exposed ORM keretrendszer használatával kerülnek megvalósításra.

1.6. Use-case diagram



ábra 2: Android app use-case

2. Rendszerterv

A rendszer magasszintű architektúrája a kliens-szerver modellt követi. Az alábbi alfejezetek ismertetik mind az Android alkalmazást, mind pedig a szervert. Ezek kommunikációja HTTPS-n keresztül történik, a szerver által biztosított REST API-n keresztül.

2.1. Az Android alkalmazás

A kliens egy Android alkalmazás. Fő funkciója a parkolóhelyek bejelentése, azok megtekintése, és felhasználói engedélyhez kötött módosítása (lefoglalás, törlés, megjegyzés hozzáfűzése). Az eszközön tárolt kép mellett lehetséges az élő kameraképen látható parkolóhely bejelentése is. Az élő kameraképen lehetséges a nagyítás és a fókusz változtatása is. Az egyes helyek az eszközön is eltárolásra kerülnek, hogy internetkiesés esetén is használható maradjon az alkalmazás. Frissítés/bejelentés/törlés azonban csak az API elfogadásával történhet, megelőzve az inkonzisztens állapotot.
A front-end fejlesztése az Android Studio Preview IDE-ben történt, Android API 30-ra targetálva. Az alkalmazás megírása Kotlin 1.4-ben történt, JDK 11 segítségével. A minimum API szint a 26-os, ami a futtatáshoz szükséges (Android 8.0, 2017-ben lett bemutatva).

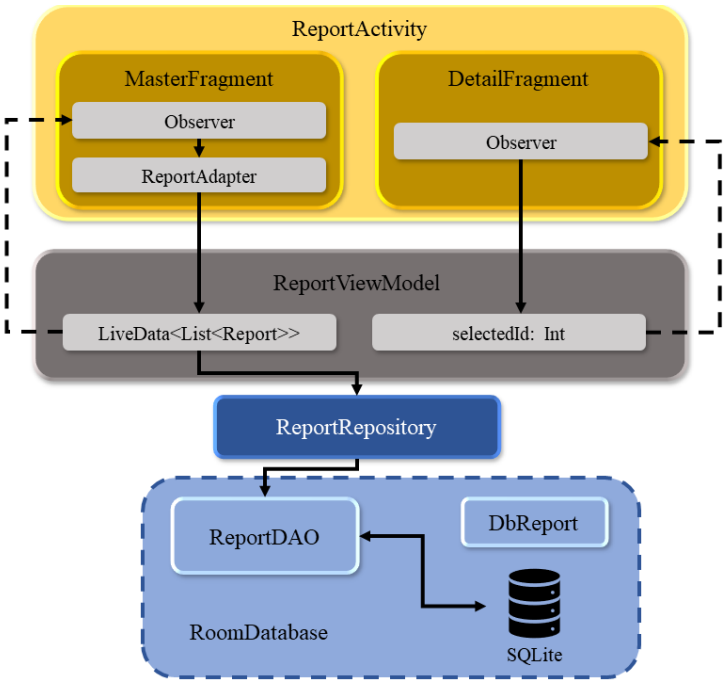
2.1.1. Architektúra

A Model View ViewModel (MVVM) felhasználói felület tervezési minta lett felhasználva az Android alkalmazás készítése során. Ez egy eseményvezérelt modell, amelyet a Microsoft talált ki az adatkötési képességek kihasználására. Az MVVM-ben a View UI felületleíró kódot tartalmaz, általában deklaratív módon (XML, XAML, HTML). A kapcsolat a ViewModellel explicit adatkötés segítségével valósul meg. Ezért kevesebb klasszikus kódolási feladat van a View-ban, és az üzleti logika komponensek könnyedén elválaszthatók.

1. ábra: MVVM komponensek és azok kapcsolatai

A magasszintű hierarchiát egy parkolóhely (Report) adatbázisban való frissítésén keresztül, az alábbiakban szemléltetem. A ReportActivity DetailFragment-je jeleníti meg az adatokat, ahol például lefoglalható egy parkolóhely. Egy Report elem listában kerül tárolásra a ViewModel-ben, ami LiveData objektumba van csomagolva (ez a UI elemek számára megfigyelhető, mert adatkötésben használható). Mikor a felhasználó frissít egy elemet a DetailFragment-ben, az adott elem a listában frissítésre kerül, mikor a Model a műveletet elfogadja. Ekkor a változás a MasterFragment-ben is azonnal megjelenik.

A ReportViewModel változás esetén értesíti a Model-ben található RepositoryService-t. Ebben található a ReportRepository, ami a Report-okhoz tartozó műveleteket (adatbázisba írás, API hívások kezelése) rejti el a külvilág számára. Egy Report változása esetén először az APIService-en keresztül a szerver a változásról értesítve lesz. Ha a backend ezt elfogadta, a Report az adatbázisban tárolt formátumra alakításra kerül (DbReport), és a ReportDAO (data access object) segítségével perzisztálva lesz. Így van biztosítva, hogy csak az kerül elmentésre, ami az API-n is érvényre jut, megelőzve az inkonzisztenciát. A Model-ben történő műveletek (API kommunikáció, adatbázis tranzakciók) háttérszálon futnak, így a felhasználói felület nem fagy be, érvényre jutáskor viszont az adatkötés miatt azonnal frissítésre kerül.



2. ábra: parkolóhely adatok adatbázisba írásának folyamata az egyes rétegeken keresztül

A parkolóhelyek adatai mellett az alkalmazás tárolja a felhasználó adatait is, amelyekkel például az automatikus bejelentkezés, illetve a fiókadatok frissítése valósíthatók meg. Ezen kívül az API-tól lekért metaadatok is eltárolásra kerülnek (időbélyeg, parkolóhelyek száma), amit minden API hívás előtt ellenőriz az alkalmazás. Ha az API által lekért metaadatok alapján annak adatai változtak az app legutóbbi frissítése óta, adatletöltés indul, ha azonban nincs változás, nincs felesleges hálózati forgalom sem.
A következő alfejezetek ismertetik az MVVM szerinti felosztásban szereplő alkalmazás komponenseket.

2.1.2. Model

Az alkalmazás Model komponense további alrétegekbe van rendezve. Legfelül találhatók a service-ek, amik egymástól független, lazán csatolt komponensek. Ezek az osztályok a singleton tervezési elv szerint lettek implementálva, azaz mindegyikből egy példány található az alkalmazásban. Ez azért célszerű, mert ezen szolgáltatások tipikusan olyan segítő metódusokat tartalmaznak, amik elérése céljából nincs értelme több példányt létrehozni (ImageConverter). Vagy olyan logikát tartalmaznak,

amik jellemzően a teljes alkalmazásra jellemzőek (például az aktuálisan bejelentkezett user információi, vagy a futtató eszköz helyadatainak kezelése). Az alábbi Service osztályok találhatók meg az applikációban:

- **AccountService**: a felhasználók kezeléséért felel. Ezen keresztül történik minden regisztrációs, bejelentkezési és kijelentkezési művelet. Minden aktuális felhasználóval kapcsolatos információ és validáció ezen a szolgáltatáson keresztül érhető el, és bizonyos szenzitív adatokat (pl. jelszó) a vele egy modulban lévő komponensek elől is elrejt.
- **DateHandler**: mivel a Java dátumkezelése egy régre visszanyúló, problémás terület, úgy döntöttem, magam implementálok a Java-s Date köré egy csomagoló osztályt, aki a string-Date konverziót konzisztens módon elvégzi (és elrejti a Date példányosítási furcsaságait, például a 0. hónapot). Erre azért van szükség, mert a JSON objektumokban a dátum string-ként jön, az alkalmazás azonban dátumként kezeli őket, hogy bizonyos kényelmi funkciók (két dátum összehasonlítása, év/hónap/nap kinyerése) megmaradjanak. Továbbá bizonyos default értékeket is szolgáltat. Ez az osztály minden dátumot egységesen UTC időre konvertál és kezel, hogy az egyes időzónák a timestamp értékeket ne zavarhassák meg.
- **ImageConverter**: segítő függvényeket biztosít, például a CameraX API-tól jövő ImageProxy példányból tud generálni Bitmap-et. Ezen kívül a kép tűkrözését is egyszerűen megoldja (előlapi kameránál igény), valamint a kép forgatása is lehetséges (kényelmi funkcióként lehetséges forgatni a betöltött képeket, ha véletlenül a metaadatok alapján rossz tájolással töltené be az alkalamzás). Ezeket a műveleteket képtranszformációk segítségével végzi el. Az itteni függvények számításigényük miatt minden esetben háttérszálakról kerülnek meghívásra.
- **LocationService**: A Geododer API szolgáltatásait csomagolja (address string -> koordináták átalakítás és vissza). Emellett az eszköz GPS helyadatait is kezeli. Mivel ennek lekérése erőforrásigényes lehet, ezt úgy teszi, hogy belső változókban tárolja az aktuális koordinátákat, de bizonyos minimum időközönként frissebb adatokért az operációs rendszerhez fordul. Így takarékosabban bánik az erőforrásokkal. Ezek is a háttérszálon futó, hosszan tartó folyamatok.
- **MediaHandler**: Az applikáció belső/külső tárhelyét kezeli. A képek beolvasásáért felel az eszközről, valamint azok Exif metaadatait is lekérdezi az operációs rendszertől (készítés ideje, elforgatás).
- **MetaProvider**: egyszerű függvényhívások mögé rejti az adott képhez tartozó metaadatok lekérését, valamint az eszköz aktuális metaadatainak (rendszeridő, koordináták) lekérését.
- **TextToSpeechService**: a szövegfelolvasó API-t egyszerá metódushívások mögé rejti. Lehetőség van hiba, elkezdődés és befejeződés callback-eket átadni ezen függvényeknek, amikkel például a felolvasó gomb ikonja (éppen lejátszik, épp nem játszik le) állapothelyese jelenhet meg minden esetben. Ezt a Kotlin a higher order függvényei segítségével teszi lehetővé.

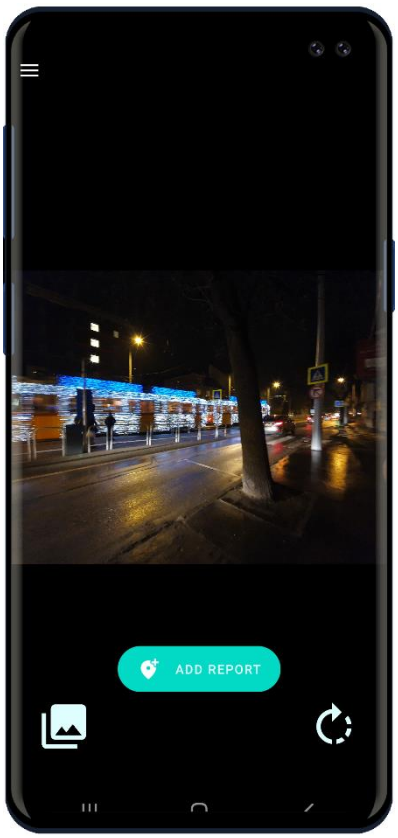
a Repository-k rétege (~Data Access Layer a külvilág számára), amik az adatműveletek mögött álló komplexebb logikát (adatbázis írási műveletek, API hívások) rejtik el a külvilág elől. Itt három osztály található: a GeneralRepository felel az általános szinkronizációs műveletekért (az API szinkronizációt végzi). A ReportRepository a bejelentések lekéréséért, adatbázisba írásáért, bejelentéséért felel, a UserRepository pedig a felhasználói account-hoz kapcsolódó funkciókért. Ebben a rétegben találhatók azok az adatmodellek, amiket a ViewModel-ek ismerhetnek (ilyen típusú Report elemeket tárolnak listában). Ezek az adatmodellek csak olyan mezőket tartalmaznak, amik UI felület számára

megjelenítendő, fontos adatok (pl. nincsenek az adatbázis tárolása szempontjából fontos, generált mezők).
A Repository-k alatt vannak az API és a DB rétegek. Mindkettőben saját maguk adatmodelljei szerepelnek, amiket mentéskor/kiolvasáskor a Repository konvertál a saját adatmodelljeivé vagy azokból. Az API komponensben található egy interface, ami leírja a hívható műveleteket a szerveren. Ez Retrofit segítségével került implementálásra az ApiService osztályban (szintén singleton). A kommunikáció hitelesítési és biztonsági paraméterei az itteni BasicAuthInterceptor segítségével adhatók meg (az API-n át történő authentikáció http basic authentikációval történik). A DB komponens tartalmazza a Room által menedzselt SQLite adattáblák DAO osztályait. Meta-, Report- ÉS UserDAO osztályok vannak. Ezen rész működését a perzisztens adattárolás alfejezete ismerteti.

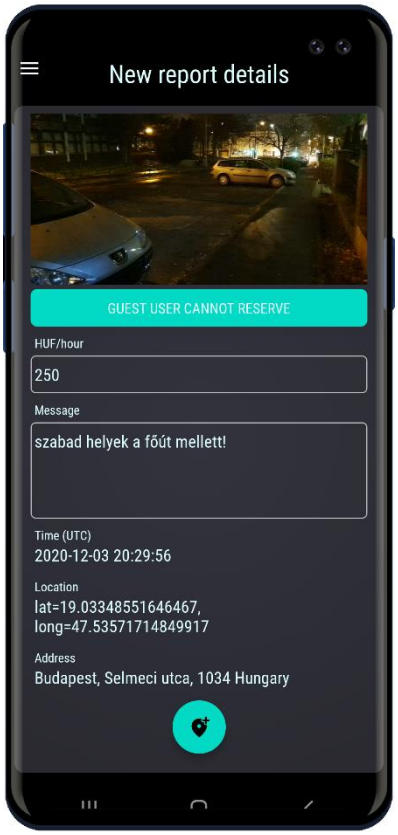
2.1.3. View

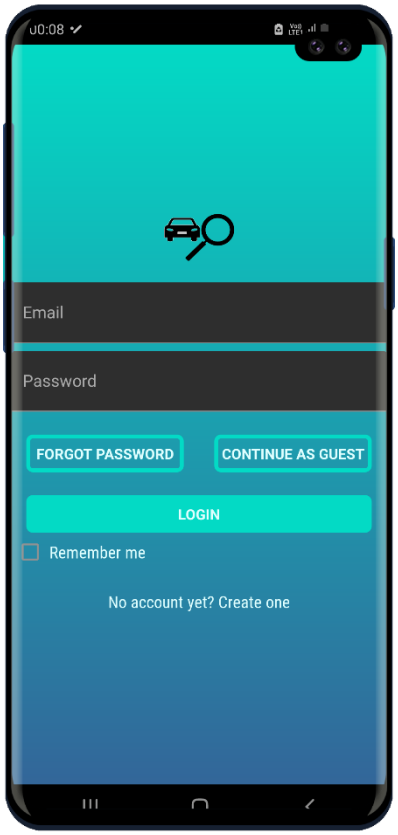
A View-ban találhatók a felhasználói felületi komponensek (activity-k, fragment-ek és azok segédosztályai). A felületleírók deklaratív módon (XML) megadott állományokban találhatók. Ez reprezentálja a Model állapotát a User számára. Ezek az elemek értesülnek a felhasználói interakciókról (írás, kattintás, gesztusok), amiket továbbítanak a megfelelő ViewModel számára. Egy ilyen osztály explicit adatkötéssel kapcsolódik a ViewModel változóihoz. Bizonyos esetekben több View ugyanazon ViewModel példányhoz kötődik (például master-detail fragment esetén).

A View-k kialakításánál a Material design irányelvei lettek figyelembe véve (View elemek, javasolt margin/padding, árnyékok, animációk használata). Az alábbi ábrák az alkalmazást mutatják élő (bal) és betöltött (jobb) képek vizsgálata közben:

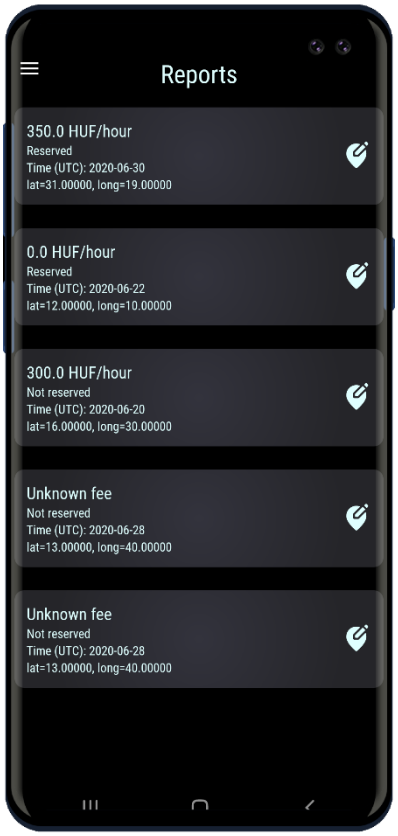


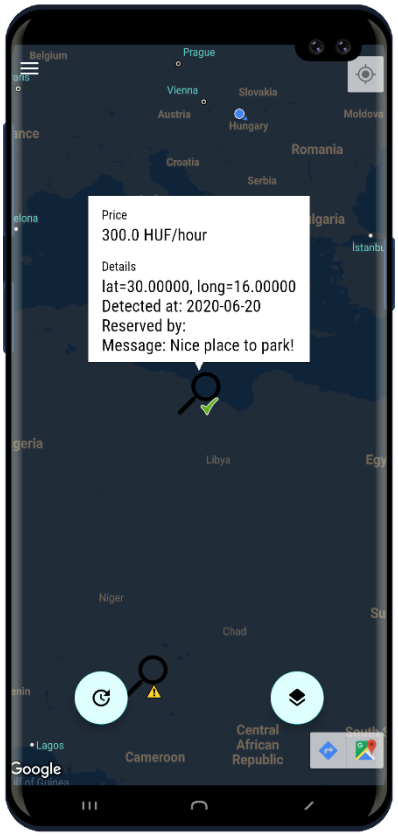
Az alábbi ábrán lehet látni egy új bejelentés részleteinek megadó képernyőjét, illetve a felhasználó bejelentkező felületét:



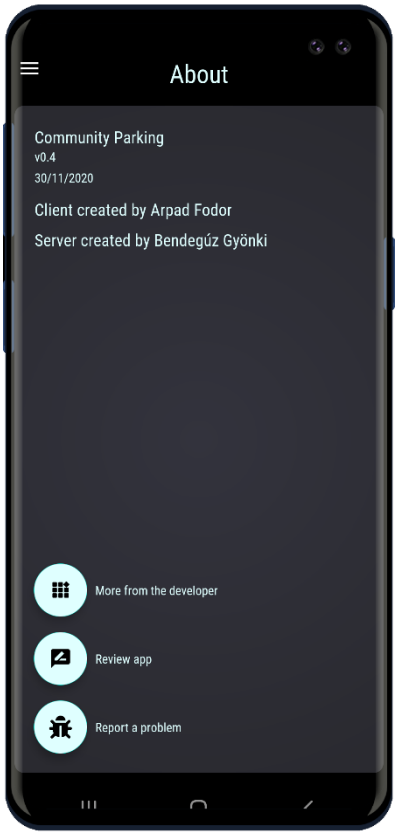


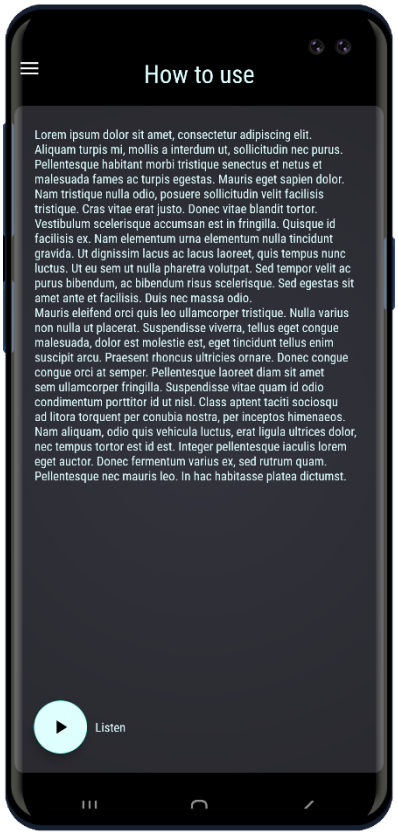
Az elérhető parkolóhelyek listás, illetve térképes megjelenítése:



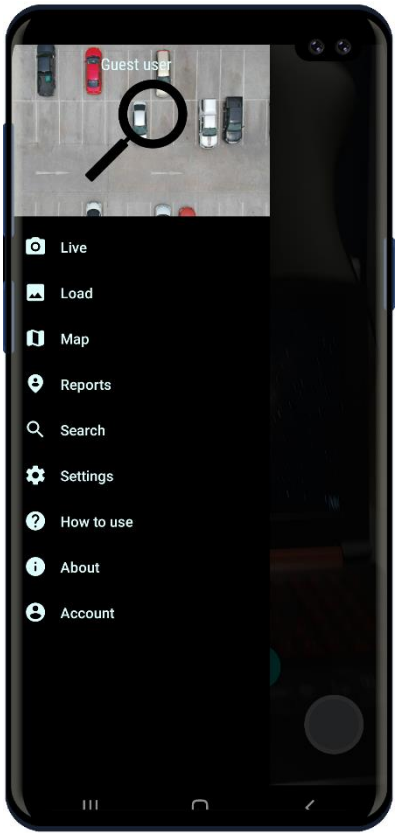


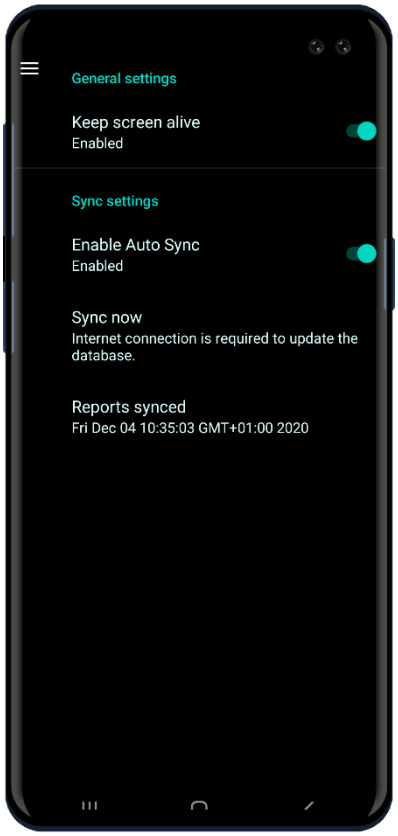
A következők mutatják az About képernyőt (bal) és a HowToUse képernyőt (jobb) :



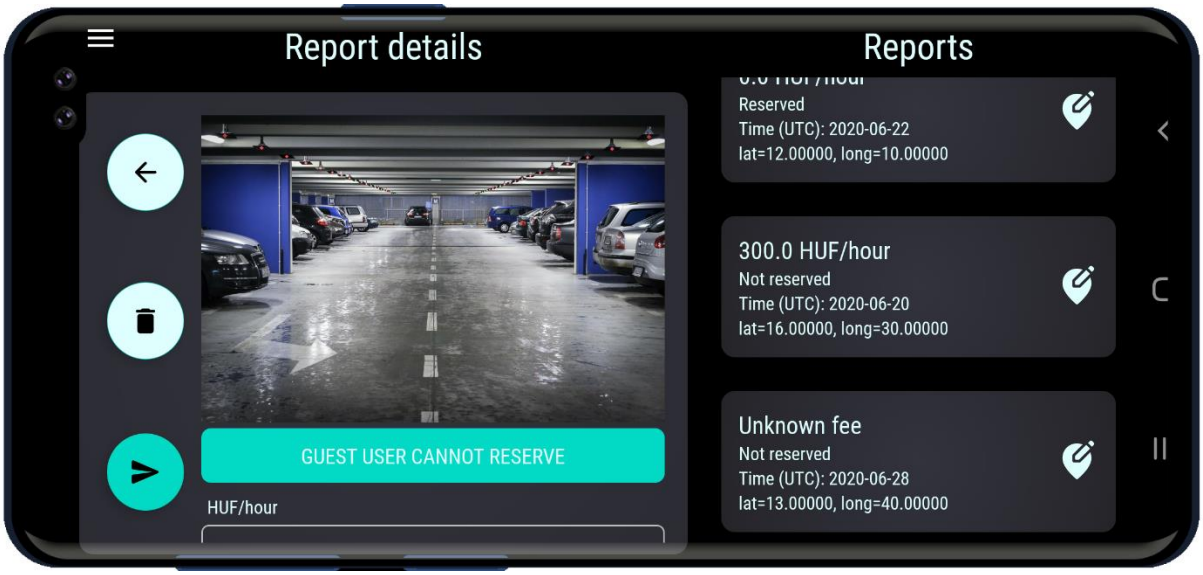


Az oldalsó alkalmazás főmenüje (bal) és a beállítások képernyő (jobb):





Az alkalmazás kezeli a képernyő elforgatást is, valamint adaptívan reagál különböző képernyők esetén (például fekvő layout) történő megjelenítésre. A master-detail fragment-ek fekvő layout esetén például egymás mellett jelennek meg:



2.1.4. ViewModel

A View absztrakciójaként is felfogható. Kezeli (validálja, továbbítja) a felhasználói eseményeket, vezérli a UI-t, a GUI logikáját tartalmazza. Az adatkötés miatt automatikusan értesül a UI eseményekről és fordítva, mikor a View elérhető. Csomagolja a modellt, és kommunikál vele. Fontos, hogy egy ilyen osztály sem tartalmaz referenciát egyetlen View elemre sem a keretrendszer adatkötési lehetőségét kihasználva. Ezzel van megoldva, hogy a például képernyőelforgatásból származó View elemek újra példányosodáskor megőrzik a korábbi állapotokat. Így pedig a felület állapota, és alapvetően az üzleti logika el tud különülni a View-k (általában rövid) életciklusától.

2.1.5. Alkalmazás adatmodellek

A kliens a szerver által meghatározott JSON formátumban kapja az adatokat. Az adatbázisban azonban a tároláshoz egyedi kulcsokra van szükség, amely kritériumnak az API válasz elemei nem biztos, hogy megfelelnek. Ezért (is) szükséges az API objektumainak relációs adatmodellé képezhető, adatbázisban megfelelő módon tárolható adatmodelleket megfeleltetni. Illetve az API változása esetén nem feltétlen kell az adatbázis sémát változtatni.
A felhasználói felületen sokszor bizonyos mezők megjelenítése nem fontos, ezért csak feleslegesen szerepelnének a ViewModel-ben. További baj lenne, ha az adatbázis adatmodell sémája megváltozik, ez a változás egészen a View-ig propagálódna fel. Ezt elkerülendő, a ViewModel-ek számára is külön adatmodellek lettek definiálva, hogy az adatbázis sémájának változása ne befolyásolja őket.
Ezen adatmodell hármasok egymás közti konverzióját a megfelelő Repository osztály végzi. Ez a külvilág számára nem látható (a ViewModel-ek csak a nekik megfelelő adattípusokat látják; az API és az adatbázis pedig szintén a saját típusaikkal válaszolhatnak a hívó Repository-nak). Az alábbiakban

összefoglalásra kerülnek az egyes adatmodell típusok az alkalmazásban. Az itt szereplő mezőnevek modelltípusonként kismértékben eltérnek.

2.1.5.1.Report

Mező neve	API adattípus	DB adattípus	VM adattípus
id	Int	Int	Int
reporterEmail	String	String	String
latitude	Double	Double	Double
longitude	Double	Double	Double
timestampUTC	String	String	String
message	String	String	String
reservedByEmail	String	String	String
feePerHour	Double?	Double?	Double?
imagePath	String	String	String

Itt fontos lehet megmagyarázni, hogy a feePerHour nevű érték miért nullable. Egy pozitív értéke (pl. 300) azt jelenti, óránként 300 Ft a parkolási díj, a 0 pedig azt, hogy ingyenes. A null érték jelentése ebben a kontextusban az, hogy nem ismert a parkolási díj értéke (az sem, hogy ingyenes-e, vagy fizetős).

2.1.5.2.MetaData

Mező neve	API adattípus	DB adattípus	VM adattípus
tableId	String	String	-
dataSize	Int	Int	-
modificationTimeStampUTC	String	String	-

A MetaData az API-tól érkező adatokat jellemzi a lekérdezés pillanatában. A kliens először ezeket kéri le; ha a kliensben tárolt legutóbbi lekérdezés metaadatánál frisebb az API válasza, a kapott érték elmentésre kerül, és elindul az adatok lekérése. A Report és User elemekhez külön-külön tartozik egy-egy ilyen példány.

2.1.5.3.User

Mező neve	API adattípus	DB adattípus	VM adattípus
email	String	String	String
password	String	String	String
name	String	String	String
hint	String	String	String
isActive	Boolean	-	-
permissions	List<Int>	-	-
validFromUTC	String	-	-

A User esetén jól látszik, miért hasznos az eltérő adatmodellek használata. Mivel az adminisztrátorok is az API segítségével kérhetnek le/változtathatnak a felhasználókon, bizonyos mezők megtalálhatók, amik a kliensalkalmazás számára érdektelenek (pl. validFromUTC, ami a regisztrálás idejét jelzi). Ezért ezek a mezők az adatbázisban már nem kerülnek eltárolásra.

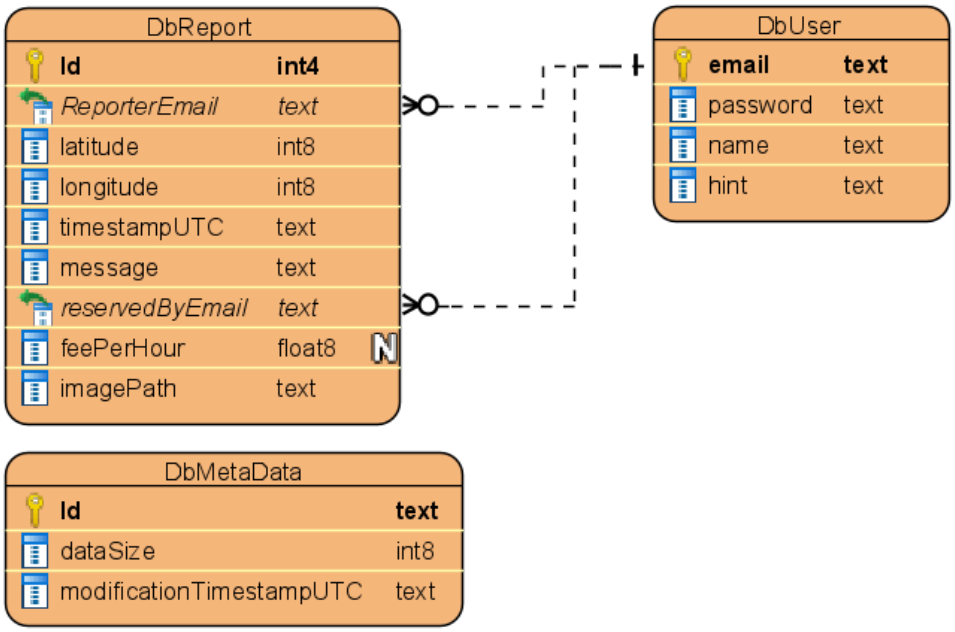
2.1.5.4.Coordinate

Mező neve	API adattípus	DB adattípus	VM adattípus
latitude	Double	-	-
longitude	Double	-	-

Ez az osztály arra szolgál, hogy az API-tól lekérdezhető legyen koordináták alapján a legközelebbi szabad parkolóhely. Ezért ez külön az adatbázisban nem kerül eltárolásra, csak a kérés indításakor példányosodik a felhasználó által megadott két Double érték alapján.

2.1.6. Perzisztens adattárolás

Az alkalmazás tartalmaz egy SQLite alapú adatbázist, ami a Room könyvtár segítségével lett kialakítva. Hogy korlátozott módon internetkiesés esetén is használható legyen az app (parkolóhelyek megtekintése, térképen böngészés) a bejelentések DbReport entitásokban vannak az eszközön tárolva. A DbMetadata arra szolgál, hogy nyilvántartsa, mikor történt az utolsó lekérés. Ha az API válaszában az adat legutóbbi módosítás lekérésekor egy újabb UTC timestamp szerepel, mint mikor a kliens legutóbb lekérte őket, frissít. Ha azonban a két időbélyeg megegyezik, felesleges a lekérés, mert a lokális SQLite adatbázisban már naprakész adatok szerepelnek. A DbUser a felhasználói adatokat tartalmazza, ami bejelentkezés esetén van használva. Lehetőség van megadni, hogy az alkalmazás megjegyezzen egy felhasználót, így automatikusan be tudjon lépni.



3. ábra: perzisztens módon tárolt adatok modellje

2.2. A szerveralkalmazás