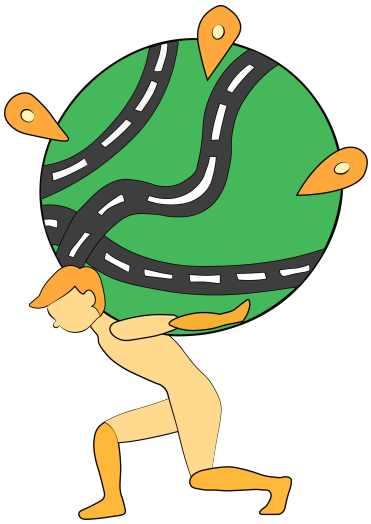
XXIII. reál- és humántudományi Erdélyi Tudományos Diákköri Konferencia (ETDK)
Kolozsvár, 2020. május 21–24.

Atlas

Városi térfigyelő kamerákon alapuló rendszer szabad parkolóhelyek keresésére és lefoglalására



Szerző:

Szabó Ágnes-Kriszta
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Témavezetők:

dr. Simon Károly, egyetemi adjunktus
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar
dr. Sulyok Csaba, tanársegéd
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar
Patakfalvi Örs-Krisztián, szoftverfejlesztő,
Codespring
Ráduly Zalán, szoftverfejlesztő,
Codespring

CODESPRING
OUR CODE YOUR SUCCESS

Kivonat

Nagyvárosokban jelentős problémát okoz a sofőrök számára a parkolóhelyek keresése. Mindez számottevő plusz költségeket eredményez, ugyanakkor a környezetszennyezéshez és a dugók kialakulásához is nagymértékben hozzájárul. A jelen dolgozatban bemutatott projekt a parkolóhelyek keresésének folyamatát hivatott megkönnyíteni városi kamerák által megfigyelt területek monitorizálásával.

Az Atlas rendszer, más közlekedési és parkolás-menedzsment szoftverekkel ellentétben, nem igényel költséges infrastruktúrát, hiszen a már meglévő térfigyelő kamerák adatait használja fel annak érdekében, hogy kijelölt és kijelöletlen parkolóhelyeket is lehessen listázni, illetve lefoglalni.

A rendszer fő részei a központi szerver, a több platformon működő mobilalkalmazás, valamint a számítógépes látásért felelős modul. Az utóbbi a projekt mentorai által fejlesztett külső modul. A dolgozat ismerteti a projekt szerkezetét, a megvalósítás részleteit, a felhasznált módszereket, eszközöket és technológiákat, illetve példákon keresztül szemlélteti az alkalmazás működését.

Tartalomjegyzék

Bevezető **1**

1. Az Atlas projekt **3**

1.1. Funkcionalitások 3

1.2. Architektúra 4

2. Központi szerver **6**

2.1. Felhasznált technológiák 6

2.1.1. Node.js 6

2.1.2. MongoDB 7

2.1.3. Docker 7

2.1.4. OSRM 7

2.2. Adatmodell 8

2.3. Adathozzáférési réteg 10

2.4. API 10

2.5. Biztonság 11

3. Mobil kliens **13**

3.1. Felhasznált technológiák 13

3.1.1. React Native 13

3.1.2. MobX 14

3.1.3. Native Base 15

3.1.4. Firebase 15

3.2. Kommunikáció a szerverrel 16

3.3. Üzleti logika 16

3.4. Megjelenítési réteg 17

3.5. Felhasználói felület 19

4. Eszközök és munkamódszerek **20**

5. Az alkalmazás működése **23**

Következtetések és továbbfejlesztési lehetőségek **26**

Bevezető

Sofőrök vagy tömegközlekedést használó személyek nap mint nap tapasztalhatják a forgalmi dugók okozta kellemetlenségeket és frusztrációt. A közúti torlódások környezetvédelmi szempontból hatalmas mennyiségű hulladék-üzemanyag- és szén-dioxid-kibocsátást jelentenek, amint ezt a Nemzetközi Parkolási Intézet 2012-es tanulmánya [25] is alátámasztja. Ugyanezen tanulmány becslései szerint az autós forgalom 30%-a parkolóhely keresése miatt tartózkodik az utakon. A Siemens 2011-es esettanulmánya [34] alapján átlagos napokon a belváros forgalmának több mint 40%-át teszik ki a parkolóhelyet kereső járművezetők, ez a szám azonban hétvégente vagy ünnepnapokon jóval nagyobb is lehet. Karácsony környékén akár a 90%-ot is elérheti.

Az Atlas projekt a parkolóhelyek keresésének folyamatát hivatott megkönnyíteni és felgyorsítani. A szükségtelen körözéseket egy mobilalkalmazáson keresztül igyekszik minimalizálni, amely megjeleníti a felhasználó számára az aktuálisan szabad kijelölt és kijelöletlen parkolóhelyeket. Az alkalmazás használatával a sofőrök időt és pénzt takaríthatnak meg miközben elősegítik a fenntarthatóságot és csökkentik karbonlábnyomukat.

Léteznek már különböző próbálkozások a parkolóhelyhiány problémájának megoldására. Ilyen például a Kolozsvár polgármesteri hivatala által kezdeményezett Cluj Parking mobilalkalmazás [2], mely a város területén található hét darab sorompós parkolóról nyújt információkat. Egy másik telefonos alkalmazás, a yeParking [5], lehetővé teszi, hogy az előfizetéssel rendelkező parkolótulajdonosok ingyenesen átadhassák helyüket, amíg ők nem használják azokat. Ezalatt a Bosch földbe épített ultrahangos érzékelők segítségével gyűjt információt szabad parkolóhelyekről. [30][29]

Az Atlas rendszer abban különbözik ezektől a megoldási kísérletektől, hogy figyelembe veszi az útszéli kijelölt és kijelöletlen parkolóhelyeket is, és még csak költséges infrastruktúrát sem igényel ehhez. A rendszer beüzemeléséhez nincs szükség új parkolóhelyek létesítésére és/vagy sok emberi erőforrásra. Az Atlas szoftverrendszer a már meglévő városi kamerák által megfigyelt területeket monitorizálva teszi lehetővé kijelölt és kijelöletlen parkolóhelyek kilistázását és lefoglalását.

A projekt fejlesztése 2019 októberében kezdődött a Codespring mentorprogram részeként. Köszönet illeti Dr. Simon Károlyt és Dr. Sulyok Csabát a projekt koordinálásáért, Patakfalvi Örs-Krisztiánt és Ráduly Zalánt a mentorálásért és szakmai segítségért, valamint a csoportos projekt tantárgy kereteiben becsatlakozott diákokat: Bajcsi Annát, Gál Róbert-Rudolfot, Gergely Noémi-Laurát, Kiss Mátét és Kovács Lillát a hozzájárulásukért.

A dolgozat 1. részében bemutatásra kerülnek a projekt előzményei, az elkészült funkcionalitások és az alkalmazás architektúrája. Az ezt követő 2. fejezetben szó esik az

alkalmazásszerver felépítéséhez használt technológiákról és a szerveroldali architekturális rétegekről. A 3. fejezet, az előző fejezet struktúráját követve, betekintést nyújt a mobil kliens megvalósításának fontosabb részleteibe. A 4. fejezet a fejlesztés során felhasznált eszközöket és munkamódszereket ismerteti, míg az 5. fejezet illusztrációk segítségével mutatja be a mobilalkalmazás működését. Az utolsó rész a levont következtetéseket és a továbbfejlesztési lehetőségeket tartalmazza.

1. Az Atlas projekt

Az Atlas projekt a parkolóhelyek keresésének frusztráló folyamatát hivatott megkönnyíteni térfigyelő kamerák által nyújtott adatokat felhasználva. Eszközként egy mobilalkalmazást kínál a felhasználónak, melynek segítségével szabad parkolóhelyeket lehet kilistázni vagy térképen megjeleníteni, illetve foglalásokat lehet végezni.

2018 júniusában, az első Digitális Kaláka [6] negyvennyolc órás hackathonán készült el az Atlas projekt prototípusa, melynek célja a szoftverrendszer megvalósíthatóságának felmérése volt. A megmérettetésen a *//TODO: Team Name* nevű csapat tagjai: Gagyi Mátyás, Patakfalvi Örs-Krisztián, Ráduly Zalán és Tüzes-Bölöni Kincső készítették el és mutatták be a projekt kezdetleges változatát. Az említett változat, mely az *intelligens informatikai rendszerek a helyi forgalom optimalizálására* kategóriát képviselte, egy nem skálázható rendszerként volt megtervezve, és csupán az alapvető funkcionalitások kivitelezhetőségét térképezte fel.

A prototípus elkészülése után merült fel az ötlet, miszerint a projektet ki lehetne próbálni valós környezetben. Ennek hatására az elkészítésben résztvevő mentorok úgy vélték, érdemes lenne újrakezdeni, részletesebben megtervezni és megvalósítani azt.

Ebben a fejezetben a különböző szerepkörökhöz tartozó funkcionalitások, valamint a teljes rendszer architektúrája kerülnek bemutatásra.

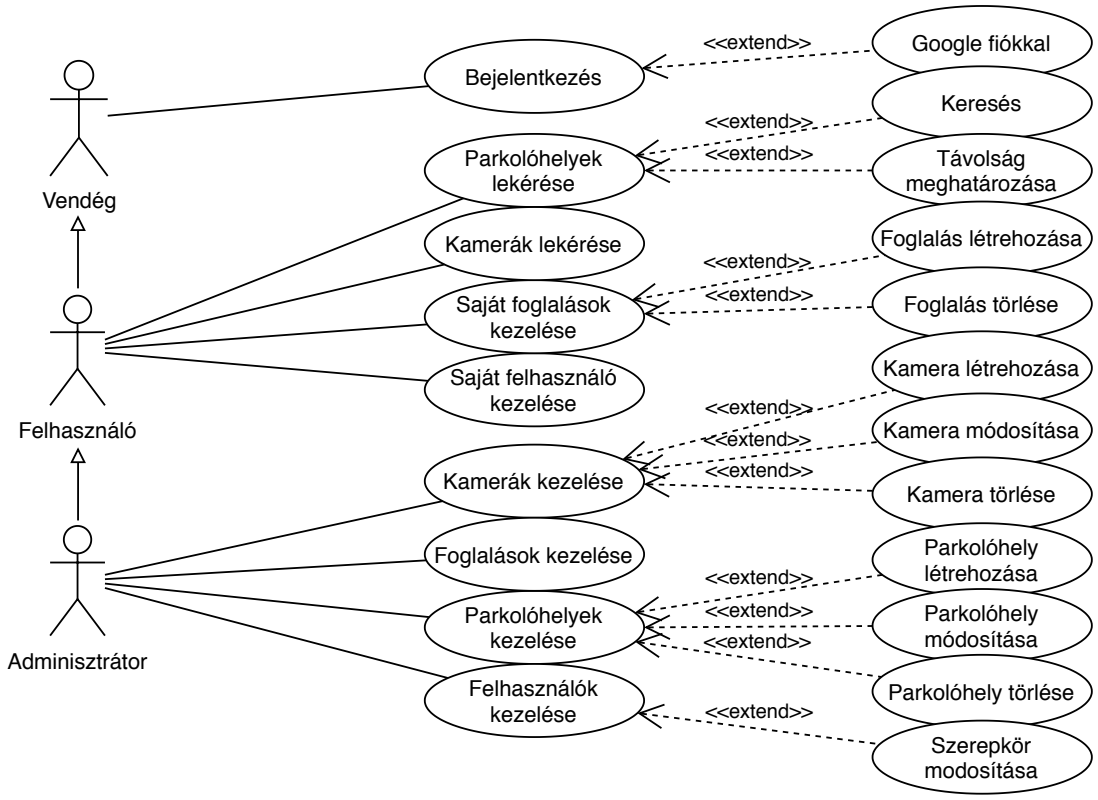
1.1. Funkcionalitások

Az alkalmazás nyújtotta szolgáltatások eléréséhez a felhasználónak szükséges bejelentkeznie Google fiókjába. A Google felhasználóval történő autentifikáció nagy mértékben felgyorsítja a bejelentkezési folyamatot, megkímélve a felhasználót a regisztrációs lépésektől és egy új fiók létrehozásától, karbantartásától. A kliens a Google API-tól [39] kapott *access token*jét továbbítja a szervernek, mely az OAuth [41] bejelentkezési keretrendszer segítségével ellenőrzi azt, amennyiben pedig érvényesnek találja, egy saját tokent küld a felhasználónak. Ez a token jogosítja fel a felhasználót a további funkcionalitások elérésére. A háttérben lezajló lépések a 2.5. alfejezetben kerülnek kifejtésre.

Az alkalmazás felhasználói három szerepkörbe sorolhatóak: vendég, egyszerű felhasználó és adminisztrátor, amint ezt az 1. ábra is szemlélteti.

A vendégnek bejelentkezésre van lehetősége, melynek során felhasználói vagy adminisztrátori szerepkörrel ruházódik fel.

Az egyszerű felhasználó minden olyan funkcionalitással rendelkezik, mely a parkolóhely keresésében segítségére válhat. Többek között joga van a parkolóhelyek és a térfigyelő kamerák adatainak lekérésére, a saját felhasználójához tartozó adatok lekérésére és módosítására, valamint foglalások létrehozására és saját foglalásainak törlésére. Mindemellett lehetősége van



1. ábra. Funkcionalitások a három szerepkör függvényében: a vendégnek bejelentkezésre van jogosultsága, a felhasználónak ezen kívül a parkolóhely-kereséshez használatos szolgáltatások is elérhetőek, míg az adminisztrátor minden funkcionalitáshoz hozzáfér.

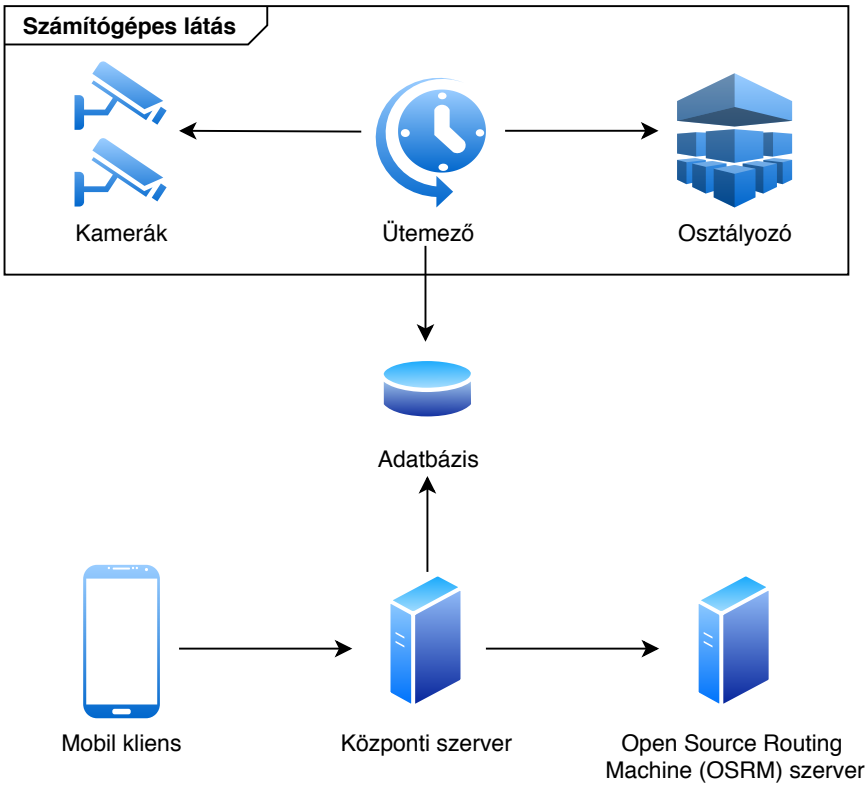
a parkolóhelyek szűrésére, illetve a személyes helyzete és a parkolóhelyek pozíciója közötti távolság lekérésére.

Az adminisztrátorok rendelkeznek az egyszerű felhasználók jogaival, és emellett további funkcionalitások is elérhetőek számukra. Az adminisztrátoroknak jogukban áll új kamerákat és parkolóhelyeket felvezetni a rendszerbe vagy módosítani a már meglévőket, szükség esetén pedig hozzáférhetnek a felhasználók adataihoz és a foglalásokhoz is.

1.2. Architektúra

A szoftverrendszer három fő részből áll, ezek a központi szerver, a mobilalkalmazás, illetve a számítógépes látásért felelős modul. A közöttük megvalósuló kapcsolatokat a 2. ábra szemlélteti.

A számítógépes látás modul a projekt mentorai által fejlesztett külső modul, mely a városi térfigyelő kamerák képeinek feldolgozását oldja meg. A képeket az ütemező komponens szolgáltatja az osztályozó számára, bizonyos időközönként lekérve ezeket az adatbázisba felvezetett kameráktól. A képek feldolgozását követően frissíti az adatbázisban tárolt parkolóhelyeket az aktuálisan elérhető szabad helyek számával. Ezt az adatbázist használja a központi szerver, melynek feladata a rendszerhez tartozó kliensek kiszolgálása.



2. ábra. A modulok közötti kapcsolatok szemléltetése

Ennek értelmében a szerver felelős a hitelesítésért, az adatbázisban tárolt naprakész adatok továbbításáért és kezeléséért, miközben egy skálázható platformot emel az adatbázis köré. A szoftverrendszerhez tartozó mobilalkalmazás, mely elérhető Android és iOS eszközökön egyaránt, lehetőséget biztosít a felhasználóknak szabad parkolóhelyek kilistázására és lefoglalására az alkalmazásszerverrel megvalósított kommunikációnak köszönhetően.

A szerver és a mobil kliens felépítését részletesebben a 2. és 3. fejezetek ismertetik.

2. Központi szerver

Az alkalmazásszerver feladata a rendszerhez tartozó kliensek kiszolgálása. A szerver felelős a hitelesítésért, a parkolóhelyekkel kapcsolatos naprakész információk szolgáltatásáért és kezeléséért, valamint az adatbázissal való kapcsolatfenntartásért, mindezt biztonságos módon megoldva.

A szerver felépítése a többrétegű architektúra [11] elveit követi, ennek értelmében a következő rétegeket különíthetjük el: adatmodell, adathozzáférési réteg és API (Application Programming Interface). Az alkalmazásszerver rétegekre bontásával modulárisan lehet dolgozni ezeken, így könnyen karbantartható és fejleszthető a rendszer. Ezen architektúra használatával ugyanakkor elkerülhető a körkörös függőségek kialakítása, hiszen a rétegek csak egy irányba folytatnak kommunikációt.

Ebben a fejezetben bemutatásra kerülnek a szerver oldalon felhasznált technológiák, ezt követően a különböző architekturális rétegekről lesz szó, végül pedig a biztonsággal kapcsolatos részletek is említésre kerülnek.

2.1. Felhasznált technológiák

Ez az alfejezet az alkalmazásszerver megvalósításához felhasznált technológiákat és fontosabb külső könyvtárakat mutatja be.

2.1.1. Node.js

A Node.js [37] egy nyílt forráskódú JavaScript futtatási környezet, amely a böngészőn kívül, egyetlen szálon valósítja meg szerver oldali alkalmazások futtatását és kitűnően hasznosítható skálázható webalkalmazások kiszolgálására.

A projekt esetén a Node.js használatát indokolja, hogy ezáltal egységesen ugyanazt a programozási nyelvet lehet alkalmazni a szerver oldalon, mint a mobil kliens esetén. Ugyanakkor ez egy nem erősen típusos nyelv, így kényelmesen lehet vele Not Only SQL (NoSQL) adatbázisokat kezelni, amilyen a választott MongoDB is.

A Node.js-hez tartozó számos modul nagyban elősegíti az alkalmazások fejlesztését. A külső modulok a Node Package Manager (npm) [35] csomagkezelő rendszerrel telepíthetőek. A RESTful [10] API megalkotásához az *Express.js* minimalista és flexibilis keretrendszer van használatba ültetve, mely ideális web és mobil alkalmazásszerverek, illetve API-ok készítésére. Az Express alapértelmezett hibakezelővel rendelkezik, azonban a projekt esetében ez felülírásra került, hogy a szerver releváns hibaüzenetekkel láthassa el a klienseket. A naplózást a *Winston* könyvtár hajtja végre, melynek segítségével rugalmassá és bővíthetővé válik ez a folyamat. A Winston használata teszi lehetővé a naplózandó üzenetek formázását.

2.1.2. MongoDB

A MongoDB egy nyílt forráskódú, dokumentumorientált adatbázis, a legnépszerűbb a NoSQL adatbázis szoftverek között [40]. A relációs adatbázisokkal ellentétben, az adatokat nem táblákban, hanem Bináris JSON (BSON) formátumú dokumentumokban tárolja [4], amely egy az emberi logikához közelebb álló megközelítés, ugyanakkor pedig egy helytakarékosabb megoldás. Jelen projekt esetében a parkolóhelyek pozíciójának tárolása teszi indokolttá a MongoDB használatát, mivel ez az adatbázis szoftver alapértelmezetten támogatja a földrajzi helyzet típusú adatot. Mindemellett a MongoDB támogatja az ideiglenes adatok tárolását is, amit szintén hasznosítani lehet a projektben.

A Mongoose [23] egy könnyen kezelhető, séma-alapú megoldást nyújt az alkalmazás adatainak modellezésére, hidat képezve a Node.js és a MongoDB között, így ennek használata oldja meg az automatikus leképezést az adatbázis bejegyzések és a lokális objektumok között. Emellett a Mongoose segítségével valósul meg a szerver oldali validálás is, hiszen a Mongoose beépített típuskonverziót és típusellenőrzést tartalmaz.

2.1.3. Docker

A Docker [28] egy alkalmazások futtatására és kitelepítésére használt virtualizációs eszköz, amely biztosítja, hogy az alkalmazás futtatható legyen bármilyen környezetben. Mindezt konténerek segítségével teszi lehetővé, melyek egy virtuális futási környezetet hoznak létre. A konténerek az adott alkalmazást és annak függőségeit tartalmazzák. Saját alkalmazás futtatásához saját Docker image készítése szükséges, mely Dockerfile segítségével valósítható meg. Jelen projekt esetében egy Node.js image képezi ezen fájl kiindulópontját.

A Docker Compose több konténerből álló alkalmazásoknál célszerű eszköz [38]. Az Atlas rendszer esetén szükségesnek bizonyul használata, hiszen így össze lehet kapcsolni az alkalmazásszervert, a MongoDB adatbázist, az adatbázist demó adatokkal feltöltő modult, illetve az Open Source Routing Machine szervert. Ezt követően egyetlen paranccsal futtatható a rendszer bármilyen környezetben.

2.1.4. OSRM

Az Open Source Routing Machine (OSRM) egy C++ implementációval rendelkező, magas teljesítményű motor a legrövidebb útvonalak meghatározásához úthálózatokban. Egyesíti a kifinomult útválasztási algoritmusokat az OpenStreetMap (OSM) projekt nyílt és ingyenes úthálózati adataival [24]. A Google Places API-val és egyéb megoldásokkal ellentétben az OSRM nem számoltat fel használati díjat az útbeli távolságok meghatározásához, ez indokolja használatát a projektben.

```
FROM osrm/osrm-backend
WORKDIR /
RUN apt-get update \
       && apt-get install -y --no-install-recommends curl ca-certificates \
       && apt-get clean \
       && rm -rf /var/lib/apt/lists/*
ARG map_url=https://download.geofabrik.de/europe/romania-latest.osm.pbf
ARG download_directory=/data/osrm/areas
RUN curl --create-dirs --output ${download_directory}/map.osm.pbf ${map_url}
RUN osrm-extract -p /opt/car.lua ${download_directory}/map.osm.pbf
COPY ./prepare.sh ./
ARG algorithm=mld
RUN chmod +x ./prepare.sh && sleep 1 \
       && ./prepare.sh ${download_directory} ${algorithm}
RUN rm ${download_directory}/map.osm.pbf
ENV ALGORITHM=${algorithm}
ENV MAP_PATH=${download_directory}
CMD osrm-routed --algorithm $ALGORITHM $MAP_PATH/map.osrm
```

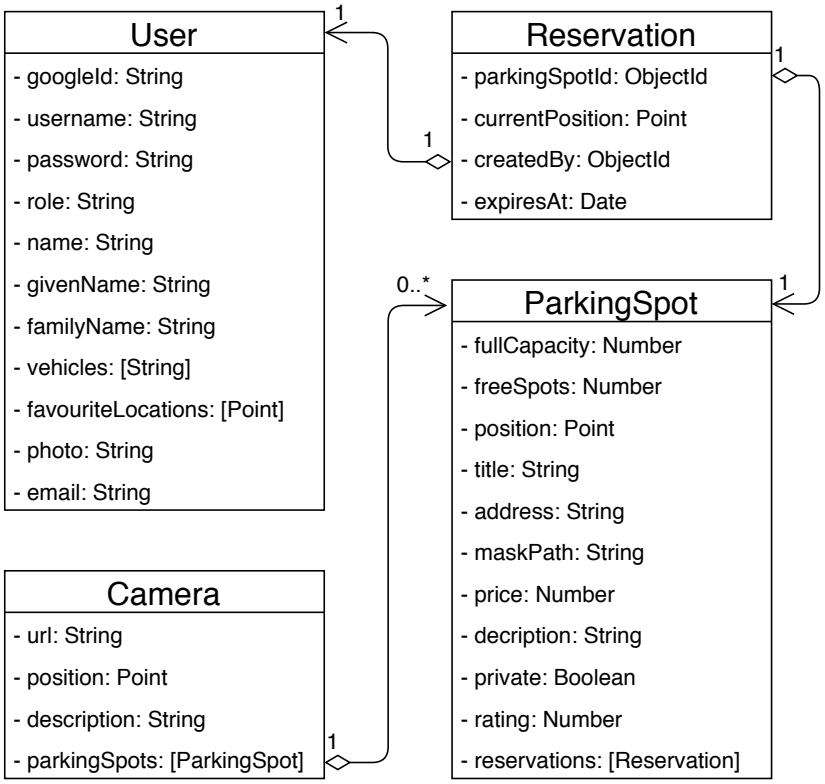
1. kódrészlet. Dockerfile az OSRM szerver konfigurálásához. Ebben van meghatározva, többek között, a használt térkép (Románia térképe) és a közlekedési profil típusa (autós közlekedés).

Egy saját útvonalválasztó motor beállításához egyszerű és gyors módszer az OSRM projekt által biztosított Docker image használata. Ez az image képezi az Atlas projekt OSRM szerveréhez tartozó Dockerfile kiindulópontját, ahogy azt az 1. kódrészlet is szemlélteti. A Dockerfile segítségével történik meg a szükséges bizonyítványok és a felhasználandó térképek letöltése, a személygépjárműves profil beállítása, valamint az útkeresési algoritmus specifikációja. Az ezáltal létrejött OSRM szerver teszi lehetővé a parkolóhelyek közötti távolságok és a távolságok megtételéhez szükséges időtartamok kiszámítását. A jelenlegi beállításoknak megfelelően az alkalmazás Románia térképét alkalmazza, ám könnyen kiterjeszthető egész Európára.

2.2. Adatmodell

A szerver által kezelt adatok egy dokumentum alapú NoSQL adatbázisban kerülnek tárolásra. Mivel egy dokumentumorientált adatbázisról van szó, az adatok kollekciókban, azokon belül pedig dokumentumokban vannak tárolva. Ezek a dokumentumok bizonyos sémákat követnek tartalmilag, melyek külön mappában kerülnek meghatározásra. A sémák határozzák meg, hogy milyen mezők használatosak a különböző kollekciókban, milyen típusúak ezek a mezők, mik az esetleges alapértelmezett értékeik, szükséges-e egyedi értékük legyen, illetve kötelezően kell-e alapértelmezett értékkel rendelkezzenek.

A szerver felépítéséhez az alábbi sémák vannak definiálva: *User*, *ParkingSpot*, *Camera*, *Reservation*; ezeken kívül pedig szükség van egy sémára a földrajzi helyzet megadásához (*Point*) és egy másikra a koordináták által alkotott sokszögek leírásához (*Polygon*). A négy



3. ábra. A szerveroldalon négy fő séma van definiálva, az ábrán az ezeknek megfeleltetett lokális objektumok és kapcsolataik láthatóak.

fő sémának megfelelő lokális objektumokat és ezek kapcsolatait szemlélteti a 3. ábra.

A *Users* kollekció a Google fiókkal bejelentkezett felhasználók adatait tárolja. A felhasználó beleegyezésével a Google szolgáltatja az alkalmazás számára az illető személy e-mail címét, nevét, Google ID-ját és Google profilképének URL címét. A sémában definiált további mezők alapértelmezett értéket kapnak, vagy pedig az előzőek alapján generált értéket: szerepkörként minden újonnan bejelentkező felhasználó egyszerű felhasználói jogosultságot kap, felhasználóneve pedig a keresztnevéből generálódik. Az egyedi felhasználónév (username) a keresztnév kisbetűsített és (több keresztnév esetén) összeolvasztott változata egy random négyjegyű számmal megtoldva. Későbbi funkcionalitásokra való tekintettel olyan mezők is definiálva vannak a sémában, mint például a jelszó vagy a felhasználó járműveinek és kedvenc helyeinek listája. A jelszó akkor lesz majd releváns, amikor implementálásra kerül az önálló regisztrációs rendszer. A két lista értéke jelenleg egy-egy üres tömb, ezek is jövőbeli felhasználásra várnak.

A *Cameras* kollekcióban kerülnek tárolásra a rendszerbe felvezetett térfigyelő kamerákhoz tartozó információk, többek között a kamera földrajzi helyzete, melyet egy Point objektummal [3] lehet megadni, és a megfigyelése alatt álló parkolóhelyek listája.

A *ParkingSpots* kollekció tartalmazza a kamerák által megfigyelt parkolóhelyek lényeges adatait. Tárolásra kerülnek olyan, többnyire állandó adatok, mint például a parkoló neve, címe,

pozíciója, maximális helykapacitása, privát vagy nyilvános jellege, ára és értékelése. Emellett ugyanakkor folyamatosan változó tulajdonságai is vannak egy-egy parkolóhelynek, ilyenek a szabad helyek száma vagy a parkolóra vonatkozó foglalások listája. A szabad helyeket a számítógépes látásért felelős modul határozza meg.

A *Reservations* kollekcióban kerülnek tárolásra a foglalások, melyeket négy tulajdonság határoz meg: a parkolóhely, ahova a foglalás szól, a felhasználó, aki a foglalást végezte, a foglalás lejárati időpontja, valamint a felhasználó kiindulási helyzete. A projektben használt MongoDB adatbázis lehetőséget ad ideiglenes dokumentumok létrehozására, így ezt felhasználva a foglalások a megadott lejárati időpontban automatikusan törlődnek, nem szükséges erőbefektetés ezeknek a kezelésére.

2.3. Adathozzáférési réteg

Az adatok elérését és manipulációját a 2.1.2 fejezetben már említett Mongoose ODM valósítja meg. A Mongoose egyszerű megoldást kínál az alkalmazás adatainak modellezésére. Ezáltal lehetőséget nyújt arra, hogy a szerveren olyan objektumokkal lehessen műveleteket végezni, melyek megfelelnek az adatbázis dokumentumainak. A létrehozott Mongoose modelleken keresztül könnyedén hozzá lehet férni az adatokhoz, nem szükséges MongoDB szkripteket írni a *Create, Read, Update, Delete*, röviden CRUD, műveletekhez sem. A CRUD a négy alapművelet, tulajdonképpen egy konvenciórendszer az erőforrások elérésére/módosítására, mely az állandó (perzisztens) adattárolást segíti elő. Az előbb említett alapműveletek megvalósítását a *controllers* réteg hajtja végre, és ugyancsak ez a réteg szolgáltat választ a kliensnek sikeres adatlekérés vagy -módosítás esetén. A *controller*-ek szoros kapcsolatban állnak a dekorátor mintát [36] követő *decorators* réteggel. Egy *decorator* feladata dinamikusan kiegészíteni a kliens kérését a módosítani vagy elérni kívánt objektummal, ezáltal csökkentvén az adott *controller* feladatainak számát.

2.4. API

A mobilalkalmazás által küldött Hypertext Transfer Protocol (HTTP) kéréseket egy RESTful API szerver szolgálja ki. Az alkalmazás ebből származó előnyei közé sorolható, hogy laza kapcsolást (loose coupling) és skálázhatóságot eredményez.

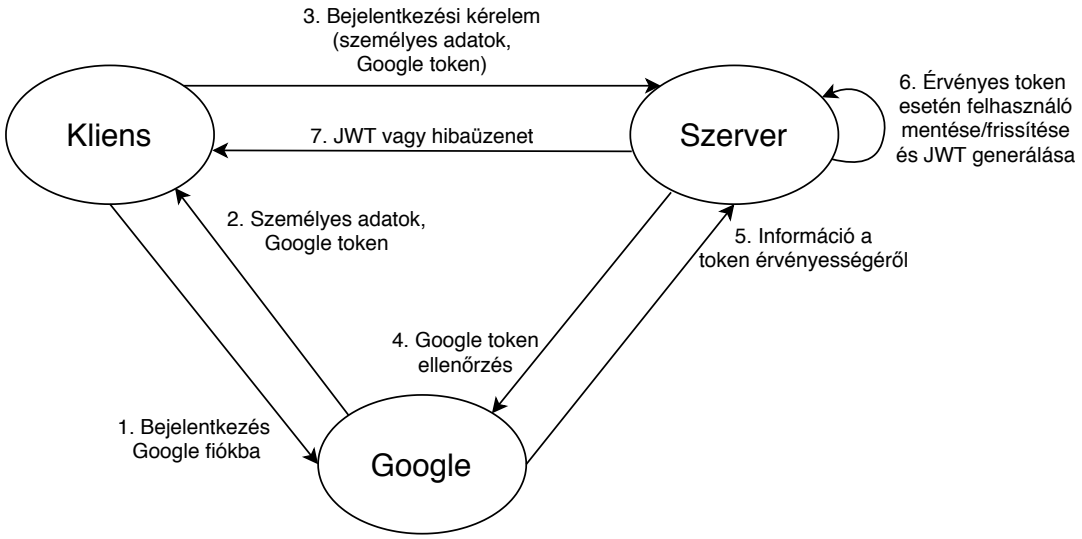
A beérkező kéréseket a *routes* réteg fogadja, és ez irányítja tovább azokat a *decorators*, majd pedig a *controllers* rétegeknek. A REST architektúra értelmében minden erőforrás egyedi URI-n keresztül érhető el. Az API szerver `/api/users`, `/api/parking-spots`, `/api/cameras` elérési pontjaihoz küldött GET kérések adják vissza a felhasználókról, parkolóhelyekről és térfigyelő kamerákról tárolt adatokat JavaScript Object Notation (JSON) formátumban,

amennyiben a kérés fejléce tartalmazza a bejelentkezett felhasználó tokenjét. Ugyanezen pontokra küldött POST, DELETE, PUT kérésekkel módosítható az adatbázis tartalma.

Egyetlen felhasználó adatainak lekérését vagy szerkesztését a `/api/users/:userId` elérési ponton keresztül lehet megtenni. Hasonlóan történik ez a parkólóhelyek és a térfigyelő kamerák esetén is. A kameráknál lehetőség van új parkolóhely felvezetésére is POST kérés küldésével a `/api/cameras/:cameraId/parking-spots` végpontra. A parkolóhelyek esetén új foglalást lehet végrehajtani vagy törölni lehet egy meglévő foglalást a megfelelő kérés küldésével a `/api/parking-spots/:parkingSpotId/reservation` elérési pontra. Mindezen pontokhoz is csak a megfelelő jogosultságokkal lehet hozzáférni, ezek hiányában válaszként egy megfelelő HTTP státuszkóddal ellátott helyzetleíró JSON objektum térítődik vissza.

2.5. Biztonság

Az Atlas API használatba vételéhez a felhasználónak először be kell jelentkeznie Google fiókjával. Ez a bejelentkezés több lépésből áll a háttérben, amint ez a 4. ábrán megfigyelhető. A kliensek egy Google hitelesítésért felelős eszköz használatával megkapott Google tokenjüket küldhetik fel a szervernek személyes adataik kíséretében. A szerver egy *OAuth2* könyvtár segítségével ellenőrzi a kérésként beérkezett adatokat. Az OAuth2 egy *autorizációs keretrendszer*, amely korlátozott hozzáférést biztosít az alkalmazások számára különböző felhasználói fiókokhoz, mint például Google, Facebook vagy Github fiókokhoz [41]. Ezen eszköz használatával intéz kérést az Atlas szervere a Google által biztosított API-hoz. Az alkalmazás adataival információkat lehet lekérni a kliens tokenjéről, így lehetőség van ellenőrizni, hogy valóban az Atlas alkalmazás bocsátotta-e ki és hogy egyáltalán érvényes-e



4. ábra. A háttérben lezajló lépések ábrázolása bejelentkezés esetén.

még. Amennyiben minden helyes, a szerver egy bizonyos időtartamra érvényes JSON Web Tokent (JWT) generál és küld vissza a kliensnek. A kliens ezáltal sikeresen be van jelentkezve, és a JWT token segítségével immár elérheti az alkalmazás nyújtotta szolgáltatásokat. A token lejárati idejét követően egy újbóli bejelentkezés biztosítja a friss token kiállítását. Az adatbázisban már létező felhasználó esetén a szerver frissíti annak Google által szolgáltatott adatait, ellenkező esetben ezek mentése mellett egyedi felhasználónevet is ad neki, és szerepkörét beállítja egyszerű *user*re.

A hitelesítést követően a kliens számára elérhetővé válnak az API végpontjai, amennyiben jogosultsága van ezekhez. Minden egyes beérkezett kérésnél a szerver egy köztesréteg (middleware) segítségével ellenőrzi a token érvényességét és a felhasználó szerepkörét. Egyszerű felhasználóként lehetőség van a parkolóhelyek és térfigyelő kamerák adatainak lekérésére, a saját foglalások kezelésére, valamint a saját felhasználó adatainak lekérésére és módosítására. Adminisztrátor szerepkör esetén minden funkció elérhető.

3. Mobil kliens

A mobil kliens feladata az alkalmazásszerverrel való kommunikáció megvalósítása és a felhasználói felület megjelenítése Android és iOS platformú eszközökön. A mobil rész a React Native keretrendszert használja, ez a 3.1.1 alfejezetben kerül részletezésre. A szerver architektúrájához hasonlóan a mobil kliens is több, jól elkülönített rétegből tevődik össze. A szerverrel való kommunikációt és az egyéb üzleti logikával kapcsolatos feladatokat a *services* réteg oldja meg. A szervertől kapott adatokat a *store* osztályok veszik át, ezek felelősek az információ tárolásáért. A *screens* réteg a telefon képernyőjén való nézetek létrehozásáért felelős, melyhez felhasználja a *components* rétegben definiált grafikai elemeket.

A fejezet további részében a mobil kliens fejlesztéséhez felhasznált technológiákról lesz szó, majd a szerverrel való kommunikáció, az üzleti logika, a megjelenítési réteg és a felhasználói réteg kerülnek bemutatásra.

3.1. Felhasznált technológiák

Ebben az alfejezetben a mobilalkalmazás fejlesztése során felhasznált fontosabb technológiák lesznek bemutatva.

3.1.1. React Native

Az egyik probléma, ami felléphet a mobil alkalmazások fejlesztésekor, a mobil piac többplatformúságából adódik. Ennek eredményeként, több platformon is elérhető alkalmazás esetén, a fejlesztés időtartama lényegesen hosszabb lesz, szükséges a különböző programozási nyelvek és környezetek ismerete, illetve az alkalmazás karbantartása is egy bonyolultabb folyamatot jelent. A megoldás erre a problémára a cross-platform mobilalkalmazások fejlesztésében rejlik. A megfelelő keretrendszer választásával minimális mennyiségű natív kód írása szükségeltetik, és az alkalmazás natív, megszokott kinézetet kap úgy Androidon, mint iOS-en.

Jelen projekt esetében a React Native-re [8] a kiválasztott cross-platform keretrendszer, mely natív mobilalkalmazások fejlesztését teszi lehetővé React, JavaScript és a natív nyelvek ötvözetének használatával. A keretrendszer lehetőséget nyújt natív kód írására a platform teljes erejéhez való hozzáférhetőség érdekében. A React dinamikus, komponens alapú és deklaratív voltának köszönhetően könnyedén alkothatók komplex, interaktív felhasználói felületek, míg a Node Package Manager számos könyvtár használatát teszi lehetővé.

Az egyik említésre méltó, jelen projekt esetében használt könyvtár a *react-navigation* 4-es verziója, mely segítséget nyújt az alkalmazás navigációs struktúrájának létrehozásában. Egy másik ilyen könyvtár a *react-native-map-clustering* [1], mely az alapértelmezett React Native

térképre épül, és ezen felül megvalósítja a markerek klaszterezését a térképen. A földrajzi helyzethez való hozzáférés felhasználó általi engedélyeztetését a *react-native-permissions* könyvtár teszi lehetővé.

A Babel egy *source-to-source kompilátor*, melynek fő feladata visszafele kompatibilissé tenni az újabb ECMAScript (ES) szabványoknak megfelelő kódot [14]. A React Native alapértelmezetten alkalmazza a Babelt, hogy a React szintaxist és az ES5+ szintaxist futtathatóvá tegye olyan JavaScript környezetben is, amely nem támogatja ezeket a funkciókat. A Babel emellett egyéb eszközöket is nyújt a fejlesztési folyamat gördülékenyebbé tételéhez. Az Atlas mobilkliensének esetében segítségül szolgál új gyökérkönyvtárak megnevezéséhez, ilyenek például a `@components`, a `@screens` vagy a `@services`, így ezek bármely fájlból egyszerűen elérhetőek és az importok is átláthatóbbak.

Expo

Kezdetben, a függőségek minimalizálása érdekében, a mobil fejlesztés az Expo [16] keretrendszer és platform használatával történt. Ez egy olyan eszköz- és szolgáltatáskészlet, melynek használatával a fejlesztés ugyanúgy megvalósítható Windowsról, mint macOS-ről (csak internetkapcsolatra és Node.js-re van szükség), és az alkalmazást közvetlenül lehet Androidon és iOS-en is tesztelni az Expo tárolóalkalmazáson keresztül. Lényegében az Expo megkíméli a kevésbé jártas szoftverfejlesztőt a natív fejlesztés összetettségétől és lehetővé teszi, hogy csak akkor kezelje ezt a komplexitást, ha feltétlenül szükséges.

Az Atlas mobilalkalmazásának esetében azonban az Expo korlátozza a fejlesztői lehetőségeket, bizonyos natív funkcionalitásokat nem támogat és a parkolóhelyeket megjelenítő térkép teljesítményét is rontja. Ennek értelmében történt meg az áttérés egyszerű React Native alkalmazásra. Ez az összes Expo függőség kiiktatását, a projekt átformálását jelentette, mely ugyanakkor magasabb fokú szabadságot eredményezett, és egy kis betekintést engedett nyerni a natív fejlesztés világába.

3.1.2. MobX

A React komponensek mindegyike saját állapottal rendelkezik, a mobilalkalmazás esetén pedig vannak olyan adatok, melyeket több komponens is használ. Annak érdekében, hogy az adatmegosztás és -karbantartás zökkenőmentesen működjön, szükséges egy állapot menedzsment rendszer használata.

Állapot menedzsment rendszereket illetően az Atlas rendszer esetében a MobX van használatban, hiszen egyszerű, könnyen skálázható és erős kombinációt alkot a Reacttal. A MobX biztosítja azt a mechanizmust az alkalmazás állapotának tárolásához és frissítéséhez, melyet azután a React Native használ fel az alkalmazás megjelenítéséhez. A rendszer

alapgondolata: „Mindent, amit az alkalmazás állapotából származtatni lehet, származtatni is kell. Automatikusan.” [31]

A MobX kiemeli az alkalmazás állapotát a komponensekből, így az könnyedén megosztható a különböző egységek között. Az adatok kezelése központi tárolókban, úgynevezett *store*-okban történik, ezekkel állnak kapcsolatban a megjelenítésért felelős komponensek. Egy-egy store-ban egymással szorosan összefüggő adatok vannak tárolva, az adatokat pedig az `@observable` annotáció teszi megfigyelhetővé. Ezek megfigyelői a felhasználói felületet alkotó komponensek, melyek `@observer` annotációval rendelkeznek.

3.1.3. Native Base

A Native Base [21] egy ingyenes és nyílt forráskódú alapvető UI komponenseket tartalmazó könyvtár, melyet React Native-hez fejlesztettek ki. Használata lehetővé teszi, hogy az alkalmazás natív kinézetet nyerjen úgy iOS-en, mint Androidon, anélkül, hogy natív kódot kellene írni. A Native Base-nek köszönhetően a felhasználókat egy kellemes, megszokott felhasználói felület fogadja.

Jelen projekt saját témával rendelkezik, melynek létrehozása a Native Base könyvtár alapértelmezett témájának *ejectelése* és felülírása által valósítható meg. Ez a testreszabás elsősorban a mobilalkalmazás színskáláját érinti.

3.1.4. Firebase

A Firebase [17] egy sokoldalú szolgáltatáscsomag a Google-től mobil- és webalkalmazásokhoz, amin keresztül számos funkció megvalósítható egyszerűen, backend infrastruktúra nélkül. Ilyen szolgáltatások például a bejelentkezés, tartalmak kedvencek közé helyezése vagy az értesítések küldése/fogadása bizonyos események után. A funkcionalitások közül a hitelesítés (Authentication modul) van használatban a felhasználók bejelentkeztetéséhez Google fiókjukkal. Ehhez szükséges egy projekt létrehozása a Firebase vezérlőpultjában, illetve a Google fejlesztői konzolján, majd a kigenerált konfigurációs fájlok, API kulcsok és OAuth2 kliens azonosítók felhasználásával valósítható meg a hitelesítés és bejelentkezés. Ehhez az Atlas mobilalkalmazás a *@react-native-community/google-signin* könyvtárat használja fel.

A fejlesztés későbbi fázisaiban a Firebase egyéb szolgáltatásainak igénybevétele is tervbe van véve. Ilyenek például az értesítések kezelése, vagy a *Test Lab* használata. A Test Lab lehetővé teszi az alkalmazás tesztelését a projekthez hozzáadott személyek által, akik automatikusan megkapják az applikációt. Ez a széleskörű, több eszközön való tesztelés segíthet a hibák felismerésében mielőtt az alkalmazás a nagyközönség által is elérhetővé válna.

```
import api from '@src/api';

class ParkingSpotsService {
        getParkingSpot(parkingSpotId) {
                return api
                        .get(`/parking-spots/${parkingSpotId}`)
                        .then(response => {
                                return response.data;
                        })
                        .catch(error => {
                                console.debug('Could not get parking spot:', error);
                                throw new Error('Could not get parking spot');
                        });
        }
        /* ... */
}
```

2. kódrészlet. A service osztályok a példányosított axiost (api) használják fel a szerverrel való kommunikáció megvalósításához.

3.2. Kommunikáció a szerverrel

A mobilalkalmazás REST kéréseken keresztül kommunikál a központi szerverrel. A REST API-hoz küldött kérések fejlécének és törzsének összeállításáért az *axios* [27] könyvtár felel. Az axios egy böngészők és Node.js számára kifejlesztett promise alapú HTTP kliens. Ez a könyvtár megkönnyíti az API kliens implementálását, lehetőséget adván aszinkron hálózati kérések küldésére és válaszok fogadására.

A különböző *service* osztályok az egyéni konfigurációval példányosított axios példányt használják fel a szerverrel való kommunikációhoz, ezt hivatott szemléltetni a 2. kódrészlet. Paraméterként az API megfelelő elérési pontját szükséges megadni ennek az instanciának, opcionálisan pedig szerver felé küldendő adatokat is lehet továbbítani. A hálózaton keresztüli kommunikáció aszinkron módon működik, tehát az axios kérés egy Promise-t térít vissza, ami egyszerűvé teszi ezeknek az aszinkron válaszoknak a kezelését és feldolgozását.

A bejelentkezést követően a szervertől visszakapott személyes token az axios példány alapértelmezett fejlécében kerül elhelyezésre, így nem szükséges minden kérésnél gondoskodni róla. A szerver ellenőrzi a token érvényességét, majd a felhasználó szerepkörének függvényében válaszol a kérésre. Kijelentkezés esetén a fejléc tartalma törlődik.

3.3. Üzleti logika

A mobilalkalmazás esetén az üzleti logikát a *services* réteg képviseli. A service osztályok tartalmazzák azokat a függvényeket, amelyek a 3.2 fejezetben leírt módon implementálják a szerverhez intézendő kéréseket, vagy a már lekért adatok alapján végeznek számításokat. Az üzleti logika külön van választva a megjelenítési rétegtől, a szerver fele csak a service osztályok

indítanak kéréseket. A store-ok ezen osztályok egy-egy példányát használják fel a szükséges adatok szervertől való lekéréséhez.

A service-ek valósítják meg a tulajdonképpeni Google hitelesítést, itt épülnek fel az adatok ellenőrzése után bizonyos kérések paraméterlistái, és ugyancsak itt kerülnek ellenőrzésre a szervertől kapott válaszok. Ha a válasz rendben van, akkor az adatokat a megfelelő store kapja meg, ellenkező esetben pedig egy Error objektumot dob vissza a függvény, amit a felsőbb rétegek kapnak el és kezelnek le megfelelően.

3.4. Megjelenítési réteg

A megjelenítési réteg három részből tevődik össze: *screens*, *components* és *stores*. A screens réteg tartalmazza azokat nézeteket, melyek között navigálhat a felhasználó. Példának okáért, itt van felépítve a bejelentkezésnél látható képernyő vagy a térképet megjelenítő képernyő is. Ezek felhasználnak különböző komponenseket, melyek a componentsben vannak definiálva. A komponensek olyan elemek, melyekre több képernyő esetén is szükség van, vagy pedig külön egységként kezelhetőek. Ugyanakkor a navigációhoz szükséges, a képernyőn állandó jelleggel jelen lévő elemek is a components részei, ilyen például a képernyő alján található navigációs sáv. A képernyők és komponensek számára az adatokat a store-ok szolgáltatják, melyek service osztályokat használnak fel a szerverrel való kommunikáció megvalósításához.

A React Native többféle lehetőséget biztosít az alkalmazásban való navigálás megvalósítására a *react-navigation* könyvtár által. Az Atlas mobilalkalmazás esetében a navigáció alapját egy *Switch Navigator* képezi. Ez köti össze a bejelentkezésre alkalmas *Login* képernyőt, a felhasználót köszöntő *Welcome* képernyőt és az „alkalmazás belsejét", így ezek csak a megadott sorrendben érhetőek el. Az alkalmazás belseje egy következő navigációs elemet foglal magába, mégpedig a *Drawer Navigator*t. A Drawer Navigator hivatott összefogni a baloldali menühüz tartozó *Stack Navigator* elemeket. Jelenlegi állás szerint itt egyetlen elem található, mely egy negyedik fajta navigátort alkalmaz: a *Bottom Tab Navigator*t. Ez a képernyő alján található navigációs sáv valósítja meg a térkép- és a listanézet közötti váltást.

A térképes nézet fő komponense értelemszerűen a térkép. Egységes kinézet érdekében az Android és iOS platformú eszközök egyaránt a Google által biztosított térképet használják. Az automatikus/gombnyomásra végzett régióváltozások animálva vannak a térképen a React Native beépített függvényének segítségével, ezáltal is hozzájárulván ahhoz, hogy az alkalmazás megfelelő felhasználói élményt nyújtson. A hatékonyság növelése érdekében a térkép mindig csak az aktuálisan megjelenített régió parkolóhelyeit kéri le a szervertől, melyek pozícióját sajátos markerekkel jelöli. Megjegyzendő, hogy az alapvető React Native térkép helyett egy olyan könyvtár van használatba ültetve, mely a markerek klaszterezését is megvalósítja.

Mindemellett a térképen engedélyezve van az utak zsúfoltságának, az aktuális forgalomnak a jelölése.

A lista nézet részletesebb információkat szolgáltat a parkolóhelyekről, többek között a felhasználó útbeli távolságát is megjeleníti ezektől. A távolság, melyet a szerver számol ki, minden ötödik másodpercben kerül frissítésre.

Úgy a térképes, mint a lista screen használ egy, a képernyő aljából felemelkedő komponenst, mely részletes információkat szolgáltat adott parkolóhelyről. Egy másik közös pontját képezi a két nézetnek a fejléc. Ez a komponens tartalmazza az oldalsó menü megnyitására alkalmas hamburger ikont, illetve a keresési fejlécet megjelenítő ikont. A keresés az adatbázisban tárolt parkolóhelyek neve és címe alapján valósul meg betű szerinti szűréssel.

Az alkalmazás által használt adatok store-okban foglalnak helyet. Ezekben a tárolókban a nézetek számára releváns adatok `@observable` annotációval vannak ellátva, míg azok a függvények, melyek az adatok értékeit módosíthatják `@action` megjelöléssel rendelkeznek, így értesíthetik az érdekelt nézeteket a változásról. Ahhoz, hogy egy nézet értesüljön az adattagok változásairól és újrarajzolja a megfelelő részeket, `@observer` annotációval kell rendelkeznie, amint ez már említésre került a 3.1.2. alfejezetben.

Jelen projekt esetében három központi tárolót különböztethetünk meg: *authentication store*, *map store* és *parking spot store*. Az első store felelős a bejelentkezett felhasználó adatainak tárolásáért, és ugyanakkor a hitelesítést végző service függvényeinek meghívásáért, amint ez a 3. kódrészletből kivehető. A map store a térképes nézettel áll szoros kapcsolatban, hisz itt kerül tárolásra a térkép által aktuálisan megjelenített régió és a felhasználó földrajzi helyzete. A harmadik tároló a szervertől lekért parkolóhelyeket tartalmazza.

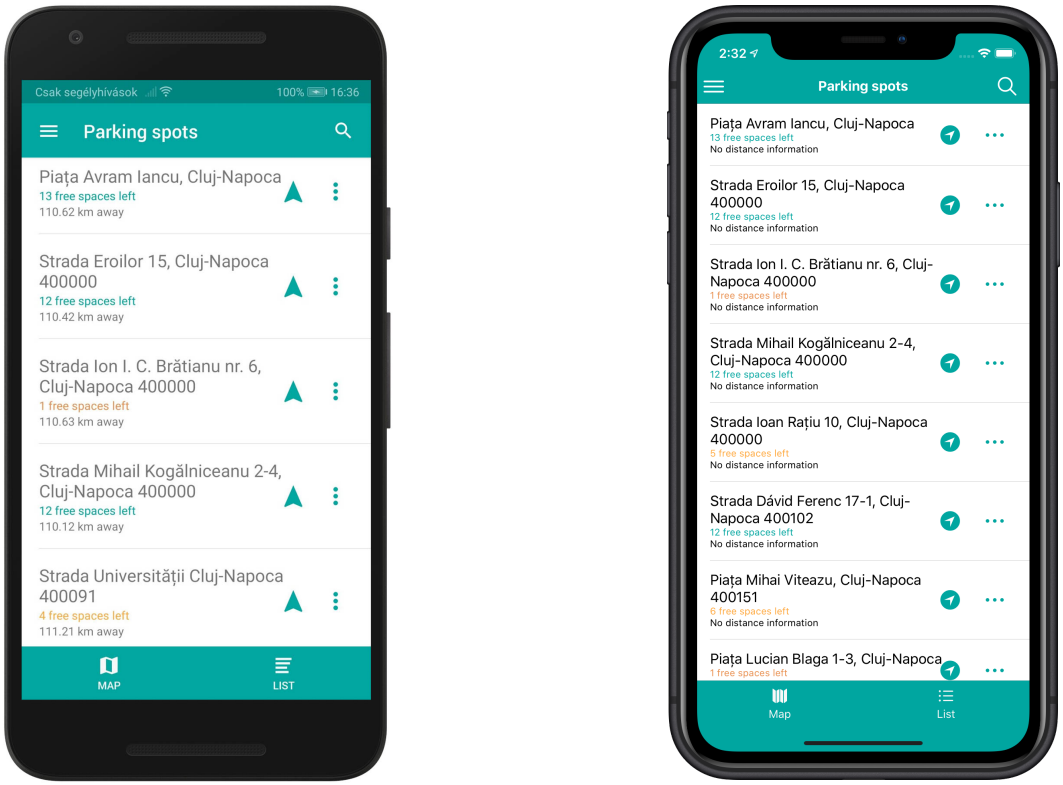
```
import { action, observable, runInAction } from 'mobx';

export default class AuthenticationStore {
        @observable user = undefined;

        constructor() {...}

        getUser() {...}

        @action googleSignIn() {...}

        @action googleSignOut() {...}
}
```

3. kódrészlet. Az authentication store felel a bejelentkezett felhasználó adatainak tárolásáért és a hitelesítést végző service függvényeinek meghívásáért. Az `@observable` annotáció megfigyelhetővé teszi a *user* adattagot. Az `@action` annotáció azon függvények előtt szerepel, melyek megváltoztathatják a megfigyelt adattag értékét. A `runInAction` az utóbbi annotációhoz hasonlóan működik, olyan függvények megjelöléséért felelős, melyek módosíthatják az adattagokat.

3.5. Felhasználói felület

React Native-ben a felhasználói felületet jellemző nézeteket komponensek építik fel, hiszen egy komponens alapú könyvtárról van szó. A teljes felületet React Native komponensek és ezeket kiterjesztő osztályok alkotják, a keretrendszer pedig gondoskodik arról, hogy az adatváltozásoknak megfelelően gyorsan frissítse a felület megfelelő részeit. A beépített elemek mellett, amilyenek a gombok vagy szövegmezők, saját komponensek is létrehozhatóak. A komponensek stíluslapok segítségével testre is szabhatóak, megváltoztatható a méretük, kinézetük és pozíciójuk.

A UI/UX elveknek [12] megfelelendő, a projekt esetén különös fontossággal bíró szempont egy felhasználóbarát alkalmazás felépítése, mely könnyen használható és konzisztenciát mutat a rendszeresen használt termékekkel. Ez megkönnyíti a felhasználó számára az új termék megismerését további tanulási költségek nélkül. Ezen elvek követését támogatja a 3.1.3. fejezetben ismertetett Native Base, hiszen ez a keretrendszer a már megszokott Android vagy iOS elemekhez hasonló komponenseket biztosít, amint ez az 5. ábrán is megfigyelhető. Annak érdekében, hogy a lehető legkevesebb interakciót igényelje az alkalmazás a sofőr részéről, ne terelje el figyelmét az úttestről, de segítse a navigációban, különösen fontos az is, hogy könnyen kezelhető legyen a felület. A kevés néha több tervezési elvet sem mellőzve, fontos tényezők a letisztultság, használhatóság és következetesség, ezáltal is csökkentvén a felhasználók kognitív költségeit.



5. ábra. A Native Base megszokott Android (bal) és iOS (jobb) komponenseket biztosít.

4. Eszközök és munkamódszerek

Ebben a fejezetben a fejlesztés során alkalmazott munkamódszer kerül bemutatásra, illetve azok a felhasznált eszközök, amelyek felgyorsították és hatékonyabbá tették ezt a folyamatot.

Verziókövetés

A projekt fejlesztése során a verziókövetés Git [18] segítségével történik. Ez egy osztott verziókövető rendszer, amely lehetőséget ad a változtatások számontartására és az egyidejű fejlesztésre. A projekt fő komponensei (szerver, mobil kliens) külön repositorykban vannak tárolva.

A GitLab [19] egy repository menedzsment rendszer, amely a projekt teljes életciklusára megfelelő fejlesztői eszközöket kínál. A projekt fejlesztése során ennek segítségével van megvalósítva a Git alapú tárolók kezelése, a forráskódok felülvizsgálata, a dokumentációk létrehozása és tárolása, a kitűzött feladatok kezelése, illetve a folyamatos integráció és folyamatos kitelepítés.

A GitLab úgynevezett Wiki oldalak létrehozására ad lehetőséget, melyek dokumentációk tárolóhelyeként szolgálhatnak. A projekt esetében itt találhatóak a leszögezett kódolási konvenciók, a projekttel szemben felállított követelmények, a *merge request*ek folyamatára vonatkozó megállapodások, illetve az architektúra és komponens diagrammok.

Projekt menedzsment

A fejlesztés során alkalmazott munkamódszer a Scrum [33], egy elterjedt agilis módszer, mely inkrementális és iteratív fejlesztést javasol. A fejlesztést rövidebb szakaszokra, úgynevezett sprintekre bontja és minden sprint végére az alkalmazás egy működő verziója kell elkészüljön.

A projekt esetén megvalósítandó feladatok kéthetes sprintekbe vannak beosztva. A szakaszok kezdetekor sprinttervező megbeszélés zajlik le, a sprint végén pedig áttekintésre kerül sor, melyet demó és visszatekintés követ.

Annak érdekében, hogy a fejlesztés alatt szimulálni lehessen a valós adatokat, az adatbázis demó adatokkal van feltöltve. Ezeket a fiktív parkolóhelyeket és térfigyelő kamerákat használja az alkalmazásszerver, és ezek kerülnek megjelenítésre a mobilalkalmazásban is.

A feladatok menedzseléséhez a Gitlab biztosít felületet, az úgynevezett kanban boardot [13]. A Kanban egy hatékony agilis szoftvermenedzsment módszer, mely jól párosítható a projekt esetén alkalmazott Scrummal [26]. Ezen módszer egy vizuális felületet ajánl a feladatok nyilvántartására, prioritizálására és követésére. A felület több oszlopra van felosztva, melyek a

fejlesztés egyes fázisait jelölik, az oszlopokban levő kártyák pedig a különböző feladatokat jelképezik.

Az átláthatóság érdekében a változtatások beépítése a kódbázisba a GitFlow [7] elágazási stratégiát követve történik. Ennek értelmében két állandó ág létezik tárolónként, ezek a *master* és a *develop*. A master ág mindig stabil, kitelepítésre kész kódot tárol, míg a develop a fejlesztés során használt központi ág. Minden új funkcionalitás bevezetése, módosítás és hibajavítás külön ágon történik, melynek típusát az ág nevében levő prefix (pl. feature, bug stb.) jelzi. Ezek az ágak a developból ágaznak el, és ugyanide épülnek vissza a változtatások végeztével. Mielőtt azonban ez a visszacsatolás megtörténne, automatikus és kézi kódellenőrzésre is sor kerül.

A változtatások átnézéséhez és a fejlesztési ág visszacsatolásához a GitLab merge request felülete nyújt megfelelő segítséget. Szerver és mobil oldalon is saját sablon alapján történik a merge requestek kitöltése, ami egységessé és könnyedén átláthatóvá teszi ezeket. Az említett felület lehetővé teszi a változtatások felülvizsgálatát és megjegyzések hozzáfűzését az egyes sorokhoz. Az Atlas projekt fejlesztése során különös figyelemben részesül a tiszta kódolási elvek (Clean Code Principles) [32] betartása, ami szintén ezen a felületen kerül ellenőrzés alá. A kód akkor tiszta, ha érthető, ami azt is jelenti, hogy könnyen olvasható, változtatható, kiterjeszthető és karbantartható.

Kódelemzés

A kódformázást a projekt esetében a Prettier [22] oldja meg, amely egy következetes és egységes stílust érvényesítő eszköz. A megfelelő *pre-commit hook* beállításával minden egyes commit előtt megtörténik a kód elemzése, majd szükség esetén a kód felülírása. Időt és energiát lehet nyerni vele azáltal, hogy mentés esetén automatikus kódformatálást végez, amennyiben az ehhez szükséges pluginok és fejlesztői környezet kiterjesztések engedélyezve vannak.

Az ESLint [15] egy nyílt forráskódú kódelemző eszköz, amely statikusan elemzi a JavaScript alapú fájlokat, így gyorsan megtalálja a problémákat. Az ESLint a kódminőséggel és a kódolási stílus kérdéseivel egyaránt foglalkozik, illetve támogatja az ECMAScript jelenlegi szabványait. Ugyanakkor az ESLint szabályai testre is szabhatóak, így ezek megfeleltethetőek a projekt kódolási konvencióinak. Konfigurációjához a Prettier plugin is használható [9]. A hibákat vagy az egységes kódolási stílusnak nem megfelelő kódrészleteket futtatás nélkül felfedezi és jelzi. A szoftverrendszer esetében használatba van ültetve úgy szerver-, mint kliensoldalon, és része a folyamatos integrációnak is.

Folyamatos integráció és kitelepítés

A fejlesztési folyamat fontos részét képezi a folyamatos integráció (Continous Integration, CI), amely biztosítja a kódbeli változtatások helyes integrációját a már meglévő rendszerbe. A GitLab CI/CD [20] hivatott ezt a célt szolgálni jelen projekt esetében, mely a *gitlab-ci.yml* fájl segítségével konfigurálható. Mobil és szerver oldalon egyaránt lefut minden egyes változtatás esetén egy kódellenőrzés. A szerver esetén a master és develop ágakra lefut a *build* fázis is, melyet kitelepítés követ.

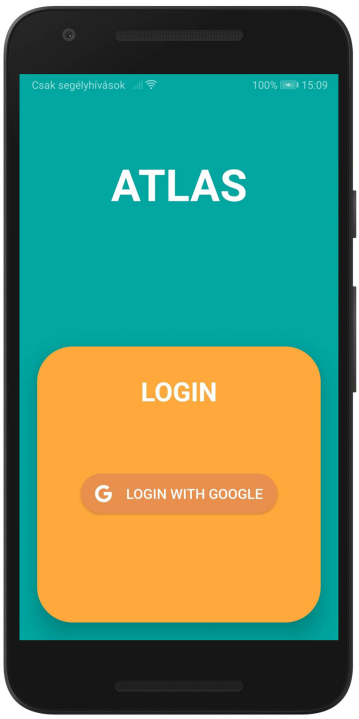
A Continous Delivery (CD) a kitelepítés automatizálását foglalja magába. Ez a folyamat is a master és develop ágak esetén fut le. Ilyenkor egy újabb docker image jön létre, mely bekerül a GitLab Container Registrybe. Az alkalmazásszerver legújabb kitelepített verziója ezt követően az *edu.codespring.ro* megfelelő szerverén érhető el. A kitelepített konténer elindítását a Docker Compose végzi, mely egyszerre több mikroszervízt indít el (a projekt esetén ezek az Atlas backend, az OSRM backend és a MongoDB).

5. Az alkalmazás működése

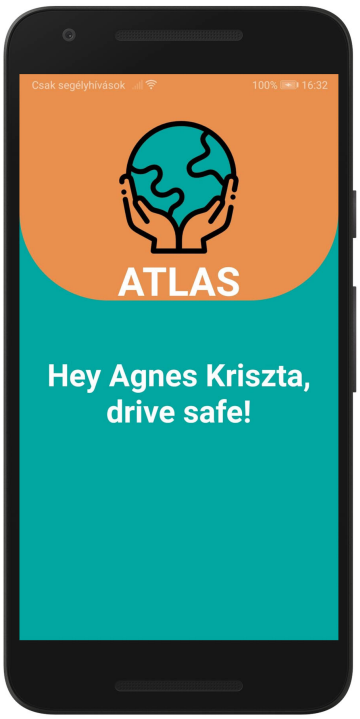
A mobilalkalmazás megnyitásakor a 6. ábrán látható bejelentkezésre alkalmas oldal fogadja a felhasználót. Ezen az oldalon egy Google-lel való bejelentkezést biztosító gomb van elhelyezve. A gomb érintésére felugrik egy ablak, melyen a felhasználó kiválaszthatja a bejelentkezéshez használandó Google fiókját. Első bejelentkezés esetén az alkalmazás engedélyt kér, hogy felhasználhassa a Google fiókhoz tartozó nevet, e-mail címet és profilképet. Ezeket elfogadván, sikeres bejelentkezés esetén a 7. ábra szerinti oldal üdvözli a felhasználót egy névre szabott üzenettel. Sikertelen bejelentkezést internetkapcsolathiány vagy szerver oldali problémák okozhatnak, ez esetben a képernyőn hibaüzenet jelenik meg.

Az alkalmazás bizonyos funkcióinak eléréséhez szükség van a készülék földrajzi helyzetéhez való hozzáféréshez. Ennek érdekében, a bejelentkezést követően egy felugró ablak kéri a helyzethez való hozzáférés jogosultságának megadását. Amennyiben ezt elfogadja a felhasználó, a későbbi bejelentkezések esetén már nem fog megjelenni. Ha a készüléken nincs bekapcsolva a helymeghatározás (Global Positioning System, azaz GPS), akkor az alkalmazás javasolni fogja ennek a bekapcsolását szintén egy felugró ablak segítségével.

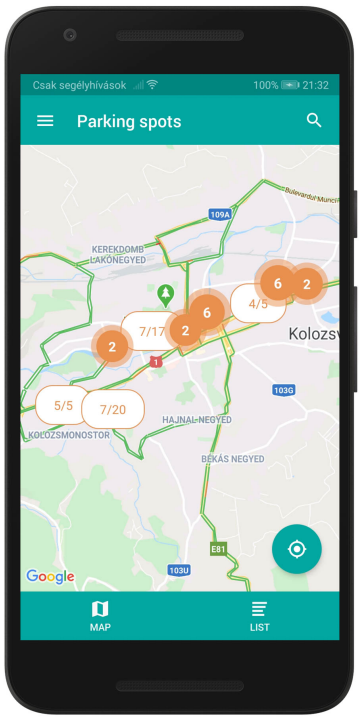
A bejelentkezett felhasználót egy térképes nézet fogadja, ez az elsődleges formája a parkolóhelyek megjelenítésének. Amint az a 8. ábrán is megfigyelhető, a parkolóhelyek pozíciójuknak megfelelően jelennek meg a térképen. A térképen az utak az aktuális forgalom függvényében vannak színezve, így a felhasználó eldöntheti, hogy merre érdemesebb



6. ábra. Bejelentkezésre alkalmas képernyő



7. ábra. Sikeres bejelentkezést követő oldal

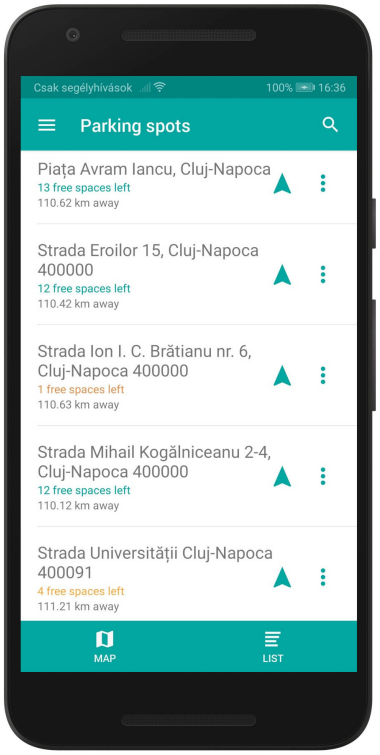


8. ábra. Térképes nézet a parkolóhelyekről

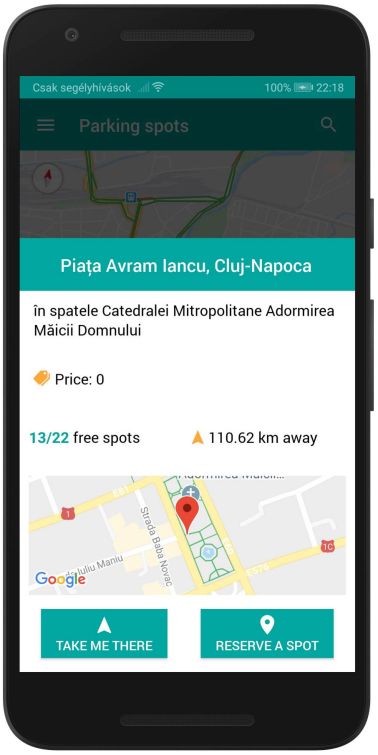
közlekednie. Annak érdekében, hogy ne legyen túlságosan zsúfolt ez a nézet, a parkolóhelyek klaszterezve vannak a térképen. Mindez dinamikusan történik, így a térképen kicsinyítés esetén egyre több parkolóhely csoportosul, míg közelítéskor a csoportosított markerek kisebb egységekké bomlanak. Klaszterre való kattintás esetén a térkép automatikus ráközelítést végez. Ha olyan mértékű a közelítés, hogy a parkolóhelyek kiválnak csoportjukból, egy sajátságos pozíciót jelölő marker közli a felhasználóval, hogy az adott parkoló esetén mennyi az összkapacitás, és ebből hány hely szabad. A térkép jobb alsó sarkában található gomb a felhasználó pozíciójához animálja a térképet, amennyiben be van kapcsolva az eszközön a helymeghatározás.

A képernyő alján található navigációs sáv (Bottom Navigation Bar) a térképes és a lista nézet közötti navigálás lehetőségét kínálja. Ha a felhasználó részletesebb információkat szeretne kapni a térképen megjelenő régió parkolóhelyeiről, a jobb oldali lista fülre kattintva válthat a 9. ábra szerinti nézetre. A listaelemek tartalmazzák az adott parkoló nevét, a szabad helyek számát, valamint a parkolóhely felhasználótól való távolságát. Az elérhető helyek száma színezve van annak függvényében, hogy hány darab van még belőlük: a türkizzöld a legbíztatóbb szín, a narancssárga esetén már viszonylag kevés szabad parkolóhely van, a sötét piros pedig a már közel megtelt parkolót jelöli.

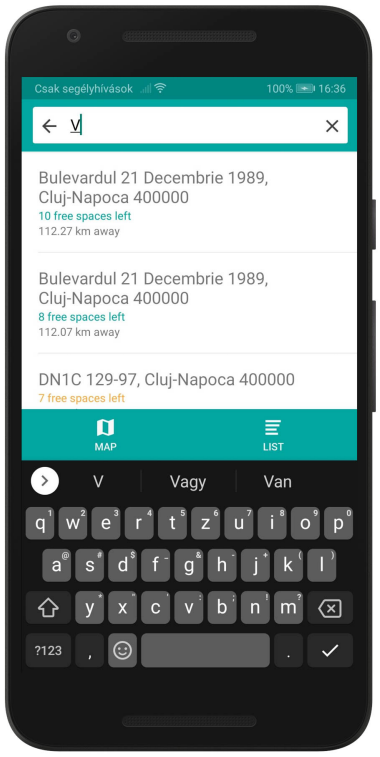
Egy kiszemelt parkolóról minden információ megjeleníthető a parkolóhelyre való



9. ábra. Lista nézet részletesebb információkkal



10. ábra. Részletes leírása egy kiválasztott parkolóhelynek

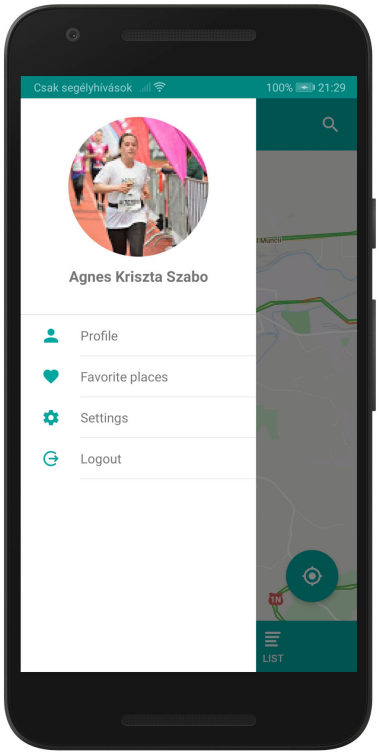


11. ábra. Név vagy cím szerinti keresés a parkolók között

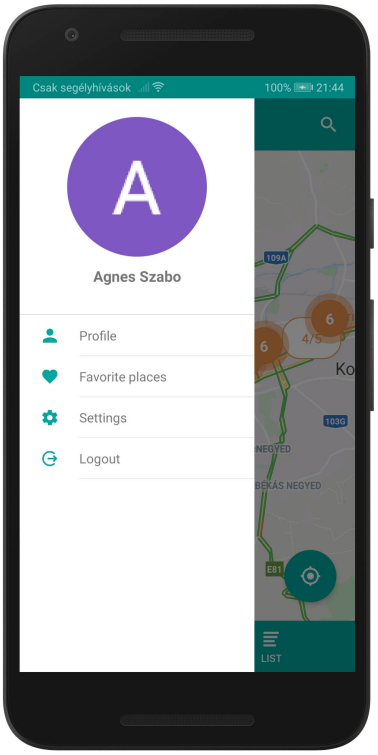
kattintással. Ez ugyanazt a képernyő aljából felemelkedő ablakot eredményezi térképes nézet esetén, mint listanézetben. Amint az a 10. ábrán is látható, ez az előtérbe kerülő ablak tartalmazza az adott parkoló nevét, pontos címét, a parkolás árát, esetlegesen az értékelését, egy kis térképet a pontos pozícióról, illetve két gombot. Ezek a gombok még nincsenek beüzemelve, ám az első feladata az lenne, hogy a felhasználót átirányítsa a Google térkép navigációjába, amely elvezetné őt a kiválasztott parkolóhoz, míg a másodikkal lehetne foglalást végezni.

Az alkalmazás fejléce két kis ikont tartalmaz. A jobb oldalon található lehetővé teszi parkolóhelyek név vagy cím szerinti keresését, szűrését. A szűrési találatok eredménye gépelés közben dinamikusan változik és a lista nézethez hasonló módon jelennek meg a találatok (lásd: 11. ábra). Ha térkép nézetben szeretne keresést végrehajtani a felhasználó, akkor az eredménylista egyik elemére kattintva a térkép úgy mozdul el, hogy a kiválasztott parkolóhely kerüljön a középpontba. Lista nézetben, a már megszokott módon, a részletes információkat tartalmazó ablak emelkedik fel (lásd: 10. ábra).

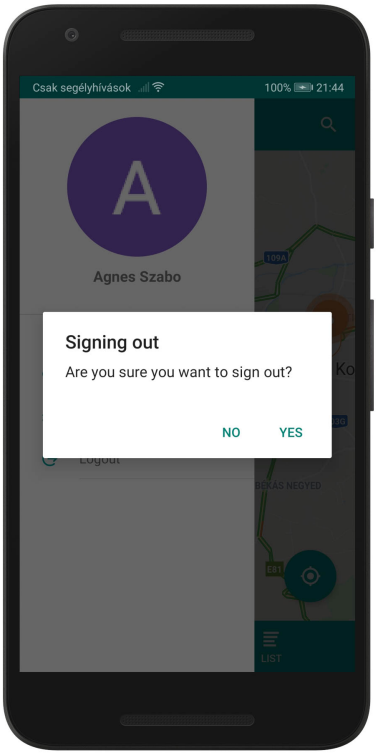
A fejléc bal oldalán található hamburger ikon egy oldalsó menüt jelenít meg, ez látható a 12. ábrán. A menü a Google fiókos profilképet tartalmazza, megjeleníti a felhasználó nevét, valamint különböző menüpontokat. Abban az esetben, ha a felhasználó nem állított be profilképet a fiókjához, nevének kezdőbetűiből generálódik egy helyettesítő kép (13. ábra). A menüben megjelenő *Logout*ra kattintva lehet kijelentkezni, mielőtt azonban ez megtörténne, az alkalmazás megerősítést vár erről a döntésről, ez látható a 14. ábrán.



12. ábra. Oldalsó menü Google profilképpel



13. ábra. Névből generált profilkép



14. ábra. Kijelentkezéskor az alkalmazás megerősítést vár

Következtetések és továbbfejlesztési lehetőségek

A jelen dolgozatban bemutatott Atlas projekt fejlesztése során sikerült megvalósítani egy olyan szoftverrendszert, mely a parkólóhelyek keresésének folyamatát hivatott leegyszerűsíteni. A rendszerhez tartozó központi szerver létesít kapcsolatot a többi komponenssel, és egy API-t biztosít kliensei számára. A telefonos alkalmazás ezen API-t felhasználva jeleníti meg az elérhető parkolóhelyeket térképen vagy lista nézetben, valamint a parkolóhelyek lefoglalására is lehetőséget kínál.

A szoftverrendszer fejlesztése során több továbbfejlesztési lehetőség is megfogalmazódott.

A későbbi fejlesztések főbb prioritásai közé tartozik a kiválasztott parkolóhelyhez való navigáció megvalósítása akár egy külső térképes alkalmazás, például a Google Maps útvonaltervező nézetének megnyitásával. Emellett hasznos lenne, ha az alkalmazás értesítést küldene a felhasználónak amennyiben a kiválasztott parkolóhely foglalása már nem elérhető.

Az oldalsó menü jelenleg inaktív gombjainak (*Profile*, *Favorite places*, *Settings*) beüzemelése is nagy fontossággal bírna. Létre lehetne hozni egy nézetet, ahol a felhasználó megtekintheti és szerkesztheti saját adatait. Akár járművek adatait is fel lehetne itt vezetni, ezáltal az alkalmazás a jármű méretének és az elérhető hely méretének függvényében ajánlhatna parkolóhelyeket. A kedvenc, gyakran felkeresett helyek mentésére is lehetőséget lehetne adni, ezáltal a felhasználó könnyedén kereshetne és foglalhatna parkolóhelyet a mentett hely közelében.

Egy web kliens létrehozása is hasznos bővítés lenne, ahol az adminisztrátorok könnyedén karbantarthatnák a rendszerhez tartozó térfigyelő kamerákat és az általuk megfigyelt helyeket.

A továbbfejlesztési tervek közé tartozik ugyanakkor a Facebookos bejelentkezés lehetőségének megadása, valamint egy önálló regisztrációs rendszer létrehozása is.

A későbbiekben célszerű lenne a mozgássérültek számára alkalmas parkolóhelyeket külön jelölni és megjeleníteni.

Mindemellett praktikus fejlesztés lenne az alkalmazás nappali és éjjeli témájának beüzemelése, így sötétben sem zavarná a sofőröket az alkalmazás által kibocsátott fény.

Végül, de nem utolsósorban egy visszajelzésre alkalmas űrlap megvalósítása is hasznos fejlesztés lenne, melynek segítségével a felhasználók bejelentést tehetnének, amennyiben az alkalmazás által jelzett szabad helyek száma nem felel meg a valóságnak, akár a számítógépes látás modul tévedése, akár a kamera meghibásodása miatt.

Hivatkozások

[1] Anton Begehr. *Cluster react-native-maps Markers with react-native-map-clustering*. 2019. URL: https://medium.com/@begehr/cluster-react-native-maps-markers-with-react-native-map-clustering-72f50db26891 (elérés dátuma: 2020. máj. 3.)

[2] Emil Boc. „Green Mobility and Quality of Life: Cluj-Napoca Case Study". *Sustainable Development and Resilience of Local Communities and Public Sector Organizations* (2018), 79. oldal.

[3] Howard Butler és mások. „The geojson format". *Internet Engineering Task Force (IETF)* (2016).

[4] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013, 389. oldal.

[5] *Cluj-Napoca: Aplicația yeParking e un park-sharing. Poate pune capăt ocupării abuzive a parcărilor plătite?* 2018. URL: https://www.stiridecluj.ro/social/cluj-napoca-aplicatia-yeparking-e-un-park-sharing-poate-pune-capat-ocuparii-abuzive-a-parcarilor-platite (elérés dátuma: 2020. ápr. 26.)

[6] *Digitális Kaláka: magunk előtt láttuk a jövőt*. 2018. URL: https://szka.org/2018/06/digitalis-kalaka-magunk-elott-lattuk-a-jovot/ (elérés dátuma: 2020. ápr. 27.)

[7] Vincent Driessen. „A successful Git branching model". *URL http://nvie.com/posts/a-successful-git-branching-model* (2010).

[8] Bonnie Eisenman. *Learning react native: Building native mobile apps with JavaScript*. " O'Reilly Media, Inc.", 2015.

[9] Eric Elliott. *Outside the box*. 2019. URL: https://medium.com/javascript-scene/streamline-code-reviews-with-eslint-prettier-6fb817a6b51d (elérés dátuma: 2020. máj. 3.)

[10] Roy T Fielding és Richard N Taylor. *Architectural styles and the design of network-based software architectures*. 7. kötet. University of California, Irvine Irvine, 2000.

[11] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] Jeff Gothelf. *Lean UX: Applying lean principles to improve user experience*. " O'Reilly Media, Inc.", 2013.

[13] Marcus Hammarberg és Joakim Sunden. *Kanban in action*. Manning Publications Co., 2014.

[14] *Hivatalos Babel dokumentáció*. URL: https://babeljs.io/ (elérés dátuma: 2020. ápr. 21.)

[15] *Hivatalos ESLint dokumentáció*. URL: https://eslint.org/docs/about/ (elérés dátuma: 2020. ápr. 20.)

[16] *Hivatalos Expo dokumentáció*. URL: https://expo.io/ (elérés dátuma: 2020. ápr. 10.)

[17] *Hivatalos Firebase dokumentáció*. URL: https://firebase.google.com/docs/guides (elérés dátuma: 2020. ápr. 20.)

[18] *Hivatalos Git dokumentáció*. URL: https://git-scm.com/doc (elérés dátuma: 2020. ápr. 20.)

[19] *Hivatalos Gitlab dokumentáció*. URL: https://about.gitlab.com/whatis-gitlab/ (elérés dátuma: 2020. ápr. 21.)

[20] *Hivatalos Gitlab dokumentáció: Continuous Integration and Delivery*. URL: https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/ (elérés dátuma: 2020. ápr. 20.)

[21] *Hivatalos Native Base dokumentáció*. URL: https://docs.nativebase.io/ (elérés dátuma: 2020. ápr. 21.)

[22] *Hivatalos Prettier dokumentáció*. URL: https://prettier.io/docs/en/index.html (elérés dátuma: 2020. ápr. 20.)

[23] Simon Holmes. *Mongoose for Application Development*. Packt Publishing Ltd, 2013.

[24] Stephan Huber és Christoph Rust. „Calculate travel time and distance with OpenStreetMap data using the Open Source Routing Machine (OSRM)". *The Stata Journal* 16.2 (2016), 416–423. oldal.

[25] International Parking Institute. „2012 Emerging Trends in Parking". *URL https://www.parking.org/wp-content/uploads/2015/12/Emerging-Trends-2012.pdf* (2012).

[26] Henrik Kniberg és Mattias Skarin. *Kanban and Scrum-making the most of both*. Lulu. com, 2010.

[27] Sudhir Kumar. *Managing API Requests in React Native Using Axios*. 2019. URL: https://medium.com/better-programming/managing-api-requests-http-https-in-react-native-using-axios-9ebf75cbca9b (elérés dátuma: 2020. ápr. 30.)

[28] Dirk Merkel. „Docker: lightweight linux containers for consistent development and deployment". *Linux journal* 2014.239 (2014), 2. oldal.

[29] *Outside the box*. URL: https://www.bosch.com/stories/artificial-intelligence-community-based-parking/ (elérés dátuma: 2020. máj. 3.)

[30] *Parking Lot Sensor, The wireless smart parking sensor for detecting parking space occupancy*. URL: https://www.bosch-connectivity.com/products/connected-mobility/parking-lot-sensor/ (elérés dátuma: 2020. máj. 3.)

[31] Pavan Podila és Michel Weststrate. *MobX Quick Start Guide: Supercharge the client state in your React apps with MobX*. Packt Publishing Ltd, 2018.

[32] Rafael Salerno. *Clean Code Summary and Key Points*. 2016. URL: https://dzone.com/articles/clean-code-summary-and-key-points (elérés dátuma: 2020. ápr. 27.)

[33] Ken Schwaber és Mike Beedle. *Agile software development with Scrum*. 1. kötet. Prentice Hall Upper Saddle River, 2002.

[34] Siemens. „Case studies for traffic solutions”. *URL https://www.academia.edu/5300681/Case_studies_for_traffic_solutions_Modern_concepts_and_technologies_help_improve_efficiency* (2011).

[35] Pedro Teixeira. *Professional Node. js: Building Javascript based scalable software*. John Wiley & Sons, 2012, 9–13. oldal.

[36] *The Decorator pattern*. URL: https://www.dofactory.com/javascript/decorator-design-pattern (elérés dátuma: 2020. ápr. 27.)

[37] Stefan Tilkov és Steve Vinoski. „Node. js: Using JavaScript to build high-performance network programs”. *IEEE Internet Computing* 14.6 (2010), 80–83. oldal.

[38] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014, 255. oldal.

[39] *Using OAuth 2.0 to Access Google APIs*. URL: https://developers.google.com/identity/protocols/oauth2 (elérés dátuma: 2020. ápr. 27.)

[40] Bob Violino. *How to choose the right NoSQL database*. 2018. URL: https://www.infoworld.com/article/3260184/how-to-choose-the-right-nosql-database.html (elérés dátuma: 2020. ápr. 30.)

[41] Ronghai Yang, Wing Cheong Lau és Shangcheng Shi. „Breaking and fixing mobile app authentication with OAuth2. 0-based protocols”. *International Conference on Applied Cryptography and Network Security*. Springer. 2017, 313–335. oldal.