ETDK-dolgozat

Hurubás Dávid

Szántó Ágnes-Bernadett

XXIV. reál- és humántudományi Erdélyi Tudományos Diákköri Konferencia (ETDK)

Kolozsvár, 2021. május 27–30.

Trucker: Logisztikai cégek belső kommunikációját elősegítő alkalmazás

Szerzők:

Hurubás Dávid

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Szántó Ágnes-Bernadett

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Témavezetők:

dr. Sulyok Csaba, egyetemi adjunktus

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Gagyi Mátyás, szoftverfejlesztő,

Codespring

Hegedüs Hunor, szoftverfejlesztő,

Codespring

Ráduly Sándor, szoftverfejlesztő,

Codespring

CODESPRING

OUR CODE YOUR SUCCESS

Kivonat

A Trucker projekt fő célja, hogy megkönnyítse a logisztikai cégeken belüli kommunikációt a sofőrök, illetve a szállításvezetők és/vagy fuvarszervezők között, valamint az, hogy segítse a szállítással kapcsolatos adatok átlátható és rendszerezett tárolását, kezelését, megjelenítését.

Két fő szerepkör különböztethető meg: sofőrök és a szállításvezetők. A sofőrök megtekinthetik a rájuk váró, vagy már elvégzett feladatokat és azok információit, szükség esetén módosíthatják a megállókat az utak során, illetve értesítéseket küldhetnek a szállításvezetőnek különböző eseményekről (pl. közúti ellenőrzés, defekt stb.). A szállításvezetők bevezethetik a rendszerbe az új utak információit, kioszthatják a feladatokat a sofőrök között, illetve követhetik, hogy a sofőrök az út melyik szakaszában vannak.

A sofőrök számára egy platformfüggetlen mobilalkalmazás, míg a szállítésvezetőknek egy webes felület áll rendelkezésükre. A két kliensalkalmazást egy REST API szerver köti össze, amely egy dokumentum alapú adatbázist, illetve in-memory cache-t használ a különböző adatok tárolására.

Tartalomjegyzék

Bevezető **1**

1. A Trucker projekt **3**

1.1. Szerepkörök és funkcionalitások 3

1.2. Architektúra 4

2. A szerver **6**

2.1. Adatmodell 6

2.2. Adathozzáférési réteg 7

2.3. Szolgáltatási réteg 8

2.4. API 8

2.5. Biztonság 9

3. A web kliens **13**

3.1. Felhasználói felület megvalósítása 13

3.2. Kommunikáció a szerverrel 14

3.3. Értesítések kezelése 15

4. A mobil kliens **17**

4.1. Felhasználói felület megvalósítása 17

4.2. Kommunikáció a szerverrel 18

4.3. Háttérfolyamatok és értesítések 19

5. Felhasznált technológiák és eszközök **21**

5.1. Szerveroldali technológiák 21

5.2. Web technológiák 22

5.3. Mobil technológiák 23

5.4. Egyéb eszközök 24

6. A kliensalkalmazások működése **25**

6.1. A mobilalkalmazás használata 25

6.2. A webalkalmazás használata 27

Következtetések és továbbfejlesztési lehetőségek **33**

Bevezető

Az áruszállító, illetve logisztikai cégekben, ahol nagy mennyiségű árut mozgatnak meg nap, mint nap országos vagy nemzetközi szinten, fontos, hogy az útvonalak optimálisan legyenek megtervezve, illetve a szállítmányok időben megérkezzenek egy adott helyszínre. Ezeknek az útvonalaknak a megtervezését a szállításvezetők hivatottak elvégezni, általában valamilyen szoftverrendszer segítségével. Ugyanakkor fontos szempont a szállításvezetők és sofőrök közti hatékony kommunikáció a szállítás lebonyolítása során.

Egyes vállalatok saját használatra tervezett és optimalizált alkalmazásokat használnak, így azok a cégek, amelyek nem rendelkeznek ilyen rendszerekkel, hátrányt szenvedhetnek. Ezen vállalkozásokban gyakran megjelenő probléma, hogy a szállításokkal kapcsolatos adatok tárolására és nyilvántartására más alternatívákat használnak (például szöveges állományok), amelyek kezelése nehézkessé válhat. Egy másik probléma a sofőrökkel való kommunikáció megvalósítása, amely jobb esetben a vállalat által biztosított mobil készülékeken történő hívásokkal vagy SMS-ekkel zajlik.

Hasonló alkalmazások ugyan léteznek már, de nem teljesen nyújtanak megoldást a fentebb említett problémákra. Többek között ilyenek a következő alkalmazások: Sygic[1], PTV Navigator[2], inRoute[3]. Mindhárom applikáció térképes navigációt tesz lehetővé a sofőrök számára. Lehetőséget adnak útvonalak tervezésére, viszont ezt a sofőrnek kell megtennie. Annak ellenére, hogy több különböző funkcionalitással rendelkeznek (offline/hang-alapú navigáció, vagy útviszonyok előrejelzése), nem biztosítanak külön olyan felületet, ahol a szállításvezetők megtervezhetnék az útvonalakat és kapcsolatot tarthatnának a sofőrökkel. Egy másik összehasonlítási szempont az alkalmazások jelenléte különböző platformokon, mivel az egyes alkalmazások nem elérhetőek mindenhol (például az inRoute alkalmazás csak iOS felhasználók számára elérhető).

A dolgozatban bemutatott Trucker alkalmazásrendszer ezekre a problémákra kínál megoldást: a szállításvezetőknek lehetőséget biztosít az útvonalak megtervezésére egy interaktív webes alkalmazás által, valamint a sofőröknek egy telefonos alkalmazás áll a rendelkezésükre, amely segítségével könnyedén tudják követni a hozzájuk rendelt feladatokat. Az alkalmazásrendszer a két munkakörben dolgozók közti kommunikációt is hivatott

[1]Forrás: https://www.sygic.com/truck, utolsó megtekintés dátuma: 2021-05-02

[2]Forrás: https://www.ptvgroup.com/en/solutions/products/ptv-navigator, utolsó megtekintés dátuma: 2021-05-02

[3]Forrás: https://carobapps.com/products/inroute, utolsó megtekintés dátuma: 2021-05-02

elősegíteni értesítések formájában, így a telefonhívások és SMS-ek helyettesíthetőek az alkalmazás által kínált lehetőségekkel. A telefonos alkalmazás mind Android, mind iOS felhasználók számára elérhető, ezzel is elősegítve az alkalmazás platformfüggetlenségét.

A dolgozat hátralevő része a következőképpen van strukturálva: az 1. fejezet bemutatja a felhasználói szerepköröket, illetve a szoftverrendszer architektúráját. A 2. fejezetben a szerver, a 3. fejezetben a web, illetve a 4. fejezetben a mobilalkalmazás felépítése kerül bemutatásra. Az 5. fejezetben a felhasznált technológiákról és eszközökről esik szó, majd a 6. fejezet részletesen tárgyalja a kliensalkalmazások használatát. A záró fejezetben a levont következtetések, illetve továbbfejlesztési lehetőségek vannak megfogalmazva.

A projekt a U-Hub Mentorprogram[4] keretein belül indult 2020 júliusában a Codespring[5] cég szakmai támogatásával. A csoportos projekt nevű egyetemi tantárgy keretein belül csatlakozott a csapathoz Fazekas Hunor, Galiger Gergő, Guti Zsombor, valamint Tamás Eszter-Viola. Köszönet illeti mentorainkat: Gagyi Mátyás, Hegedüs Hunor és Ráduly Sándor szoftverfejlesztőket, valamint témavezető tanárunkat dr. Sulyok Csaba egyetemi adjunktust, illetve diáktársainkat is.

[4] https://u-hub.ro/mentorprogram
[5] https://edu.codespring.ro

1. A Trucker projekt

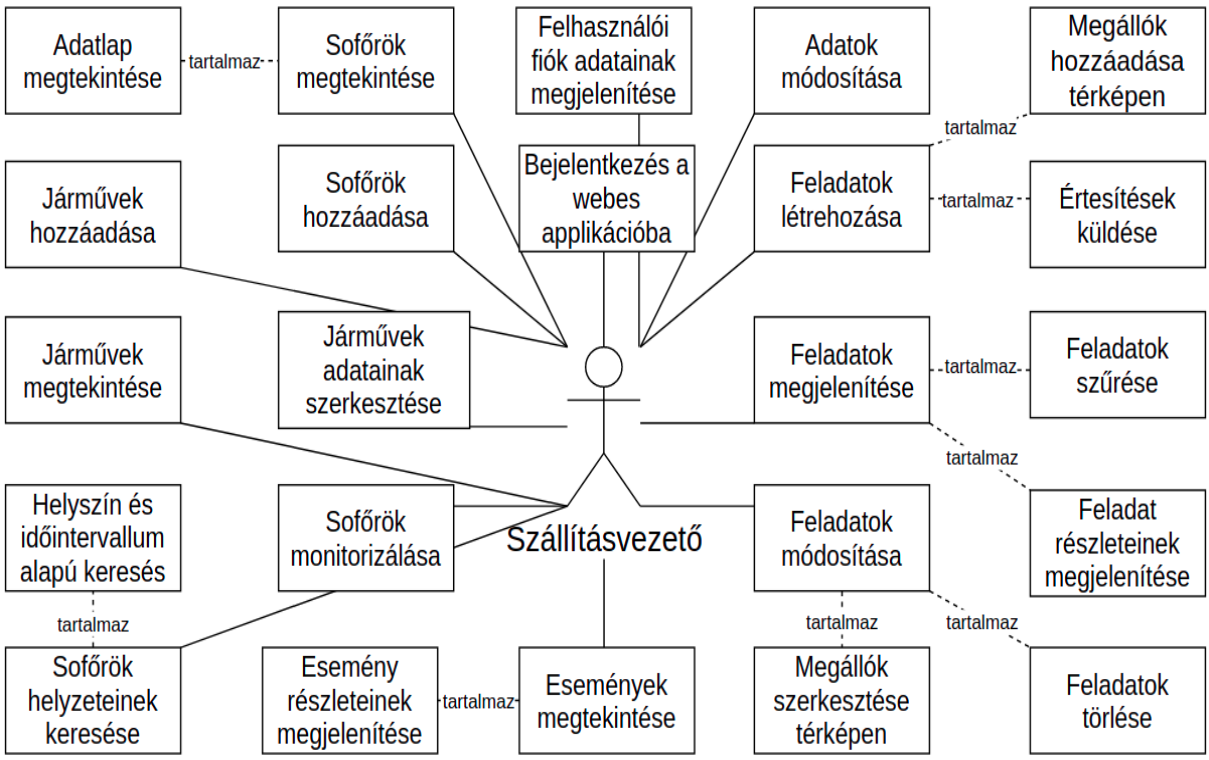
A fejezet bemutatja az alkalmazásban megjelenő két szerepkört, a szállításvezetőt és sofőrt, továbbá részletezi a projekt architektúráját, és annak főbb alkotó részeit.

1.1. Szerepkörök és funkcionalitások

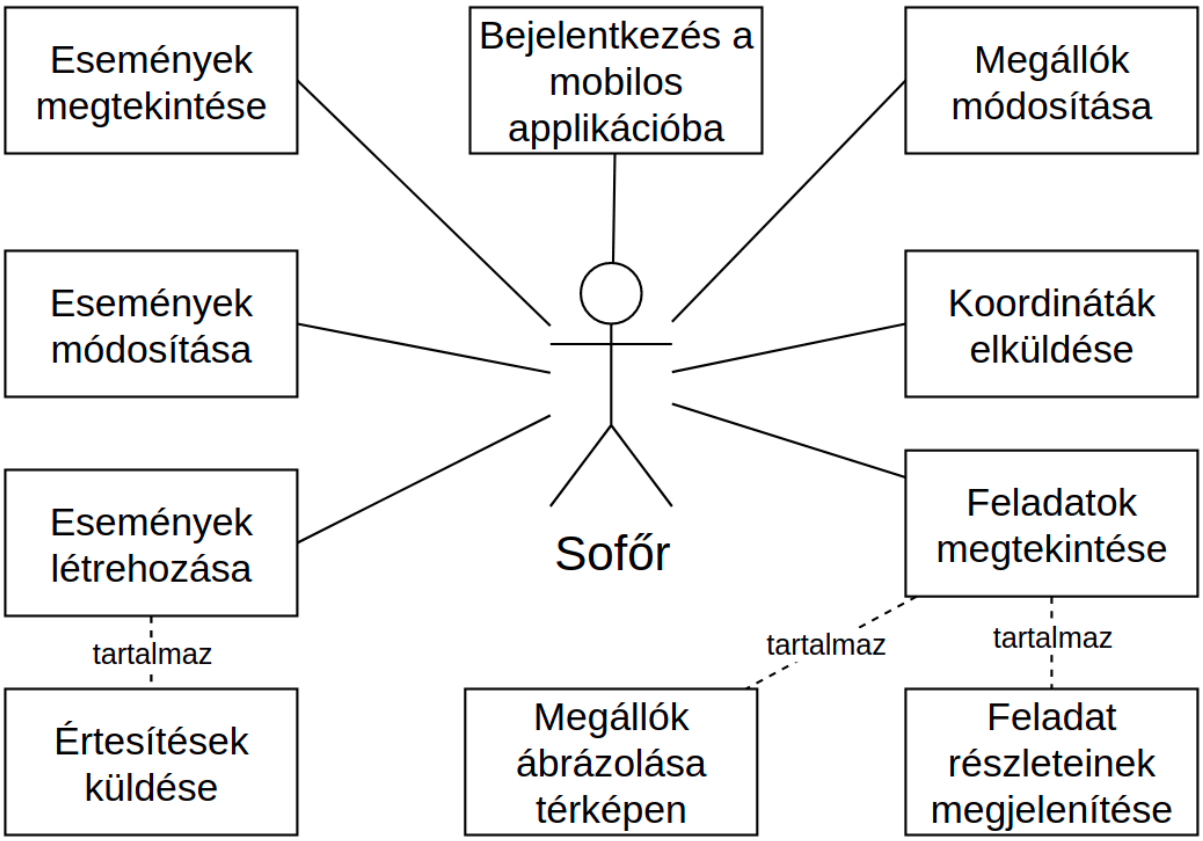
A Trucker szoftverrendszerének szerves részét képezik a sofőrök és a szállításvezetők. Mind a két szerepkör jelenléte szükséges az alkalmazáson belül, mivel a munkájuk is összefűződik. A továbbiakban az általuk elérhető funkcionalitások kerülnek bemutatásra.

Szállításvezetők

A szállításvezetők számára a 1. ábrán bemutatott funkcionalitások a webes alkalmazásba való bejelentkezés után lesznek elérhetőek. A szállításvezetők feladata, hogy megtervezzék a szállítások útját és idejét, valamint a megállók helyszíneit. Létrehozásukkal a sofőrök értesítést kapnak, ha új feladatot rendeltek hozzájuk. Lehetőségük van új sofőröket és járműveket hozzáadni a rendszerhez. A sofőrök útjának monitorizálását egy élő térképnézet, valamint szűrők segítségével tehetik meg.



1. ábra. A szállításvezető funkcionalitásai



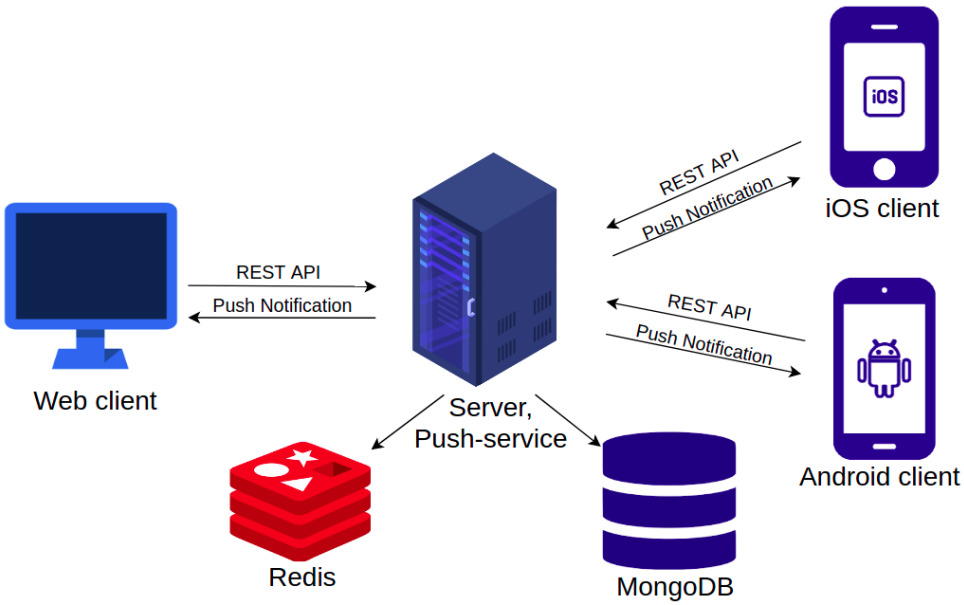
2. ábra. A sofőr funkcionalitásai

Sofőrök

A sofőrök számára a mobilosalkalmazás áll rendelkezésre, amelybe bejelentkezve a 2. ábrán látható funkcionalitások érhetőek el. Ők felelnek az áruk elszállításáért, ezáltal teljesítik a hozzájuk rendelt feladatokat. Az alkalmazás lehetőséget nyújt a feladatok részletes megtekintésére, a megállók térkép nézeten is megjeleníthetők, ezzel is segítve a felhasználót a navigációban. Ha valamilyen gond adódik szállítás során (pl. baleset, ellenőrzés, szervíz) a sofőr értesítéseket küldhet a szállításvezetőnek, így azok időben tudni fognak az esetleges problémákról. Az egyes feladatoknál megjelenő megállók listája szerkeszthető, így ha például a sofőrnek több pihenésre lenne szüksége, akkor hozzáadhat egy új megállót, így a szállításvezető tudni fogja, hogy a sofőr miért nincs úton.

1.2. Architektúra

Az alkalmazás architektúrája a 3. ábrán látható, központi eleme a szerver, amely a kliensek kéréseinek kiszolgálását egy RESTful API [24] segítségével végzi. A szerver *push service*ként [17] is funkcionál, így lehetővé teszi, hogy értesítéseket küldjön a klienseknek. A szerver az



3. ábra. A szoftverrendszer elemei

adatok tárolására egy MongoDB [27] adatbázist, valamint egy Redis [6] memória alapú *cache*-t használ. Előbbiben az alkalmazások által használt adatok, utóbbiban az értesítések elküldéséhez szükséges információk kerülnek eltárolásra.

Az alkalmazásrendszerben kétfajta kliens van jelen: a webes, valamint a mobilalkalmazás, amelyek HTTP kéréseken keresztül kommunikálnak a szerverrel. A webalkalmazás React [15] nyelvben íródott és a szállításvezetők számára elérhető. A sofőröknek a mobil kliens áll rendelkezésükre, amely a React Native [7], valamint Expo [9] keretrendszerek segítségével jött létre. A mobil kliens egy *cross-platform* alkalmazás, így Android, illetve iOS felhasználók számára egyaránt elérhető.

2. A szerver

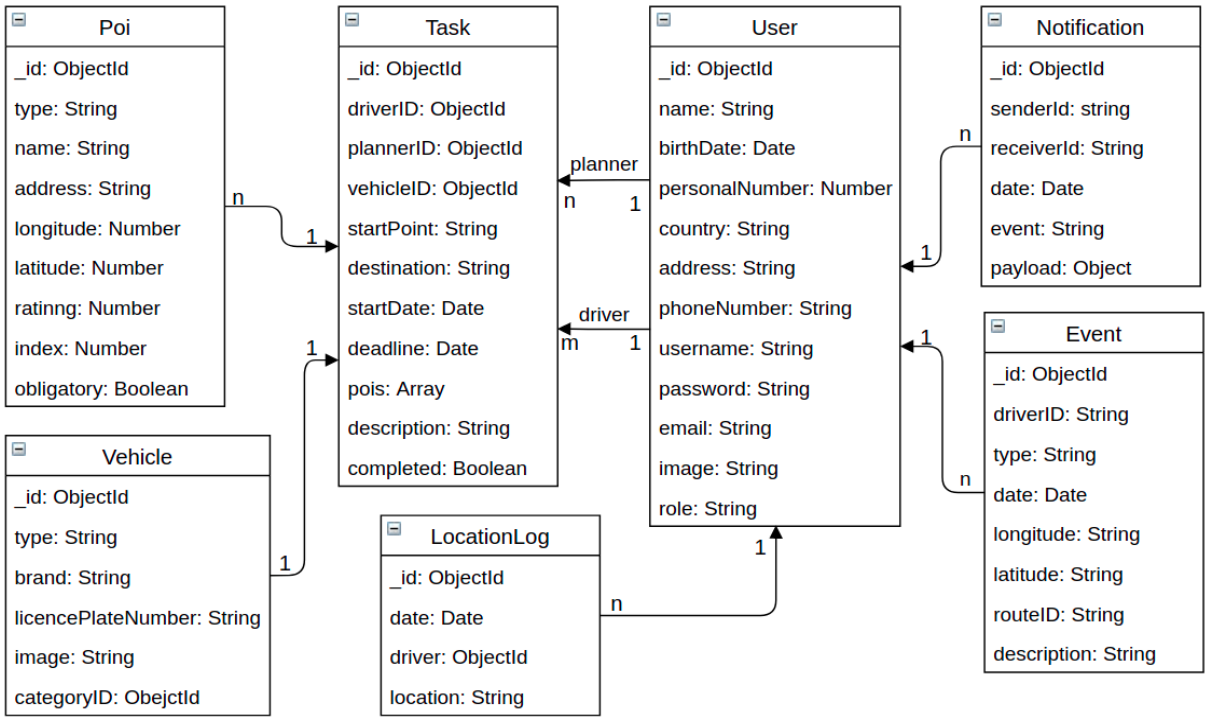
Az alkalmazás központi része egy **JavaScript** [16] alapú szerver, amely a kliensek kéréseinek kiszolgálásáért, az értesítések elküldéséért és az adatok kezeléséért felel. A futási környezetet a **Node.js** (ld. 5. fejezet) biztosítja, amely a Chrome V8 JavaScript motorját használja.

A standard háromrétegű architektúra elemei: az adathozzáférési réteg, a szolgáltatási réteg, és a megjelenítési réteg. A szerver felel az adathozzáférési, valamint szolgáltatási réteg megvalósításáért, a megjelenítés a kliensalkalmazások feladata.

2.1. Adatmodell

Az adatok sémáit a **db/models** modul tartalmazza. Ezek közül a főbb modellek a következők: a feladatokat és megállókat, felhasználókat, közúti eseményeket és értesítéseket, sofőrök koordinátáit, valamint a járműveket reprezentáló modellek.

Ahogy a 4. ábrán látható, a **Task** modell tartalmaz minden szükséges adatot, amely az áru szállításához tartozik. A felhasználókat a **User** modell jelképezi, a lényeges különbséget a **role** nevű mező tartalmazza, amely az alkalmazáson belül betöltött szerepet tárolja (*planner* vagy *driver*). A **Poi** (*point of interest*) modell azokat a helyszíneket tartalmazza, amelyekhez



4. ábra. Az alkalmazás fő entitásai

a sofőrök el kell menjenek útjuk során. Ezek a megállók lehetnek kötelezőek vagy éppen opcionálisok.

A **Notification** nevű modell tárolja a felhasználók által küldött értesítéseket és azok tartalmát, a **Vehicle** pedig a feladathoz hozzárendelt jármű adatait tartalmazza, amellyel a feladatot kell végeznie a sofőrnek.

Az **Event**, illetve **LocationLog** modellek főként a sofőrökről tartalmaznak információt. Egy *event* objektumban megtalálható, hogy hol, mikor és melyik sofőrrel történt valami az útja során. Ilyen események lehetnek például a baleseti jelentések, közúti ellenőrzések vagy esetleges lopások. A LocationLog a sofőrök pillanatnyi helyzetét rögzíti, amely a sofőr azonosítóját, koordinátáit és a rögzítés időpontját tartalmazza.

2.2. Adathozzáférési réteg

Az adathozzáférési réteget a **controllers** modulban található vezérlők alkotják, amelyek biztosítják az adatokon végezhető **CRUD** műveleteket: létrehozás (***C**reate*), olvasás (***R**ead*), frissítés (***U**pdate*) és törlés (***D**elete*).

Az 2.1. fejezetben bemutatott modellekből létrejövő fő, valamint egyéb segéd entitások, mint például a sofőrök utolsó helyzetét képviselő adatok tárolására a **MongoDB** szolgál. A szerver és az adatbázis kapcsolatát a **Mongoose** [40] könyvtár biztosítja. A Mongoose segítségével a kódban szereplő modellek az adatbázisban levő modellekre lesznek leképezve, így lehetővé téve az adatok kezelését.

A fő adatbázis mellett a szerver a Redis memória alapú tárolót is használja, amely a kliensalkalmazások *push token*-jeit tartalmazza, amelyek az értesítések küldéséhez és fogadásához szükségesek. Ezek tárolását két hasítótábla végzi a gyors adatelérés érdekében. Az egyik hasítótábla a szállításvezetők, a másik pedig a sofőrök tokenjeit tárolja. Egy token kulcsát a hozzá tartozó felhasználó azonosítója képezi.

A szerver a *push token*-eket a felhasználók bejelentkezésekor kapja meg, amit a kliens által küldött kérés tartalmaz. Ekkor a tokent a szerepkörnek megfelelő táblába lesz elmentve, amelyeket értesítések küldésekor fog használni a szerver.

A Redisben tárolt adatok a szerver által biztosított API-n keresztül nem elérhetőek, csak a szerver képes lekérni, illetve módosítani azokat.

2.3. Szolgáltatási réteg

A szolgáltatási réteg felel az alkalmazás logikájának megvalósításáért, a kliensek kérésének kiszolgálásáért, valamint az értesítések elküldéséért. Kapcsolatban áll az adathozzáférési réteggel, így minden adatmodellnek külön egység felel meg, amelyek együtt a **services** modult alkotják. Aszerint, hogy a kliens milyen műveletet szeretne elvégezni egy adott entitáson, a hozzá tartozó vezérlő által biztosított függvények itt kerülnek meghívásra.

Az értesítések küldésére két esetben kerül sor: amikor a szállításvezető létrehoz egy feladatot, ekkor a szerver megkeresi a feladathoz rendelt sofőr tokenjét és értesíti őt az új feladatról. A másik eset, amikor a sofőr bejelent egy új eseményt. Ebben az esetben a szerver megkeresi azt a feladatot, amely közben a sofőr bejelentette az eseményt, majd a Redisben tárolt táblákból kiválasztja a megfelelő szállításvezető tokenjét, aki létrehozta az adott feladatot, majd elküldi neki az értesítést az új eseményről.

Az értesítések elküldését klienstől függően két modul biztosítja: a webes kliensnek a *web-push*, a mobil kliensek számára az *expo-server-sdk* modul segítségével küldhetők az értesítések.

2.4. API

A szerver a kliensektől érkező HTTP [14] (*HyperText Transfer Protocol*) alapú kérések fogadására és kiszolgálására az **Express** [23] keretrendszert használja. A szerver által figyelt *endpoint*-ok a REST (*Representational State Transfer*) alapelveinek megfelelően vannak kialakítva. Az állapot nélküli kommunikációnak köszönhetően a szerver további skálázhatóságra tesz szert. Az API [26] (*Application Programming Interface*) úgy van kialakítva, hogy a szerver erőforrásaihoz, mint a mobil, mint a webes kliensek ugyanazokon az URI-ken [5] (*Uniform Resource Indentifier*) keresztül tudnak hozzáférni.

Az API két fő útvonalon biztosít kommunikációt a szerverrel. Az egyik útvonal az `/auth`, a másik az `/api` gyökérre épülő URI. A támogatott *endpoint*-ok a **routes** modulban vannak definiálva, ahol mindegyik *endpoint*-on végezhető műveletekhez a szolgáltatási réteghez tartozó megfelelő kezelő függvény van rendelve.

Ahhoz hogy a szerver hitelesíteni tudja a felhasználókat, bejelentkezés szükséges, amely a `/auth/login` útvonalon egy *POST* metódusú kérés küldésével történik. A kérés törzsének tartalmaznia kell a felhasználónevet, jelszót, valamint egy *subscription* nevű mezőt is, amely a felhasználó eszközéhez tartozó a *push token*-t tartalmazza. Bejelentkezéskor egy helyes

felhasználónév-jelszó páros mellett a *push token* jelenléte szükséges, mivel anélkül a szerver nem tudja megvalósítani az értesítések küldését. Abban az esetben, ha az értesítések le vannak tiltva, az alkalmazás továbbra is megfelelően működik, viszont a felhasználó nem fog értesülni az új eseményekről. Sikeres bejelentkezés esetén a szerver egy *access*, illetve egy *refresh token*-t állít ki a kliens számára, amely tartalmazza a felhasználó azonosítóját, illetve az általa betöltött szerepkört (tokenekkel kapcsolatos részletesebb leírást a 2.5. alfejezet tartalmaz).

Ahhoz, hogy egy felhasználó új token párost igényeljen, nincs szüksége újbóli bejelentkezésre. Egy GET kérést küldve a `/auth/refresh` útvonalra, a szerver új *access* és *refresh token*-t generál. A kliensnek egy *Refresh* nevű fejlécet kell hozzácsatolnia a kéréshez, amely a szerver által előzően kiállított *Refresh token*-t tartalmazza.

A szerveren tárolt erőforrásokhoz, a kliens a `/api/{resource}/:id` útvonalon keresztül a következő metódusok segítségével férhet hozzá: **GET**, **POST**, **PATCH** és **DELETE**. A /api előtag mindegyik URI-ben közös. Az URI `/{resource}` része jelzi, hogy a kliens melyik erőforráshoz szeretne hozzáférni (szállításvezető esetében, ha le szeretné kérdezni a hozzá tartozó összes task-ot, akkor az ennek megfelelő útvonal a /api/tasks lenne). Abban az esetben, ha `:id` paraméterrel is el van látva, akkor a szerver az adott id-val rendelkező entitást fogja visszatéríteni vagy módosítani, a kérésben szereplő HTTP metódus alapján. Ezen felül bizonyos URI-k esetében a szerver `query` paramétereket is vár feldolgozásra, amelyeket az egyes entitások közti kereséshez használ szűrőként (például egy adott területen elhaladt sofőrök és koordinátáik: `GET /api/drivers?lat=46.57&lng=23.86&radius=20000`).

Mivel ezeken az útvonalakon a szerver hozzáférést biztosít az adatokhoz, ezért a kérésekhez szükséges az *Authorization* beállítás csatolása a fejléchez, amely egy érvényes *access token*-t tartalmaz. Ez által hitelesíthető a felhasználó, valamint korlátozható az egyes erősforrásokhoz való hozzáférés a kliens szerepkörének megfelelően. Abban az esetben, ha a felhasználó nem csatol *access token*-t a kéréshez, vagy a token érvényességi ideje lejárt, a szerver a 401-es (*unauthorized*) státusz kódot, valamint megfelelő hibaüzenetet térít vissza válaszként.

2.5. Biztonság

A biztonságért a JWT[6] [18] (*JSON Web Token*) alapú felhasználó hitelesítés, az RBAC [12] (*Role Based Access Control*) rendszer, valamint a VAPID [39] (*Voluntary Application Server Identity*) kulcsok felelnek.

[6] 5. ábra forrása: `https://jwt.io`

5. ábra. A bal oldalon egy JWT token és részei láthatóak, a jobb oldalon az adatok, amiket a szerver felhasznál ezek előállítására.

A JWT tokenek használata egy biztonságos módja a szerver és a kliensek közti kommunikáció megvalósításának, mivel minden token digitálisan aláírt. Az aláírásnak köszönhetően ellenőrizhető, hogy az információ megbízható forrásból származik-e. Egy tokennek három fő része van: fejléc (*header*), az információt tartalmazó rész (*payload*), valamint az aláírás (*signature*). Mindegyik részt, egy-egy *JSON* [3] objektum reprezentál, amelyek végül titkosítás után egy három részből álló karakterláncot fognak alkotni, amelyben az egyes részeket '.' karakter választ el egymástól (5. ábra).

A fejléc a tokenről tartalmaz információt, amelynek két része van: *alg*, amely az aláíráshoz használt algoritmust jelöli, és a *typ*, amelynek az értéke 'JWT'. A használt algoritmust a rendszer környezeti változóból olvassa be. A fejlécből egy *Base64* [36] kódolású karakterlánc készül. Ahogy a 5. ábrán is látható, a *payload*-ban tetszőleges információ szerepelhet szintén *JSON* formátumban. A szerver a token ezen részébe fogja elhelyezni a kliens azonosításához szükséges információkat: a felhasználó azonosítóját, valamint az általa betöltött szerepkört. A fejléchez hasonlóan a *payload* is egy Base64-el kódolt karakterlánccá lesz átalakítva.

Az aláírást a token fejlécében megadott algoritmussal végzi a rendszer. Az aláírást generáló algoritmus két paramétert vár. Az első egy karakterlánc, amely az előre titkosított fejléc és *payload* összefűzésével kapunk, amiket egy '.' karakter választ el egymástól. A második paraméter egy *secret*, amelyet a szerver egy környezeti változóban megadott *.key* kiterjesztésű fileból olvas be. A JWT tokenekhez használt secretek RSA [29] algoritmussal előállított privát és publikus kulcspárok, amelyek az `util/resources/keys/jwt` mappában találhatóak.

Bejelentkezéskor a kliensalkalmazás két JWT kulcsot kap. Az egyik egy rövidebb lejárati idővel rendelkező *Access token*, amelyet a kliensalkalmazás az egyes kérések mellé küld a

```
if (!can(role, 'task:edit', { userId, role, creatorId: task.plannerID })) {
  res.status(status.UNAUTHORIZED).json({ message: 'Operation not granted!'});
}
```

1. kódrészlet. RBAC használata kódban. A can metódus a neki megadott paraméterek esetében ellenőrzi, hogy az adott szerepkörrel rendelkező felhasználó jogosul-e egy feladat elvégzésére.

szervernek. A másik az egy *Refresh token*, amely csak akkor kerül használatra, ha az *Access token* érvényességi ideje lejár. Ennek a segítségével a felhasználó új, érvényes tokenekhez juthat.

Az RBAC hitelesítési rendszer a felhasználók szerepköre alapján határozza meg, hogy egy adott kérést kiszolgál-e a rendszer, vagy sem. Ezeket az ellenőrzéseket még az adathozzáférési műveletek végrehajtása előtt elvégzi a kontrollerek metódusaiban. A kliensek jogosultságának ellenőrzését az RBAC osztály `can(role, operation, params)` függvénye végzi, amelyet a 1. kódrészlet szemléltet. Az első paraméter az aktuális felhasználó szerepköre, amelyet az előzetes *JWT* token ellenőrzésekor, illetve annak dekódolásával határoz meg a rendszer. A második paraméter a művelet, amit el szeretne végezni, a harmadik pedig a paraméterek listája, ahol újabb megszorítások ellenőrizhetőek. Például (2. kódrészlet), ha a szerver azt ellenőrzi, hogy egy felhasználó frissítheti-e egy feladat paramétereit, akkor két feltétel helyességét kell ellenőrizze: a kliens szállításvezető szerepkörrel kell rendelkezzen, valamint ő kell legyen az adott feladat létrehozója, így a szállításvezetők nem szerkeszthetik egymás feladatait. Ha egy kliens egy számára nem engedélyezett műveletet szeretne elvégezni a szerver 401-es státusz kóddal és *'Operation not granted'* hibaüzenettel válaszol.

A VAPID kulcsok szükségesek ahhoz, hogy a szerver *push service*-ként azonosíthassa magát, így a klienseknek *push* értesítéseket küldhet anélkül, hogy egy külső szolgáltatáson keresztül juttatná el azokat. Ezek olyan Base64 kódolású karakterlánccá alakított JWT tokenek, amelyek többek között a szerver identitását biztosító adatokat tartalmaznak. Ilyenek lehetnek

```
{
  name: 'task:edit',
  when: (params) => {
    return prams.role === 'planner' && params.userId === params.creatorId;
  },
}
```

2. kódrészlet. A 1. kódrészlet a `when` függvényt fogja kiértékelni, hogy eldöntse, hogy egy felhasználónak joga van-e elvégezni a `name` mező által jelölt feladatot

például az `aud` mező, amely a tokent egy *push service*-hez [4] rendeli hozzá, az `exp` mező, amely a token lejárati idejét tartalmazza, vagy akár egy kapcsolati email cím.

A VAPID a kulcsokat a `/util/resources/keys/vapid` mappából olvassa ki a szerver. Így mind a webes, mind a mobil kliens számára küldhetők az értesítések a 2.3. fejezetben említett külső könyvtárak segítségével. A webes kliensek számára a *web-push* modul által biztosított `sendNotification` metódus áll rendelkezésre. Mivel a mobil kliens felhasználja az Expo keretrendszert, az értesítések fogadását is az Expo kezeli. Ezért van szüksége a szervernek az *expo-server-sdk* modulra, amely a `sendPushNotificationsAsync` metódus segítségével küldhet értesítéseket a mobil klienseknek.

3. A web kliens

A webalkalmazás egy interaktív felületet biztosít a szállításvezetők számára, ahol könnyedén kezelhetik a feladatokat vagy nyomon követhetik a sofőröket. Ezek mellett olyan lehetőségeket is biztosít, amelyek leegyszerűsítik a felhasználó munkáját, akár a sofőröktől kapott információkról, vagy egy adott helyen és időben történő keresésről legyen szó, éppen ezért a webes kliens csak a szállítástervezők számára elérhető.

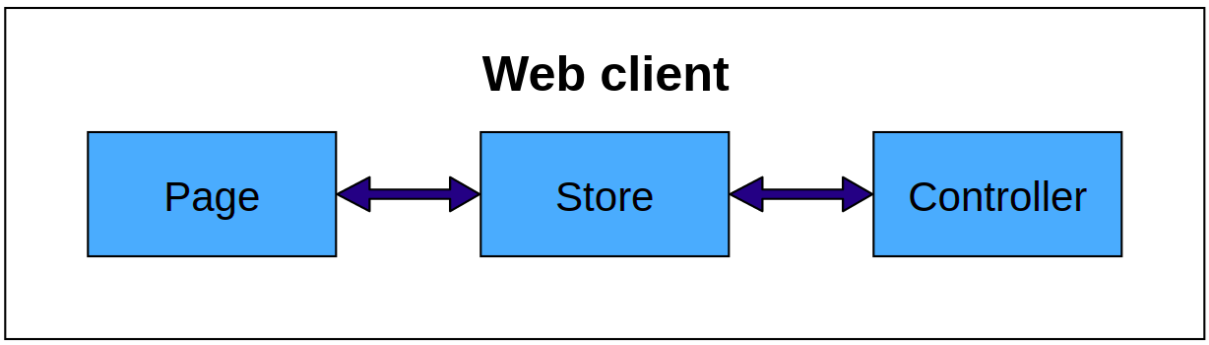
3.1. Felhasználói felület megvalósítása

A felhasználói felület a **React** [37] könyvtár segítségével jött létre, az alkalmazás állapotának kezeléséért a **MobX** [34] felel (további részletek az 5. fejezetben).

A webalkalmazás három fő modulból épül fel: oldalak, *store*-ok és vezérlő egységek, amelyek kapcsolatát a 6. ábra szemlélteti. Az oldalak felelnek a megjelenítésért, a *store*-okban vannak eltárolva az adatok, a vezérlők kommunikálnak a szerverrel és töltik fel a *store*-okat adattal. Az utóbbi modul a 3.2. fejezetben kerül bemutatásra.

Az oldalak a különböző komponensből állnak, illetve a nagyobb komponensek kisebb egységekből épülnek fel, amelyek a **components** mappában találhatóak (például az oldalakon található fejléc vagy egy feladatot megjelenítő listaelem). A különálló komponenseknek köszönhetően az alkalmazás könnyen bővíthetővé, a komponensek pedig újra felhasználhatóvá válnak. Az felhasznált komponensek kinézete a **styles** mappában található CSS [28] (*Cascading Style Sheets*) állományok segítségével lettek kialakítva. Minden oldalhoz, illetve komponenshez külön CSS fájl tartozik.

Az oldalak közti navigáció a **React Router**rel [38] van megvalósítva. Az oldalakhoz vezető útvonalak a **config** mappában találhatóak. Itt egy *JSON* objektum van definiálva, amely két listát



6. ábra. A webalkalmazás szerkezete.

```
const routes = {
  authorized: [
    {
      path: '/planner/tasks/:taskId',
      name:'TaskDetailsPage',
      component: TaskDetailsPage,
    },
    ...
  ],
  unAuthorized: [
    {
      path: '/auth/login', name: 'LoginPage', component: LoginPage,
    },
  ],
}
```

3. kódrészlet. A webalkalmazás által használt router objektumából egy részlet. A bejelentkezett felhasználó egy adott azonosítóval rendelkező feladat részleteit megjelenítő oldalhoz (*component* attribútum), valamint egy nem bejelentkezett felhasználó által elérhető bejelentkezéshez vezető útvonalakat szemléltet.

foglal magába. Az egyik az *authorized*, amely a bejelentkezett felhasználó által elérhető oldalak útvonalait tartalmazza. A másik az *unauthorized*, amelyben a bejelentkezés nélkül látogatható oldalakat tartalmazza.

Az alkalmazásban található térkép nézeteket a **Leaflet** [20] biztosítja, amelyek felhasználását és funkcionalitásait a 6.2. fejezet mutatja be.

3.2. Kommunikáció a szerverrel

A szerverrel való kommunikáció megvalósításáért a kliens *service* modulja felel. Az alkalmazásban található minden oldalhoz külön *service* tartozik, amelyek együtt az alkalmazás szolgáltatási rétegét alkotják.

A *service* osztályokban definiált metódusok a **Fetch API** [13] segítségével kéréseket küldenek a szervernek, amelyre érkezett válaszból kinyert adatok ezekhez az osztályokhoz tartozó *store*-okban kerülnek eltárolásra. Ezekben a metódusokban egy `options` nevű objektum is felépítésre kerül, amely a kérés típusát (*method*), fejlécét (*headers*), vagy törzsét (*body*) tartalmazza. A kliens a kérést az *util* mappában található `customizedFetch(url, requestOptions)` metódus segítségével küldi el.

A `customizedFetch` két paramétert vár: az első egy karakterlánc, amely a kérés

```
const defaultOptions = {
  method: method.GET,
  headers: {
    Accept: 'application/json',
    Authorization: accessToken,
  },
};
```

4. kódrészlet. A kliensalkalmazások által küldött kérések alapértelmezett beállításai. A kérés GET metódusú, amire JSON formátumú választ vár a szervertől, valamint a kliens *access* tokenje is csatolva van.

URI-jét tartalmazza. A második egy objektum, amely a kérés beállításait tartalmazza. A *customizedFetch*-ben található egy objektum, amely előre definiált beállításokat tartalmaz, így megelőzi a kód ismétlést a hasonló fejléccel rendelkező kérések küldése esetén. Az alapértelmezett beállításokat a 4. kódrészlet szemlélteti. A kérés elküldésére a megadott URI-re és a véglegesített beállításokkal ezután kerül sor a *fetch* parancs segítségével. A kérés küldése úgy lett kialakítva, hogy figyelje a szerver válaszát. Ha a válasz a 401-es státusz kódot tartalmazza, akkor a kliensalkalmazás automatikusan új tokeneket igényel a szervertől, amelyeket a *localStorage*be [22] ment el, majd újra elküldi az adott kérést.

3.3. Értesítések kezelése

Az értesítések fogadását és megjelenítését a böngészőben regisztrált **service worker**ek [1] végzik, amelyek *proxy* szerverekként funkcionálnak a webes alkalmazások és a böngésző között, elérhetővé téve olyan funkcionalitásokat, mint például a *push notification*-ök megvalósítása vagy a háttérben történő szinkronizáció.

Az alkalmazás indulásakor ellenőrzi, hogy a böngészőben létezik-e már a használni kívánt *service worker*, és amennyiben szükséges, regisztrálja azt. Ahhoz, hogy az alkalmazás értesítéseket tudjon fogadni, fel kell iratkoznia a szervernél, amely bejelentkezéskor történik meg. Ekkor a `NotificationService` osztály `getPushToken()` metódusa lekéri a böngésző *push token*jét, amihez engedélyezni kell az értesítések megjelenítését az alkalmazáson belül.

A webes kliens által használt *service worker* két különböző eseményt figyel, amelyet az `addEventListener(event, callback)` függvény segítségével lehet definiálni, amely a `WorkerGlobalScope` interfészre mutató `self` referencián keresztül érhető el:

- `'push'`: értesítések érkezésekor váltódik ki, itt kerül megjelenítésre a

showNotification metódus által

- notificationclick': az értesítésre való kattintásra reagál, ekkor a böngészőben egy új ablak nyílik meg, ahol az értesítésben szereplő esemény részletei láthatóak

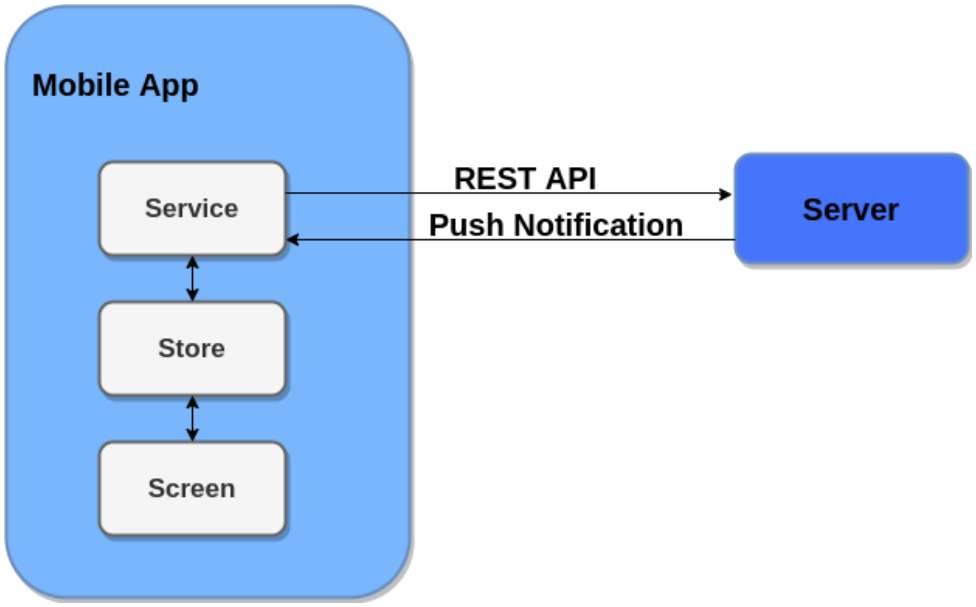
4. A mobil kliens

Ez a kliens egy olyan platformfüggetlen mobilalkalmazás, amely *React Native*-ben [35] van implementálva az *Expo* [9] keretrendszer segítségével. Felhasználói szemszögből ez a felületet csakis a sofőrök számára készült. Ezen keresztül ők képesek arra, hogy elérjék a központi szervert, és ezáltal a szállításvezetőket. Megtekinthetik a rájuk váró feladatokat, azok részleges információival együtt, és szükség esetén módosíthatnak az egy-egy úthoz tartozó megállókon.

4.1. Felhasználói felület megvalósítása

A telefon képernyőjén megjelenő különböző nézetek hátterében egy-egy olyan osztály rejtőzik, amelyek a *React* komponens ősosztályból vannak származtatva. Ezek a komponensek csak az adatok megjelenítéséért felelősek. Viszont ehhez szükség van olyan rétegre is, amely szolgáltatja és kezeli az adatokat. Ezt a feladatot fogják elvégezni a *store*-ok és a *service*-ek.

A *screen*-ekhez tartozik egy-egy megfelelő *store*, ezekhez pedig egy-egy *service*. A *store*-ok különböző adattagokkal rendelkezhetnek, melyekben a szervertől kapott információt tárolják és kezelik. Illetve ők végzik az állapotmenedzsmentet is a **MobX** könyvtár segítségével, melynek a működési alapelve megtalálható a 5.2 fejezetben. A *screen*-eket megvalósító osztályokba ezeknek a *store*-oknak egy-egy példánya van beinjektálva, és ezeken keresztül lehet elérni azokat az adattagokat, melyek a megjelenítésre váró információkat tárolják. A *service*-k a szerverrel való kommunikációért felelnek, lényegében ők a kapcsolattartók a mobilkliens és



7. ábra. A mobilalkalmazás felépítése

a központi szerver között. Ez látható a 7. ábrán.

Az alkalmazás használata során a következő *screen*-ek jelennek meg:

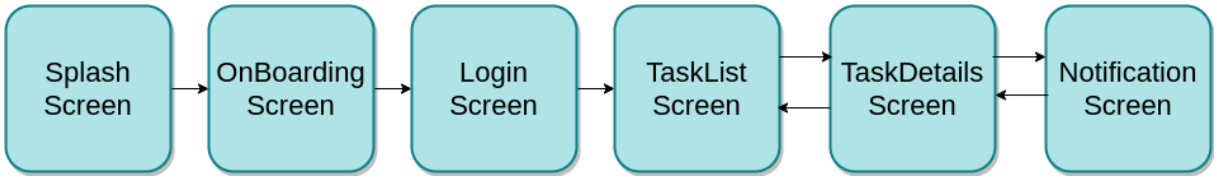
- **OnBoardingScreen**: az alkalmazás legelső indulásakor ez jelenik meg a felhasználónak, mint egy kezdőoldal, amelyről tovább léphet a bejelentkezéshez.
- **LoginScreen**: itt történik maga a bejelentkezés.
- **TaskListScreen**: a sikeres bejelentkezés után, ide navigál az alkalmazás. Ezen a nézeten kerülnek kilistázásra azok a sofőrhöz tartozó feladatok, amelyek rá várnak, illetve amelyeket már elvégzett.
- **TaskDetailsScreen**: ezen az oldalon tekinthetjük meg egy kiválasztott út minden információját, illetve hozzáadhatunk az úthoz új megállót, ha az indokolt.
- **NotificationScreen**: a felhasználó értesítést küldhet a szállításvezetőnek a történésekről.

A fent felsorolt nézetek közötti navigációt a **Stack Navigator** teszi lehetővé, amely egy verem alapelveit követi, tehát betesz és kivesz egy-egy elemet a verembe, ha a felhasználó oda szeretne navigálni, vagy csak visszalépni, ezzel leegyszerűsítve a navigációs folyamatot.

A fejlesztés során nem került használatba vételre semmilyen *UI library*, a nézetek egységes kinézete, mind a fejlesztők implementációi.

4.2. Kommunikáció a szerverrel

Ahogyan az előző alfejezetben is említésre került, a nézetek csak az adatok megjelenítésére szolgálnak. Ám ahhoz hogy ezeket az adatokat el lehessen érni, elengedhetetlen a szerverrel való kommunikáció. Itt van szükség a *service*-ek implementációjára. Majdnem minden *store*-hoz tartozik egy *service*, amely HTTP hálózaton keresztül kéréseket küld a szervernek, ha valamilyen adatra van szükség, új adatot kell bevezetni a rendszerbe vagy módosítani kell rajtuk.



8. ábra. Navigáció a nézetek között

Abban az esetben, ha a felhasználói felületen olyan esemény váltódik ki, amely során szükség van a szerverrel való kapcsolatfelvételre, akkor a nézetről továbbítódik a kérés a *store*-okhoz, majd a *service*-ek felé. Itt egy a Fetch API [13] használatára épülő metódus segítségével küldődik el a kérés (GET, POST, DELETE, PATCH) egy megadott útvonalra, a megfelelő fejléccel és *body*-val szükség esetén.

A bejelentkezést követően a szervertől kapott válaszban (amennyiben helyesek voltak a bejelentkezési adatok) érkezik két *JWT token* is: az *Access* és a *Refresh token*. Az ezután küldött kéréseknek tartalmaznia kell az *Access tokent*, hogy a *Rest API*-hoz beérkezett kéréseken keresztül a rendszer azonosítani tudja a felhasználót.

Ennek érdekében a *service*-ek a **customizedFetch()** függvényt használják, melynek implementációja a **util.js**-ben található. A függvény automatikusan hozzácsatolja a kéréshez a tokent. Amennyiben a szervertől az a válasz érkezne, hogy az *Access token* lejárt, ebben az esetben egy következő kérés keretein belül elküldésre kerül a *Refresh token is*. Az erre érkező válaszból kihámozott új *Access tokent* a rendszer eltárolja, majd az eredeti kérést már ennek segítségével újra elküldi.

A tokenek az **Async Storage**-ban [2] vannak eltárolva kulcs-érték párosként. Ezeket elérni vagy módosítani a **getData()** és **storeData()** függvények segítségével lehet, melyek a **util/asyncStorageFunctions.js**-ben vannak implementálva.

A szervertől az adatok *JSON* formátumban érkeznek. Ezeket az alkalmazás a beépített *JSON.parse()* metódus segítségével alakítja át objektumokká, melyek felépítése megegyezik az adatbázis adatmodelljében szereplőkkel a 2.1 fejezetből.

4.3. Háttérfolyamatok és értesítések

A mobilalkalmazás indulásakor, a bejelentkezést követően elindul egy háttérfolyamat, melynek az implementálása az *services* modulon belül a *BackgroundService.js*-ben található. Ez a folyamat egy előre megszabott időközönként a hálózaton keresztül elküldi a szerver felé a sofőr aktuális helyadatait az *expo-location* [10] modul használatával, mely elérhetővé teszi az eszköz geo lokációs adatainak az olvasását. Ehhez szükség van arra, hogy előzőleg engedélyezve legyen az eszköz beállításai között az alkalmazás számára a helyadatok használata (ezzel kapcsolatosan automatikus engedélykérés is érkezik az alkalmazás indulásakor). Ezek az adatok arra szükségesek, hogy a webes felületről a szállításvezető egy térkép segítségével élőben követni tudja, hogy a sofőrök az út milyen szakaszában vannak.

A sofőrök a mobiltelefonjaikon keresztül képesek arra, hogy értesítéseket fogadjanak, illetve küldjenek. Amikor a szállításvezető egy új feladatot hoz létre és azt kiosztja egy bizonyos sofőrnek, akkor az illető egy értesítést kap. A sofőrök pedig valamilyen esemény következtében értesítést küldhetnek a felettesüknek, mely során egy új esemény kerül az adatbázisba. Ez a *push notification*-ök segítségével valósul meg az *Expo Notifications API* [11] használatával. Ehhez szükség van az *ExpoPushToken* lekérésére, mely segítségével fel lehet iratkozni a szervernél az értesítések fogadására.

5. Felhasznált technológiák és eszközök

Ebben a fejezetben a szerver, valamint a kliensalkalmazásokban felhasznált technológiák, valamint a fejlesztés során segítségül vett eszközök kerülnek bemutatásra.

5.1. Szerveroldali technológiák

Node.js

A Node.js [25] egy pehelysúlyú, esemény-vezérelt futási környezetet biztosít JavaScriptben írott programok számára. Aszinkron módon működik, a program viselkedését *callback* függvények segítségével lehet definiálni. Az események és lekezelések egy FIFO-szerű várakozási sorba kerülnek, ahol a prioritásuktól függően, sorrendben hajtódnak végre egyetlen szálon.

Az alkalmazás függőségei a *package.json* állományban szerepelnek, ahol a `dependencies` objektumban találhatóak azok a függőségek, amelyekre a kitelepítéskor, illetve lokális futtatáskor van szükség. A `devDependencies` objektumban csak olyan függőségek jelennek meg, amelyek az alkalmazás fejlesztésekor hasznosak, mint például a *nodemon* [31], amely a kód minden frissítésekor újraindítsa az alkalmazást, vagy az *eslint* [8], amely a kódolási konvenciók betartásában nyújt segítséget. A függőségek kezeléséért az npm [32] (*node package manager*) felel.

Express

Az Express egy keretrendszer, amelyet egy Node.js által futtatott alkalmazás használhat külső függőség formájában. Az általa szolgáltatott `Router` osztálynak köszönhetően az alkalmazás webszerverként viselkedhet, amely *routerek*ben definiált útvonalon figyel. Segítségével *middleware*-k definiálhatók, amelyekkel előfeldolgozás végezhető a kliensek által küldött kéréseken. A 2.5. fejezetben bemutatott RBAC által vizsgált előfeltételek, valamint a kérésekben szereplő JWT tokenek dekódolása és érvényességük ellenőrzése *middleware*-k használatával történik.

MongoDB

A MongoDB egy dokumentum alapú (*NoSQL*) adatbázis, amely az adatokat JSON típusú dokumentumokban tárolja. Mivel a tárolt adatok formátuma nincs megszabva, mint

a hagyományos relációs adatbázisokban, ezért flexibilis és skálázható marad nagymértékű adathalmaz esetén is.

Mongoose

A Mongoose egy ODM [33] (*Object Document Mapper*) könyvtár, ami az ORM-re [21] (*Object Relational Mapper*) épít. Az ORM rendszerek a kódban használt objektumokat relációs adatbázis béli táblákká képezik le, így az objektumoknak pontosan kell követniük a táblák szerkezetét. A Mongoose a kódban szereplő objektumokat a MongoDB által használt JSON objektumokra alakítja, így a tárolt adatok szerkezete eltérő lehet, ami kellő rugalmasságot biztosít. A Mongoose a MongoDB natív vezérlőjére [30] épül, így lekérdezések, adatbázisba való beszúrások, vagy az adatok módosítása mellett séma ellenőrzést is biztosít, amely nagyban megkönnyíti az adatbázisba kerülő adatok helyességének ellenőrzését.

Redis

A Redis (*Remote Dictionary Server*) egy nyílt forráskódú, memória alapú gyorsítótár, amely ANSI C nyelvben íródott. Kulcs-érték párosok formájában különböző adatstruktúrák tárolására ad lehetőséget. Hatékonysága annak köszönhető, hogy más adatbázisokhoz képest nem háttértárolókat, hanem a memóriát használja az adatok tárolására, viszont a ritkán használt kulcsokat idővel állományokba menti le. A hasítótábla-alapú tárolás előnye a gyors keresés, így a szerver hatékony maradhat nagy felhasználóbázis esetén is.

5.2. Web technológiák

React

A React egy népszerű JavaScript könyvtár, amely felhasználói felületek létrehozására szolgál. Komponens alapú, vagyis a megjelenített felület különböző komponensekből épül fel, amelyek egymásba ágyazódnak, így egy fa szerkezetet alkotnak. Minden komponensnek saját állapota van, amelyek együtt meghatározzák a teljes alkalmazás állapotát.

MobX

A MobX egy állapotkezelést megvalósító könyvtár, amely az alkalmazás aktuális állapotát úgynevezett *store*-okban tárolja. A *store*-nak köszönhetően az alkalmazás állapota nem lesz

szétszórva a komponensek között, így végig könnyen kezelhető és koherens marad, elkerülve az összeférhetetlen állapotok miatt fellépő meghibásodásokat. Ennek érdekében különböző annotációkat biztosít: @*observer*, egy olyan objektum, amely megfigyeli az állapothoz tartozó attribútumokat, amelyek az @*observable*-el vannak ellátva. Az alkalmazásban fellépő eseményekre az @*action*-el jelölt függvények reagálnak, megváltoztatva az állapothoz tartozó értékeket, ezzel kiváltva különböző mellékhatásokat, mint például megjelenítés vagy frissítés.

Leaflet

A Leaflet egy nyílt forráskódú JavaScript könyvtár, amely lehetőséget nyújt interaktív térképek létrehozására nem csak webes alkalmazásokban, hanem mobil kliensekben is. A Leaflet API által biztosított eszközök mellett lehetőség van további *plug-in*-ok hozzáadására is, amellyel tovább bővíthető a funkcionalitások listája.

5.3. Mobil technológiák

React Native

A React Native [7] egy keretrendszer, amelyet a Facebook, mint cég, fejlesztett ki, annak érdekében, hogy kiküszöböljék a HTML használatát a Facebook mobilalkalmazásán belül. A React Native által használható a React keretrendszer, kibővítve ezt a különböző felületek által nyújtott natív lehetőségekkel. Ennek értelmében a működési elve szinte teljesen megegyezik a React-éval, amely az előző alfejezetben került említésre.

Expo

Az Expo egy nyílt forráskódú felület és fejlesztői eszköz is egyben, amely lehetővé teszi platformfüggetlen applikációk fejlesztését Android-on, iOS-en és webes felületen egyaránt, JavaScript és React/React Native segítségével.

A projekt keretein belül azért esett ennek az eszköznek a használatára a választás, mivel nagyban megkönnyíti a fejlesztést azáltal, hogy az Expo CLI letöltése után egy bármilyen mobiltelefonra, szinte azonnal futtatható az alkalmazás, hiszen egy platformfüggetlen applikációról van szó. Tehát nem szükséges külön forráskódot írni külön operációs rendszerekre. Ha bármilyen módosítás történne a kódbázisban, az alkalmazás éppen működő változata automatikusan frissül. Emellett az Expo felkínál számos olyan API-t amely hasznos

volt az implementáció során, például: az Expo notification API, amely által elérhető az ExpoPushToken és könnyedén küldhetők a push notification-ök vagy az Expo location, mely lehetőséget nyújt a geo lokációs adatok felhasználására.

5.4. Egyéb eszközök

Git

A csoportos munka megkönnyítése érdekében a projekt fejlődésének figyelése a Git osztott verziókövető szoftverre lett bízva. A Git egyik nagy erőssége a gyorsaság, mivel minden parancs lokálisan hajtódik végre, és nem a központi szervert terheli. Egy másik nagy előnye az, hogy minden lokális tároló, egy teljes biztonsági másolattal rendelkezik a projektről, tehát bármilyen probléma esetén, sokkal kisebb az esély az adatvesztésre.

Ennek használatában a GitLab felhasználói felülete nyújtott segítséget, mely rendelkezik számos olyan kiegészítővel is, amelyek a csoportmunkát és a munkamenet megszervezését teszik könnyebbé, mint például az Issue Board vagy a Wiki.

A GitLab-on belül 3 különböző projekt került létrehozásra, mind a szervernek, mind a mobil és webes kliensnek egy-egy repó állt a rendelkezésre.

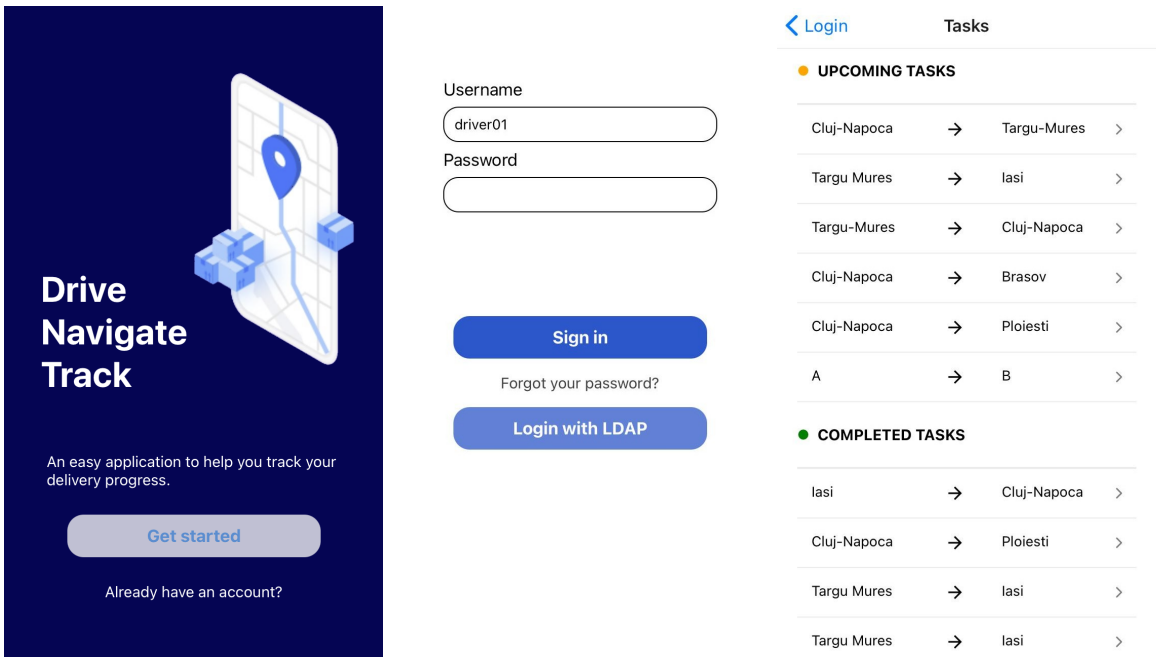
6. A kliensalkalmazások működése

Az eddigiekben a kliensalkalmazások, valamint a szerver felépítése volt bemutatva. Ez a fejezet körbejárja mindkét kliens által biztosított felhasználói felületet, valamint azok működését.

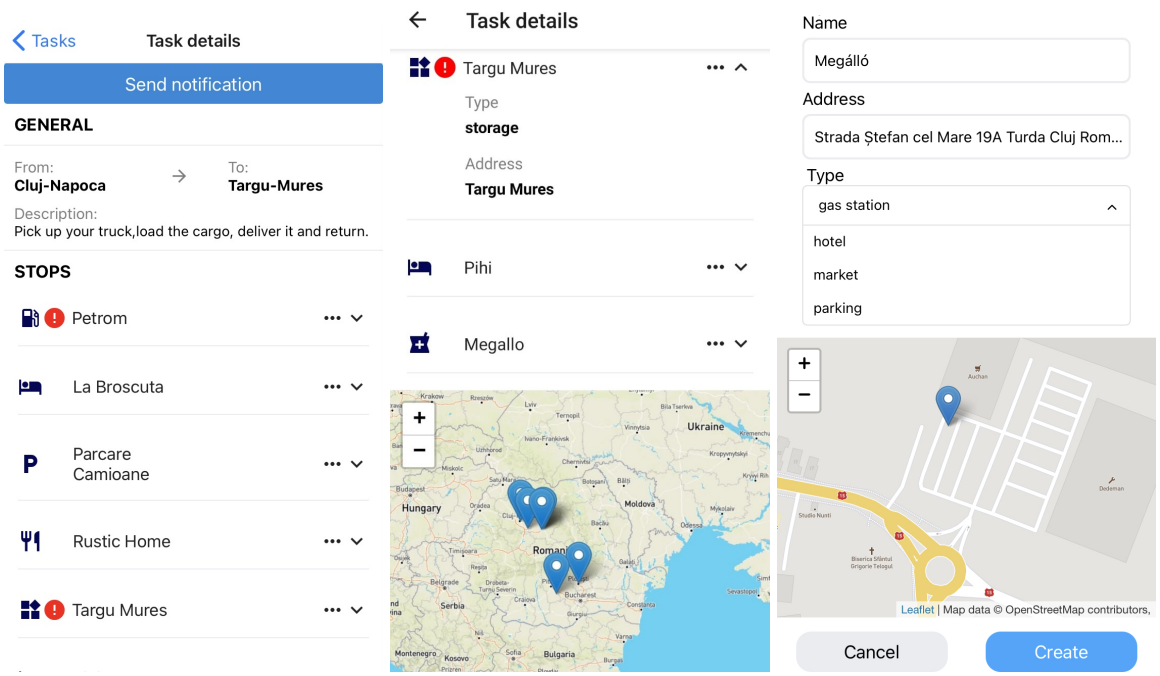
6.1. A mobilalkalmazás használata

A mobilalkalmazás legelső megnyitásakor egy SplashScreen jelenik meg, amit egy OnBoarding screen követ. Innen lehet eljutni a bejelentkezési felületre, ahol egy helyes felhasználónév és jelszó beírásával lehet továbbhaladni (ld. 9. ábra). A bejelentkezési oldalon még megjelenik egy „Login with LDAP” feliratú gomb is, amely jelenleg nem elérhető. Az LDAP [19] a *Lightweight Directory Access Protocol* rövidítése és olyan protokoll, mely a címtárak gyors elérésére és kezelésére szolgál. A címtár mivel egy olyan adatbázistípus, melyen belül nagyon gyorsan lehet keresni, rendkívül alkalmas a felhasználók bejelentkezési adatainak a tárolására.

Fontos kiemelni, hogy a mobilalkalmazás csak a sofőröknek szól. Ennek értelmében a bejelentkezést követően megjelenik a következő képernyő, melyen két lista szerepel egymás alatt. Az első listában a sofőr jövőbeli feladatai szerepelnek, míg a másodikban azok, amelyeket már teljesített. Ezek az utak a kezdőponttal és végponttal vannak feltüntetve.



9. ábra. OnBoarding, Login valamint TaskList nézetek



10. ábra. TaskDetails nézet és az AddPoi felugró ablak

A lista elemei közül bármelyikre rákattintva megjelenik egy következő nézet a kiválasztott út részletes leírásával (ld. 10. ábra). Itt megjelennek az általános információk (kezdőpont, végpont, leírás), majd egy lista elemein belül a megállok, illetve ezek egy térképen feltüntetve pinek segítségével. A megállók kétfélék lehetnek: kötelezőek (mint például a kiindulópont, a szállítmány felvételének vagy letételének helye, stb.) vagy opcionálisok (pihenők helye, étkezés, stb.). Ezt egy piros felkiáltójel jelzi, amennyiben ez szerepel a megálló neve mellett, abban az esetben ez kötelező, ellenkező esetben csak opcionális. Emellett még az elemek bal oldalán a különböző ikonok a megállók típusait jelzik, ezek lehetnek: tankolás, pihenés, étkezés, gyógyszervásárlás, gumidefekt és még pár. A lista elemeinek jobb oldalán megjelenik még két különböző ikon. A jobb oldali lefele mutató nyíl segítségével a lista kiválasztott eleme lenyílik és megjelennek részletes információk (a megálló típusa és címe feltüntetve). A mellette lévő ikon megnyomásakor egy lenyíló menü jelenik meg, ahol két opció áll rendelkezésre: törlés és módosítás. Egy opcionális megállót lehet törölni, illetve módosítani.

A sofőr új megállót is hozzáadhat a listához a térkép alatti ‘+’ gomb megnyomásával. Ekkor egy felugró ablak jelenik meg, ahol megadható a megálló neve, kiválasztható a típusa, illetve a térképre kattintva meghatározható a pontos helye. Majd a „Create” gombbal ez elmenthető vagy a „Cancel” gombbal vissza lehet lépni. A sofőr által hozzáadott megálló opcionális lesz. Kötelező megállókat csak a szállításvezető vezethet be a rendszerbe.

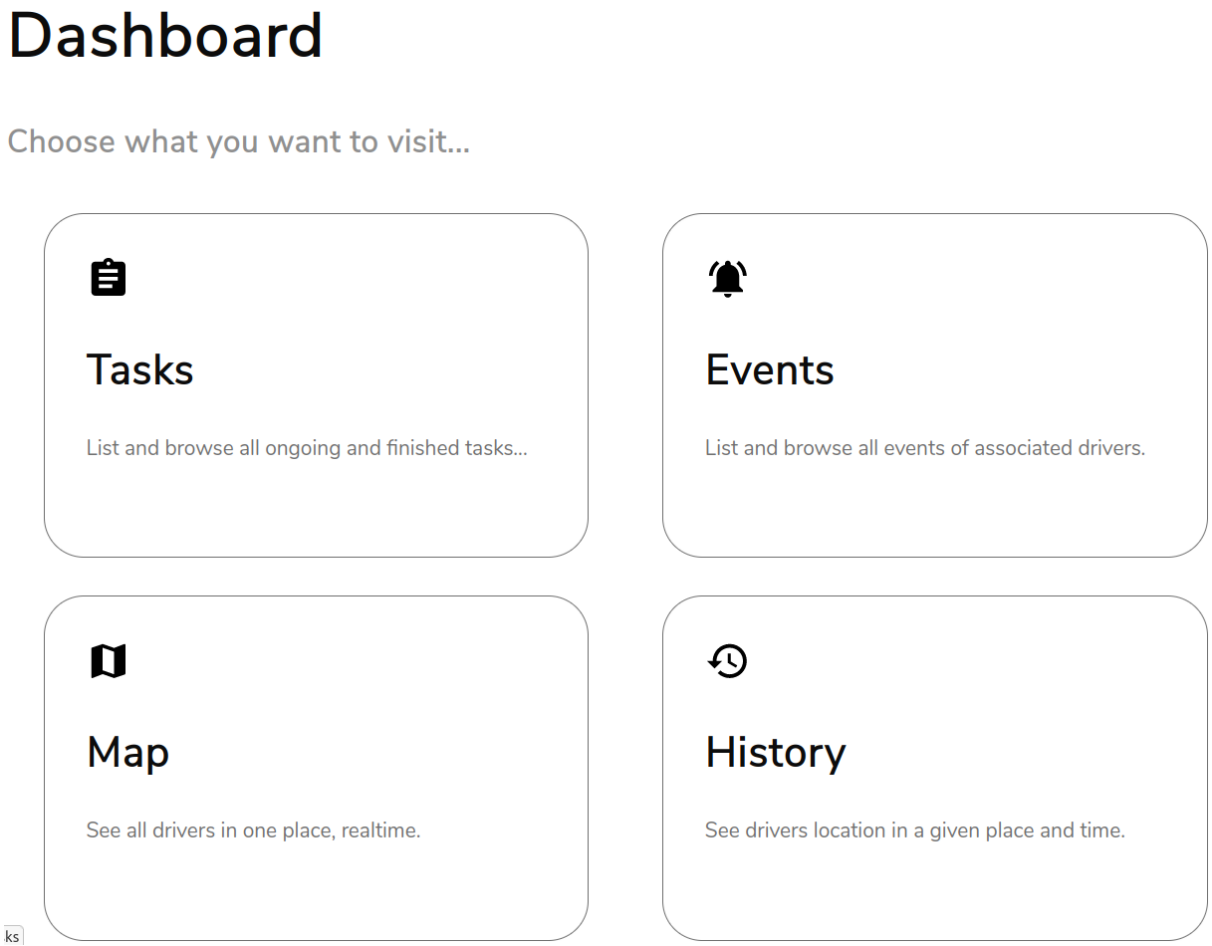
Az értesítések küldésére a szállításvezető számára, rendelkezésre áll egy külön nézet, ide

a „Send notification” nevű gomb megnyomásával lehet eljutni. Ezen a nézeten két fontos elem szerepel: egy lista, amiből kiválasztható az értesítés címkéje, valamint egy szövegdoboz, amiben megadható egy magyarázat is. Ezek kitöltése után a „Send” gomb megnyomásakor elküldésre kerül az értesítés, majd az alkalmazás visszanavigál az előző oldalra, vagyis a teljesítendő feladat részleteinek nézetére.

6.2. A webalkalmazás használata

A fejlécen megtalálható a projekt logójával ellátott rész, amelyre kattintva a főoldalra vezeti a felhasználót. Az itt található gombok segítségével az alkalmazás különböző pontjaihoz irányít át, mint például a feladatok listázása, az események vagy az élő térkép. Ezeket egy avatar követ, amely segítségével a felhasználó megtekintheti a profilját, valamint a kijelentkezés gomb.

Az alkalmazás indulásakor egy bejelentkezési oldal fogad, ahol a felhasználónév és jelszó megadásával a szállításvezető tovább léphet a főoldalra.



11. ábra. A webalkalmazás főoldala

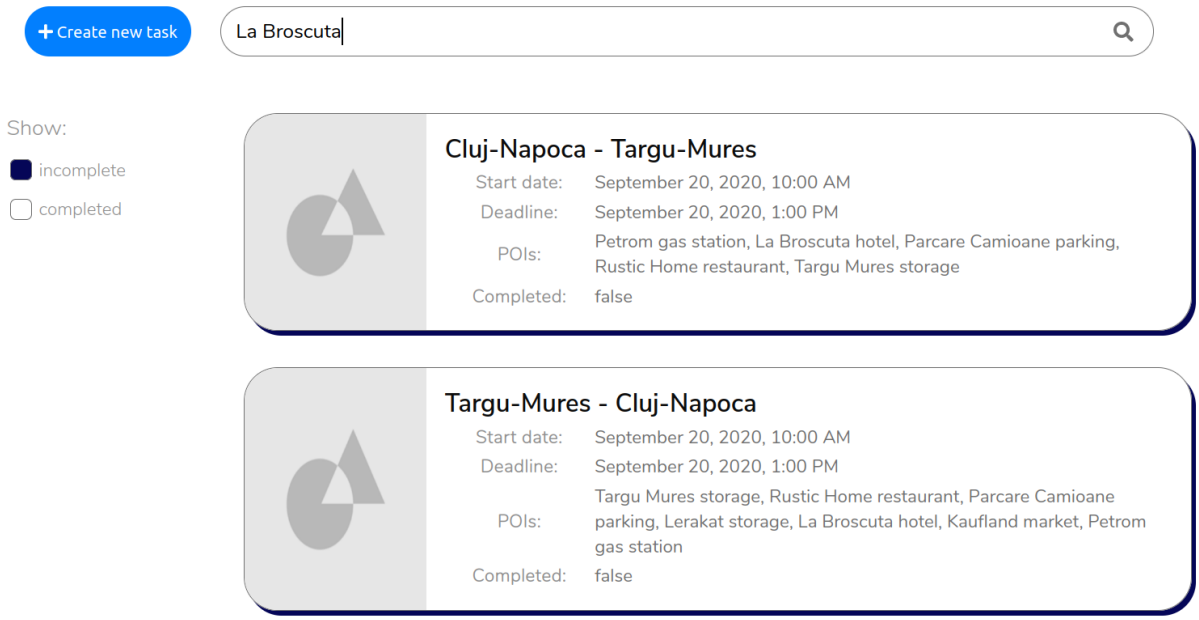
Bejelentkezés után az alkalmazás a *Dashboard*ra irányítja a felhasználót. Ez az alkalmazás fő oldala ahol négy kártya látható (ld. 11. ábra), amelyek a fejléchez hasonlóan a megjelölt oldalra navigálnak át.

A 12. ábrán látható a feladatokat megjelenítő oldal. A szállításvezetők itt tekinthetik meg az általuk létrehozott feladatokat, illetve itt található a *Create new task* gomb, amelyre kattintva új feladatot vezethet be a felhasználó.

Az oldal lehetőséget ad a feladatok szűrésére. A felhasználó választhat, hogy milyen típusú feladatok jelenjenek meg a listában: a még nem teljesített (*incomplete*) vagy a már teljesített (*completed*) feladatok is legyenek feltüntetve. A szűrő használatával a szállításvezető kereshet a feladat azonosítója kiinduló pontja, végállomása, feladatokhoz rendelt sofőr azonosítója, vagy akár a feladatokban szereplő megállók nevei alapján.

A feladatok a képernyő közepén egy listaként jelennek meg, ahol minden kártya egy-egy feladat főbb információit tartalmazza. A feladatokra kattintva megtekinthetők azok részletei.

A 13. ábrán látható oldalon tekinthetőek meg egy feladat részletei. A térkép nézet a sofőr megállóit jelöli kék markerek segítségével. Egy markerre kattintva annak színe pirosra vált, így kiemelhető a többi közül. A térkép alatt helyezkedik el a megállók részleteit tartalmazó lista. Minden elem tartalmaz két ikont, az egyik a megálló típusát jelöli (benzinkút, parkoló, étterem stb.). A másik egy figyelmeztető ikon, aszerint, hogy a sofőrnek az adott megálló kötelező-e vagy sem. Ezek mellett a név, illetve pontos cím is látható. A térképen található markerekhez



12. ábra. Azon *feladatok kilistázása*, amelyek még nincsenek teljesítve és a megállók közt szerepel a *La Broscuta* nevű hely

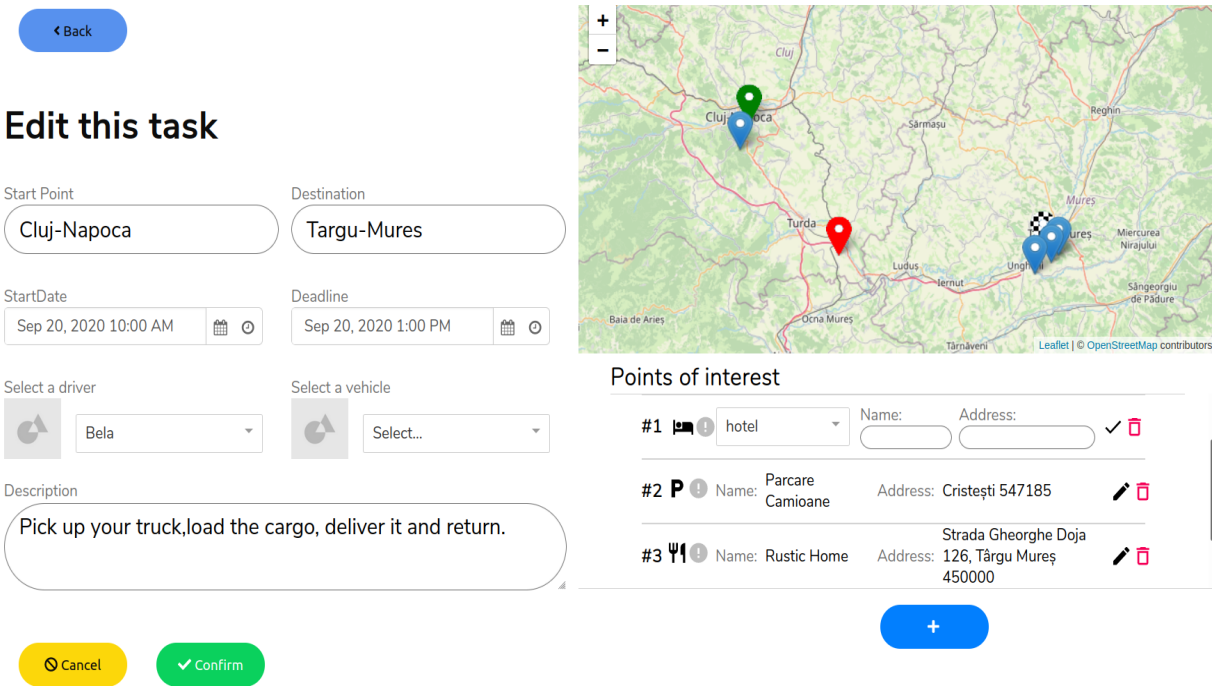
hasonlóan az azok adatait tartalmazó elemre rákattintva annak háttérszíne szürkére, valamint a megfelelő marker pirosra vált. A kiemelt markerre vagy listaelemre kattintva eltávolítható a kiemelés.

A 14. ábrán a feladatok szerkesztését biztosító oldal látható. Ez az oldal teszi lehetővé új feladat létrehozását, vagy egy meglevő módosítását.

Az itt található űrlap segítségével a feladat részleteit adhatja meg a felhasználó, illetve itt rendelheti hozzá egy sofőrhöz, valamint kiválaszthatja a feladat elvégzésére alkalmas járművet. A megállók listáját a térkép, valamint az alatta elhelyezkedő panel segítségével szerkesztheti a felhasználó. A feladatban felvezetett megállók különbözően vannak megjelenítve a térképen. A kiindulási pontot egy zöld, a célállomást pedig egy fekete-fehér kockás mintájú marker jelez. Ennek a két markernek az elhelyezése opcionális, csak a megállók megkülönböztetésére szolgálnak. Minden más megálló kék színnel van jelölve.

Egy megálló adatait csak szerkesztési módban változtathatja meg a felhasználó, amelyet az adott megállóhoz tartozó ceruza ikon jelöl. Egyszerre csak egy megálló nyitható meg szerkesztésre. Ha a felhasználó egy másikat is szeretne módosítani, arra csak akkor kerülhet sor, amikor a már elkezdett szerkesztési folyamatot befejezi. Amennyiben nincs megálló megnyitva szerkesztésre, akkor a bal egérgombbal kattintva a térképre a kiindulási pontot jelölő zöld

13. ábra. Feladat részleteinek megjelenítése



14. ábra. Feladat szerkesztése

marker, a jobb egérgombbal pedig a célpontot jelző marker helyezhető el.

Szerkesztési nézetben a felhasználó egy legördülő listából kiválaszthatja a megálló típusát, aminek megfelelően változik a megálló típusát jelző ikon. A típus kiválasztása mellett található egy másik ikon, amellyel megszabható, hogy a megálló kötelező-e, amelyet piros színnel jelöl, illetve szürkével, ha opcionális. A felhasználónak a térképen is meg kell jelölnie a megálló helyét, amelyet egy piros színű marker jelez. A marker a térképen a bal egérgombbal helyezhető el, illetve visszavonásra is lehetőség van, amit a jobb egérgombbal lehet megtenni. A változtatások elmentése után, normál nézetre visszatérve a térképen a piros marker kékre vált, ezzel is jelezve, hogy az adott megálló adatait nem lehet módosítani. A megálló a kuka ikonra kattintva törölhető. A változtatások elmentése után az alkalmazás egy rövid felugró üzenettel jelez vissza a felhasználónak, ha a módosítás sikeres volt, vagy hiba lépett fel.

A sofőrök útjai során felmerülő eseményeket egy külön oldalon tekintheti meg a szállításvezető, ahol a feladatokhoz hasonlóan, az események is listázva jelennek meg. Az eseményekre kattintva megtekinthetők azok részletei: az értesítést küldő sofőr neve mellett megjelenik az esemény típusa, mint például baleset, ellenőrzések, terelések. Ez alatt található az értesítés dátuma, helyszíne, valamint a sofőr telefonszáma. A *Task* címke mellett megjelenő linkre kattintva megtekinthetők a feladat részletei, amely során fellépett az adott esemény.

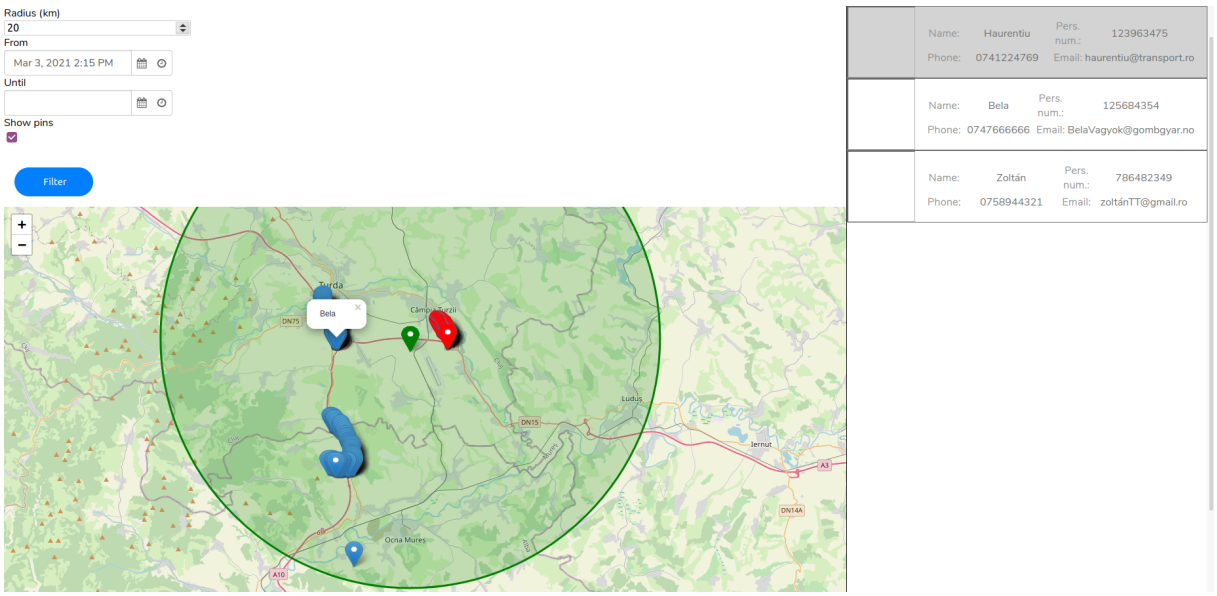
A *Live Map* segítségével a szállításvezetők valós időben követhetik nyomon, hogy a sofőrök

15. ábra. Az élő térkép segítségével a szállításvezetők valós időben követhetik nyomon a sofőrök útját.

éppen hol tartanak útjuk során, amely a 15. ábrán látható. A térkép folyamatosan frissül, így a sofőrök által naplózott legutolsó pozíció kerül megjelenítésre markerek segítségével, amelyekre kattintva megjelenítik az adott sofőr információit (például név, telefonszám, email cím).

A szállításvezetőnek a 16. ábrán látható oldalon lehetősége nyílik arra, hogy ellenőrizze, hogy egy adott pont bizonyos körzetében melyik sofőrök haladtak el. Ez abban az esetben lehet fontos, ha valamilyen probléma lépett fel a sofőrnél, itt megtekinthető az általa megtett útvonal.

A pontot, ahol történik a keresés a térképre kattintva állítható be, amely körül a sugárnak



16. ábra. Sofőrök helyzetének keresése időintervallum és helszín alapján.

megfelelő kör jelenik meg, ezzel elősegítve a keresési helyszín pontosabb meghatározását. A keresési kör sugara, valamint a kívánt időintervallum egy űrlap segítségével adható meg. A képernyő jobb oldalán egy panel helyezkedik el, amely a keresési feltételeknek megfelelő sofőrök adatait tartalmazó kártyák találhatóak. Amennyiben a keresés nem eredményezett találatot, a *No drivers found.* feliratot tartalmazza. Ha a felhasználó bekapcsolja a *Show pins* funkciót, akkor az adott hatókörön belül a megtalált sofőrök által megjelölt összes helyszín megjelenik a térképen kék markerek formájában. Egy sofőr kártyájára kattintva a hozzá tartozó markerek színe pirosra vált, kiemelve az érintett pontokat. A markerekre kattintva megjelenik felettük egy *popup* ablak, amely a hozzá tartozó sofőr nevét tartalmazza.

Következtetések és továbbfejlesztési lehetőségek

A Trucker alkalmazás fejlesztése során létrejött egy rendszer, amely két fő szerepkörre épül: szállításvezetők, valamint sofőrök. Az alkalmazásrendszer két kliensalkalmazást biztosít a felhasználók számára szerepkörüktől függően. A szállításvezetőknek a webes alkalmazás áll rendelkezésükre, amely segítségével létrehozhatják és kezelhetik a feladatokat, ellenőrizhetik a sofőrök aktuális és múltbéli helyzeteit, valamint értesülhetnek a sofőrökkel történt eseményekről. A mobil alkalmazás lehetőséget nyújt a sofőröknek a hozzájuk rendelt feladatok könnyű áttekintésére, valamint az útvonalak megállóinak személyre szabására. Az új feladatokról kapott, valamint a szállításvezetőknek küldött értesítések segítségével megvalósulhat a hatékony kommunikáció a két munkakört betöltő felhasználó között.

Annak ellenére, hogy a fejlesztés alatt sikerült megvalósítani a főbb funkcionalitások egy részét, továbbra is maradnak továbbfejlesztési lehetőségek:

- a sofőrök számára időjárás előrejelzés biztosítása az alkalmazáson belül, figyelmeztetés a rossz körülményekre;
- értesíteni a sofőrt, ha a hozzárendelt feladattal kapcsolatban változás történt (útvonal, megállók);
- a mobilos alkalmazás kibővítése útvonaltervezési, valamint navigációs funkciókkal;
- pontozási rendszer kialakítása, amelyben a sofőrök pontokat kaphatnának a feladatok elvégzéséért (például időben elszállított áru);
- weboldal kinézetét egységesíteni a mobil alkalmazással;
- a mobilalkalmazás kibővítése a szállításvezetőknek szánt funkcionalitásokkal;
- értesíteni a szállításvezetőt, ha a sofőr módosítja a megállók listáját vagy sorrendjét;
- a webalkalmazás kibővítése a szállított áruval kapcsolatos információk kezelésére, mint például a szállítmány típusa, mennyisége, tömege;
- lehetőséget nyújtani a szállításvezetőknek, hogy értékeljék a sofőröket, így visszajelzéseket adhatnak az alkalmazottaknak.

Hivatkozások

[1] Sean Amarasinghe. *Service worker development cookbook*. Packt Publishing Ltd, 2016.

[2] *Async Storage API dokumentáció*. URL: https://react-native-async-storage.github.io/async-storage/docs/api (elérés dátuma: 2021. máj. 4.)

[3] Lindsay Bassett. *Introduction to JavaScript object notation: a to-the-point guide to JSON*. " O'Reilly Media, Inc.", 2015.

[4] Kris M Bell, Darryl N Bleau és Jeffrey T Davey. *Push notification service*. US Patent 8,064,896. 2011. nov.

[5] Tim Berners-Lee, Roy Fielding, Larry Masinter és mások. *Uniform resource identifiers (URI): Generic syntax*. 1998.

[6] Maxwell Dayvson Da Silva és Hugo Lopes Tavares. *Redis Essentials*. Packt Publishing Ltd, 2015.

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

[8] *ESlint hivatalos dokumentáció*. URL: https://eslint.org/docs/about (elérés dátuma: 2021. máj. 3.)

[9] *Expo hivatalos dokumentáció*. URL: https://docs.expo.io (elérés dátuma: 2021. máj. 4.)

[10] *Expo Location API hivatalos dokumentáció*. URL: https://docs.expo.io/versions/latest/sdk/location/ (elérés dátuma: 2021. máj. 4.)

[11] *Expo Notification API hivatalos dokumentáció*. URL: https://docs.expo.io/versions/latest/sdk/notifications (elérés dátuma: 2021. máj. 4.)

[12] David Ferraiolo, D Richard Kuhn és Ramaswamy Chandramouli. *Role-based access control*. Artech House, 2003.

[13] *Fetch API hivatalos dokumentáció*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (elérés dátuma: 2021. máj. 3.)

[14] Roy Fielding és mások. *Hypertext transfer protocol–HTTP/1.1*. 1999.

[15] Cory Gackenheimer. *Introduction to React*. Apress, 2015.

[16] Arjun Guha, Claudiu Saftoiu és Shriram Krishnamurthi. „The essence of JavaScript". *European conference on Object-oriented programming*. Springer. 2010, 126–150. oldal.

[17] Hiroshi Ikeda. *Push service system and push service processing method*. US Patent 6,999,991. 2006. febr.

[18] *JWT hivatalos dokumentáció.* URL: https://jwt.io/introduction (elérés dátuma: 2021. máj. 2.)

[19] *LDAP hivatalos dokumentáció*. URL: https://ldap.com/ (elérés dátuma: 2021. máj. 5.)

[20] *Leaflet hivatalos dokumentáció.* URL: https://leafletjs.com/reference-1.7.1.html (elérés dátuma: 2021. máj. 3.)

[21] Patrick Connor Linskey és Marc Prud'hommeaux. „An in-depth look at the architecture of an object/relational mapper". *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 2007, 889–894. oldal.

[22] *Local Storage hivatalos dokumentáció.* URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage (elérés dátuma: 2021. máj. 3.)

[23] Azat Mardan. *Express. js Guide: The Comprehensive Book on Express. js*. Azat Mardan, 2014.

[24] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[25] Brett McLaughlin. *What is node?* " O'Reilly Media, Inc.", 2011.

[26] Mehdi Medjaoui és mások. *Continuous API Management: Making the right decisions in an evolving landscape*. O'Reilly Media, 2018.

[27] Peter Membrey és mások. *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Springer, 2010.

[28] Eric A Meyer. *CSS: The Definitive Guide: The Definitive Guide*. " O'Reilly Media, Inc.", 2006.

[29] Richard A Mollin. *RSA and public-key cryptography*. CRC Press, 2002.

[30] *MongoDB Node Driver hivatalos dokumentáció.* URL: https://docs.mongodb.com/drivers/node/current (elérés dátuma: 2021. máj. 3.)

[31] *Nodemon hivatalos dokumentáció.* URL: https://npmdoc.github.io/node-npmdoc-nodemon/build/apidoc.html (elérés dátuma: 2021. máj. 3.)

[32] *npm hivatalos dokumentáció.* URL: https://docs.npmjs.com/about-npm (elérés dátuma: 2021. máj. 3.)

[33] *ODM forrás.* URL: https://medium.com/spidernitt/orm-and-odm-a-brief-introduction-369046ec57eb (elérés dátuma: 2021. máj. 3.)

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

[35] *React Native hivatalos dokumentáció.* URL: https://reactnative.dev/docs/getting-started (elérés dátuma: 2021. máj. 4.)

[36] Prabath Siriwardena. „Base64 URL Encoding”. *Advanced API Security*. Springer, 2020, 397–399. oldal.

[37] Stoyan Stefanov. *React: up & running: building web applications*. " O'Reilly Media, Inc.", 2016.

[38] Vasan Subramanian. „React Router”. *Pro MERN Stack*. Springer, 2019, 233–265. oldal.

[39] M Thomson és P Beverloo. „Voluntary Application Server Identification for Web Push”. *IETF Tools* (2017).

[40] Sufyan bin Uzayr, Nicholas Cloud és Tim Ambler. „Mongoose”. *JavaScript Frameworks for Modern Web Development*. Springer, 2019, 309–375. oldal.