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

Truxi

Áruszállítók és ügyfeleik kommunikációját segítő platform

Szerzők:

Domokos Bernát
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év
Vad Bertalan
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Témavezetők:

dr. Simon Károly, egyetemi adjunktus
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar,
Codespring
Kiss Anna, szoftverfejlesztő,
Codespring
Nagy Roland, szoftverfejlesztő,
Codespring
Orbán Ákos, szoftverfejlesztő,
Codespring

CODESPRING
OUR CODE YOUR SUCCESS

Kivonat

A Truxi alkalmazás célja kapcsolatba hozni szabadúszó teherautó sofőröket olyan felhasználókkal, akik árut szeretnének szállíttatni.

A szoftverrendszer egy webes és egy mobil kliensből áll, melyek egy központi szerverrrel kommunikálnak. Az áruhirdetők a weboldalon vezethetik fel szállítandó javaikat, illetve ugyancsak itt nézhetik meg és fogadhatják el a sofőrök jelentkezését. A mobil alkalmazás segítségével a szállításra alkalmas járművet birtokló sofőrök könnyen böngészhetnek a hirdetések között, és ajánlatokat adhatnak a szállítmányokra. Továbbá, a sofőröknek lehetőségük van az elfogadott ajánlatok kezdő- és végpontjaiból álló optimális útvonalat generálni. Miután a meghirdető elfogadta az ajánlatot, a felek megkapják egymás elérhetőségeit, így közvetlen kapcsolatba léphetnek egymással.

A dolgozat bemutatja a projekt megvalósításához használt technológiákat, a webalkalmazás architektúráját, és egy rövid leírást is tartalmaz az alkalmazás használatáról, illetve a továbbfejlesztési lehetőségekről.

Tartalomjegyzék

Bevezető **1**

1. Szerepkörök és funkcionalitások **3**

1.1. Munkahirdető . . . 3

1.2. Sofőr . . . 3

1.3. Nem regisztrált felhasználó . . . 3

2. A szoftverrendszer **5**

2.1. Szerver . . . 5

2.1.1. Adatmodell . . . 5

2.1.2. Perzisztencia réteg . . . 9

2.1.3. Üzleti logika réteg . . . 10

2.1.4. API réteg . . . 13

2.1.5. Biztonság . . . 17

2.1.6. Hibakezelés és validáció . . . 18

2.1.7. Optimális útvonalgenerálás . . . 19

2.2. Web kliens . . . 20

2.2.1. Architektúra . . . 20

2.2.2. Komponensek . . . 21

2.2.3. API kliens . . . 22

2.3. Mobil kliens . . . 23

2.3.1. Komponensek . . . 23

2.3.2. Adatmodell . . . 25

2.3.3. API kliens . . . 25

3. Technológiák és eszközök **27**

3.1. Eszközök . . . 27

3.2. Szerveroldali technológiák . . . 27

3.3. Web technológiák . . . 29

3.3.1. React . . . 29

3.3.2. Typescript . . . 29

3.3.3. Mobx állapotmenedzser . . . 30

3.3.4. Dependency Injection . . . 30

3.3.5. Semantic UI React . . . 31

3.3.6. React Leaflet . . . 31

3.3.7. Formik . . . 31

3.4. Mobil technológiák 31

4. Az alkalmazás működése **33**

4.1. Web alkalmazás 33

4.2. Mobil alkalmazás 36

Következtetések és továbbfejlesztési lehetőségek **41**

Bevezető

Napjainkban gyakran felmerülő probléma, hogy valaki el szeretne szállíttatni valamit, - legyen az a régi, lecserélendő nagyszobakanapé, vagy akár egy sajátkezűleg lebontott régi ház törmeléke, - de önerőből nem tudja a szállítást megoldani. A hirdetőújságok felütése sem sokat segít, mivel eléggé ritkák a szállítási szolgáltatást nyújtók hirdetései ezekben. Az internethez fordulva is nehéz sikerrel járni, mivel kevés az ilyen jellegű keresőszolgáltatás. Romániában csak egy ilyen weboldal létezik (Cargopedia [16]), viszont az ő célközönségüket inkább a vállalkozások és cégek képezik, magánszemélyek számára az oldal nem a legideálisabb.

Az országban kevés teher- vagy furgonsofőr vállalja be, hogy szabadúszóként dolgozzon, mert nehéz klienseket találni és így nagy a kockázat. Ennek egyik oka éppen a fentebb említett probléma: egy kompetens platform hiánya. Legtöbben népszerű használttermék-elárusító oldalakon hirdetik szolgáltatásukat, mint például az OLX [46], de az ilyen portálok inkább termékek és nem szolgáltatások hirdetésére vannak optimalizálva.

Itt jön képbe a dolgozat tárgya: egy olyan alkalmazás, ami a két fentebb említett célcsoport (a munkahirdetők és sofőrök) között hivatott kapcsolatot kiépíteni. A Truxi segítségével munkahirdetőként bárki egyszerűen létrehozhat szállítmányokat, böngészheti az ezekre érkező ajánlatokat, majd kapcsolatba léphet azzal a sofőrrel, akinek az ajánlata a legjobban tetszett. Sofőrként kényelmesen lehet böngészni a meghirdetett szállítmányokat, ajánlatokat lehet tenni azokra, amelyek megfelelnek, felvezetni és menedzselni lehet saját teherautókat vagy furgonokat. Ezen kívül, sofőrként akár a megkapott munkáknak a felvevési és leadási pontjaiból álló optimális útvonalat is kigeneráltathatjuk az alkalmazással.

A szoftverrendszernek három alkotóeleme van: egy webalkalmazás, mely a munkahirdetőknek biztosít felhasználóbarát felületet, egy mobilos alkalmazás a sofőröknek (elérhető egyaránt Android-on és iOS-en), hogy bárhol is tartózkodjanak, könnyen használni tudják a rendszert és végül egy központi szerver, amely a két, előbb említett kliensalkalmazást köti össze és ugyanakkor felelős az adatok kezeléséért.

A dolgozat hátralevő része a következőképpen van strukturálva: az 1. fejezet bemutatja a rendszeren belüli szerepköröket és az ezekhez tartozó funkcionalitásokat. A 2. fejezet a szoftverrendszert és annak elemeit fogja elemezni. A 3. fejezetben a projekt megvalósításához használt technológiák és eszközök lesznek taglalva. A 4. fejezetben a rendszer működése lesz részletezve. A dolgozat végén (4.2. fejezet) találhatóak a következtetések és szó lesz a továbbfejlesztési lehetőségekről is.

A projekt megvalósítása 2019 júliusában kezdődött a Codespring által szervezett nyári gyakorlat alkalmával. A koordinálásért köszönettel tartozunk mentorainknak: Kiss Anna, Nagy Roland és Orbán Ákos szoftverfejlesztőknek a Codespring részéről. Ugyancsak

köszönetet mondunk témavezető tanárunknak, dr. Simon Károly egyetemi adjunktusnak. A projekt fejlesztése folytatódott az egyetemi *Csoportos projekt* nevű tárgy keretén belül, ahol csatlakoztak hozzánk Fehér Evelin, Magdó Ágnes-Etelka, Majoros Máté és Simon Péter évfolyamtársaink és egy fél évig besegítettek a fejlesztésbe.

1. Szerepkörök és funkcionalitások

A Truxi rendszer két fő felhasználói felülettel rendelkezik: egy web és egy mobilalkalmazással. A rendszerben három típusú felhasználó van, amelyek a szerepkörüktől függően használhatják az alkalmazást és különböző funkcionalitásokat érhetnek el.

Ebben a fejezetben a felhasználói szerepkörök és ezek funkcionalitásainak bemutatására kerül sor.

1.1. Munkahirdető

A munkahirdető számára a web kliens áll rendelkezésre, amelynek funkcionalitásait csakis regisztrált és sikeresen bejelentkezett felhasználóként tudja elérni: böngészheti a már meghirdetett saját szállítmányait, új szállítmányt hirdethet meg, böngészhet a szállítmányra érkező ajánlatok között és elfogadhat egyet közülük, kapcsolatba léphet az elfogadott ajánlatot tevő sofőrrel. A szállítmányok hatékony böngészése érdekében a rendszer két kategóriát különít el: elérhető vagy nem elérhető szállítmány. Új szállítmányok létrehozása során a munkahirdetőnek lehetősége van képfeltöltési opcióra vagy a felvevési és leadási helyszín pontos megadására interaktív térkép használatával.

1.2. Sofőr

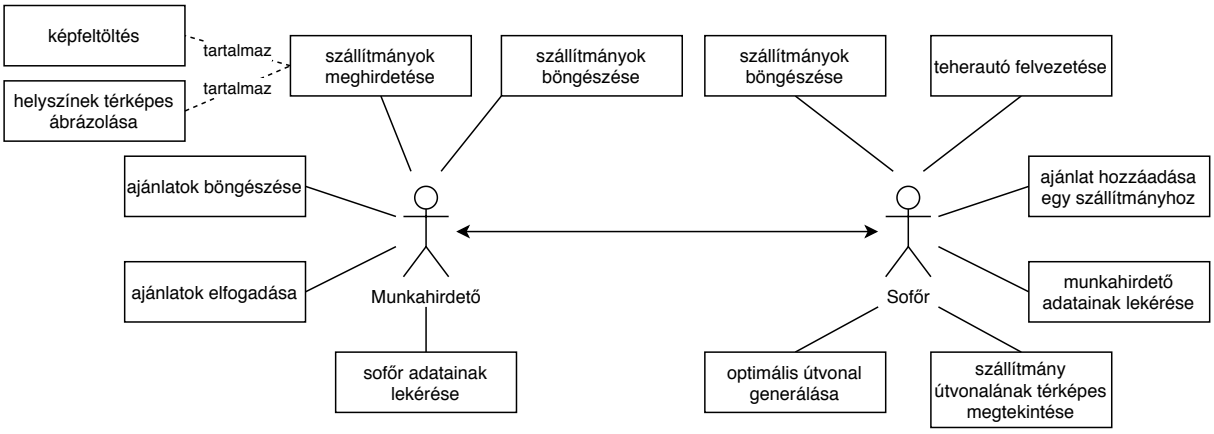
A sofőr a mobilalkalmazás segítségével tudja használni a rendszer által nyújtotta szolgáltatásokat. A sofőr funkcionalitásait csak egy regisztrált és sikeresen bejelentkezett felhasználó érheti el. A bejelentkezett sofőr böngészhet a munkahirdetők által közzétett szállítmányok között, és ajánlatot tehet az elérhető szállítmányokra. Az ajánlat megtételéhez a sofőrnek meg kell adnia egy árat és egy járművet, aminek a kapacitása nagyobb, mint a szállítmány súlya. Az ajánlat elfogadása esetén a sofőrnek elérhetővé válik a munkahirdető telefonszáma a két fél közötti kommunikáció megteremtésének érdekében.

Az elfogadott szállítmányok kiszállítására a sofőrnek lehetősége van optimális útvonalterv generálására is, ami az áru felvevési és leadási csomópontjai mellett figyelembe veszi a teherautó teherbírását is. A sofőr egy listát kap eredményül, ami a szállítmányok felvevési és leadási csomópontjainak sorrendjét tartalmazza.

1.3. Nem regisztrált felhasználó

A nem regisztrált felhasználó számára a felület betekintést nyújt az alkalmazás működéséről.

A weboldalon a nem regisztrált felhasználó nem kap semmilyen hozzáférést a munkahirdető funkcionalitásaihoz, hanem csak egy kezdőoldal áll rendelkezésére.



1. ábra. Funkcionalitások

A mobilalkalmazásnál a vendég felhasználónak rendelkezésére áll egy biztonságos böngészés a meghirdetett szállítmányok között és ezek részletes leírásának megtekintése, melynek célja, hogy felkeltse a felhasználó érdeklődését. Ha a vendég felhasználó úgy dönt, hogy szeretné használni az alkalmazást, akkor rendelkezésére áll a regisztrációs funkció, ahol a megfelelő információk megadásával létrehozhatja a rendszerben a felhasználói fiókját. A felhasználó szerepkörét a rendszerben az határozza meg, hogy a két kliensalkalmazás közül melyiken hajtódik végre a regisztráció: munkahirdető, ha web kliensen keresztül vagy sofőr szerepkör, ha a mobilalkalmazáson keresztül történik. A bejelentkési funkció csak akkor sikeres, ha a felhasználó a szerepköréhez rendelt megfelelő kliensalkalmazást használja.

2. A szoftverrendszer

A szoftverrendszert 3 fő komponens alkotja: egy Spring keretrendszer segítségével megírt központi *szerver* (2.1. fejezet), egy React.js *webalkalmazás* (2.2. fejezet), illetve egy Androidon és iOS-en egyaránt működő *mobilalkalmazás*, ami React Native-val lett megvalósítva (2.3. fejezet). A rendszer adatait egy MySQL adatbázisba menti le a rendszer.

A kommunikáció a szerver és a kliensalkalmazások között a szerver által publikált *Application Programming Interface*-en (API) keresztül történik. Ez az API betartja a *REpresentational State Transfer* (REST) architektúrai stílus által diktált konvenciókat. Napjaink webszerverei között a REST egy széles körben elterjedt minta, amit Roy Fielding dolgozott ki és publikált 2000-ben a doktori disszertációjában [55]. Mivel a szerver API-ja RESTful módon lett megírva, automatikusan örökölte a minta jellemzőit, amelyek közül néhány fontosabb: állatpotmentesség, kliens-szerver kommunikáció, különálló rendszerek rétegelése.

2.1. Szerver

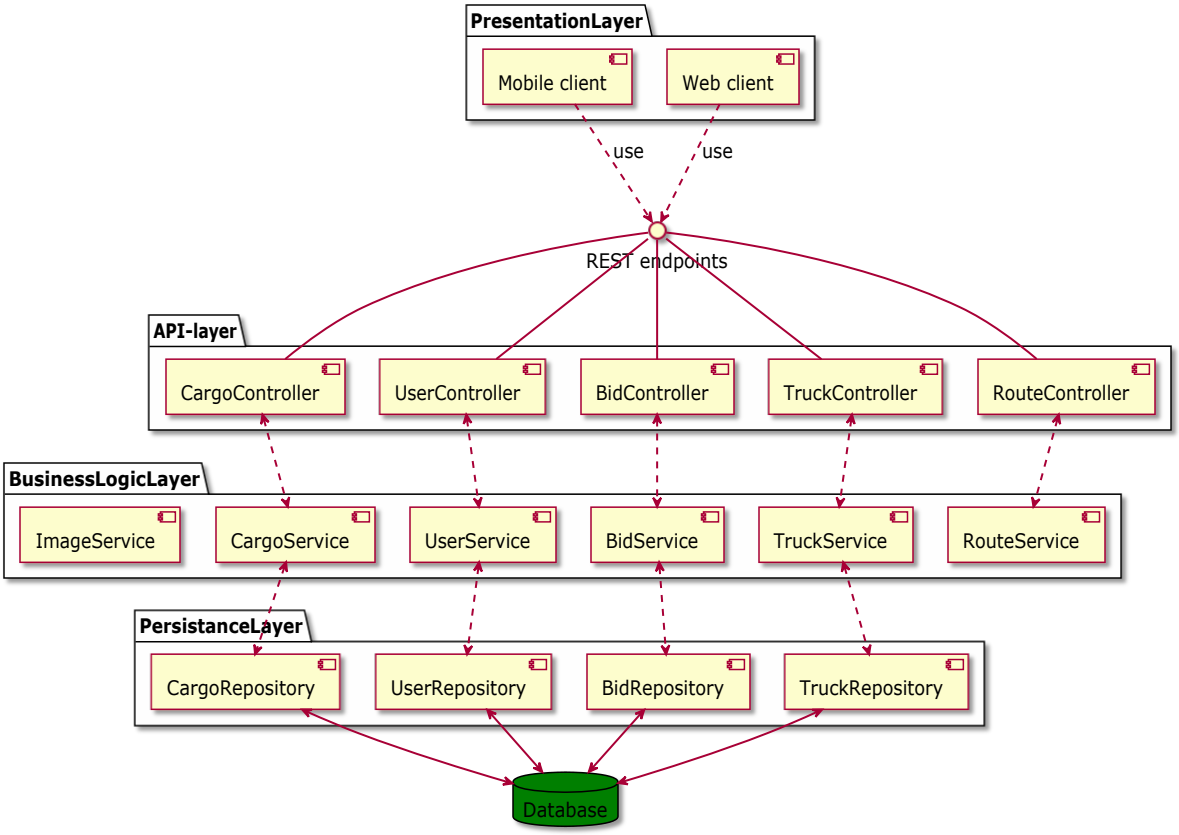
A szerveralkalmazás a *többrétegű architektúra* mintát követi, ami miatt az adatmenedzsment, adatfeldolgozás és adatkiszolgálás logikailag és fizikailag is szét lett választva. Ez egy olyan kód születését eredményezte, mely jól szétválasztható, emiatt a rétegeket lehet különállóan módosítani, az architektúra flexibilis, könnyen bővíthető és tesztelhető.

Három elkülönített réteg van jelen, egy API-réteg, mely a kliensalkalmazásoktól érkező HTTP kérések és válaszok kezelésére hivatott, egy üzleti logika réteg, (szervíz-réteg, business logika réteg), mely a komplexebb feladatokat és az adatmodell változtatásokat végzi és egy perzisztencia réteg, melynek feladata az adatok keresése és mentése. A rétegek egyik fontos tulajdonsága, hogy zártak [64], ami abban nyilvánul meg, hogy minden réteg kizárólag a szomszédjával kommunikál (lásd 2 ábra) és nincsenek "ugrások".

Ami a rétegek közti kommunikációt illeti, a szervíz-réteget az API-réteg hívja egy-egy HTTP kérés beérkezése után, majd - a szügséges logika elvégzése után - továbbhív a perzisztenciaréteg felé, ami az adatbázis-műveleteket megvalósítja. A válasz ugyanezen a láncon jut vissza az API-hoz, aki HTTP válaszon keresztül továbbítja ezt a kliensnek.

2.1.1. Adatmodell

A 3 ábrán látszanak a rendszer által használt modellosztályok. A Domain Model osztályai mind *Plain Old Java Object*-ek (POJO) [36], ami szükséges feltétele annak, hogy a szerver által használt *Object Relational Mapping* (ORM) keretrendszer (a Spring Data JPA[60]-t

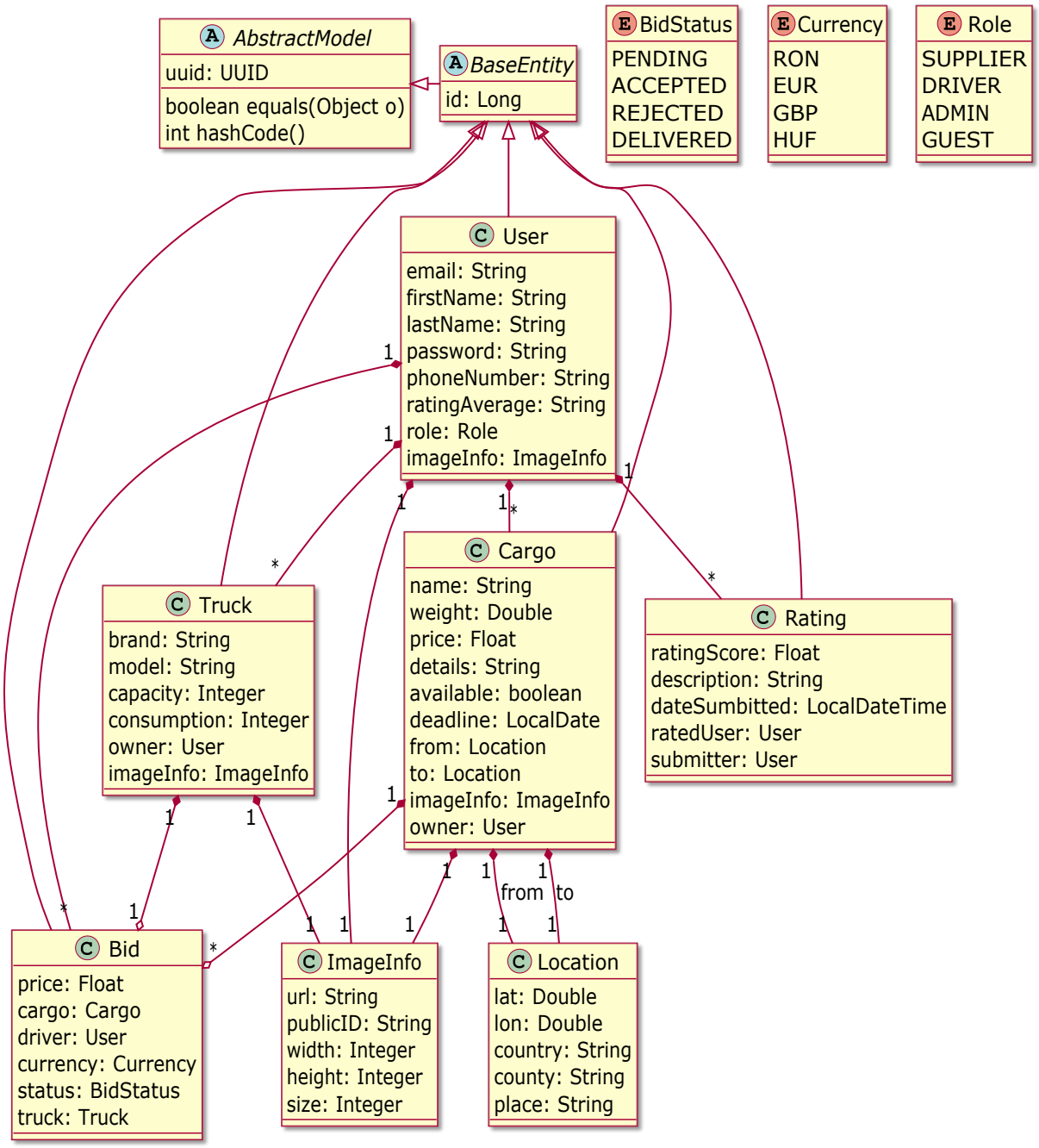


2. ábra. A szerver komponens diagrammja. A komponensek izoláltak, könnyen karbantarthatóak, tesztelhetőek és cserélhetőek.

implementáló *Hibernate* [31]) meg tudjon feleltetni minden modellosztálynak egy táblát az adatbázisban (lásd a 4. ábrán). Ebben a fejezetben a modellosztályok lesznek részletezve.

Az összes **model** csomagban található entitásosztály örökli a **BaseEntity**, illetve ezen keresztül az **AbstractModel** absztrakt ősosztályokat, amint a 3 ábrán is látszik. Ennek az öröklődésnek köszönhetően minden modellosztály tartalmazni fog egy rendszerszinten egyedi *Universally unique identifiert*-t (UUID) és egy ID-t, ami az adatbázisbeli elsődleges kulcsát fogja tartalmazni.

Az egyik központi entitás a rendszerben a *Szállítmány* (`Cargo`). A Szállítmány a legfontosabb entitás a rendszerben, minden funkcionalitás erre épül. Egy Szállítmány adattagjai közé tartozik a neve (*name*), a súlya (*weight*), a részletei (*details*), a határideje (*deadline*). Minden szállítmány esetében el kell tároljuk, hogy honnan, illetve hova kell eljusson. Erre a célre beágyazott adattagokat használunk (*from* és *to*), melyeknek a típusa a beágyazható *Helyszín* (`@Embeddable Location`). A beágyazottság az adattagokra helyezett `@Embedded` [23] annotáció segítségével történik. Azért szükséges, mert így nem jön létre külön tábla a helyszín típusnak, hanem az adattagjai leképződnek attribútumoknak a `Cargo` táblában. Az `@AttributeOverrides` annotációval [12] van meghatározva, hogy a létrejövő attribútumok milyen nevet viseljenek. Minden helyszín esetében eltárolandó a földrajzi hosszúság és



3. ábra. A rendszer által használt modellosztályok. Az entitások globálisan az `AbstractModel`-től örökölt *uuid*-val vannak beazonosítva.

földrajzi szélesség, egy országnév, egy városnév és opcionálisan egy tartománynév is. Minden *Szállítmányról* nyilvántartjuk egy *elérhető* (*available*) adattagban, hogy a sofőrök számára van-e lehetőség még ajánlatot tenni erre a szállítmányra vagy sem. A szállítmány tulajdonosának az információit is eltároljuk minden szállítmány esetében a *tulajdonos* (*owner*) adattagban, melynek típusa *Felhasználó* (*User*), ami a későbbiekben lesz részletezve. Fontos adattag még az *ár*a (*price*), melyet a tulajdonosnak a létrehozáskor kell megadnia, mint irányár. Ennél az árnál kisebb ajánlatokat tudnak majd tenni a sofőrök. Az *ár* mellé el van tárolva a pénznem is, amiben a munkahirdető azt feltüntette. Az applikáció által jelenleg támogatott

valuták: *RON*, *EUR*, *GBP*, *HUF*.

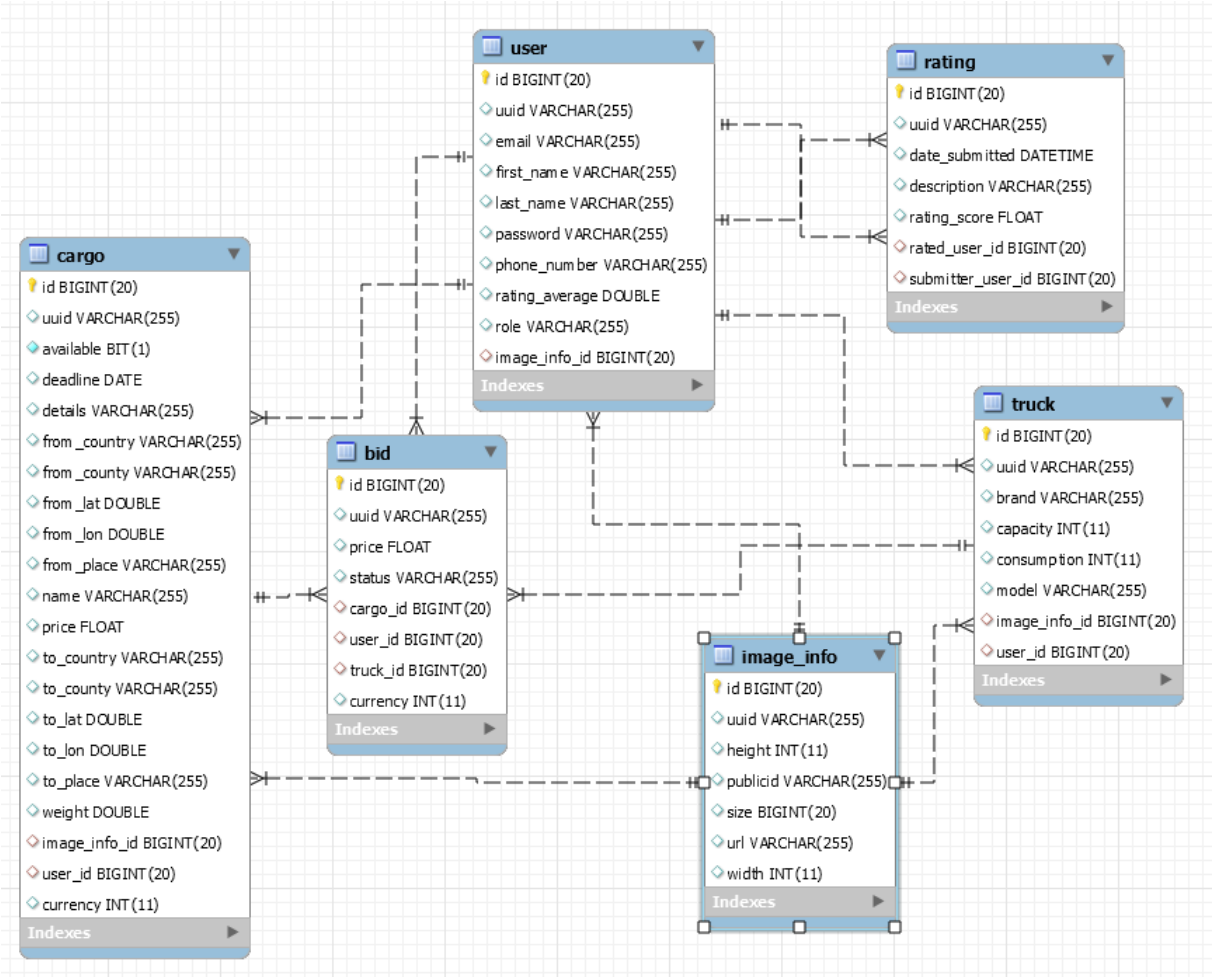
A második központi entitás a rendszerben a *Felhasználó* (`User`). A felhasználók regisztráláskor meg kell adják *család-* és *keresztnevük*et (*firstName*, *lastName*), egy *telefonszám*ot (*phoneNumber*), illetve egy *email cím*et és egy *jelszó*t (*email* és *password*). Az utóbbi kettő lesz használatos a felhasználó bejelentkeztetésekor. Az applikáció lehetőséget teremt a felhasználók értékelésére, így minden felhasználó esetében perzisztálva lesz egy *értékelés átlag* (*ratingAverage*). Fontos továbbá, hogy el legyen tárolva minden felhasználó *szerepköre* (*role*). Egy felhasználó lehet *sofőr* (*driver*), *munkahirdető* (*supplier*), *adminisztrátor* (*admin*) és *vendég* (*guest*).

A Truxi alkalmazásban a felhasználók tudják egymást értékelni, hogy visszajelzést adjanak például az elvégzett munka minőségéről. A rendszerben egy értékelést az *Értékelés* (`Rating`) osztály ír le, amely rendelkezik egy *értékeltPontszám*-mal (*ratingScore*). Ez a pontszám a $[0.0, 5.0]$ intervallumból vehet fel értéket, egy opcionális *leírás*-sal, és egy dátum típusú adattaggal (*dateSubmitted*). Ezen kívül tartalmaz referenciát az értékelő felhasználóról és az értékelt felhasználóról is.

Mielőtt a sofőr ajánlatokat tehetne, fel kell vezetnie *Teherautókat* (`Truck`). A teherautók rendelkeznek a *márka* (*brand*), *modell* (*model*), *teherbírás* (*capacity*) és *fogyasztás* (*consumption*) adattagokkal. Ezen kívül eltároljuk a járművek *tulajdonos*át (*owner*) egy *Felhasználó* típusú adattagban.

Az eddig említett három központi entitáshoz (szállítmányhasználó, jármű) hozzárendelhető egy-egy kép is, így mindhárom entitás esetében el van tárolva még egy *KépInformáció* (`ImageInfo`) típusú adattag. Maguk a képek nincsenek eltárolva az adatbázisban, hanem egy Cloudinary nevű felhőszolgáltatásba [17] vannak feltöltve. Egy képről eltárolt információk közé tartozik egy *URL* (*url*), ahonnan a képet le lehet tölteni a felhőből, egy *publikus kulcs* (*publicID*), ami beazonosít minden képet a felhőben, a kép *magassága* és *szélessége* (*height* és *width*).

Az ajánlatokat reprezentáló központi entitás a rendszerben az *Ajánlat* (`Bid`). Mikor a sofőrök ajánlatokat hoznak létre, az adatbázisban egy ilyen *Ajánlat* típusú rekord lesz lementve. Egy *Ajánlat* tartalmazza, hogy ki helyezte el, vagyis a *sofőr* (*driver*), ami *Felhasználó* típusú, egy *Szállítmány* típusú adattag: *szállítmány* (*cargo*), mely referenciát tárol a szállítmányról, amire az ajánlatot tették. Minden ajánlat magába foglalja azt az *ár*at és *pénznem*et (*price* és *currency*), amennyi pénzt a sofőr szeretne kapni a munkáért. Ez az érték kisebb vagy egyenlő, mint a *szállítmány*ban megjelenő irányár. A sofőrök kötelezően meg kell adjanak egy *jármű*vet (*truck*), amivel a szállítást véghez fogják vinni. Minden ajánlat rendelkezik egy *állapot*tal (*status*). A lehetséges állapotok: *függő* (*pending*), *elfogadott* (*accepted*), *elutasított* (*rejected*) és *elszállított* (*delivered*). Az ajánlat létrehozáskor alapértelmezetten *függő* állatpotba kerül.



4. ábra. Az adatbázis sturktúrája. A Java modellosztályok le vannak képezve egy relációs adatbázis tábláivá a *Hibernate* által.

2.1.2. Perzisztencia réteg

A szerver a Spring keretrendszer technológiáinak segítségével volt megírva, így a perzisztenciáért a Spring Data [33] projekt felelős. A perzisztencia réteg által használatos Spring Data JPA [60] része a Spring Data családnak és egy extra absztrakciós réteget biztosít a használt JPA implementáció felett (a Truxi projekt esetében a Hibernate [31] felett). A Spring Data JPA jelentősen megkönnyíti a fejlesztést, mivel minden entitásra biztosítja az alap *Create-Read-Update-Delete* (CRUD) műveletek implementációját, így rengeteg redundáns kódrészletet kiküszöbölhetünk. A forráskód szintjén csak annyit kellett tenni ennek az eléréséhez, hogy minden entitásnak létre kellett hozni egy-egy *Java interface*-t, amelyek kiterjesztik a Spring Data csomagbeli generikus *CrudRepository* interface-t, ami két sablonparamétert vár. Az egyik az entitás típusa, a másik az elsődleges kulcsnak a típusát jelöli. Ezt követően, ha akár üresen is hagyjuk ezeket az interface-eket, az alapműveletek biztosítva lesznek a keretrendszer által. Az alkalmazás futásakor a rendszer biztosít *Spring Bean*-eket, amelyek az *@AutoWired* [61] annotációval válnak beinjektálhatóvá. Ezek a bean-ek a Spring által biztosítot Inversion-of-Control (IoC) konténerben találhatóak és a Spring keretrendszer

menedzseli őket. A Spring Data által biztosított bean-ek beinjektálása után a perzisztencia műveletek elvégezhetőek a megfelelő metódusok hívásával.

Gyakran előfordult, hogy a CRUD műveleteknél komplexebb lekérdezésekre volt szükség. A Spring Data erre is biztosít megoldást. A Truxi rendszerben minden entitás az **AbstractModel**-től örökölt *uuid* adattaggal van beazonosítva, ami ugyan le van mentve az adatbázisba, de nem elsődleges kulcs. Mivel a Spring Data által kigenerált CRUD metódusok közül az azonosító alapján kereső metódus (*findById*) csak az elsődleges kulcs alapján tud keresni, ezért szükséges minden entitás esetén egy *uuid* alapján kereső függvény is. Ezt úgy lehet kódból elérni, hogy megírjuk a *findByUuid* metódust a megfelelő interface-ben és a Spring Data a függvény fejléce alapján legenerálja az implementációt. Hasonlóan lehet eljárni a többi lekérdezés létrehozásánál is. Az összetettebb lekérdezéseket a megfelelő annotáció segítségével JPQL query-ként is meg lehet adni az interfészek metódusainak a szintjén.

A perzisztencia réteg feladata kizárólagosan az adatbázissal való kommunikáció. A logikát és ellenőrzéseket a üzleti logika réteg végzi, amely rendelkezik a perzisztenciát végző objektumokkal (Dependency Injection), amelyeknek a megfelelő metódusain keresztül jön létre a kommunikáció a két réteg között. A szerver kódbázisában a **repo** csomagon belül találhatóak a fent említett interface-ek. Az összes központi entitásnak (szállítmány, felhasználó, jármű, ajánlat) létre lett hozva egy ilyen interface. Ha az üzleti réteg egy adott típusú entitással szeretne perzisztencia műveletet végezni, akkor csak az annak megfelelő repository komponenst kell beinjektálnia.

2.1.3. Üzleti logika réteg

A kódbázisban a **service** csomagban található minden, ami ehhez a réteghez kapcsolódik. A perzisztencia réteghez hasonlóan, itt is a szolgáltatások entitásonként külön vannak választva. Egy entitás szolgáltatása egy *Java interface*-ből áll, ahol a szervízhez tartozó metódusok szignatúrája van megadva és az ehhez tartozó konktrét osztályból, ami implementálja az interface-t. Több indokból is hasznos volt szétválasztani az interface-eket az implementációktól: az Inversion-of-Control könnyebben megvalósítható, a tesztelése a szervízeknek könnyebbé válik, illetve megengedhető lesz, hogy egy interface-nek több implementációját is elkészítsük. A konkrét osztályból futáskor létrejön egy *Spring bean*, amit bármelyik osztály felhasználhat, miután beinjektálta. Az üzleti réteget az API-réteg hívja, viszont létezik kommunikáció különböző szervízek között is, tehát egy szervíz-implementáció injektálhat akár egy másikat is.

Az üzleti logika réteg az a része a szervernek, ahol bonyolultabb logikát tartalmazó kódrészletek vannak. Ez sokszor abban nyilvánul meg, hogy a hívások a perzisztenciaréteg felé és az ellenőrzések váltakozva történnek, vagy több perzisztenciahívás található ugyanabban

a metódusban, így bárhol hiba merül fel, minden eddigi módosítást vissza kell állítani. Az ilyen eljárások annotálva vannak a `@Transactional` annotációval [65], hogy törzsük tranzakcióban fusson le. A továbbiakban a többi közül kiemelendő szervíz-metódusok kerülnek nagyító alá, entitásonként csoportosítva. Amik nincsenek felsorolva, azok az egyszerű CRUD műveleteket elvégző metódusok, amik általában csak minimális ellenőrzéseket végeznek, majd továbbhívnak a perzisztencia réteg felé.

A üzleti logika réteg **szállítmány**ok kezelését megvalósító komponensének fontosabb funkcionalitásai:

- A `List<Cargo> findCargoesByUserUuid(UUID supplierUuid)` metódus egy *uuid*-val azonosított felhasználónak az összes létrehozott szállímányát téríti vissza. A metódus ellenőrni, hogy a beazonosított felhasználó munkahirdető szerepkörrel rendelkezik-e.

- A `List<Cargo> findAvailableCargoes()` metódus visszatéríti az összes még elérhető szállítmányt, aminek még nem került sofőr.

- A `void setAvailableFlag(UUID cargoUuid, boolean value)` eljárás adott *uuid*-val rendelkező szállítmány objektumnak beállítja az *available* mezőjét a szintén paraméterként kapott *Igaz* vagy *Hamis* értékre.

A rendszer **felhasználó**it menedzselő komponens kiemelendő eljárásai:

- A `void registerUser(User user)` eljárás először ellenőrzi, hogy a paraméterként kapott felhasználó email címe nem-e foglalt, majd a felhasználó jelszavát BCrypt hashelő függvénnyel [15] titkosítja és végül egy hívással a perzisztencia réteg felé továbbítja az új felhasználó objektumot, ahol megtörténik annak mentése.

- A `List<Rating> findRatingsOfUser(UUID userUuid, boolean hasDescription, Integer page, Integer size)` függvény visszatéríti a *userUuid*-val beazonosított felhasználó értékeléseit. A *hasDescription* flag mondja meg, hogy a visszatérített értékelések tartalmazzanak-e leírást. A *page* és *size* paraméterek segítségével lapozást lehet konfigurálni.

- A `@Transactional Rating submitRating(UUID ratedUserUuid, Rating newRating)` metódus hívásával a *ratedUserUuid*-val rendelkező felhasználóra a *newRating* értékelést lehet elhelyezni. A függvény kiolvassa a Spring által biztosított `SecurityContextHolder` osztály segítségével a kérést intéző felhasználó azonosítóját és szerepkörét, azért, hogy további ellenőrzéseket tudjon elvégezni. Ellenőrzi, hogy a felhasználó nem-e próbálja meg önmagát értékelni, illetve, hogy az értékelő és az értékelt

felhasználók ne legyenek azonos szerepkörűek, ugyanis a rendszerben ugyanolyan szerepkörű felhasználóknak nincs lehetőségük értékelni egymást. Ha az értékelő már értékelte korábban az értékelt felhasználót, ez az értékelés fölülíródik, különben beállítja a *submitter* és *ratedUser* (2.1.1) adattagokat. Minden értékelés során az értékelt felhasználó értékelési átlaga (*ratingAverage* mező) frissül.

A business-logika réteg **ajánlat**okért felelős komponense számos fontos metódussal rendelkezik:

- A `Bid createOrUpdateBid(UUID contestedCargoUuid, Bid bid)` metódus elhelyezi a paraméterként érkező ajánlatot a *contestedCargoUuid* azonosítójú szállítmányra. Ellenőrzésre kerül a célszállítmány és a kiválasztott jármű létezése a rendszerben, a szállítmány elérhetősége (*cargo.available*), az, hogy a kiválasztott jármű tulajdonosa valóban az-e, aki az ajánlatot elhelyezte (a sofőr kiléte a *SecurityContextHolder* osztály segítségével van kiolvasva a kontextusból). Továbbá ellenőrzésre kerül, hogy az ajánlott ár (*bid.price*) kisebb, mint az eredeti irányár és a kiválasztott jármű (*bid.truck*) teherbíró képessége elég nagy, hogy elvigye a szállítmányt. Ha már helyezett el ez a sofőr ajánlatot erre a szállítmányra, akkor a régi ajánlat módosul, különben egy új rekord jön létre az adatbázisban.

- A `List<Bid> findBidsByCargoUuid(UUID cargoUuid)` metódus segítségével egy munkahirdető felhasználó le tudja kérni egy adott szállítmányára az összes beérkező ajánlatot. A munkahirdető a tulajdonosa kell legyen a szállítmánynak.

- A `List<Bid> findOwnBidsWithStatus(BidStatus status)` metódus visszatéríti egy sofőr számára az összes elhelyezett ajánlatát, amely a *status* paraméterben megadott állapottal rendelkezik.

- A `@Transactional void acceptBid(UUID cargoUuid, UUID bidUuid)` komplex eljárással tudnak munkahirdetők elfogadni egy ajánlatot egy szállítmányukra. A *cargoUuid* paraméter mondja meg, hogy melyik szállítmány az alany, a *bidUuid* meg az elfogadandó ajánlatot azonosítja be. Validálásra kerül a megfelelő entitások létezése, az, hogy az ajánlat még függőben van-e, illetve az, hogy a kérést intéző munkahirdető valóban a szállítmány tulajdonosa. A függvény elfogadott állapotba helyezi az ajánlatot, majd elutasított állapotba az összes többi ajánlatot erre a szállítmányra. Végül a szállítmányokért felelős szervíz-osztály segítségével beállítja nem elérhetőre az adott szállítmányt (2.1.3).

- A `Bid findLowestBidOfCargo(UUID cargoUuid)` függvény visszatéríti a

cargoUuid azonosítójú szállítmányra beérkező ajánlatok közül a legalacsonyabb értékűt.

- A `void markBidAsDelivered(UUID bidUuid)` metódussal sofőrök tudják jelezni a rendszernek, hogy sikeresen leszállítottak egy árut. Eredetileg a *bidUuid* azonosítójú ajánlat lett elfogadva a szállítmány tulajdonosa által, az eljárás ezt az ajánlatot helyezi leszállított állapotba. Ellenőrzésre kerül előbb az ajánlat létezése, az, hogy elfogadott állapotban van-e, illetve az is, hogy a sofőr valóban tulajdonosa az ajánlatnak.

2.1.4. API réteg

A szerver API-ja egy *RESTful* szolgáltatás [54], tehát a szerver által kiszolgált erőforrások előre meghatározott, a kliensek által is ismert URI-kon (*Uniform Resource Identifier*) elérhetőek.

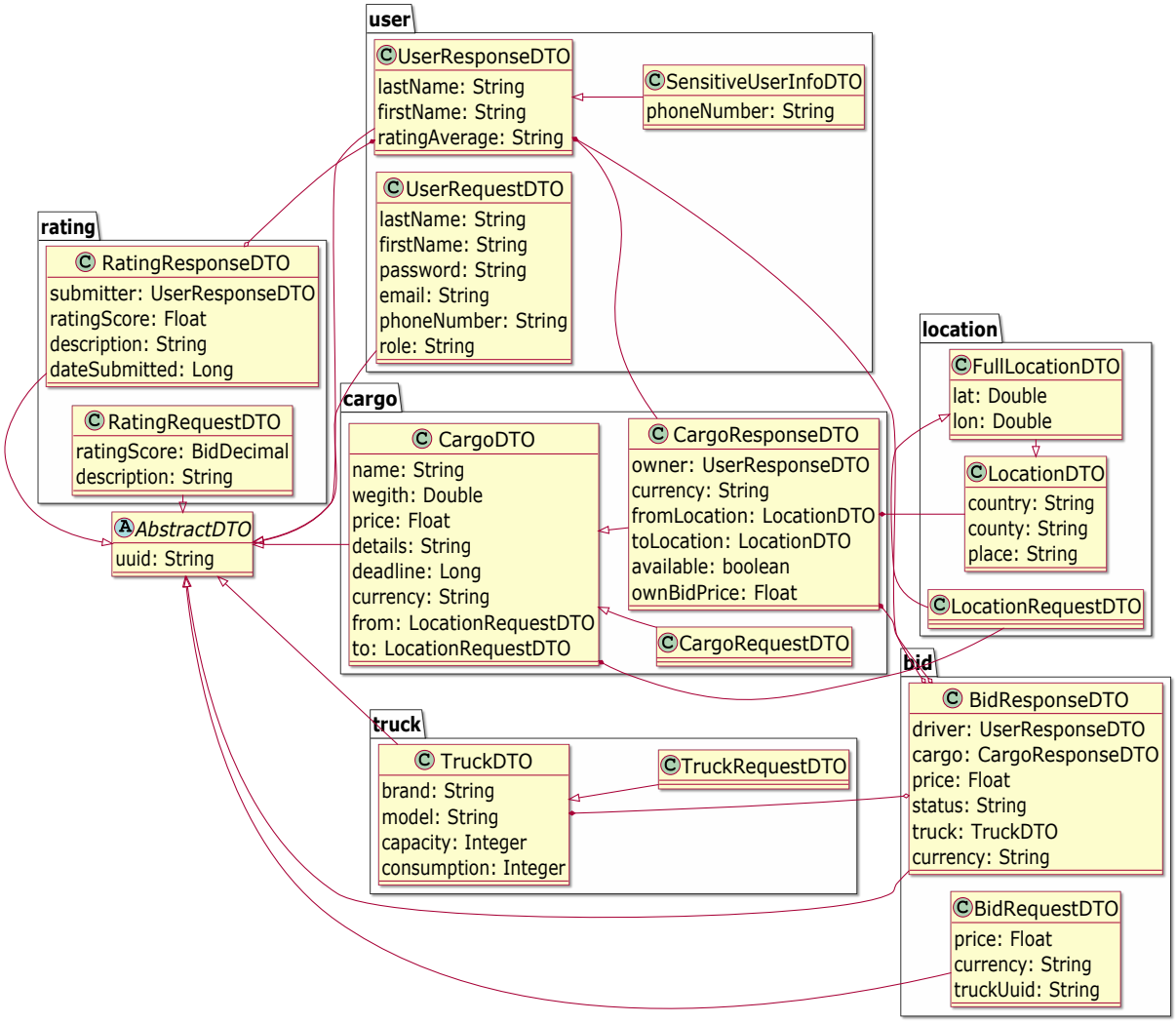
Az API-réteg, másnéven kontroller-réteg, a szerver legkülsőbb rétege. Az API-réteg feladata a kliens-alkalmazások által küldött HTTP-kérések fogadása és ezek továbbítása az üzleti logika réteg felé, majd a művelet sikerességét vagy sikertelenségét magába foglaló HTTP-válasz kiküldése.

A kliens-alkalmazások és a szerver közötti adatcsere DTO minta (Data Transfer Object [22]) alapján történik, ahogy a 5 ábrán is látszik. Minden modellosztálynak megfelel egy vagy több DTO osztály és mindenik DTO osztály más és más reprezentációját írja le egy entitásnak. Ezért a DTO osztályok sokszor csak részleges változatai a modellosztályoknak, így minimalizálódik az adatforgalma a szoftverrendszernek, mert a küldendő adatmennyiség kisebb. Ezenfelül, a DTO osztályok kizárólag szerializálható adattagokkal rendelkeznek és használatukkal a projekt komponensei egymástól jobban függetlenné válnak. A DTO osztályok halmaza a **restapi/dto** csomagban található.

Az adatcsere JSON formátumban [37] történik, kivéve a képek esetében, amik Multipart Form Data [44] típusban érkeznek a szerverre. Fontos, hogy a kliensek ismerjék a szerver által fogadott és küldött DTO osztályokat, hogy a JSON formátumban levő adatok struktúrája kompatibilis legyen a kommunikáció során.

Az API réteg csakis DTO objektumokkal kommunikál, ezért mielőtt meghívná a business-réteg valamelyik metódusát, át kell alakítania a DTO-kat az üzleti réteg által is használt modell-osztálybeli megfelelőivé. Itt jönnek képbe az *assembler* osztályok, amelyek statikus metódusokkal rendelkeznek és képesek bármelyik DTO-t modellé- és visszaalakítani. Az assembler osztályok a **restapi/assembler** csomagban találhatóak és a jobb átláthatóság miatt entitásonként szét vannak bontva.

A kéréseket az entitásoknak megfeleltetett kontroller osztályok várják, melyek annotálva



5. ábra. A kommunikációra használt *Data Transfer Object*-ek (DTO-k). A DTO-kat az assembler osztályok alakítják modellosztályokká és vissza.

vannak a *Spring Boot Web Starter* csomag `@RestController` annotációjával. Ezeknek az osztályoknak a metódusai be vannak konfigurálva, hogy milyen URI-kon, milyen HTTP metódusra figyeljenek. Az alfejezet hátralevő részében a szerver REST API végpontjai lesznek felsorolva. A kontroller osztályok a **restapi/controller** csomagban találhatóak.

A **szállítmányok**at kezelő végpontokra érkező kéréseket a *CargoController* osztály metódusai várják.

- `GET /api/cargoes`. Visszatéríti az összes elérhető szállítmányt. Bármilyen szerepkör számára elérhető, még nem regisztrált vendégeknek is.
- `GET /api/cargoes/{cargoUuid}`. Visszatéríti a *cargoUuid* azonosítójú szállítmányt. Bármilyen szerepkör számára elérhető.
- `POST /api/cargoes`. Szállítmányt hoz létre, majd visszatéríti azt. Az ide intézett kérés testében kell megadni az új szállítmány adatait. Munkahirdetők számára elérhető.

- `PUT /api/cargoes/{cargoUuid}`. A megadott azonosítójú szállítmányt módosítja. Az ide intézett kérés testében meg kell adni az összes adattag értékét, még ha nem is változik az érték. Munkahirdetők számára elérhető.
- `PATCH /api/cargoes/{cargoUuid}`. A megadott azonosítójú szállítmányt módosítja részlegesen. Csak azok az adattagok fognak frissülni, amik meg lettek adva a kérés testében. Munkahirdetők számára elérhető.
- `DELETE /api/cargoes/{cargoUuid}`. A megadott azonosítójú szállítmányt törli a rendszerből. Munkahirdetők számára elérhető.
- `GET /api/cargoes/{cargoUuid}/image`. A megadott azonosítójú szállítmányhoz csatolt kép az elérhetőségét és tulajdonságait téríti. Bármilyen szerepkör számára elérhető.
- `PUT /api/cargoes/{cargoUuid}/image`. A megadott azonosítójú szállítmányhoz lehet képet feltölteni. Ha már van kép feltöltve hozzá, akkor az fölülíródik. Munkahirdetők számára elérhető.

A **felhasználó**k menedzsmentjével kapcsolatos végpontok működése a *UserController* osztály metódusaiban van leírva.

- `GET /api/users/account-info`. Visszatéríti a bejelentkezett felhasználó számára a saját információit. Vendégen kívül bármelyik szerepkörű felhasználó hívhatja.
- `GET /api/users`. Visszatéríti a felhasználók listáját. Adminisztrátorok számára elérhető csak.
- `GET /api/users/{userUuid}`. Visszaadja a beazonosított felhasználó adatait. Bármilyen szerepkör számára elérhető.
- `GET /api/users/{userUuid}/cargoes`. Visszaadja a beazonosított felhasználó létrehozott szállítmányait. Bármilyen szerepkör számára elérhető.
- `POST /api/users`. Beregisztrál egy új felhasználót a rendszerbe a kérés testében található adatokkal. Bármilyen szerepkör számára elérhető.
- `GET /api/users/{userUuid}/image`. Visszaküldi egy felhasználó profilképének az elérhetőségét és tulajdonságait. Bármilyen szerepkör számára elérhető.
- `GET /api/users/account-info/image`. Visszaküldi a bejelentkezett felhasználó profilképének az elérhetőségét és tulajdonságait. Bármilyen szerepkör számára elérhető.

- `PUT /api/users/account-info/image`. Multipart Form Data segítségével fel lehet tölteni vagy ki lehet cserélni profilképet. Bármilyen szerepkör számára elérhető.
- `GET /api/users/{userUuid}/ratings`. Visszatéríti egy adott felhasználó értékeléseit. Bármilyen szerepkör számára elérhető.
- `POST /api/users/{userUuid}/ratings`. Létrehoz egy értékelést a beazonosított felhasználóra. Bármilyen szerepkör számára elérhető.

Az **ajánlatok**ért felelős végpontok a *BidController* osztályban vannak felsorolva.

- `GET /api/bids`. Visszatéríti egy bejelentkezett sofőrnek az összes eddig tett saját ajánlatát. Opcionális lekérdezési paraméterrel lehet állapot szerint szűrni ezeket. Sofőrök számára elérhető.
- `GET /api/bids/{bidUuid}`. Visszatéríti egy bejelentkezett sofőrnek egy eddig tett ajánlatát, ami a megadott *bidUuid*-val rendelkezik. Sofőrök számára elérhető.
- `GET /api/cargoes/{cargoUuid}/accepted-bid`. Visszatéríti egy adott szállítmányra az elfogadott ajánlatot. Sofőrök számára elérhető.
- `GET /api/cargoes/{cargoUuid}/lowest-bid`. Visszatéríti egy adott szállítmányra érkezett legkisebb árértékű ajánlatot. Bármilyen szerepkör számára elérhető.
- `DELETE /api/bids/{bidUuid}`. Törli a rendszerből a megadott azonosítóval rendelkező ajánlatot. Sofőrök számára elérhető.
- `GET /api/cargoes/{cargoUuid}/bids`. Visszatéríti egy adott szállítmányra érkezett összes ajánlatot. Munkahirdetők számára elérhető.
- `PUT /api/cargoes/{cargoUuid}/bids`. Adott azonosítójú szállítmányra helyezi el a kérés testében megadott ajánlatot. Sofőrök számára elérhető.
- `PATCH /api/cargoes/{cargoUuid}/bids/{bidUuid}`. Az adott azonosítójú szállítmányra tett, *bidUuid*-jú ajánlatot elfogadottnak nyílvánít. Munkahirdetők számára elérhető.
- `PATCH /api/bids/{bidUuid}`. Módosítja a beazonosított ajánlatot a kérés testében megadott adattagok szerint. Sofőrök számára elérhető.

A **jármű**vekért felelős végpontok a *TruckController* osztályban vannak felsorolva.

- `GET /api/trucks`. Visszatéríti egy sofőrnek az összes saját járművét. Sofőrök számára elérhető.

- `GET /api/trucks/{truckUuid}`. Visszatéríti a sofőr *truckUUid* szerint beazonosított járművét. Sofőrök számára elérhető.
- `POST /api/trucks`. Beregisztrálja a kérés testében megadott járművet a rendszerbe. Sofőrök számára elérhető.
- `PUT /api/trucks/{truckUuid}`. A megadott azonosítójú járművet módosítja. Az ide intézett kérés testében meg kell adni az összes adattag értékét, még ha nem is változik az érték. Sofőrök számára elérhető.
- `PATCH /api/trucks/{truckUuid}`. A megadott azonosítójú járművet módosítja részlegesen. Csak azok az adattagok fognak frissülni, amik meg lettek adva a kérés testében. Sofőrök számára elérhető.
- `DELETE /api/trucks/{truckUuid}`. Adott járművet töröl a rendszerből. Sofőrök számára elérhető.
- `GET /api/trucks/{truckUuid}/image`. A megadott azonosítójú jármű képének az URL-jét és tulajdonságait téríti. Bármilyen szerepkör számára elérhető.
- `PUT /api/trucks/{truckUuid}/image`. A megadott azonosítójú járműhöz tölt fel képet. Ha már van kép feltöltve hozzá, akkor az fölülíródik. Sofőrök számára elérhető.

Az útvonalgeneráláshoz egy végpont elérhető, aminek a lekezelő metódusa a *RouteController* osztályban található.

- `GET /api/route` Visszatéríti a legoptimálisabb útvonalat a sofőr konfigurációi alapján. Lekérdezési paraméterként kötelezően be kell azonosítani a járművet (*truckUuid* paraméter értékeként), amivel a körút meg lesz téve és meg kell adni, hogy az utolsó leadási pont után vissza szeretne-e térni a kiindulási pontba a sofőr (*closeCircle* paraméter, értéke vagy *Igen* vagy *Nem*). Opcionális paraméterként meg lehet adni, hogy melyik szállítmánnyal szeretné kezdeni a körutat a sofőr. Sofőrök számára elérhető.

2.1.5. Biztonság

A szerveren a biztonságért a Spring-nek a Spring Security [63] alprojektje felel. A Spring Security testreszabható autorizáción és autentifikáción kívül, webes támadások ellen is védelmet biztosít. Az alkalmazás autorizációja *JSON Web Token*-ek (*JWT*) [39] segítségével történik. A felhasználók autentikációja a `/api/login` végponton történik, ami során a szerver generál nekik egy *access token*-t [11] és egy *refresh token*-t [53]. A tokenek tartalmába (*payload*) bekerülnek a felhasználó adatai és a szerepköre. Az access token

fogja a kliens-alkalmazásoknak a hozzáférést az erőforrásokhoz engedélyezni, ezért azok bejelentkezés után minden kérés fejlécében el kell helyezzék a tokent. A kérés, mielőtt beérkezne a kontroller-osztályok valamelyikébe, végigmegy a *JwtLoginFilter* szűrőn, ami ellenőrzi az access token jelenlétét és a token aláírását, majd elküldi az Spring Security által biztosított autorizációs szervernek, ami ellenőrzi a tartalmában szereplő felhasználó létezését. Ha minden megfelel az elvárásoknak, akkor a felhasználó adatai bekerülnek a *SecurityContextHolder* objektumba foglalt kontextusba, ami a teljes kérés lekezelése alatt elérhető. Ebből az objektumból van minden alkalommal kiolvasva a kérést intéző felhasználó azonosítója és szerepköre az üzleti logika rétegen.

Miután az access token lejár, a felhasználónak nem kell explicit módon újra bejelentkeznie a kliens-alkalmazáson keresztül, mert az automatikusan újra be tudja jelentkeztetni a háttérben a felhasználót. Ez akkor történik meg, mikor a kliens-alkalmazás valamelyik kérésére az autorizációs szerver a 403-as HTTP státuszjelző kódot küldi (*Forbidden*) vissza, lejárt access token miatt. A kliens-alkalmazások ilyenkor a refresh tokent elküldik a `/api/refresh` végpontra, ahol, ha ez megfelel minden szempontból, akkor új access és refresh token generálódik. Az új access tokennel a fejlécben, a kliens-alkalmazások újraküldik az eredetileg sikertelen kérést.

2.1.6. Hibakezelés és validáció

Az API egyes végpontjaiban adatokat fog kiolvasni a kérés testéből. Ezeket az adatokat DTO-k formájában várja, melyeknek a tartalma a `@Valid` annotáció segítségével van ellenőrizve. A `@Valid` annotáció engedélyezi a *Hibernate Validator* [32] megszorításait. A csomagban számos hasznos beépített annotáció található, de lehetőség van saját megszorításokat is írni. Ha a beérkezdő DTO valamelyik értéke nem teljesíti a rá szabott megszorítást, kivétel dobódik, még mielőtt a végpontot lekezelő metódus meghívódna.

A szerver többrétegű architektúrájában bármelyik rétegen hiba merül fel, rétegspecifikus kivétel dobódik. A perzisztencia rétegen történő hibák mind a `DataAccessException`-ből származnak, így polimorfizmus miatt elegendő ennek a kivételosztálynak a példányait kezelni a *catch* ágakon az üzleti logika rétegen, ahol ezek a kivételek be lesznek burkolva a rétegspecifikus `ServiceException` kivételosztályba. Azokban az esetekben, mikor a hiba oka nem a rendszer belső állapotából ered, hanem függhet a bemeneti paraméterektől, akkor eset-specifikus kivételt dobunk, azért, hogy a hiba üzenete segítsen megértetni a felhasználóval, hogy mi nem ment rendjén. Néhány, gyakrabban használt kivétel:

- `ItemNotFoundException`. Akkor dobódik, mikor valamilyen keresés az adatbázisban sikertelen.

- `ItemNotOwnedException`. Akkor dobódik, mikor a kérést intéző felhasználó nem birtokol valamit, amire hivatkozott.
- `ActionNotAllowedException`. Akkor dobódik, mikor a felhasználó általi cselekvés valami miatt nem megengedett.
- `ActionCannotBeRepeatedException`. Akkor dobódik, mikor a felhasználó meg szeretne ismételni egy egyszer elvégezhető cselekvést.

Az API-rétegen a kontroller osztályok hibakezelő kódtól mentesek, mert a Hibernate Validator vagy az üzleti réteg által dobott kivételek mind a `GlobalExceptionHandler` osztályban vannak lekezelve. Ez az osztály rendelkezik a `@ControllerAdvice` annotációval [18] és `@ExceptionHandler`-ként [24] annotált metódusokat tartalmaz, mert ez az osztály kezeli le az összes kontroller osztályban dobódott kivételt. Így minden kivételnek van írva egy saját metódus, mely testreszabja a válaszként küldendő üzenetet és a HTTP státuszkódot megfelelően beállítja. Ezáltal nem a Spring Web alapértelmezett hibaüzenetei jelennek meg és a szerver REST API-ja el tudja rejteni a technológiai részleteket a külvilág elől.

2.1.7. Optimális útvonalgenerálás

A sofőr szerepkörű felhasználó a `/api/route` végpontra *GET* kérést küldve veheti igénybe az optimális útvonalat kigeneráló funkcionalitást. Az optimális útvonal a sofőr elfogadott ajánlataihoz tartozó szállítmányok felvevési és leadási pontjaiból fog állni. A sofőr a kérés intézésekor meg kell adja paraméterként, hogy melyik felvevési pontból akar indulni (`startingCargoUuid`), melyik járművét szeretné használni (`truckUuid`) és, hogy a körút végén szeretne-e visszatérni a kiindulási pontba (`closeCircle`). Az kérést fogadó kontroller osztály (`RouteController`) továbbítja a kérést az üzleti logika rétegen található `RouteService` szervízosztálynak, meghívva a `findRoute` metódusát.

A `RouteService` elvégzi a paraméterek helyességének és érvényességének ellenőrzését, majd átadja ezeket az algoritmust megvalósító `getOptimalRoute` függvénynek. Az optimális útvonal kigenerálása hasonlít az utazóügynök problémára, tehát ez is egy *NP-teljes* probléma. A `getOptimalRoute` függvény a probléma megoldására egy evolúciós algoritmust használ, pontosabban a 2-opt algoritmus [1] egy testreszabott változatát. Távolságmérő függvényként az *euklidészi távolság* volt használva. A Truxi alkalmazás esetében léteznek bizonyos megszorítások, amik a feladatot tovább nehezítik. A sorrendbe rakandó földrajzi pontok két típusúak lehetnek: felvevési pont és leadási pont, éppen ezért az utóbbi nem szerepelhet az előbbi előtt, mert az nem lehetséges, hogy a szállítmány hamarabb legyen leadva, mint felvéve. Arra is kell figyelni, hogy az aktuálisan szállított termékek súlya egy pontban sem haladhatja meg a jármű teherbírását. Ezeket figyelembe véve, az algoritmus egy listát gyárt, amiben a

szállítmányok leadási és felvevési pontjai jelennek meg látogatási sorrendben. Ha a sofőr a lista sorrendjét betartva szállítja el a szállítmányokat, akkor a lehető leghamarabb tudja megtenni a körutat. Az algoritmus a pontok egy véletszerűen generált sorából indul ki és addig végzi el az úgynevezett *2-opt cserét* a listán különböző paraméterekkel, amíg az össztávolságot meghatározó függvény már nem mutat javulást. A 2-opt csere két adott index között megcseréli a listaelemek sorrendjét, a specifikus megszorításokat tiszteletben tartva. Ha az így kapott lista rövidebb össztávval rendelkezik, mint a csere előtti, akkor az aktuálissá válik. A ciklus leállta utáni lista fog visszajutni a kérést intéző felhasználónak.

2.2. Web kliens

A web kliens felületet nyújt a felhasználó számára, amely az adatok kezelését és a szerverrel való kommunikációt támogatja. A web kliens *React*ban íródott, ami deklaratív és komponens alapú architektúrával rendelkező JavaScript könyvtár [52].

Ebben a fejezetben részletes bemutatásra kerül a web kliens szerkezeti felépítése és ennek a megvalósítása.

2.2.1. Architektúra

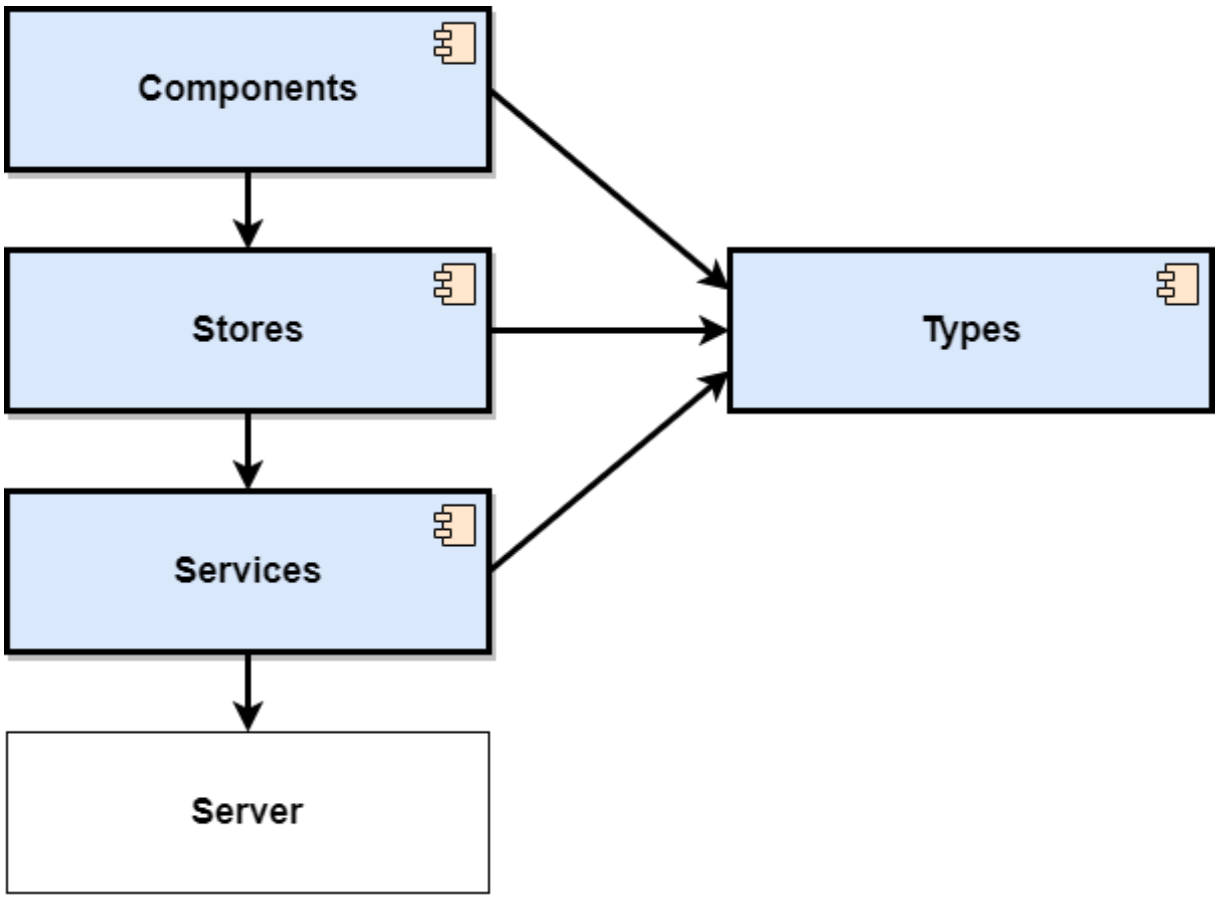
A web projekt architektúrája **négy** fő komponenst foglal magába (lásd ábra): *Types*, *Services*, *Components*, és *Stores*. (lásd 6. ábra)

A ***Types*** komponens feladata, hogy reprezentálja a kliens oldali entitásokat.

A ***Services*** komponens a kommunikációért felelős szerepet tölti be a szerver és a kliens között. A kommunikáció REST API segítségével történik, típusa aszinkron jellegű, így az adatok lekérése dinamikus. Az entitások megfelelő kliensoldali vagy szerveroldali konvertálása szintén itt végződik el DTO-k segítségével. (lásd kommunikáció a szerverrel)

A ***Stores*** modul tárolja és menedzseli a felhasználói felületért felelős komponensek állapotát. Az állapottal rendelkező komponensek egymástól elkülönített *store*okkal rendelkeznek, amelyben a nézettől elválasztva tartalmazzák az adattagokat és a hozzá tartozó műveleteket. A *store*okban található műveletek kétfélék lehetnek: lokális adatmanipulációs műveletek, amelyekhez nem szükséges szerver kommunikáció, valamint szerver kommunikációt igénylő műveletek.

A ***Components*** a felhasználói felületért felelős komponenseket tartalmazza. A webalkalmazás minden oldala egy vagy több komponensből épül fel, amelyek a szerverrel történő interakciót a *Stores* modulon keresztül érhetik el. A komponensek folyamatosan figyelik a nekik kirendelt *store*ok változását, majd dinamikusan frissítik a felhasználói felületet ennek alapján.



6. ábra. A web kliens négy fő modulja

2.2.2. Komponensek

A Truxi webalkalmazás React könyvtár segítségével van megírva, amely az összetett felhasználói felületet kisebb, egyszerűbb darabokra bontja fel, ezekt nevezzük komponensnek[47]. Minden oldala az alkalmazásnak egy konténerben található, ami egy sajátos komponens, melynek egyszerűen csak az a feladata, hogy összefoglalja az adott oldalhoz tartozó további komponenseket. Ahhoz, hogy jól átláthatóak és könnyen bővíthetőek legyenek a komponenseink jól meghatározott struktúrát követnek, amely az `Components` modulban helyezkednek el.

A komponensek osztályokként vannak reprezentálva, melyek a **`React.Component`**[9] osztályt terjesztik ki és kötelezően implementálják a **`render()`** metódusát, amely az adott komponens nézetét fogja ábrázolni a felhasználói felületen. Ha egy komponensnek komplexebb logikai műveleteket is kell végeznie, akkor az adott komponens két részből fog összetevődni, ahol az egyik rész csupán a nézetet tartalmazza, míg a másik rész a nézethez kapcsolódó műveleteket. Ennek a megoldásnak a segítségével a bonyolult komponensek átláhatóbbá és könnyebben fejleszthetővé válnak, és érvényesül az az elv, hogy egy komponensnek csak egy feladata legyen (*single-responsibility principle* [57]).

Az oldalak útvonalának megfeleltetését a **react-router-dom**[8] navigációs könyvtárból

a **BrowserRouter**[2] speciális komponensel valósítja meg az alkalmazás, ami a HTML5 history API-t használja [4], hogy szinkronban tartsa a felhasználói felületet az oldalhoz tartozó URL-el. A **react-router-dom** továbbá olyan komponenseket is biztosít, amellyekkel az oldalak közötti navigálás könnyen elvégezhető, és ezek közül a legfontosabb a **Redirect** komponens, amely kikeresi a keresett útvonalat és az átirányítás után frissíti a nézetet. A következőkben bemutatásra kerülnek azok a komponensek, amelyek a különböző oldalak nézetét alkotják.

- `HomePage`. A felhasznalót az alkalmazás megnyitásakor ez a nézet fogadja. A nézet kétféle tartalmat ábrázolhat attól függően, hogy a felhasználó bejelentkezett vagy sem. A komponens szerepe, hogy egy barátságos nyitólapot bíztosítson a felhasználóknak vagy vendégeknek.
- `OwnCargoPage`. A bejelentkezett munkahirdető számára listanézetet biztosít a saját szállítmányairól, ahol áttekintheti ezek állapotát.
- `AddCargoPage`. Ez a nézet biztosítja a munkahirdetőnek azt, hogy új munkahirdetést tegyen fel, egy form kitöltésével, amelyben megadja a szállítmányhoz tartozó információkat.
- `BidListPage`. A nézet egy szállítmányhoz tartozó árajánlatokat listázza, melyek a sofőröktől érkeznek.

2.2.3. API kliens

A komponensek a megjelelenítést szolgáltatják a felhasználói felületen, viszont a kommunikációs logikát nem ők végzik, hanem az API kliens réteg, amely a ***Services*** modulban található. A szerver és komponensek közötti minden kommunikációs interakció a REST szabályokat követve működik. Az API kliens az `src/services/apiClient` csomagban található, amely magába foglal egy ApiClient osztályt, valamint minden entitás számára elkülönít egy API kliens osztályt.

Az `ApiClient` osztály az API hívásokért felel, amelyet az *axios* könyvtár[13] segítségével valósít meg. Az osztály a szerverhívásokat a publikus HTTP metódusok szerint kategorizálja (GET, POST, PUT, PATCH, DELETE). A metódusok, minden kérésnél paraméterként megkapják a relatív útvonalat és a kérés törzsét ha szükséges. Az osztály szerepköréhez hozzátartozik, hogy megfelelően konfigurálja a hívásokat. A kérések autorizációjának szempontjából a szerver egy *JWT tokent* [39] vár el, amit a rendszer a sikeres bejelentkezés után kap meg a szervertől és a `localStorage`[6] tárolóba menti el, amelyben az adatoknak nincs lejárati idejük. Azok a hívások, amelyek védett adatokkal dolgoznak, *JWT token* megadását

igénylik, amit az `ApiClient` a hívás konfigurálásakor a kérés fejlécébe állít be. Ha a token lejár, akkor a felhasználónak újra be kell jelentkeznie a rendszerbe.

Az API kliensben minden entitáshoz tartozik egy osztály, ami azokat a műveleteket tartalmazza, amelyek kizárólag az adott entitásra vonatkoznak. Ezeknek a metódusoknak az a szerepe, hogy előkészítsék a kérést, meghívják az `ApiClient` osztályból a megfelelő HTTP metódust. Mivel a szerver a kommunikáció során DTO osztályokat (*Data Transfer Object*) [22] használ, ezért minden kérés előtt a paraméterként megkapott adatokat át kell transzformálni egy olyan DTO-ra, ami megfeleltethető azzal, ami a szerveren van. Ha a kérésre érkező DTO válaszobjektum formája nem azonos azzal, amit a hívó fél elvár, akkor egy újabb konverzió végrehajtásával létrehozza a kívánt eredményt. A DTO osztályok gyűjteménye a `src/services/dtos` csomagban található, ami magába foglalja az összes szerverrel kompatibilis kérés és válasz DTO objektumokat. A konverzióhoz tartozó műveletek megvalósítását assembler osztályok végzik, amelyek az `src/services/assemblers` csomagban találhatóak.

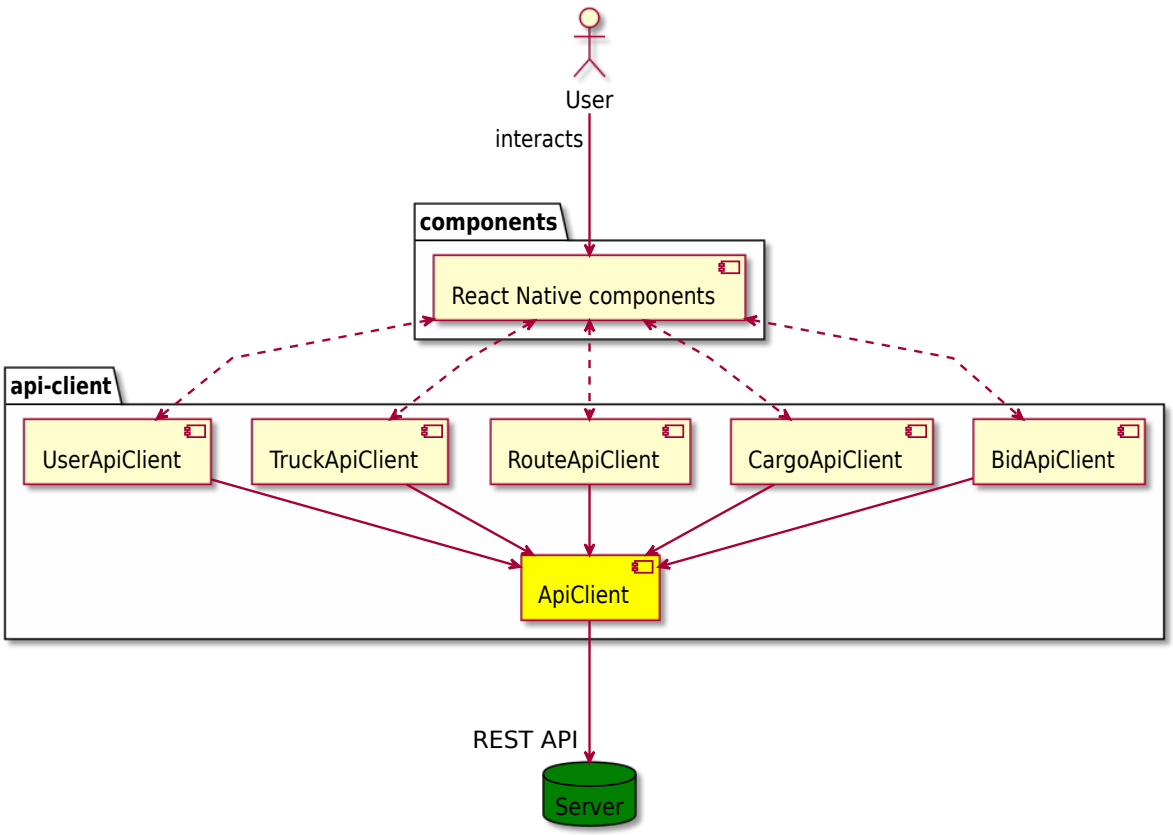
2.3. Mobil kliens

A mobilalkalmazás a szoftverrendszer megjelenítési rétegének fontos komponense, mert a sofőr szerepkörű felhasználók ezen keresztül használhatják a rendszert. Az alkalmazás barátságos és könnyen használható felhasználói felületet nyújt, miközben a háttérben kommunikál a szerverrel (2.1).

2.3.1. Komponensek

A *React Native* [49] keretrendszer segítségével fejlesztett alkalmazásban a kinézeti elemek *React komponensek*nek (*React components* [47]) felelnek meg, melyek egy komplex, viszont átlátható és könnyen bővíthető *komponens-hierarchiá*t alkotnak. A hierarchia gyökerében a *React Native Router Flux* navigációs könyvtárból [50] származó speciális `Router` komponens [51] van, amely előre definiálja a navigációs útvonalakat az alkalmazásban.

Egy alkalmazás-szintű konvenció szerint az összes React komponens osztálykomponensként lett megírva, így a komponensek a `src/components` mappa `.jsx` kiterjesztésű fájljaiban levő TypeScript osztályokban vannak definiálva. A `screens` almappában találhatóak azok a komponensek, amelyek külön nézetet határoznak meg és dedikálva van számukra egy útvonal a React Native Router Flux navigációs komponensében. Ezek a nézet-komponensek elsősorban konténerként viselkednek, ugyanis további komponenseket foglalnak magukba. Az alkalmazásban minden komponensnek egy feladata van (*single-responsibility principle* [57]), ezért, amikor megoldható volt, minden komponens



7. ábra. A mobilalkalmazás szerkezete. A felhasználó a felületen megjelenített komponensekkel végez interakciót, amik az esetleges hálózati hívásokat a API klienssel végeztetik.

további komponensekre lett bontva. A dolgozatban nem esik szó minden komponensről, viszont a fontosabb konténer-komponensek bemutatásra kerülnek. A következő nézetek kinézete (képernyőmentések formájában) és részletesebb működésük leírása a 4. fejezetben található.

- `CargoListScreen`. Az alkalmazás indulásakor ez a nézet köszönti a felhasználót. Ez csak az elérhető szállítmányok listáját megjelenítő komponenst foglalja magába. Ezt láthatják be nem jelentkezett felhasználók is, viszont a hatáskörük eddig tart.

- `CargoDetailsScreen`. Minden szállítmányról részletes leírást kaphat a felhasználó, ha erre a nézetre navigál. Itt látható a legalacsonyabb ajánlott ár, és ugyancsak itt van lehetőség ajánlatot létrehozni vagy egy meglévőt módosítani.

- `BidListScreen`. Ezen a nézeten tudja a sofőr az eddigi ajánlatait megtekinteni és menedzselni. Az ajánlatok szét vannak válogatva állapotuk szerint. A függőben lévő ajánlatok módosíthatóak vagy törölhetőek. Az elfogadtott ajánlatok elszállítottnak jelölhetőek, míg a már elszállított szállítmányoknak értékelhetjük a tulajdonosait.

- `TruckListScreen`. A sofőr a rendszerbe már beregisztrált járműveinek listáját láthatja.

- `AddTruckScreen`. A sofőr beregisztrálhat új járművet, miután megadta ennek az adatait. Képet is feltölthet hozzá opcionálisan.
- `RouteFinderScreen`. Ezen a nézeten igényelheti a sofőr, hogy a rendszer egy optimális útvonalat generáljon a már elfogadott ajánlatok szállítmányainak a helyszíneiből. A felhasználó kiválaszthatja melyik helyszínről szeretné az útját indítani és melyik járművével szeretné véghezvinni. A járműves listában csak azok a járművek jelennek meg, amelyek képesek elvinni bármelyik elfogadott szállítmányt.
- `RouteExposerScreen`. Ez a nézet az optimális útvonalat jeleníti meg. A szervertől válaszként érkezett optimális útvonal leadási és felvevési pontjai megkülönböztetve és helyes sorrendben vannak feltüntetve.
- `RegisterUserScreen`. A regisztrációért felelős nézet. Elérhető vendégek számára.
- `LoginScreen`. A bejelentkezésért felelős nézet. Elérhető vendégek számára.

2.3.2. Adatmodell

A mobilalkalmazás a szerver esetében bemutatott entitásokkal dolgozik (2.1.1), így az adatmodellje tükrözi a szerver adatmodelljét. A TypeScript típusosságot biztosít a projektnek, így a modelleket minden használatkor adott struktúrához lehet kötni. Az entitások struktúrája TypeScript *interface*-k [67] segítségével van leírva. A modellek gyűjteménye a `src/types` mappában található.

2.3.3. API kliens

A komponensek kizárólag a megjelenítésért felelősek, így nem ők fogják végezni a bonyolultabb logikát igénylő feladatokat. Az alkalmazásban az állapotmenedzsmentet és bonyolultabb üzleti logikát a *Mobx* állapotmenedzser [43] könyvtár *store*-jai valósítják meg. Mikor hálózati kommunikációra van szükség (a szerverhez kell egy kérést intézni), a komponensek továbbhívnak az API kliens vagy a *store*-ok felé.

Az API kliens az a rétege az alkalmazásnak, amelyik megvalósítja a kommunikációt a szerverrel, azáltal, hogy fogyasztója az általa publikált API-nak. Azért, hogy a kommunikáció megtörténhessen és az adatcsere kompatibilis legyen, a mobilalkalmazás is ugyanazokat a *Data Transfer Object* (DTO) osztályokat használja, mint a szerver által kiszolgált API (2.1.4). Ez azt vonja maga után, hogy szükség van *assembler* osztályokra, melyek a beérkező DTO-kat a mobilalkalmazásban használt típusokra (2.3.2) konvertálják, vagy fordítva. A DTO-k a `src/service/dtos`, míg az assemblerek a `src/service/assemblers` csomagokban találhatóak.

Az API kliens réteg TypeScript osztályokból áll, amelyek a `src/services/api` csomagban találhatóak. Két fontosabb részre lehet osztani az API kliens réteget: az `BaseApiClient` osztályra és az entitásonkénti API kliens osztályokra.

Az `BaseApiClient` osztály csak egy absztrakciós réteg az *axios* [13] HTTP hívások kezelésére megírt könyvtár felett. Azért volt szükség erre az osztályra, hogy minden kommunikációval kapcsolatos konfigurációt egy helyen végezzünk el. Az osztály rendelkezik egy axios objektummal, amin a konfigurációkat végezzük. Többek közt be van állítva, hogy mi a szerver API-jának az alap útvonala, illetve be van konfigurálva, hogy minden kérés fejlécébe helyezze el az autorizációs JWT tokent. Ha a token érvényességi ideje lejár, a `BaseApiClient` osztályban aktiválódik egy JWT access-token megújítási mechanizmus. Az osztály a refresh-token segítségével új tokeneket kap a szervertől, kicseréli az alkalmazásban tárolt tokeneket az újakra, majd újraküldi a kérést. Csak abban az esetben hajtja ezeket a lépéseket végre, ha a felhasználó jelezte bejelentkezéskor, hogy szeretetne bejelentkezve maradni, ez az opció szintén el van tárolva az alkalmazásban. A mechanizmusnak az implementálásában segített az *Interceptor* minta, melyre implementációt biztosít az axios [14]. Az axios *interceptor*-ral el lehet fogni a szervertől érkezett választ, mielőtt az az eredeti axios hívóobjektumba beérkezne. Az axios instanciának a konfiguráció során meg lehet adni a *callback* függvényben, hogy mihez kezdjen az elfogott válaszobjektummal. Ha a válasz lejárt access-tokent jelez, akkor az előbb leírt lépések el lesznek végezve.

Az `BaseApiClient` osztály egy olyan osztály, ami az API hívásokat intézi a szerver felé. Az osztály rendelkezik a rendszer által használt HTTP metódusnak (GET, POST, PUT, PATCH, DELETE) megfelelő publikus metódusokkal. A függvények várják paraméterként a relatív útvonalat, ahova a megfelelő HTTP metódusú kérés intéződni fog. A `post(...)`, `put(...)` és `patch(...)` függvényeknek meg lehet adni a kérés törzsét is opcionálisan.

Minden rendszerentitáshoz tartozik egy különálló API kliens-osztály, amiben a vele kapcsolatos műveleteknek vannak metódusok létrehozva. Ezek a metódusok előkészítik a szerverhez intézett kérést, elvégzik a szükséges DTO konverziókat, majd meghívják a megfelelő HTTP metódusnak megfelelő függvényét a `BaseApiClient` osztálynak. Ez a függvényhívás kompozíció segítségével történik, mert az API kliens-osztályok rendelkeznek egy `BaseApiClient` típusú adattaggal, ami injektálva lesz ezekbe. Mikor megkapják a választ, konvertálják a beérkezett DTO-t modellé, esetlegesen elvégezik a bonyolultabb logikát igénylő feldolgozását az adatnak, majd visszatérítik azt a hívó félnek. Az API kliens-osztályok ismerik a szerver REST API-jának az URL-jeit, mert az `BaseApiClient` metódusainak meg kell adniuk a megfelelő relatív útvonalat.

3. Technológiák és eszközök

Ebben a fejezetben részletezve lesznek a projekt megvalósításához felhasznált technológiák és eszközök mindhárom alprojekt esetében.

3.1. Eszközök

Az alkalmazás forráskódja verziókövető alá van helyezve és a használt verziókövető rendszer a *Git* [3]. Az alkalmazás fejlesztése alatt a csapat betartotta a *git-flow* által előírt konvenciókat [27]. A konvenciók szerint a fejlesztés egy `develop` nevű ágon történt, ahonnan minden új funkcionalitásnak saját ág lett indítva. A funkcionalitásokhoz tartozó kód elkészülte után, ha sikeresen végigment a minőségi ellenőrzésen (*code review*), be lett *merge*-elve a `develop` ágra.

Mindhárom projekt (szerver, webkliens, mobilkliens) külön Git alapú tárolóval (*repository*) rendelkezik. A tárolók a GitLab [29] által vannak menedzselve. A GitLab egy ingyenesen is használható, teljeskörű *DevOps* platform, melyet a Truxi a távoli tárolók (*remote repository*) kezelésén kívül sok más szolgáltatása miatt is használt. A GitLab projektmenedzselő felülete volt felhasználva a funkcionalitások állapotának követésére (*Gitlab issue tracking*), a GitLab platformján történt a minőség ellenőrzése (*merge request*), valamint használva voltak a GitLab folyamatos integráció és folyamatos kitelepítés szolgáltatásai is (*continuous integration, continuous deployment* vagy *CI/CD*) [28].

Az alkalmazás fejlesztése csapatban történt a *Scrum keretrendszer* [56] szabályai szerint. A fejlesztés *sprint*ekre volt bontva, melyek időtartama két hét volt. Minden sprint elején volt egy tervező (*planning*) gyűlés tartva, ahol a csapat eldöntötte, hogy milyen funkcionalitások (*user story*-k) lesznek implementálva a sprint alatt. A funcionalitások (feladatok) nehézsége *story* pontokban volt mérve, ami segített eldönteni, hogy pontosan mennyi feladat fér bele az adott sprintbe. A sprintek végén minding volt egy *demo*, ahol be lettek mutatva az adott sprint alatt elkészült funkcionalitások, majd egy *retrospektív* megbeszélés, ahol a csapat visszatekintett az elmúlt iterációra, elemezve azt, majd megállapította, hogy mi ment jól és, hogy mi mehetett volna jobban.

3.2. Szerveroldali technológiák

A webalkalmazás a *Spring keretrendszer* [62] segítségével van megírva, fejlesztése a Java programozási nyelvben történt. A Spring egy összetett projekt, a konfigurálását és könnyű beüzemelését a *Spring Boot* [58] alprojekt biztosítja. A szerver legkülső rétegén, az *API-réteg*en (2.1.4), a RESTful szolgáltatások a *Spring Web MVC* [59] projektből

származó kontroller osztályokkal vannak megvalósítva. A perzisztencia réteg (2.1.2) a *Spring Data JPA* [60] alprojekt szolgáltatásaira épül. A Spring Data JPA interface-eire a Spring biztosít implementációkat, a háttérben alapértelmezetten a *Hibernate* ORM keretrendszert [31] alkalmazva JPA implementációként. A Hibernate képezi le a megfelelően megjelölt Java osztályokat egy relációs adatbázis tábláivá.

Az alkalmazás adatai egy *MySql* [45] relációs adatbázisba mentődnek le. Az adatbázis séma menedzsmentje a *Liquibase* [40] nevű könyvtárral van megvalósítva. A Liquibase segítségével az adatbázis séma verziókövető alá kerül, tehát minden állapota le lesz mentve. A projekt **src/main/resources/db** mappájában találhatóak a Liquibase állapotmódosításokat tartalmazó *.xml* kiterjesztésű állományai (changelog-ok).

A szerveren a build- és függőségmenedzsmentért a *Gradle* [30] felelős. Az állománystruktúra törzsében található `build.gradle` állományban vannak listázva a build-fázisban használt pluginok, ezek konfigurációs beállításai, illetve a projekt függőségei. A felhasznált pluginok a standard `Java` plugin, ami a forráskód fordításáért felelős task-okat biztosítja, a `org.springframework.boot` plugin, aminek a *bootRun* task-jával a webalkalmazás indítása történik és az `org.liquibase.gradle` plugin, ami parancssorból futtatható Gradle task-okat biztosít az adatbázisséma menedzselésére és manipulációjára. A `build.gradle` állományban két saját Liquibase *activity* is definiálva van: *main* és *diff*. Liquibase activity-k segítségével előre be lehet állítani bizonyonos paramétereket, azért, hogy a parancssorból történő futtatáskor csak a szükséges activity-t kelljen megadni. A *diff* activity akkor használatos, mikor ki kell generálni az aktuális adatbázisséma és a Java osztályok közötti különbséget Liquibase changelog formájában, míg a *main* az alapértelmezett Liquibase activity, amely a többi Liquibase task-nál aktiválódik. Így az adatbázisséma változtatása a következő lépésekben történik: először a Java modellosztályok vannak kiegészítve az új adattagokkal, majd a `gradle diffChangeLog -PrunList=diffMain` parancssori utasítással a Liquibase kigenerálja a különbségeket tartalmazó changelog *.xml* állományt. Ez az állomány automatikusan megfelelő helyre kerül, így a módosítások a `gradle update` utasítással érvénybe lépnek az adatbázison belül is.

Az alkalmazásban lehetőség van képeket feltölteni a központi entitásokhoz. Ezek a képek nem az adatbázisban vannak tárolva, hanem a *Cloudinary* [17] felhőszolgáltatásban. A Cloudinary biztosít Java integrációt is, így a `com.cloudinary:cloudinary-http44` függőség hozzáadása után lehetőség van Java kódból kommunikálni a felhőszolgáltatással. A képek bináris formában Multipart Form Data encoding típussal érkeznek a megfelelő végpontokra, majd továbbítva lesznek az `ImageService` üzleti logika rétegbeli osztálynak. A szervízosztály a fogyasztója az előbb beimportált csomagnak, így fel tudja küldeni a felhőbe a képet, majd ellátja a megfelelő azonosítóval, hogy a lekérés is megvalósítható legyen

ugyanilyen módon.

A GitLab folyamatos kitelepítés szolgáltatásának köszönhetően, a *Docker* virtualizációs eszköz [20] minden *push* műveletkor a törzsben található `Dockerfile` alapján elkészít egy *image*-et az alkalmazásból és feltölti azt a projekt GitLab-os, távoli tárolójához rendelt *registry*-jébe. Ez az image tartalmazza az alkalmazás build-elt változatát, amit futtatni tud a Docker egy *konténer*ben. A Dockerfile többlépcsős (*multistage Dockerfile*), mert van egy átmeneti lépés, ami kizárólag a build-eléssel foglalkozik, míg a végső lépés futtatja a build eredményét. A *Docker-compose* eszközzel [21] van megvalósítva az, hogy az adatbázis és az alkalmazásszerver két különböző konténerbe fussanak és tudjanak kommunikálni egymással. A kommunikáció megvalósításához a rendszer egy *hálózatot* (*network*-ot) hoz létre. A Docker-compose leíróállománya szintén a gyökérben található: `docker-compose.yml`. A leírófájlban be vannak konfigurálva a futtatandó konténerek és a kommunikáció ezek között (egy *network* segítségével). Az adatbázis konténeréhez hozzá van rendelve egy *volume*, hogy a lementett adatok maradjanak perzisztensek akkor is, ha újra kell indítani a rendszert. A Docker-compose minden futtatáskor letölti a szerveralkalmazás registry-ben található image-ét, majd elindítja a szerver és az adatbázis konténereit a megfelelő sorrendben. Az adatbázisszervernek már futnia kell ahhoz, hogy a szerveralkalmazás csatlakozni tudjon hozzá. Az alkalmazásszerver jelenleg egy konténerbe kitelepítve működik a Codespring cég egyik szerverén és az általa kiszolgált REST API-nak az URL-je truxi-api.test.edu.codespring.ro.

3.3. Web technológiák

3.3.1. React

A webalkalmazás a React [52] könyvtár felhasználásával íródott. Ez egy JavaScript könyvtár, amelyet interaktív webes felhasználói felületek építésére fejlesztett ki a Facebook, közösen más független fejlesztőkkel és cégekkel. Az alapvető építőelemei a felületeknek komponensekből állnak, amelyek a komplex nézetet kisebb részekre osztják fel. A komponensek összesége alkotja a *Virtual Document Object Modelt*, ami a *DOM*-ot az állapot változásának hatására hatékonyan tudja frissíteni. A felhasználói felületet alkotó komponensek leírása JSX (JavaScript XML) [38] nyelvezetet használ, ami a JavaScript-nek egy szintaktikai kiterjesztése. Az XML-hez hasonlító kód az előfeldolgozás során átalakúl megfelelő JavaScript utasításokká.

3.3.2. Typescript

A Typescript egy nyílt forráskódú, JavaScript-re épűlő nyelv, statikus típusdefiníciók hozzáadásával [66]. A statikus kódírás esetén a típus a változóval szöveges kifejezéssel van

társítva és kompilálási időben van ellenőrizve. A dinamikus kódírás esetén viszont a változó típusa az értékkel van társítva és futási időben van ellenőrizve. A típusok kikövetkeztetése legtöbb esetben egyértelműen végrehajtható, ha viszont nem, akkor léteznek erre dedikált algoritmusok. A típusok létrehozása interfészeken kereszül történik. A TypeScript nyelv használata előnyt jelent a fejlesztők számára, mert a típusoknak köszönhetően a fejlesztői környezetek intellisense mehanizmusa bíztosítja a kódkiegészítést és a típusok leírását, ezáltal gyors és helyes kód írására ad lehetőséget. Mivel a Typescript a JavaScript-et terjeszti ki, ezért a JavaScript kód is elfogadható a Typescript számára, de a JavaScript alapú könyvtárak használatakor szükséges egy állomány, amely a típusok definicióját és szerkezetét írja le.

3.3.3. Mobx állapotmenedzser

A komponensek állapotának kezelésére a webalkalmazás a Mobx [43] könyvtárat használja, mivel a funkcionális reaktív programozási elvek követésével egyszerű és skálázható rendszert alkot. A Mobx alapgondolata abból áll, hogy minden, ami származtatható az alkalmazás állapotából, azt automatikusan származtatni kell. Minden olyan adattag, amely a komponens állapotát képezi, *observable* (megfigyelhető) dekorátorral megjelölhető és az *observer* (figyelő) pedig reagálni tud a változásukra. A Mobx segít abban is, hogy a kompnensek állapotát tartalmazó adattagokat és az ezekkel végzett bonyolultabb üzleti logikát ne a komponensbe tároljuk, hanem Mobx *store* osztályokba csoportosítsuk. A *storeok* létrehozásával a komponens állapota egy átláthatóbb egységbe szerveződik, valamint könnyebben megosztható más komponensekkel.

3.3.4. Dependency Injection

A TypeScript megjelenésével lehetőség van objektumorientált programozásra JavaScriptben. Mivel az osztályok között szoros kapcsolatok alakulhatnak ki, szükség van egy olyan tervezési mintára, ami feloldja ezeket a függőségeket. A Dependecy Injection [19] tervezési mintának köszönhetően az osztályok függőségeit egy külön objektum menedzseli.

Az *InversifyJS* könyvtár által nyújtott megoldás [34] egy pehelysúlyú inversion of control (IoC) [35] konténert biztosít TypeScript és JavaScript alkalmazásoknak. A konténerben azoknak az osztályoknak a példányai lesznek eltárolva, amelyek egy másik osztály függőségét képezik. Ezeket az osztályokat az *inversify* az @injectable() dekorátorral jelöli, míg a beinjektálásra a @lazyInject() dekorátort használja. Ezáltal a React komponensekbe könnyen beinjektálhatóak például a megfelelő *Mobx store*ok, vagy az API hívásokat megvalósító osztályok példányai.

3.3.5. Semantic UI React

A *Semnatic UI* egy webfejlesztési keretrendszer, ami stílusos, reszponzív felhasználói felületek létrehozását segíti, tömör HTML osztályok biztosításával[10]. A Semantic UI React a Semantic UI hivatalos React-os integrációja, ami a HTML alapú stílusosztályokat komponensekbe szervezi. A deklaratív API segítségével a keretrendszer biztosítja a komponensek robusztus funkcióit és a tulajdonságok validációját is a hibák hatékony beazonosítására. Lehetőség van alkomponensek használatára is, amelyek elngedhetetlenek a komponensek testreszabásának rugalmasságához.

3.3.6. React Leaflet

A *Leaflet*[5] egy nyílt forráskódú, pehelysúlyú JavaScript könyvtár interaktív térképek ábrázolására, web és mobil platformokra. A térképek kinézetének testreszabására és funkcióinak kibővítésére több plugin is létezik, így az aklamazás igényeinek megfelelően konfigurálható a térkép. A React Leaflet[7] absztrakciót biztosít a Leaflet fölött. Az absztrakció nem helyettesíti a Leafletet, hanem csak kihasználja a React életciklus módszereit, amelyekkel meghívja a releváns Leaflet handler metódusokat.

3.3.7. Formik

A *Formik*[26] könyvtár a React form-ok egy egyszerűbb megoldását biztosítja. A könyvtár három feladat megoldásában segít: a form változóinak kinyerése és frissítése, validáció és hibaüzenetek kezelése, a form elküldésének lekezelése. Ezeknek egy egységbe való elhelyezésével a Formik rendezettebb formok létrehozásához vezet, megkönnyítve a form megértését, tesztelését, refaktorálását, karbantartását, bővítését. A ismétlődő programrészletek elkerülésére újrafelhasználható metódusokat és akció-kezelőket biztosít.

3.4. Mobil technológiák

A mobilalkalmazás *React-Native* [49] technológiával íródott. A React-Native egy *JavaScript* keretrendszer, ami cross-platform mobil applikációk fejlesztésére alkalmas. A React-Native a szintén Facebook által fejlesztett *React* könyvtárra [52] épül, a primitív React komponensek pedig natív UI platform (Android vagy iOS) elemekként renderelődnek. A Truxi alkalmazás mobilos projektje teljes mértékben JavaScript-ben volt megírva, bár szükség esetén a React-Native lehetőséget biztosít natív kód írására is. A kinézetet biztosító React-Native komponensek *JSX* nyelvezettel [38] íródtak meg, mely egy szintaktikai kiterjesztése a JavaScript-nek. A mobilalkalmazás könnyebb fejlesztése érdekében az *Expo* keretrendszer [25] volt használva. Az Expo olyan eszközök gyűjteménye, melyek segítenek az alkalmazás

fejlesztésében, buildelésében, tesztelésében és kitelepítésében egyaránt Android és iOS platformra. A navigáció a komponensek között a *React Native Router Flux* könyvtár [50] segítségével volt megvalósítva.

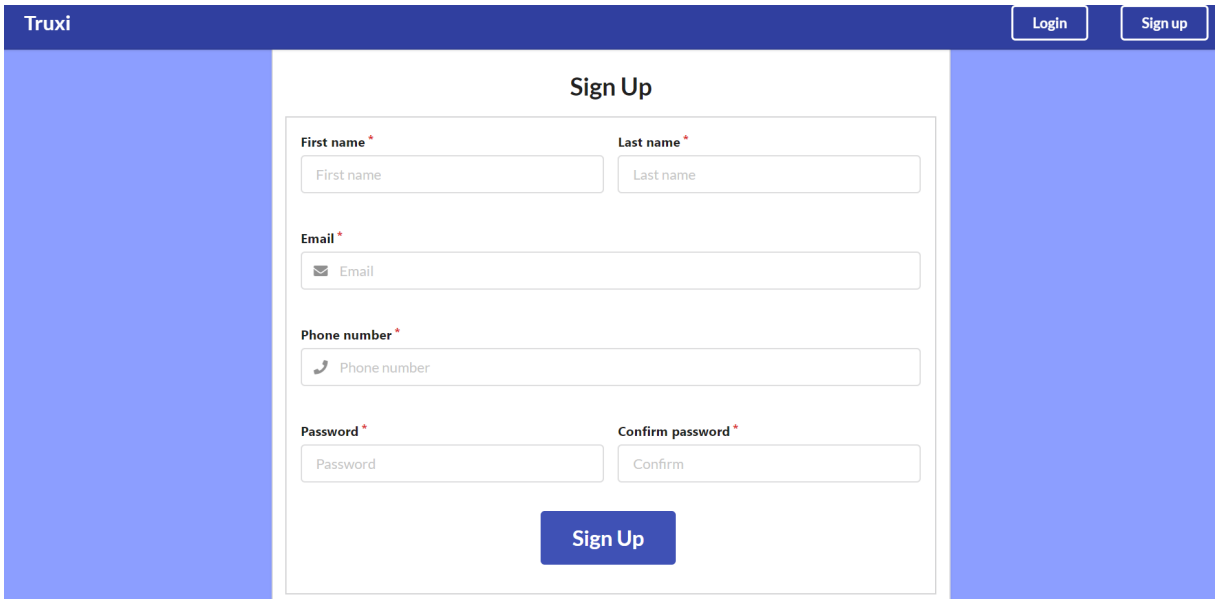
Az alkalmazás a Microsoft által készített *TypeScript* programozási nyelvben [66] íródott. A TypeScript a JavaScript-nek egy kiterjesztett változata, mely statikus típusosságot biztosít a kódnak, ami miatt sok esedékes hiba már kompilálási időben jelentkezik, így hibabiztosabb kód születik. Használatos volt a *Mobx* állapot-menedzsment könyvtár [43], mely segítségével az alkalmazás állapotában történő változások egyszerű és kényelmes módon voltak lekezelve. A Mobx lehetőséget nyújt több *store* készítésére, melyekben az alkalmazás állapotának különböző, logikusan szétválasztott részeit tárolhatjuk. Ezek a store-ok elérhetőek az összes komponens számára. A projekt használja a *Dependency Injection* mintát, amit az *InversifyJs* keretrendszer [34] segítségével valósít meg. Az Inversify egy pehelysúlyú Inversion-of-Control konténert biztosít az alkalmazásnak, melynek előre meg kell adni az injektálandó osztályokat és ezek injektálásának a konfigurációját. Ezek a konfigurációk a `src/ioc` mappában találhatóak. Az injektálás a React-Native komponensek esetében a **`@lazyInject`**, míg a hagyományos TypeScript osztályokban az **`@inject`** TypeScript dekorátorral történt. A mobilalkalmazásban megjelenő form-ok megvalósítása a *Formik* [26] JavaScript könyvtár segítségével történt, mely megkönnyítette a form-ok megírását és jelentősen lecsökkentette az ismétlődő kódrészleteket. A komponensek stilizálásáért a *React Native Elements* komponenskönyvtár [48] felelős, mely bármilyen mértékben testreszabható, elegáns komponenseket biztosít.

4. Az alkalmazás működése

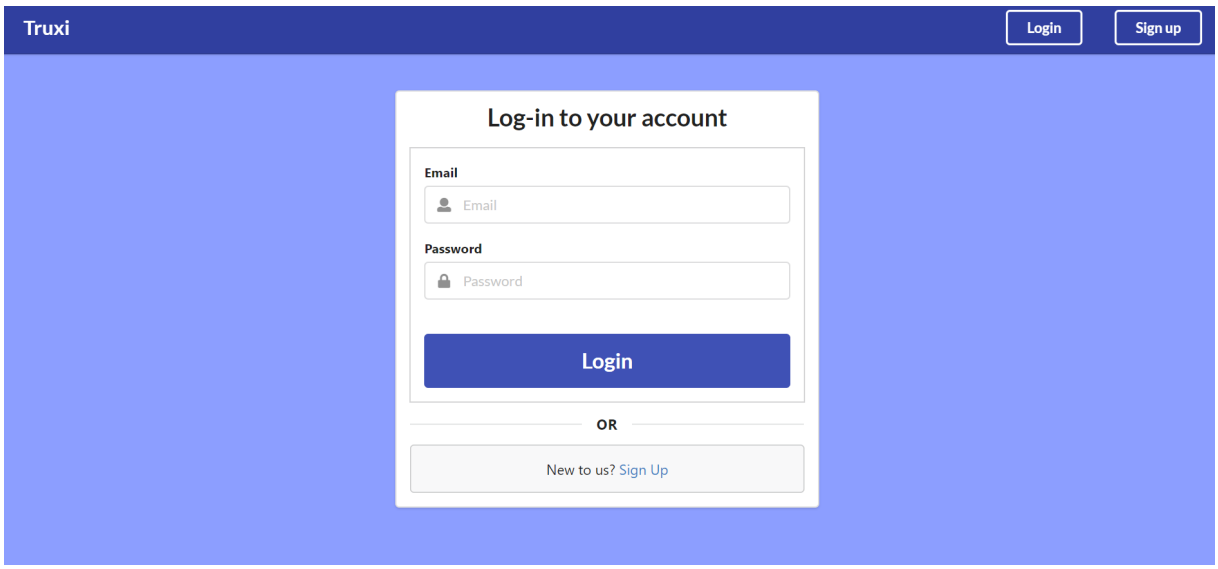
A Truxi alkalmazás felhasználói a rendszert közvetlenül csak a web- és mobilalkalmazásokon keresztül használhatják, így ebben a fejezetben ezeknek a működése lesz részletesen bemutatva. A rendszer által nyújtott szolgáltatások korlátozva vannak az autentikált felhasználók számára, elérésükhöz a felhasználóknak be kell jelentkezniük a rendszerbe. A vendégek, akik szintén el akarják érni a szolgáltatásokat, mint munkahirdető, vagy mint sofőr, ezt megelőzően be kell regisztráljanak. A rendszerbe regisztrálni ingyenesen lehet, viszont meg kell adni egy email címet, előnevet, utónevet, telefonszámot, jelszót. Regisztrálni mindkét kliensalkalmazásból lehet, ugyanazokat az adatokat kell megadni. A weben (8. ábra) és mobilon (13. ábra) is validálva vannak a beviteli mezők, így a regisztráció nem lesz sikeres egészen addig, míg az adatok eleget nem tesznek az összes feltételnek. Sikeres regisztráció után a felhasználó bejelentkezhet a szerepkörének megfelelő alkalmazásba, munkahirdetők a webalkalmazásba (9. ábra), sofőrök a mobilos alkalmazásba (13. ábra). Bejelentkezéskor ki lehet választani a *Jegyezz meg* opciót, aminek hatására a felhasználó automatikusan be lesz jelentkeztetve a rendszer által a munkameneti idejének lejárta után. Ha nem él a felhasználó ezzel a lehetőséggel, akkor két óra után újra be kell jelentkeznie.

4.1. Web alkalmazás

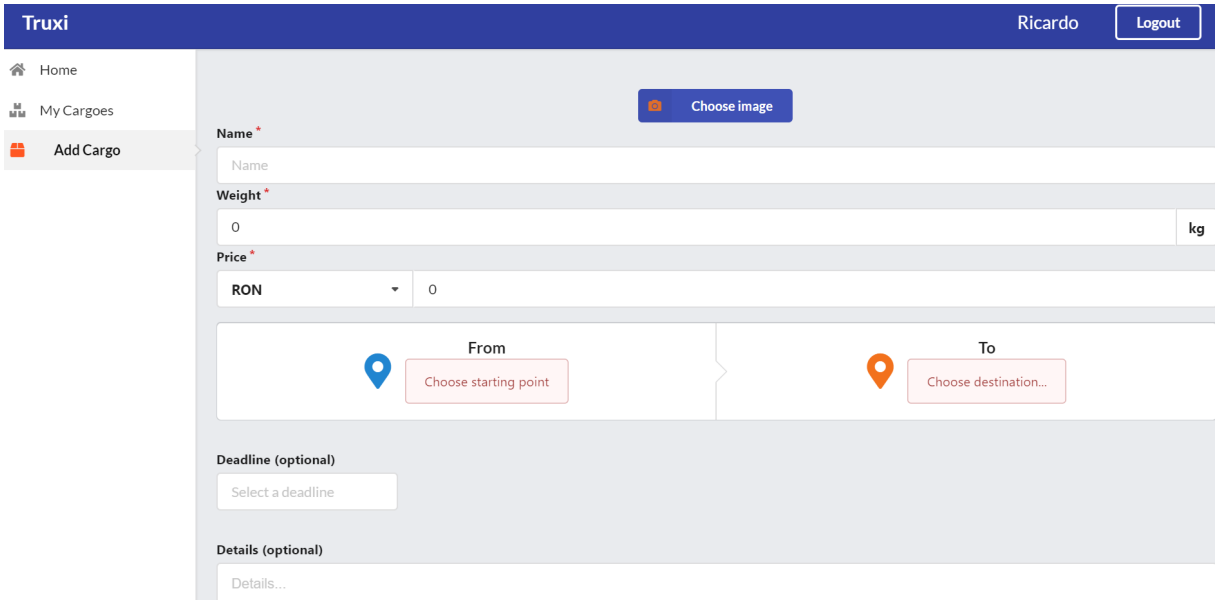
A webalkalmazás a munkahirdető szerepkörű felhasználóknak biztosít működési felületet. A webalkalmazás nem használható, csak miután a felhasználó bejelentkezett, ezért vendégek számára csak a 8. és 9. ábrán látható oldalak elérhetőek.



8. ábra. A regisztrációs nézet a webalkalmazásban. A munkahirdetők meg kell adják a nevüket, az email címüket, a telefonszámukat és egy erős jelszót.

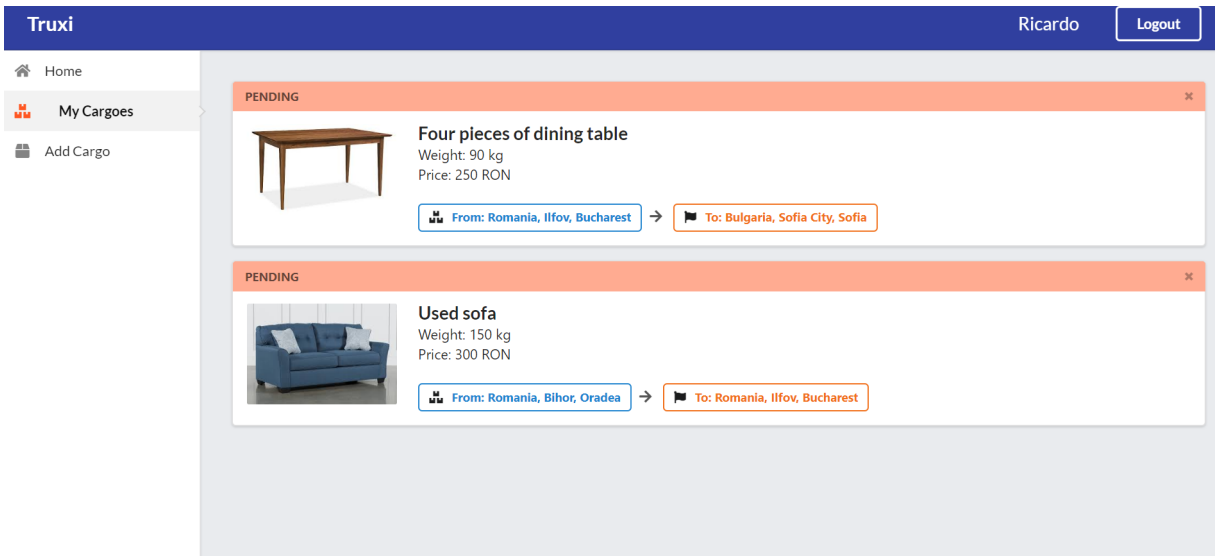


9. ábra. A bejelentkezési felület a webalkalmazásban a munkahirdetők számára. A felhasználók az email címükkel és a jelszavukkal kell azonosítsák magukat.

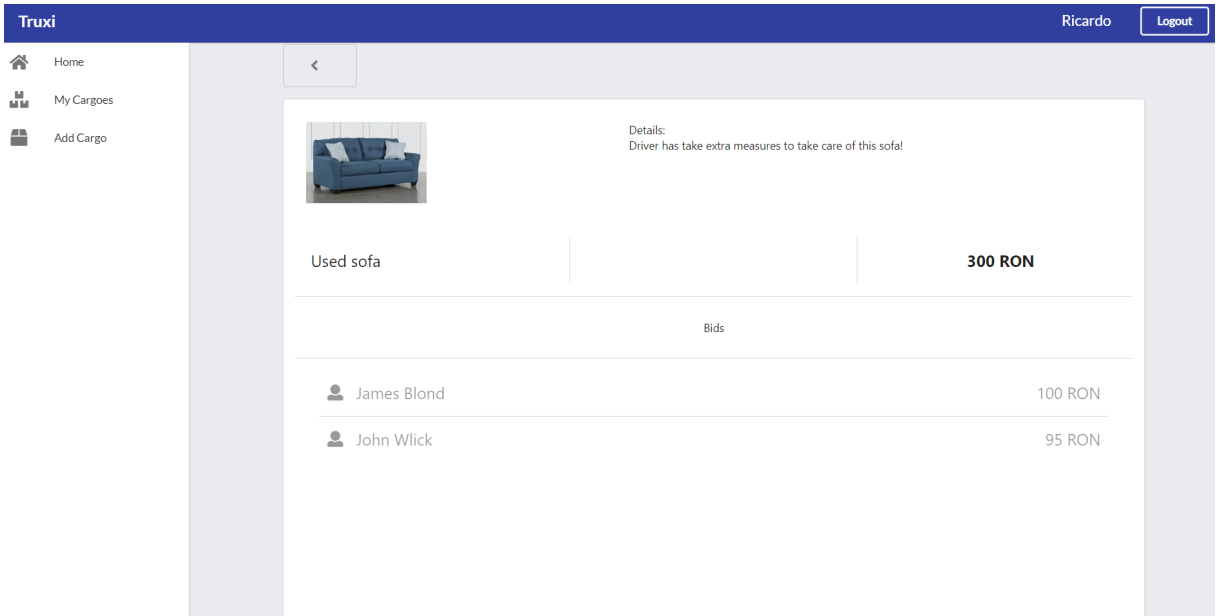


10. ábra. Új szállítmány létrehozására használt nézet. A munkahirdetőnek meg kell adnia a szállítmány nevét, súlyát, árát, határidejét, részleteket róla. Ezenfelül, ki kell választania egy térképen, hogy honnan, illetve hova szeretné, hogy el legyen szállítva.

Bejelentkezés után a munkahirdető hozhat létre szállítmányokat (10. ábra). Szállítmány létrehozásakor kötelezően meg kell adnia a szállítmány nevét, súlyát, árát, illetve azt, hogy honnan, illetve hova szeretné elszállíttatni. A helyszínek megadásához rendelkezésre áll egy interaktív térkép és két keresőmező. A keresőmezőkkel ki lehet keresni a két lokációt (honnan és hova), vagy a térképen be lehet jelölni egy markerrel a pontos helyszíneket. Opcionálisan lehet adni egy rövidebb leírást a szállítmányról, egy határidőt, illetve képet is fel lehet tölteni hozzá.



11. ábra. Listanézet egy munkahirdető által létrehozott szállítmányokkal. Egy szállítmány már nem elérhető állatpotba kerül, ha a munkahirdető elfogadja az egyik rá érkezett ajánlatot. Ajánlat elfogadása után ezen a nézeten kapja meg a választott sofőr elérhetőségeit.

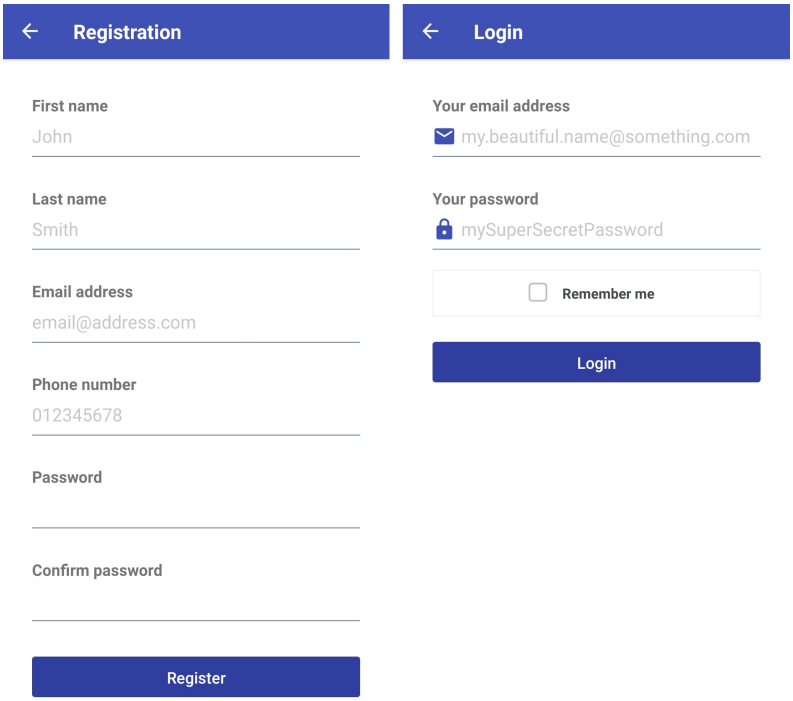


12. ábra. Nézet, amin egy munkahirdető látja, hogy kik helyeztek el ajánlatot egy adott szállítmányra. Adott sofőrre kattintva el lehet fogadni az ajánlatot.

A munkahirdető felhasználó böngészheti az általa létrehozott szállítmányokat (11. ábra). Kétfelé osztható a szállítmányok listája: elérhető szállítmányok, amelyeknek esetében még nem volt elfogadva egy sofőr ajánlata sem, illetve már nem elérhető szállítmányok, amelyeknek már megvan a sofőrje. A nem elérhetőek esetében, ha rájuk kattint a munkahirdető, láthatja a győztes ajánlatot megtévő sofőr privát információját (telefonszámát). A még elérhető szállítmányok esetében, kattintásra a beérkező ajánlatokat listázó nézetre lesz navigálva a felhasználó (12), ahol láthatja, hogy a sofőrök milyen árajánlatokat tettek a szállítmányára. Bármelyik árajánlatra kattintva, a megerősítés után, elfogadottnak lehet nyílvánítani azt, és az

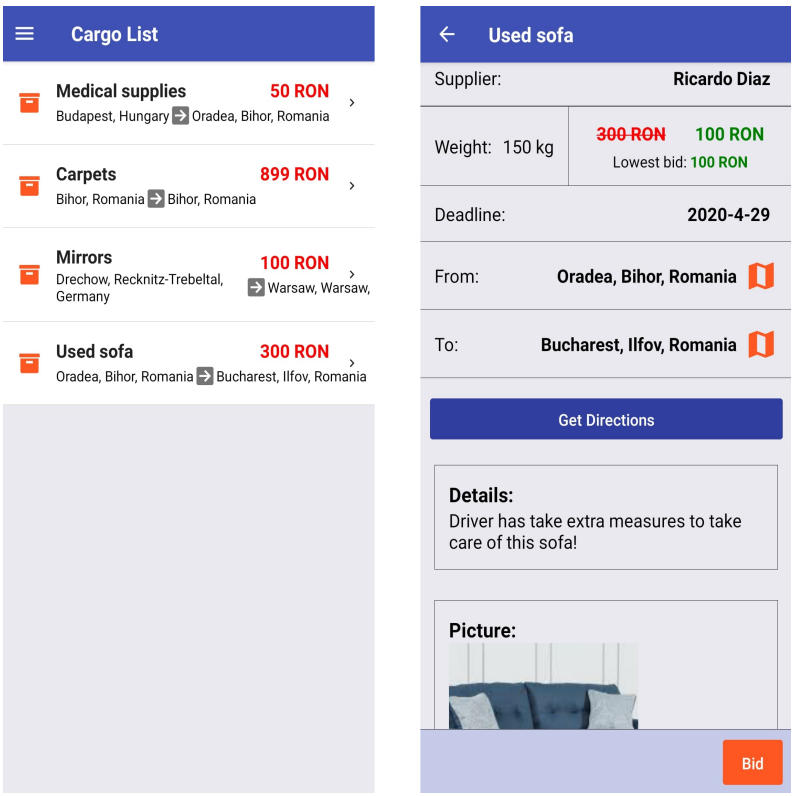
adott szállítmány azután nem lesz már elérhető a többi sofőr számára. Ha volt több ajánlat is az adott szállítmányra, bármelyik elfogadásakor a többi ajánlat állapota elutasítottá válik. Ajánlat elfogadásakor a munkahirdető megkapja a sofőr telefonszámát, ami addig nem elérhető számára. A munkahirdetőnek lehetősége van törölni a még el nem fogadott szállítmányait.

4.2. Mobil alkalmazás

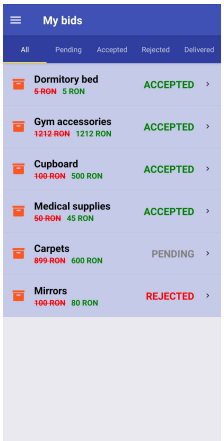


13. ábra. A regisztrációs és a bejelentkezési nézetek a mobilalkalmazásban. A sofőrök meg kell adják a nevüket, az email címüket, a telefonszámukat és egy erős jelszót. Az utóbbi kettővel tudnak bejelentkezni.

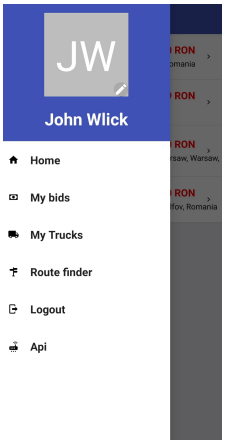
A mobilalkalmazást a sofőr szerepkörű felhasználók tudják regisztrálás és bejelentkezés (13. ábra) után használatba venni. A vendégek az applikáció elindulása után csak a kezdőképernyőt érik el, amin az elérhető szállítmányok listája van megjelenítve (14. ábra). A listában egy szállítmányra koppintva, a felhasználó át lesz navigálva a szállítmányrészletező nézetre, ahol a szállítmány minden adatát megtekintheti. Itt látható a tulajdonos által megadott irányár, a sofőr esetleges ajánlatának az értéke és a legalacsonyabb ajánlott ár is. A kis térképek valamelyikére kattintva a helyszínek nevei mellett az alkalmazás megjeleníti a lokációt a térképen, a telefon lokális térkép-alkalmazásában, amely *Google Maps* [41] Android, Apple Maps [42] az iOS operációs rendszer esetében. Amíg nincs elfogadva a sofőr ajánlata egy szállítmányra, addig a honnan és hova helyszínek pontos koordinátái nem elérhetőek számára, csak a helyszínek városa/helysége.



14. ábra. Az elérhető szállítmányok listája és a szállítmány-részletező nézet. A sofőr bármelyik szállítmányra kattintva extra információhoz juthat azzal kapcsolatban.



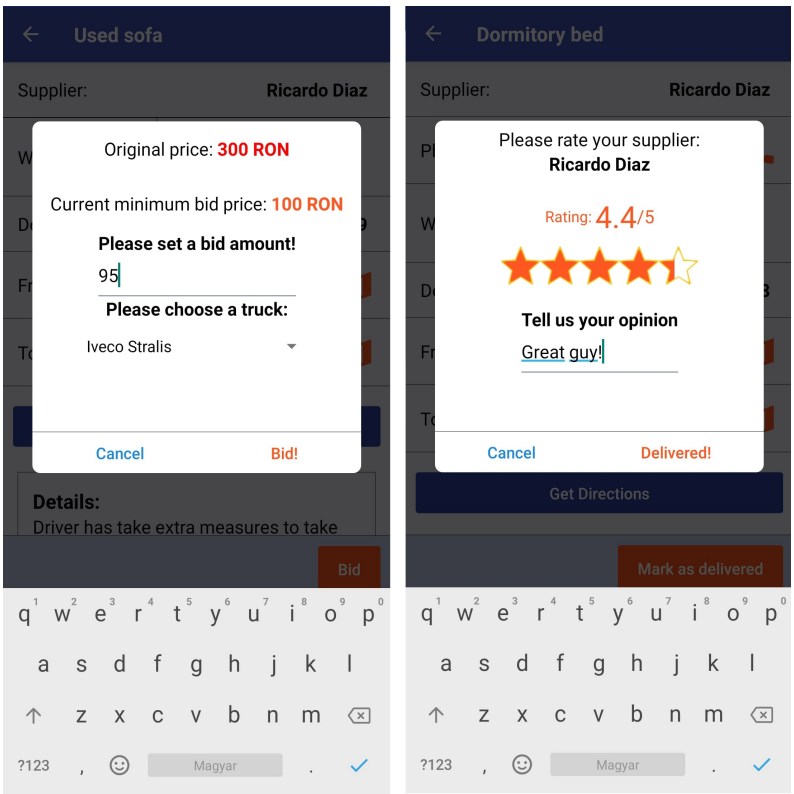
15. ábra. A sofőr eddig tett ajánlatainak a listázására és menedzselésére alkalmas nézet. Az ajánlatokat vegyesen és szűrve is lehet böngészni.



16. ábra. A navigációt biztosító kihúzható oldalsó menü. Egyszerűen el lehet érni segítségével az összes nézetet. Emellett megjeleníti bejelentkezett felhasználó teljes nevét és képét, vagy, ha nincs képe, akkor nevének kezdőbetűit.

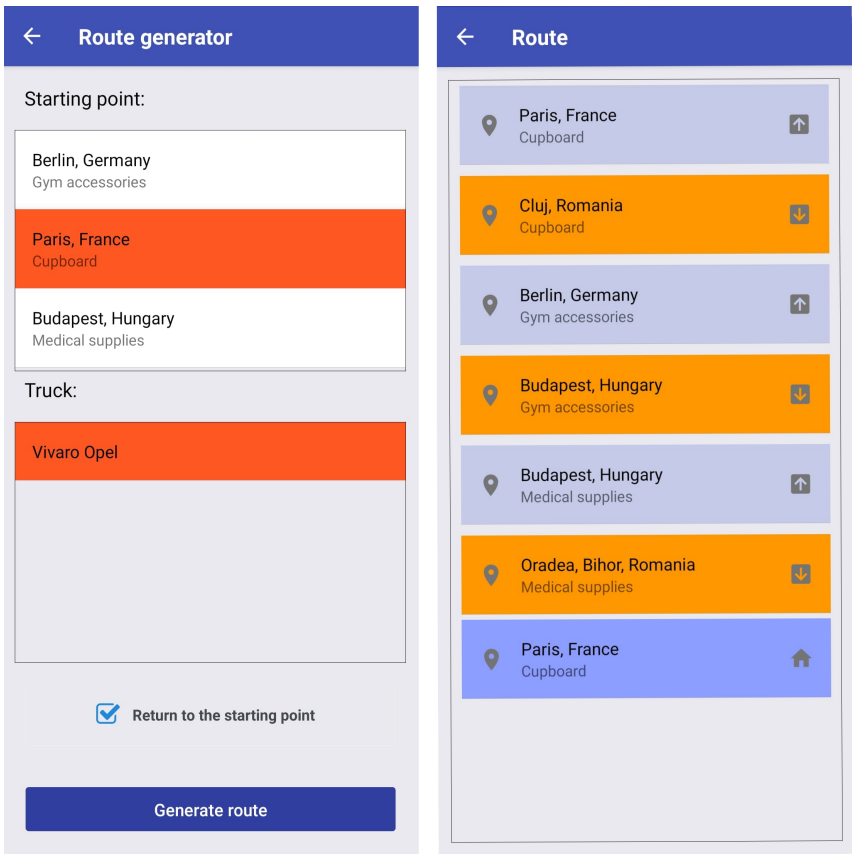
A navigáció az alkalmazásban a bal oldalról kihúzható menü (16. ábra) segítségével történik. Ha az *Ajánlataim* (*My Bids*) menüpontra navigál a sofőr, bejön a nézet, ami a sofőr ajánlatainak listázására és menedzselésére alkalmas (15. ábra). Az ajánlatok állapotuk szerint szűrve vannak egy fülnézet segítségével, ahol minden fül különböző állapotú ajánlatokat tartalmaz, de van egy fül, ami az összes ajánlatot tartalmazza vegyesen. A függő állapotban levő ajánlatokat jobbra

húzva lehet törölni, a többi állapotú ajánlatot nem lehet törölni. Ha rákoppint a sofőr bármelyik ajánlatra, akkor a szállítmány megjelenítődik a szállítmány-részletező nézetben (14. ábra). Az, hogy milyen akciók jelennek meg a részletező nézet alján, az függ az ajánlat állapotától. Ha az ajánlat állapota függő, akkor az ajánlat értékének a módosítására és a használandó jármű kicserélésére alkalmas modálablak lesz elérhető. Ha az ajánlat állapota elfogadott, akkor a sofőr, miután elszállította a szállítmányt, a rendszerben is elszállított állapotba helyezheti az ajánlatot és egyben értékelheti is a szállítmány tulajdonosát az értékelő modállal (17. ábra). Ha az ajánlat elszállított állapotban van, akkor a sofőr módosíthatja a tulajdonosra tett értékelést.



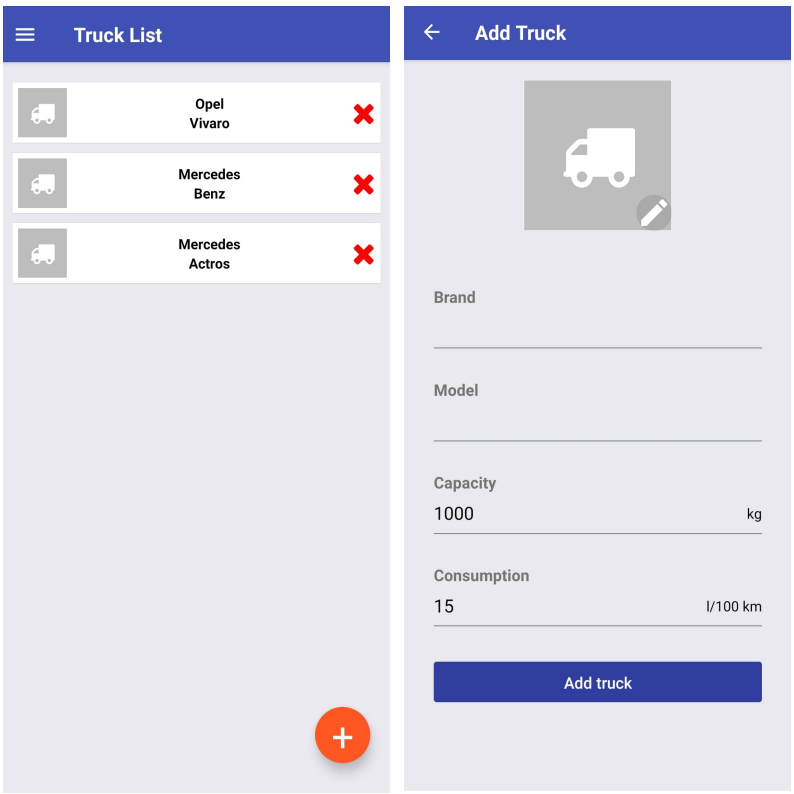
17. ábra. Új ajánlat megtételére és munkahirdető értékelésére alkalmas modál-nézetek.

A *Járműveim* (*My trucks*) menüpont alatt a járműmenedzser nézet (19. ábra) jön be, ahol a sofőr láthatja az eddig beregisztrált járműveit, illetve át tud navigálni az új jármű regisztrálására alkalmas nézetre.



18. ábra. Az optimális útvonal paramétereinek beállítására és a kigenerált útvonal pontjainak listázására alkalmas nézetek. A beállított paraméterek alapján kigenerált útvonal pontjai látogatási sorrendbe vannak rendezve. A halványkék háttérszínű listaelemek a felvevési pontok, míg a narancsszínű elemek a leadási pontok.

Az *Útvonal kereső* (*Route finder*) menüpont alatt egy olyan nézetre lesz navigálva a felhasználó, amelynek segítségével be tudja állítani az optimális útvonalgenerálás paramétereit (18. ábra). Az útvonal olyan szállítmányokból áll, amikre az ő ajánlata lett elfogadva a tulajdonosok által. Ki kell választania, hogy melyik szállítmány felvevési címéről akarja indítani a körutat és, hogy melyik járművel szeretné a körutat véghezvinni. Itt azt feltételezzük, hogy ugyanazt a járművet fogja használni a körút teljes ideje alatt, illetve a járműkiválasztásnál a listában csak olyan járművek jelennek meg, amelyek a körút szállítmányainak bármelyikét képesek elszállítani. Utolsó paramétként, egy jelölőnégyzet segítségével be lehet állítani, hogy a körút ott fejeződjön-e be, ahonnan indult. A paraméterek beállítása és a szerver válaszának megérkezése után az útvonalat megjelenítő nézet jön be, ahol a szállítmányok *honnan* és *hova* helyszínei látogatási sorrendben vannak listázva. A könnyebb érthetőség érdekében, a felvevési pontok kék, a leadási pontok narancssárga háttérszínnel vannak feltüntetve, illetve rendelkeznek egy kis ikonnal is, ami segít a megkülönböztetésben. Ha a sofőr vissza szeretne térni a kiindulási pontba, akkor az útvonal végén sötétkékkel lesz ez jelölve.



19. ábra. A sofőr beregisztrált járműveinek a listázására és új jármű létrehozására alkalmas nézetek.

Következtetések és továbbfejlesztési lehetőségek

A dolgozat részletesen bemutatta a Truxi alkalmazást, a szoftverrendszer felépítését, a felhasznált technológiákat és a megvalósítás lépéseit. A rendszer két kliensalkalmazást biztosít a felhasználóknak, attól függően, hogy milyen típusú szerepkört töltenek be. A munkahirdetőnek a webalkalmazás segítségével lehetősége van saját szállítmányok meghirdetésére, amelyeket majd a sofőrök a mobilalkalmazáson keresztül böngészhetnek és árajánlatokat tehetnek rájuk. A munkahirdető a szállítmányokra beérkező árajánlatok közül kiválasztja a neki legmegfelelőbbet, melynek következtében elérheti a söfőr telefonszámát, és létrejöhet a kapcsolat a két fél között. A sofőr megadhatja a teherautóját, valamint optimális útvonalat generálhat a szállítmányok kiszállítására annak érdekében, hogy időt és pénzt spóroljon az út során.

A Truxi alkalmazás tervezése és fejlesztése során felmerült több továbbfejleszthetőségi lehetőség:

- kibővíteni a bejelentkezési opciót a Facebook és Google felhasználókkal való bejelentkezési lehetőségekkel, mindkét kliensalkalmazásban
- a sofőrök kapjanak értesítést arról, ha a munkahirdető elfogadta a szállítmányra tett ajánlatot és megbízta az áru kiszállításával, vagy elutasította az ajánlatot, és egy másik sofőr ajánlatát fogadta el
- nemzetköziesíteni a web és mobilalkalmazásokat, hogy a felhasználók számára több nyelven is elérhetővé váljon a felhasználói felület
- a sofőröknek és munkahirdetőknek lehetőségük legyen az árak átkonvertálására más pénznemekbe
- a felhasználó szerepköréhez tartozó funkcionalitások legyenek mindkét alkalmazásban elérhetőek
- chat funkció hozzáadása, ami a felhasználók közötti kommunikációt segítené
- automatikus árbecslő funkció, amely az áru tömege és az út hossza alapján kiszámolja a munkahirdető számára a szállítás költségét
- az optimális útvonalgeneráló algoritmusban a távolságfüggvény által használt euklidészi távolság lecserélése földrajzi távolságra

Hivatkozások

[1] *2-opt algoritmus leírása*. URL: https://www-e.ovgu.de/mertens/TSP/node3.html (elérés dátuma: 2020. ápr. 23.)

[2] *A BrowserRouter hivatalos oldala*. URL: https://reacttraining.com/react-router/web/api/BrowserRouter (elérés dátuma: 2020. ápr. 27.)

[3] *A Git hivatalos dokumentációja*. URL: https://git-scm.com/doc (elérés dátuma: 2020. máj. 2.)

[4] *A history object React Router*. URL: https://reacttraining.com/react-router/web/api/history (elérés dátuma: 2020. ápr. 27.)

[5] *A Leaflet hivatalos oldala*. URL: https://leafletjs.com/ (elérés dátuma: 2020. máj. 3.)

[6] *A localStorage hivatalos dokumentációja*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage (elérés dátuma: 2020. ápr. 27.)

[7] *A React Leaflet hivatalos oldala*. URL: https://react-leaflet.js.org/ (elérés dátuma: 2020. máj. 3.)

[8] *A React Router hivatalos oldala*. URL: https://reacttraining.com/react-router/core/api/Router (elérés dátuma: 2020. ápr. 27.)

[9] *A React.Component hivatalos leírása*. URL: https://reactjs.org/docs/react-component.html (elérés dátuma: 2020. ápr. 27.)

[10] *A Semnatic UI React hivatalos dokumnetációja*. URL: https://react.semantic-ui.com/ (elérés dátuma: 2020. máj. 2.)

[11] *Access Tokens leírás*. URL: https://auth0.com/docs/tokens/concepts/access-tokens (elérés dátuma: 2020. ápr. 15.)

[12] *AttributeOverride hivatalos dokumentáció*. URL: https://docs.oracle.com/javaee/6/api/javax/persistence/AttributeOverride.html (elérés dátuma: 2020. ápr. 8.)

[13] *Axios GitHub oldal*. URL: https://github.com/axios/axios (elérés dátuma: 2020. ápr. 26.)

[14] *Axios Interceptor-ok hivatalos leírás*. URL: https://github.com/axios/axios#interceptors (elérés dátuma: 2020. ápr. 26.)

[15] *BCryptPasswordEncoder hivatalos dokumentáció*. URL: https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html (elérés dátuma: 2020. ápr. 12.)

[16] *Cargopedia*. URL: https://www.cargopedia.ro/ (elérés dátuma: 2020. márc. 24.)

[17] *Cloudinary felhőszolgátatás képtárolásra, Java integráció hivatalos dokumentációja*. URL: https://cloudinary.com/documentation/java_integration (elérés dátuma: 2020. ápr. 8.)

[18] *ControllerAdvice hivatalos dokumentáció*. URL: https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html (elérés dátuma: 2020. ápr. 15.)

[19] *Dependecy injection leírása*. URL: https://en.wikipedia.org/wiki/Dependency_injection (elérés dátuma: 2020. ápr. 29.)

[20] *Docker hivatalos weboldal*. URL: https://www.docker.com/ (elérés dátuma: 2020. ápr. 22.)

[21] *Docker-compose hivatalos dokumentáció*. URL: https://docs.docker.com/compose/ (elérés dátuma: 2020. ápr. 22.)

[22] *DTO minta, Martin Fowler*. URL: https://martinfowler.com/eaaCatalog/dataTransferObject.html (elérés dátuma: 2020. ápr. 14.)

[23] *Embeddable hivatalos dokumentáció*. URL: https://docs.oracle.com/javaee/6/api/javax/persistence/Embeddable.html (elérés dátuma: 2020. ápr. 8.)

[24] *ExceptionHandler hivatalos dokumentáció*. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html (elérés dátuma: 2020. ápr. 15.)

[25] *Expo hivatalos dokumentáció*. URL: https://docs.expo.io/ (elérés dátuma: 2020. ápr. 23.)

[26] *Formik hivatalos dokumentáció*. URL: https://jaredpalmer.com/formik/docs/overview (elérés dátuma: 2020. ápr. 23.)

[27] *Git-Flow workflow leírása*. URL: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (elérés dátuma: 2020. ápr. 22.)

[28] *GitLab CI/CD leírása*. URL: https://docs.gitlab.com/ee/ci/ (elérés dátuma: 2020. ápr. 22.)

[29] *GitLab hivatalos weboldal*. URL: https://about.gitlab.com/ (elérés dátuma: 2020. ápr. 22.)

[30] *Gradle hivatalos weboldal*. URL: https://gradle.org/ (elérés dátuma: 2020. ápr. 21.)

[31] *Hibernate hivatalos oldal*. URL: https://hibernate.org/ (elérés dátuma: 2020. ápr. 15.)

[32] *Hibernate Validator hivatalos dokumentáció*. URL: http://hibernate.org/validator/ (elérés dátuma: 2020. ápr. 15.)

[33] *Hivatalos Spring-Data dokumentáció*. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference (elérés dátuma: 2020. ápr. 1.)

[34] *InversifyJs hivatalos dokumentáció*. URL: https://github.com/inversify/InversifyJS (elérés dátuma: 2020. ápr. 23.)

[35] *Inversion of control leírása*. URL: https://en.wikipedia.org/wiki/Inversion_of_control (elérés dátuma: 2020. ápr. 29.)

[36] *Java POJO*. URL: https://www.martinfowler.com/bliki/POJO.html (elérés dátuma: 2020. ápr. 1.)

[37] *JSON formátum hivatalos weboldal*. URL: https://www.json.org/json-en.html (elérés dátuma: 2020. ápr. 14.)

[38] *JSX hivatalos dokumentáció*. URL: https://reactjs.org/docs/introducing-jsx.html (elérés dátuma: 2020. ápr. 23.)

[39] *JWT hivatalos dokumentáció*. URL: https://jwt.io/introduction/ (elérés dátuma: 2020. ápr. 15.)

[40] *Liquibase hivatalos dokumentáció*. URL: https://www.liquibase.org/documentation/index.html (elérés dátuma: 2020. ápr. 22.)

[41] *Maps térkép-alkalmazás a Play Store-on*. URL: https://play.google.com/store/apps/details?id=com.google.android.apps.maps&hl=en (elérés dátuma: 2020. ápr. 29.)

[42] *Maps térkép-alkalmazás az App Store-on*. URL: https://apps.apple.com/us/app/maps/id915056765 (elérés dátuma: 2020. ápr. 29.)

[43] *Mobx hivatalos dokumentáció*. URL: https://mobx.js.org/README.html (elérés dátuma: 2020. ápr. 23.)

[44] *Multipart form data*. URL: https://tools.ietf.org/html/rfc2388 (elérés dátuma: 2020. ápr. 14.)

[45] *MySql hivatalos weboldal*. URL: https://www.mysql.com/ (elérés dátuma: 2020. ápr. 22.)

[46] *Olx transport marfa*. URL: https://www.olx.ro/oferte/q-transport-marfa/ (elérés dátuma: 2020. márc. 24.)

[47] *React components hivatalos leírás*. URL: https://reactjs.org/docs/components-and-props.html (elérés dátuma: 2020. ápr. 25.)

[48] *React Native Elements hivatalos weboldal*. URL: https://react-native-elements.github.io/react-native-elements/ (elérés dátuma: 2020. ápr. 23.)

[49] *React-Native hivatalos weboldal*. URL: https://reactnative.dev/ (elérés dátuma: 2020. ápr. 23.)

[50] *React-Native-Router-Flux GitHub oldala*. URL: https://github.com/aksonov/react-native-router-flux (elérés dátuma: 2020. ápr. 25.)

[51] *React-Native-Router-Flux Router dokumentáció*. URL: https://github.com/aksonov/react-native-router-flux/blob/master/docs/API.md#router (elérés dátuma: 2020. ápr. 25.)

[52] *React.js hivatalos dokumentációja*. URL: https://reactjs.org/ (elérés dátuma: 2020. ápr. 15.)

[53] *Refresh Token leírás*. URL: https://auth0.com/docs/tokens/concepts/refresh-tokens (elérés dátuma: 2020. ápr. 15.)

[54] *REST leírás*. URL: https://restfulapi.net/ (elérés dátuma: 2020. ápr. 22.)

[55] *Roy Fielding disszertáció, 5. fejezet*. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (elérés dátuma: 2020. ápr. 1.)

[56] *Scrum keretrendszer leírása*. URL: https://www.scrum.org/resources/what-is-scrum (elérés dátuma: 2020. ápr. 22.)

[57] *Single-responsibility principle hatása a React technológiára, leírás*. URL: https://en.wikipedia.org/wiki/Single-responsibility_principle (elérés dátuma: 2020. ápr. 25.)

[58] *Spring Boot hivatalos dokumentáció*. URL: https://spring.io/projects/spring-boot (elérés dátuma: 2020. ápr. 21.)

[59] *Spring Boot Web Started dokumentáció*. URL: https://www.javatpoint.com/spring-boot-starter-web (elérés dátuma: 2020. ápr. 22.)

[60] *Spring Data JPA hivatalos dokumentáció*. URL: https://spring.io/projects/spring-data-jpa (elérés dátuma: 2020. ápr. 15.)

[61] *Spring Dependency Injection*. URL: https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-collaborators (elérés dátuma: 2020. ápr. 8.)

[62] *Spring keretrendszer hivatalos weboldal*. URL: https://spring.io/ (elérés dátuma: 2020. ápr. 21.)

[63] *Spring Security hivatalos dokumentáció*. URL: https://spring.io/projects/spring-security (elérés dátuma: 2020. ápr. 15.)

[64] *Többrétegű architektúra, zárt rétegek*. URL: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html#sapr_0102_img (elérés dátuma: 2020. ápr. 15.)

[65] *Transactional annotáció hivatalos dokumentáció*. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html (elérés dátuma: 2020. ápr. 12.)

[66] *TypeScript hivatalos dokumentáció*. URL: https://www.typescriptlang.org/docs/home.html (elérés dátuma: 2020. ápr. 23.)

[67] *TypeScript interface-k hivatalos dokumentáció*. URL: https://www.typescriptlang.org/docs/handbook/interfaces.html (elérés dátuma: 2020. ápr. 26.)