



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
XXX Tanszék

Szalkai Krisztián

SZEMÉLYFUVAROZÁST TÁMOGATÓ ALKALMAZÁS KÉSZÍTÉSE

KONZULENS

Albert István

BUDAPEST, 2021

Tartalomjegyzék

Összefoglaló	5
Abstract.....	6
1 Bevezetés	7
1.1 A téma.....	7
1.2 A választott platformok	7
1.3 A dokumentum felépítése	7
2 Technológiák	8
2.1 .NET Core	8
2.1.1 Entity Framework Core	8
2.1.2 ASP.NET Core Identity	9
2.1.3 Swashbuckle	9
2.1.4 Barion Client.....	9
2.1.5 ASP.NET Core SpaServices	9
2.2 React	10
2.2.1 React router.....	11
2.2.2 Material UI.....	12
2.2.3 Axios.....	12
2.2.4 Google Maps Places.....	13
2.3 TypeScript.....	13
2.3.1 Redux	14
3 Tervezés	15
3.1 Funkcionális követelmények	15
3.2 A weboldal működésével kapcsolatos elvárások.....	16
3.3 Architektúra	17
3.4 Felülettervek	18
3.5 Az adatbázis felépítése.....	25
4 Megvalósítás	28
4.1 Backend	28
4.1.1 Felépítés	28
4.1.2 Az adatbázis létrehozása	29
4.1.3 TaxiService.Web.....	30

4.1.4 TaxiService.Dal	40
4.1.5 TaxiService.Dto	42
4.1.6 TaxiService.Bll	44
4.2 Frontend	51
4.2.1 Felépítés	52
4.2.2 Funkciók	61
5 Összefoglaló	68
5.1 Továbbfejlesztési lehetőségek	68
5.2 Végző	68
6 Irodalomjegyzék.....	70

Hallgatói nyilatkozat

Alulírott **Szalkai Krisztián**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzé tegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 12. 09.

.....
Szalkai Krisztián

Összefoglaló

Napjainkban a rohamosan növekvő népesség és igényei nehéz helyzetbe kényszeríti a fővárosi közlekedést, mivel a kialakult struktúra nehezen változtatható, új utak, parkolóhelyek létrehozása sok helyen nehezen kivitelezhető. Az átlag életszínvonal növekedésével a tömegközlekedés nehezen tud lépést tartani, így az emberek jelentős része járna autóval, ez pedig az utak nagyobb leterheltségéhez vezet. A probléma mérséklésére számos alternatíva jelent meg. Ezek közül a kényelmes, de legtöbbször drága taxi szolgáltatások és a kevésbé komfortos, de általában olcsóbb közösségi közlekedési opciók (Mol Limo stb.) ötvözése az alkalmazás célja, mely egy megfizethető és kényelmes alternatívát nyújtana a városi közlekedésre.

Az alkalmazás három szolgáltatást nyújt, használható hagyományos taxi szolgáltatásként, azonnali autórendeléssel, lehetőséget nyújt adott időpontra előre történő autófoglalásra egy útra, például repülőtér vagy színház megközelítésekor, illetve bérelhető autó sofőrrel egy megadott időtartamra, például bevásárlás vagy ügyintézés esetére. A szolgáltatás előnye, hogy összefogja a hasonló szektorokat, így a városi közlekedésre készülő leg több igényét ki tudja elégíteni, ezzel növelve a potenciális felhasználók számát.

Jelen dokumentumban bemutatom a megoldás során használt technológiákat, a rendszer megtervezésének lépéseit, feltárom a megvalósításom lépéseit, az aközben felmerülő problémákat, végül leírom az alkalmazás továbbfejlesztési lehetőségeit, a megvalósítás során szerzett tapasztalataimat. A rendszer egy web szerverből és egy webes felületből áll.

Abstract

This day and age, the rapidly growing population and their needs place capital cities' traffic management in a predicament, since the already built infrastructure is hard to change, new roads, parking spaces are mostly hard to build. Public transport networks have a hard time with keeping up with the ever-rising average living standards of the metropolitan population, so more and more people prefer to use a car, and that leads to the roads being even more pressured. To face this problem several solutions arose: From the many, this application attempts to merge the more comfortable, but usually less affordable taxi services, with the more affordable but usually less comfortable community transport options (such as Mol limo etc.), which would result in a both affordable and comfortable solution for metropolitan transport.

The application incorporates three services: It works as a regular taxi service, where you can pay for a ride from one place to another immediately, you have the option to book a ride in advance, for example if you know you will have to be at the airport or at a theatre at a given time, additionally, you can rent a chauffeur and a car for a given period of time, useful for shopping, or if you need to take care of business in multiple places throughout the day. The advantage of such a service is that it incorporates similar sectors, and it satisfies the needs of most who want to travel in/to the city, thus increasing the number of potential users.

In this document I will introduce the technologies I used, the steps of designing and implementing the system and show how I solved the problems that occurred while developing the system. Lastly, I will present some ideas for further development and summarize what I've learned during the project. The system consists of a web server, and a web application.

1 Bevezetés

A következő fejezetben kifejtem miért ezt a témát választottam, illetve kitérek néhány technológiai/platform választás kérdésére.

1.1 A téma

Napjainkban a rohamosan növekvő népesség és igényei nehéz helyzetbe kényszeríti a fővárosi közlekedést, mivel a kialakult struktúra nehezen változtatható, új utak, parkolóhelyek létrehozása sok helyen nehezen kivitelezhető. Az átlag életszínvonal növekedésével a tömegközlekedés nehezen tud lépést tartani, így az emberek jelentős része járna autóval, ez pedig az utak nagyobb leterheltségéhez vezet. A probléma mérséklésére számos alternatíva jelent meg. Ezek közül a kényelmes de legtöbbször drága taxi szolgáltatások és a kevésbé komfortos de általában olcsóbb közösségi közlekedési opciók (Mol Limo stb.) ötvözése az alkalmazás célja, mely egy megfizethető és kényelmes alternatívát nyújtana a városi közlekedésre.

1.2 A választott platformok

A téma kiválasztása után el kellett döntenem milyen platformon fogom megvalósítani a szolgáltatásokat. A döntésben figyelembe vettem a jelenlegi szakmai trendeket, a hasonló alkalmazások megvalósításait, illetve szerepet játszott a saját preferenciám, az általam ismert programozási nyelvekben szerzett tapasztalataim is.

1.3 A dokumentum felépítése

A továbbiakban bemutatom a használt technológiákat, használatuk előnyeit. Azután felvázolom a tervezés folyamatát, a megvalósítandó konkrét feladatokat, a létrehozandó adatbázis felépítését. Végül rátérek a megvalósítás tényleges folyamatára, az elkészült program bemutatására.

2 Technológiák

Az alábbiakban az általam választott technológiákat mutatom be általánosságban, illetve amennyiben nem egyértelmű a választás, megemlítem a választási lehetőségeket és a mérlegelési pontok felsorolásával alátámasztom a választásaimat.

2.1 .NET Core

Mivel a relációs adatbázis megvalósítására a .Net fejlesztői platformot választottam, így a használt technológia a .Net Core, hiszen a sokak által használt és szeretett .Net Framework továbbfejlesztett utódja (Scott Hanselman, 2019).

A .Net Core egy nyílt forráskódú, platform független keretrendszer, mely ideális szerveralkalmazások elkészítésére. Az ASP.NET Core lehetővé teszi egy állapotmentes Web Application Programming Interface (API) készítését, ami egy Hyper Text Transfer Protocol (HTTP) alapú kommunikációra alapuló szolgáltatás, így több platformról is elérhető, legyen az webes, asztali vagy telefonos.

A backend megvalósításához meggondoltam még a Java használatát, mint a másik vezető backend platform a piacon. A két platform körül folyamatos a „vita” a fejlesztők között, eddig úgy tűnik, hogy mindkét nyelv nagyon hasonló tulajdonságokkal rendelkezik (Patel, 2020), így leginkább a személyes preferencia játszott szerepet a döntésben.

2.1.1 Entity Framework Core

Az Entity Framework Core (EF Core) egy lecsupaszított, platformfüggetlen verziója az Entity Frameworknek, ami egy Microsoft által fejlesztett objektum relációs leképző (ORM) keretrendszer (Microsoft team, 2018). Segítségével egyszerű .NET-es objektumokon végezhetünk műveleteket C# nyelven, amit az EF Core változáskövetője átalakít az adatbázis által értelmezhető parancsokká és végrehajttatja őket. A Language Integrated Query (LINQ) szintakszissal együtt használva könnyen értelmezhető kódbázist készíthetünk az adatsémánk manipulálása céljából. Ezen felül többféle adatbázis rendszert is támogat, így a fejlesztő számára transzparens módon tud ugyanabból a C# kódból Oracle, MSSQL, SQLite, PostgreSQL vagy akár NoSQL által futtatható kódot generálni.

2.1.2 ASP.NET Core Identity

Az ASP.NET Core Identity egy úgynevezett „tagsági” rendszer, mellyel bejelentkezési funkcionalitást adhatunk .Net Core alkalmazásunkhoz. Az Identity nem csak az adott alkalmazásban létrehozott felhasználói fiókokkal történő belépést támogatja, hanem külső szolgáltatóktól származó bejelentkeztetés is támogatható (Ilyen például a Facebook, Google, vagy Twitter) (Rick Anderson, 2019).

Ennek nem csak az az előnye, hogy felhasználóinkat azonosítani tudjuk-hitelesíteni-, a rendszer támogatja a felhasználók meghatalmazási szintjének ellenőrzését is. Az általam fejlesztett rendszer szempontjából például ilyen meghatalmazási csoportok lehetnek a felhasználók adminisztrátorok és a dolgozók.

2.1.3 Swashbuckle

A Swashbuckle egy olyan könyvtár mely segítségével a fejlesztő könnyebben dokumentálhatja, tesztelheti Representational State Transfer (REST) architektúrájú alkalmazását. Képes a megírt C# Application programming interface (API)-t felhasználva automatikusan egy olyan felületet generálni, melyről a kérdéses végpontok tesztelhetők, illetve a kód megfelelő kommentezésével az alkalmazás dokumentálását is nagyban elősegíti.

2.1.4 Barion Client

Az alkalmazáshoz többféle fizetési módszer is integrálható, én a Barion szolgáltatót választottam, mint elsőként megvalósított rendszer. A szolgáltatás integrációját hivatott segíteni a Barion Client NuGet csomag, mely a leggyakoribb API hívásokat és JSON struktúrákat csomagolja C# osztályokba.

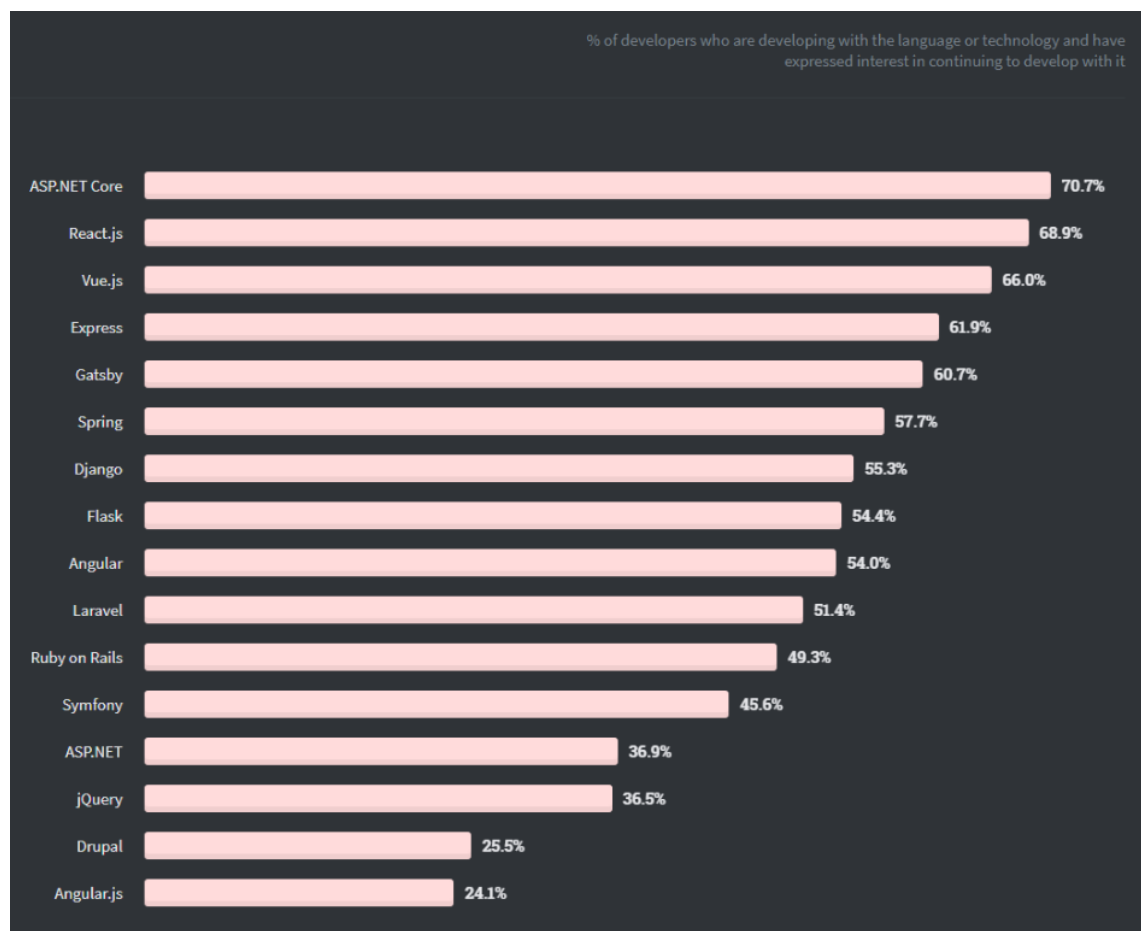
2.1.5 ASP.NET Core SpaServices

Az alkalmazás felépítése lehetővé teszi, hogy külön-külön, vagy egyben telepítsük ki a backendet és a frontendet. Az egyben történő telepítést könnyíti meg az Spa Services nevű NuGet csomag, mely elfedi a frontend által használt statikus fájlok kezelését egyetlen konfigurációs hívás mögé. A fordítást és a telepítést ezen kívül a backend fordítási direktíváinak beállításával támogatnunk kell, ezekről a Megvalósítás fejezetben részletesen is beszélni fogok.

2.2 React

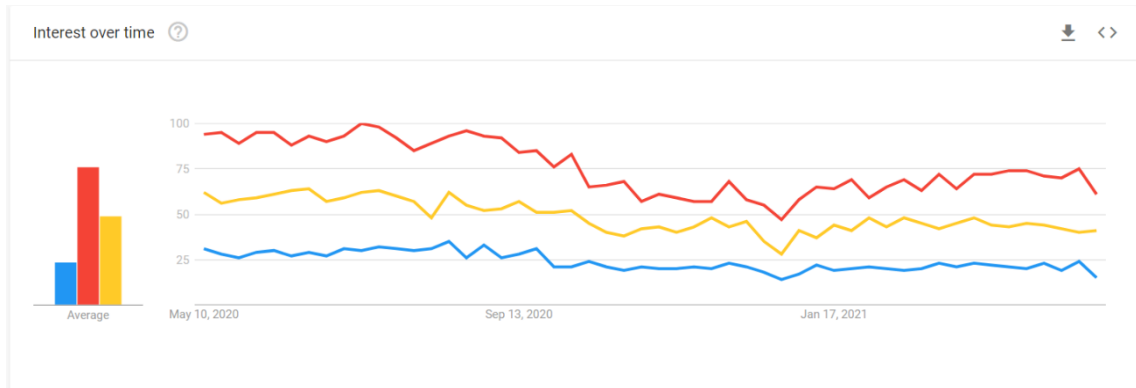
2021-ben elképzelhetetlen a web fejlesztés Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), és JavaScript nélkül. Manapság a JavaScript a frontend fejlesztés lelke, felvetül tehát a kérdés, hogy melyik User Interface (UI) könyvtárat válasszuk a számos lehetőség közül (Jigar Mistry, 2021). A választás nem könnyű, hiszen a piac és a vélemények folyamatosan változnak, azonban az utóbbi időben kialakuló statisztikák alapján három nagyobb keretrendszer emelkedett ki. Ezek a React, az Angular és a Vue.

A stackoverflow online lap kutatása szerint (Stack Overflow, 2021) a legkedveltebb web keretrendszer – az Modell-View-Controller (MVC) alapokon működő ASP.NET Core után – A React volt, azt pedig a Vue követte, az Angular pedig igencsak lemaradva jelent meg a listában.



2.1.5-1 A stackoverflow 2020-as statisztikája

A Google keresési statisztikái szerint (Google, 2021) viszont, ugyan szinten a React vezet, de az Angular követi, a Vue pedig utolsóként jelenik meg érdeklődést tekintve (keresések száma).



2.1.5-2 A google 12 hónapos keresési statisztikája 2021.05.07.-én

A statisztikák alapján tehát a React nyeri a versenyt, nézzük is meg, hogy miért lehet népszerűbb a többinél, illetve mi is a React.

A React ahogy fentiekben említettem egy UI keretrendszer, JS könyvtár, melynek segítségével egy weblap felhasználói felületét készíthetjük el. Lényeges különbség például az Angular-hoz képest, hogy ez egy rendkívül könnyűsúlyú csomag, igazából nem is nevezhető keretrendszernek inkább csak egy könyvtár. Ez azt jelenti a gyakorlatban, hogy minden olyan feladathoz, ami nem kapcsolódik szorosan a megjelenéshez külső könyvtárakat kell használnunk, például a navigációhoz a React routert, a hálózati kommunikációhoz az Axios-t vagy hasonló segédkönyvtárakat. Nagy előnye legtöbb vetélytársával szemben, hogy virtuális Document Object Model-t (DOM) használ, mellyel sokkal jobb teljesítményt nyújt, mint azok a megoldások amik valós DOM-ot használnak- Például az Angular. Tanulhatóság és a fejlesztés görbülékenysége szempontjából szintén érdemes olyan eszközt választani, amely nagy elterjedtséggel bír, mivel könnyebb hozzá segédanyagokat találni.

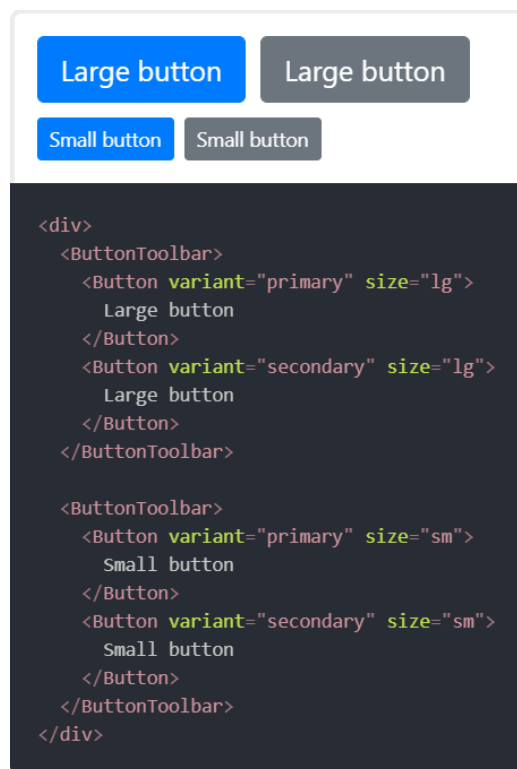
2.2.1 React router

A legtöbb modern web alkalmazás úgynevezett Single page application (SPA) ami azt jelenti, hogy a megjelenítendő oldalak nem több weblapból tevődnek össze, hanem csak egy lapon változik a tartalom a megjelenítendő komponensek függvényében. Ezt a navigációt segíti elő a React router, anélkül navigálhatunk az oldalunkon, hogy ez

az „egy oldal” amivel dolgozunk újra töltődne, fehér töltőképernyőt hagyva a két oldal megjelenítése között.

2.2.2 Material UI

A React UI elemei komponensekből épülnek fel, ami azért hasznos, mert könnyen készíthetünk újra felhasználható HTML elemeket, akár megadható, felhasználásonként változtatható paraméterekkel melyek megváltoztatják a komponens viselkedését. A Material UI egy olyan könyvtár mely a bootstrapből jól ismert előre elkészített kinézeteket adja az alkalmazásunkhoz React komponensek formájában, így a komponensek tulajdonságainak kihasználásával élvezhetjük a bootstrap előnyeit, például paraméter segítségével adhatjuk meg a komponens kinézetének tulajdonságait.



2.2.2-1 A Material UI nyomógomb komponensének egy példája

2.2.3 Axios

Egy web alkalmazás természetesen nehezen képzelhető el valamiféle backend kommunikáció nélkül. Ezt segíti elő az Axios, ami egy úgynevezett „promise-based” azaz ígéret alapú JS könyvtár HTTP kérések kezelésére, ami azt jelenti, hogy aszinkron kéréseket valósíthatunk meg, illetve a szerver válasza alapján tudunk elemeket betölteni, hibát kezelni (Flavio Copes, 2018). Az Axios előnyei közé tartozik még az natív fetch

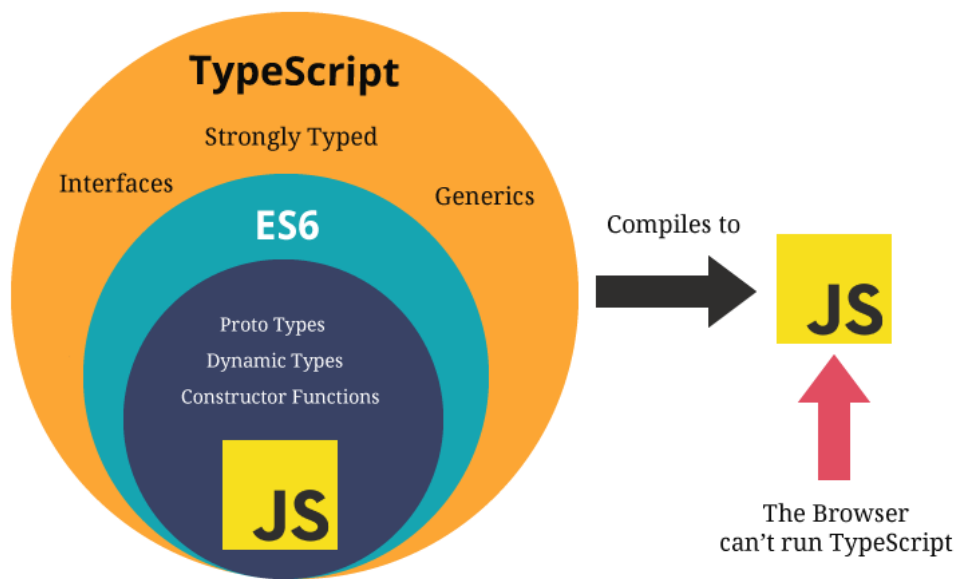
API-val szemben, hogy támogatja a régebbi böngészőket is, képes a kérések megszakítására, illetve beállítható rajta számos alapértelmezett beállítás, mint a válasz időtúllépésének korlátja, vagy olyan fejlécek melyek minden kéréssel elküldésre kerülnek – például egy autorizációs fejléc. Más hasonló könyvtárakkal szemben előnye, hogy beépített Cross-site request forgery (CSRF) védelemmel rendelkezik, illetve automatikus JavaScript Object Notation (JSON) adat átalakításra képes (Nnamandi, 2018).

2.2.4 Google Maps Places

A kiindulási/ érkezési helyek meghatározására, illetve a kettő közötti útvonal hosszának meghatározására, így az ár számítására a Google Places API-t használom. Ennek a szolgáltatásnak része a Google Places Autocomplete, mely a frontenden egy külső JavaScript komponens inicializálásával egy kész megoldást biztosít az API irányába történő kérések kezelésére. A Megvalósítás fejezetben részletesen is ki fogom fejteni ennek a komponensnek a működését.

2.3 TypeScript

A TypeScript (TS) egy JavaScripten (JS) alapuló programozási nyelv, melyet nagyszabású webes felületek kialakításakor célszerű használni, úgy is gondolhatunk rá mint a JavaScript továbbfejlesztése (Rawat, 2019). A TS kódunk egyenesen JS-re fordul, így egy minden platformra forduló biztonságos kódot kapunk. Leglényegesebb előnye a JS-el szemben, hogy erősen típusos és objektum orientált. Ez rengeteget segít, mikor a kódbázis nagyon nagy, elírások és hasonló hibák esetén a fordító hibát fog jelezni, így nem kevés időt megspórolhatunk magunknak fejlesztés során. Népszerűségének még egy oka, hogy mivel közvetlenül JS-re fordul, nyugodtan használhatunk kizárólag JS-ben írt könyvtárakat (Unknown, 2018).



2.2.4-1 TypeScript felépítése, kapcsolata a JavaScripttel

A TypeScript nyelv ezen kívül lehetővé teszi, hogy az új ECMAScript funkciókat használhassuk régebbi böngészőkön is, mivel a fordító az új funkciókból régi-a böngésző által támogatott- verziókra is képesek lefordulni.

Talán egyetlen hátránya – de legalábbis nehézsége – a JavaScripttel szemben, hogy pont a típusosság miatt nehezebb generikus megoldások alkalmazása, mint például a reflection, vagy az Enum értékek Stringként kezelése, ezekről a későbbi fejezetekben részletesebben is beszélni fogok.

2.3.1 Redux

Az SPA alkalmazás állapotainak kezelésére a React-Redux TypeScript könyvtárat használtam. A Redux lehetővé teszi az oldalakon megjelenő információk, állapotok egy úgynevezett „store” -ban történő tárolását. Ennek előnye, hogy nem kell külön karbantartanunk több helyen megjelenő információkat, azokat egységesen tudjuk kezelni, így elkerülhetjük az olyan hibákat, mikor elfelejtjük módosítani egy helyen az adatot, így inkonzisztensen jelenítjük meg a felhasználóknak.

3 Tervezés

Ebben a fejezetben ismertetem a funkcionális követelményeket, a rendszer felépítését, a rendszer részei közti kapcsolatokat, illetve bemutatom az adatbázis struktúráját, az adatok elérésének, tárolásának módját.

3.1 Funkcionális követelmények

Az elkészítendő alkalmazás egy taxi szolgálat weboldala, melyen a felhasználók tudnak regisztrálni, bejelentkezni, időpontokat foglalni a különböző szolgáltatásokhoz, illetve kezelni tudják a saját adataikat, eddigi foglalásaikat. Az alkalmazás ezen kívül lehetőséget nyújt az adminisztrátoroknak az összes leadott foglalás kezelésére, az adminisztrációs tevékenységek elvégzésére, a dolgozók pedig kezelhetik, áttekinthetik a hozzájuk rendelt munkákat (fuvarokat).

A taxiszolgalat által nyújtott szolgáltatások a következők:

- Pontból pontba történő fuvar meghatározott időpontban.
- Adott időtartamra autó bérlete folyamatos sofőrszolgálattal.
- „Normál” taxi működés, azonnali fuvar kérése.

A két előbbi szolgáltatás esetén meghatározható a foglaláshoz, hogy milyen típusú autóval, milyen extra szolgáltatásokkal (dohányzó/nem dohányzó, légkondi kisállatos stb.) kéri a felhasználó.

A weboldalon a felhasználó a következő funkciókat kell elérje:

- Regisztráció
- Belépés
- Foglalás a háromféle szolgáltatásra
- Saját adatok megváltoztatása, törlése
- Előző foglalások megtekintése
- Foglalás lemondása (Adott határidőn belül)
- Kijelentkezés
- Online fizetés

Adminisztrátori belépéssel:

- Foglalások megtekintése
- Foglalások dolgozókhoz rendelése
- Felhasználók és foglalásaik megtekintése

Dolgozói belépéssel:

- Dolgozóhoz rendelt foglalások megtekintése, kezelése.

3.2 A weboldal működésével kapcsolatos elvárások

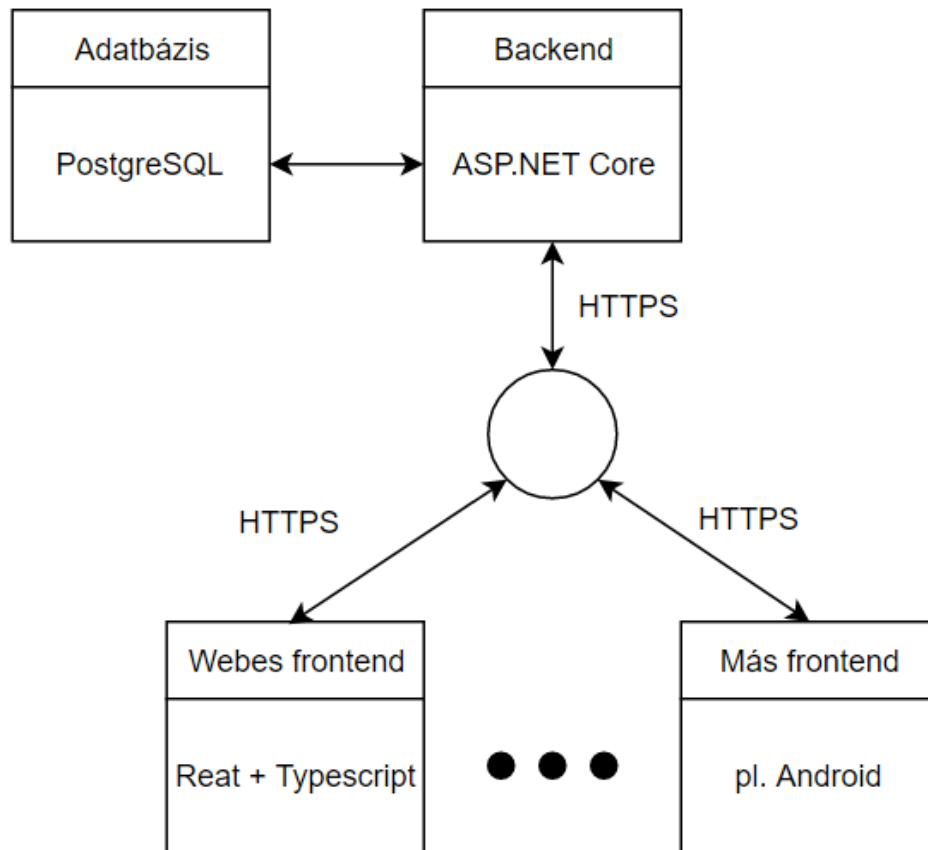
A weboldalnak minden személyes adatot biztonságosan kell tárolnia, az adatokhoz csak a megfelelő jogosultságú felhasználók férhetnek hozzá. A személyes adatok, illetve a felhasználói profil a felhasználó kérésére bármikor törölhető kell legyen.

Foglalás során a rendszernek automatikusan kell elvégezni a következőket:

- Ár számítása a preferenciák, extra szolgáltatások, szolgáltatás típusa és az esetleges felárak (pl.: Dugódíj) alapján.
- Promóciós kódok automatikus kezelése és figyelembevétele a foglaláskor.
- Átirányítás biztonságos fizetési lehetőségekhez (pl: Stripe, PayPal, Barion). Fizetés esetén az esetleges problémák kezelése (Sikertelen fizetés, helytelen összeg, hálózati probléma).
- A foglaláshoz egyedi azonosító generálása és megerősítő email küldése a felhasználónak.
- Határidőn belül történő foglalás lemondása esetén az összeg visszautalása a felhasználónak.

3.3 Architektúra

A kialakított rendszer több részből épül fel, ezek felépítését alább a 2.3.1-1ábra mutatja.



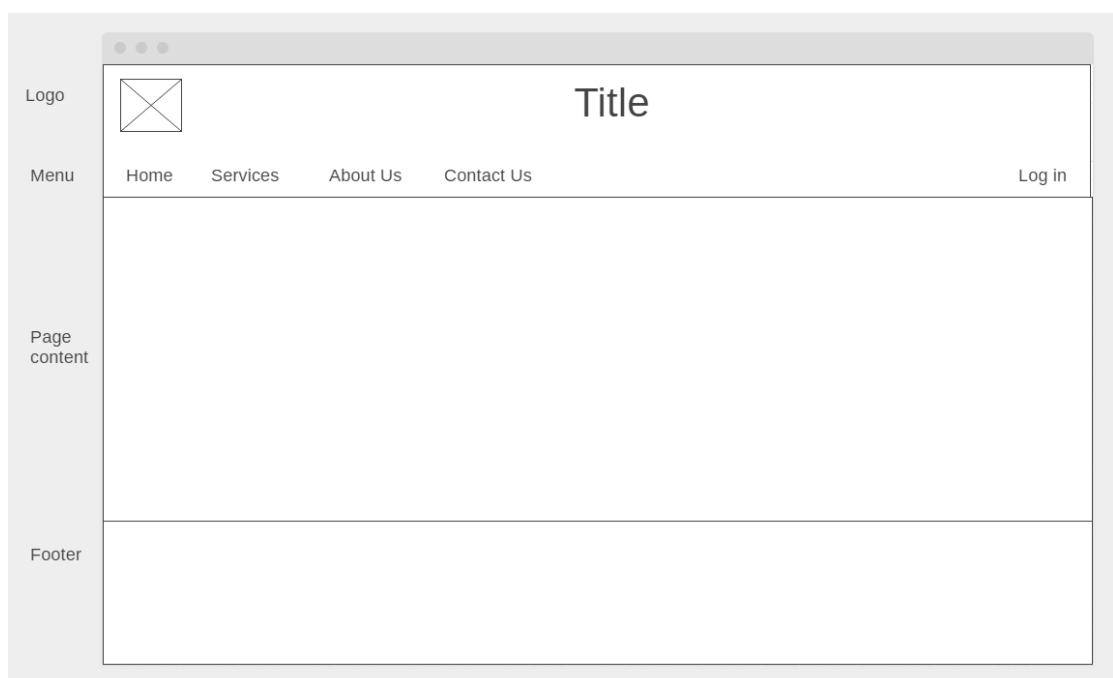
2.3.1-1 Az architektúráis felépítés

A backend egy ASP.NET Core-ban készült web API mely egy PostgreSQL adatbázissal kommunikál, ám úgy lett kialakítva, hogy az adatbázis típusa kevés konfigurációval cserélhető legyen. A webes felület melyet a felhasználók használnak React keretrendszer használatával készült és jelenleg a backend által kezelt SPA-ként működik, ám önálló alkalmazásként is telepíthető, ezen kívül a web API képes feldolgozni kéréseket más rendszerektől is, így a későbbiekben csatlakozhat a rendszerhez további alkalmazás is, például egy telefonos alkalmazás. Az interneten keresztüli kommunikáció HTTPS üzenetekkel történik, mivel mindkét felületen szükséges bejelentkezés, a jelszavak biztonságos továbbítását pedig kizárólag így lehet biztosítani, a HTTP üzenetek

lehallgatásakor még a jelszavak frontenden történő hashelése esetén is visszafejthetőek a bizalmas adatok.

3.4 Felülettervek

Az alkalmazás tervezését a követelményekből kiindulva a felülettervek elkészítésével kezdtem. A felülettervek készítése segíthet a megrendelővel lefektetett követelményekben fennmaradó esetleges félreértések felderítésében, illetve a webszerver és az adatbázis megalkotásánál is hasznos támpontot adnak. A tervekhez létrehozásához a WireFrame online tervező eszközt használtam.



2.3.1-1 A webes felület általános kinézete

Az alkalmazás alapvető felépítése a 2.3.1-1 A webes felület általános kinézete ábrán látható. A lap tetején mindig egy Header látszik, melyen megjelenik a logó a szolgáltatás neve és a navigációs menü. A lap alján egy Footer található, melyen helyet kaphatnak a kapcsolatfelvételi információk, felhasználási feltételek, oldaltérkép, stb. A kettő között helyezkednek el a különböző oldalak tartalmai. Látszik, hogy a tervezés ezen szakaszában még nem koncentrálnunk a konkrét működésre, tényleges felületi elemekre – például nem jelenik meg a navigációs menü különbsége a különböző felhasználó típusok között.

Az alábbi 2.3.1-2 Foglalások képernyő terve képen a foglalási oldalak tervezett kinézete látható. Az adatok megadásával a felhasználó lekérdezi a szerverről az árat és véglegesítheti a foglalást.

Title

[Home](#)
[Services](#)
[About us](#)
[Contact us](#)

[Account](#)
[Log out](#)

One way

By the hour

Right now

From Address:

To Address:

Date:

Car Type 1

Select

Car Type 2

Select

Car Type 3

Select

Select preferences:

-
-
-
-
-
-
-
-
-

Comment:

Discount code:

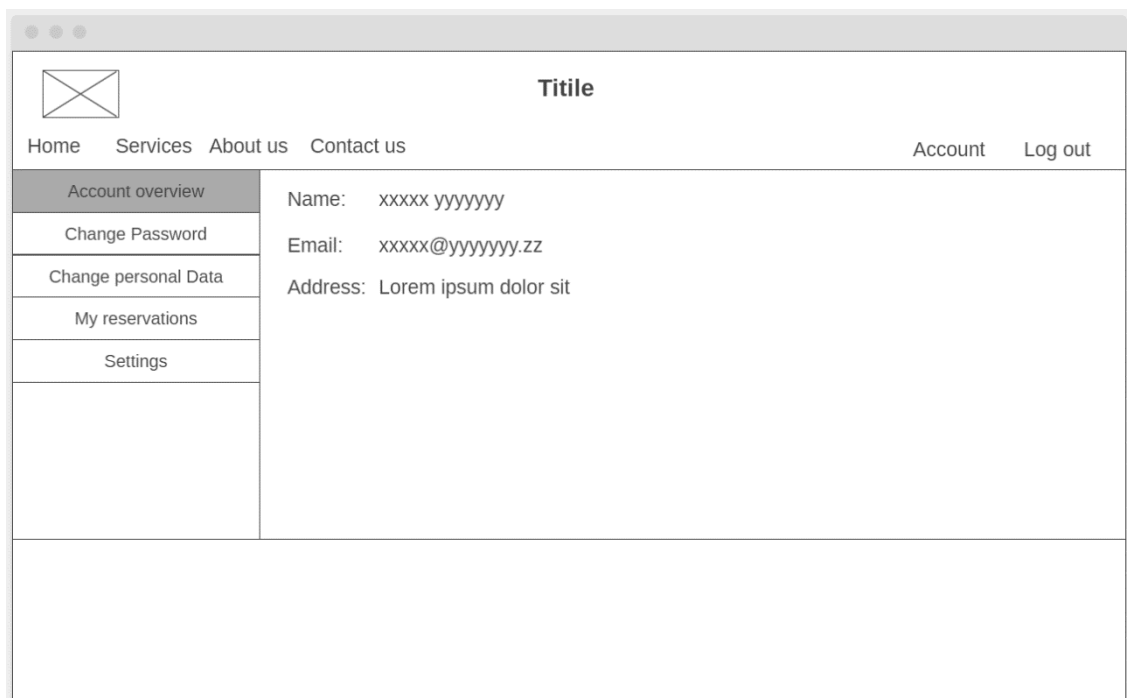
Calculate

XXX .-

Make reservation

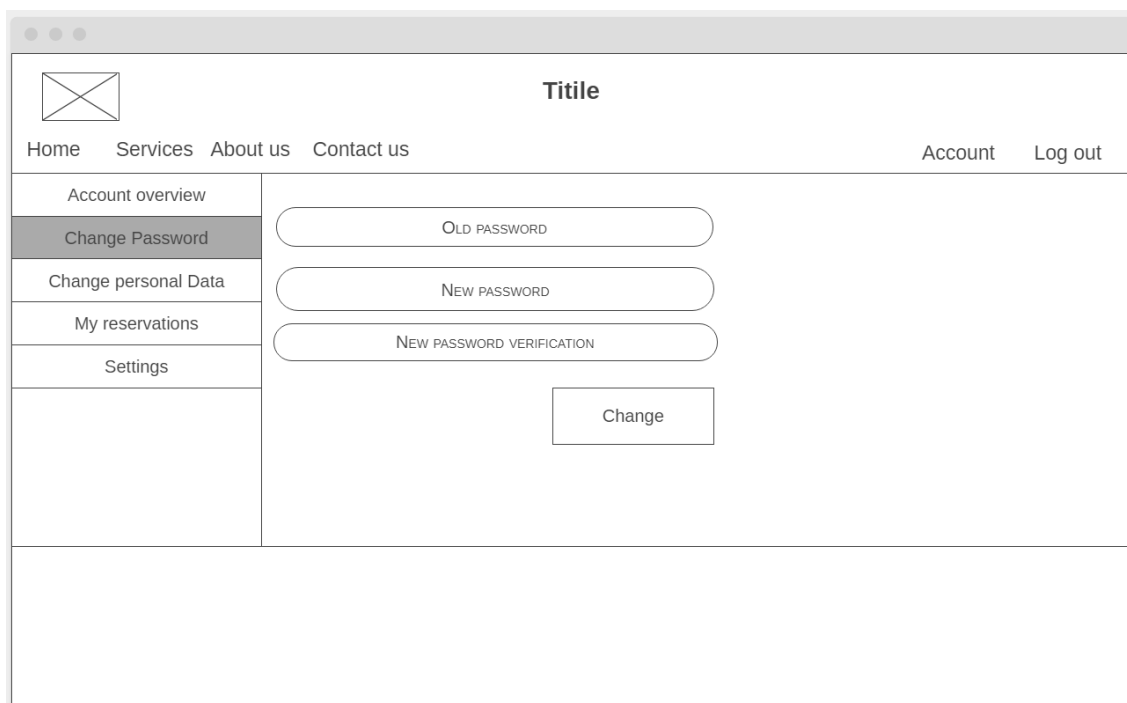
2.3.1-2 Foglalások képernyő terve

A 2.3.1-3 - 2.3.1-7 ábrákon a felhasználói oldalak láthatók. A felhasználónak lehetősége van a jelszava megváltoztatására, személyes adatainak kezelésére, saját foglalásainak megtekintésére, illetve beállításainak változtatására.



The screenshot shows a web browser window with a title bar. The page has a header with a logo (a square with an 'X') and the title 'Titile'. Below the header is a navigation bar with links: Home, Services, About us, Contact us, Account, and Log out. The main content area is divided into two columns. The left column contains a sidebar with links: Account overview (highlighted), Change Password, Change personal Data, My reservations, and Settings. The right column displays the user's profile information: Name: xxxxx yyyyyyy, Email: xxxxx@yyyyyy.zz, and Address: Lorem ipsum dolor sit.

2.3.1-3 Felhasználói adatok áttekintése




The screenshot shows a web browser window with a title bar. The page has a header with a logo (a square with an 'X') and the title 'Titile'. Below the header is a navigation bar with links: Home, Services, About us, Contact us, Account, and Log out. The main content area is divided into two columns. The left column contains a sidebar with links: Account overview, Change Password (highlighted), Change personal Data, My reservations, and Settings. The right column displays the password change form with three input fields: OLD PASSWORD, NEW PASSWORD, and NEW PASSWORD VERIFICATION. Below these fields is a 'Change' button.

2.3.1-4 A felhasználó jelszavának megváltoztatása

The screenshot shows a web browser window with a title bar. The page has a header with a logo (a square with an 'X') and the title 'Titile'. Navigation links include 'Home', 'Services', 'About us', 'Contact us', 'Account', and 'Log out'. A sidebar on the left contains a menu with 'Account overview', 'Change Password', 'Change personal Data' (highlighted), 'My reservations', and 'Settings'. The main content area displays a form for updating personal information. It includes labels for 'Name:', 'Email:', and 'Address:', each followed by a text input field containing placeholder text: 'XXXX YYYY', 'XXXX@YYYY.ZZ', and 'XXXX YYYY ZZZ LL' respectively. A 'Change' button is positioned below the address field. The bottom of the page features a large, empty rectangular box.

2.3.1-5 Személyes adatok változtatása

Az alábbi 2.3.1-6 ábrán látható a felhasználó foglalásainak megtekintése. A dokumentum későbbi fejezeteiben eltérést fedezhetünk fel az itt látható képernyőterv és a megvalósított program között – A kész megoldásban a foglalás lemondását kezdeményező gom átkerült a foglalás részleteit mutató oldalra. Az éles projektekben is gyakori a kezdeti felülettervektől való eltérés, hiszen az ilyen fajta tervek készítésének nem a végleges kinézet megalkotása a célja, fejlesztés közben számos változás következhet be.

Logo	 <div>Title</div>	
Menu	Home Services About Us Contact Us	Account Log out
Navigation	Account overview	<div>2020.10.02</div>
	Change password	<div> <div>From:</div> <div>To:</div> <div>Selected preferences:</div> <div>price .-</div> <div>Cancel</div> </div>
	Change personal data	
	My reservations	
Footer	Settings	<div>1998.10.02</div> <div> <div>From:</div> <div>To:</div> <div>Selected preferences:</div> <div>price .-</div> </div>

2.3.1-6 Foglалások megtekintése

<div>  <div>Title</div> </div>	
<div> Home Services About us Contact us </div> <div> Account Log out </div>	
Account overview	<div> <div>Notifications: <input checked="" type="checkbox"/></div> <div>Email news: <input checked="" type="checkbox"/></div> <div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div> <div>Delete Account</div> </div>
Change Password	
Change personal Data	
My reservations	
Settings	

2.3.1-7 Beállítások módosítása

A 2.3.1-8 Adminisztrátori belépéssel elérhető foglalások lap ábrán látható képernyőn tudnak a megfelelő jogosultságú felhasználók a kifizetett rendelésekhez dolgozókat hozzárendelni.

Title

[Home](#)
[Services](#)
[About us](#)
[Contact us](#)
[Account](#)
[Log out](#)

From Address:

To Address:

Date:

Selected preferences:

Reservation type:

Reservation status:

Page number: 20

<<

<

1/10

>

>>

Reserved by: Teszt Elek

Preferences: Non-Smoking, Pets allowed

Assign

Reservation date: 2021/02/05

Reservation type: One way

Car type: S-class

Reserved by: Teszt Elek

Preferences: Non-Smoking, Pets allowed

Assign

Reservation date: 2021/02/05

Reservation type: One way

Car type: S-class

Reserved by: Teszt Elek

Preferences: Non-Smoking, Pets allowed

Assign

Reservation date: 2021/02/05

2.3.1-8 Adminisztrátori belépéssel elérhető foglalások lap

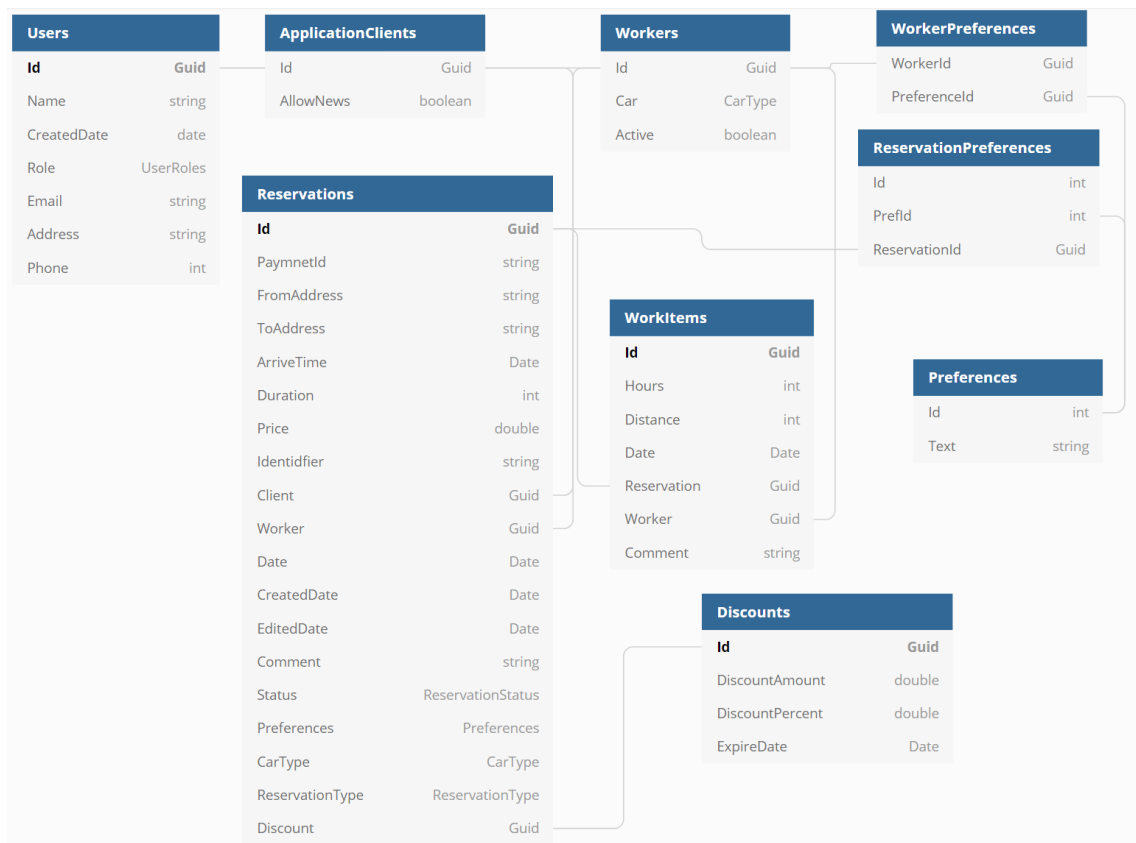
A 2.3.1-9 Dolgozói felület ábrán pedig a dolgozók által elérhető oldal látszik, melyen megtekinthetik és kezelhetik a hozzájuk rendelt foglalások állapotát.

2.3.1-9 Dolgozói felület

Az utolsó wireframen a contact formot láthatjuk, mellyel automatikus emailt küldhetnek a felhasználók. A statikus oldalakról nem készült felületterv, mivel főleg szöveges tartalmak lesznek backend integráció nélkül.

3.5 Az adatbázis felépítése

Az adattábla megtervezésekor az egyszerűséget és a bővíthetőséget tartottam szem előtt



2.3.1-1 Az alkalmazás adatbázis terve

A felhasználók adatainak biztonságos tárolására az ASP.NET Core Identity könyvtárát használtam, mely számos funkciót tartalmaz, mely transzparens a fejlesztő számára, például a jelszavak hashelt tárolása, vagy a felhasználók bejelentkeztetése, szerepkör-kezelése. Az alapvető adatokon kívül a felhasználók esetében tárolásra kerül egy „AllowNews” változó is, mely azt hivatott tárolni, hogy az adott ügyfél kíván-e promóciós anyagokat kapni. Dolgozók esetében tároljuk a használt autó típusát, illetve a dolgozó által szolgáltatott extra szolgáltatásokat (pl.: dohányzó/nem dohányzó, állatos stb.). Az extra szolgáltatások listáját – preferenciák – a könnyű karbantarthatóság, dinamikus kezelhetőség érdekében külön táblában tároljuk. A dolgozók lezárt munkáinak tárolására a „WorkItems” tábla szolgál. Egy foglalás lezárásakor a dolgozó bejegyezheti a munka adatait, például, hogy mennyi időt vett igénybe a fuvar, hány km-et ment az autó, ezen kívül tartozik ehhez is egy komment, arra az esetre, ha valami különleges esemény történne az út alatt (pl.: sajnálatos baleset miatt az autó kárpitja tisztításra szorul – a kliens hibájából). A megrendelésekhez tartozhat még egy leértékelő kód is. A leértékeléseket szintén külön táblában tároljuk az előzőekben kifejtett okok miatt, az adminisztrátorok megadhatják magát a promóciós kódot, a hozzá tartozó leértékelést vagy százalékos vagy

érték formájában, illetve a promóciós kód lejáratí dátumát, ami után a rendszer már nem fogadja el a kódot.

4 Megvalósítás

Ebben a fejezetben bemutatom a tervezett rendszer implementálását. Az egyes komponensek megvalósításának lépéseit a komponensek feladatait, a fejlesztésük során felmerült problémákat és megoldásokat. Kitérek még a felvetülő biztonsági kérdésekre is, azok megoldására.

4.1 Backend

Az alkalmazásban a backend szolgáltatja az adatokat a kliensek felé. A kliens HTTPS üzenetekkel kommunikál a backeddel, ami feldolgozza a kérést, összegyűjti a kért adatokat és azokat a kliens számára könnyen értelmezhető módon visszaküldi.

4.1.1 Felépítés

Az alkalmazás elkészítése során az volt a cél, hogy egy többretegű architektúrát hozzak létre, melyben az egyes rétegek kizárólag az alattuk lévő rétegtől függenek.

TaxiService.Web: Ez az alkalmazás futtatható rétege, egy .NET Core projekt, ami egy RESTful Web API formájában szolgálja ki a kliensek kéréseit, a kiszolgálás közben felmerülő problémákat kezeli és azonosítja/engedélyezi a felhasználókat. A jelenlegi konfigurációban ez a réteg felel a frontend szolgáltatásáért is egy SPA formájában.

TaxiService.Dto: A DTO rövidítés feloldása Data Transfer Object. Ez tulajdonképpen nem önálló réteg, egy .NET standard library, mely azokat az osztályokat tartalmazza melyeket a kliens felé menő üzenetekben használunk. Azért hasznos ilyen objektumokat létrehozni, mert nem mindig szeretnénk a teljes adatbázis rekordot visszaküldeni. Erre egy jó példa, hogy a felhasználók listázásánál nem feltétlenül szeretnénk elküldeni a kliensnek minden felhasználó jelszavát. Ezzel a megoldással szintén növelhető a teljesítmény, hiszen a teljes objektum helyett csak a kérés szempontjából lényeges adatokat küldhetjük el

TaxiService.Dal: Önmagában nem futtatható .NET Core project. Az adatelérési réteget (Data Access Layer- DAL) tartalmazza. Itt határozzuk meg a használni kívánt adatbázis struktúrákat, ezek modelljeit és változásuk esetén azok migrációit, illetve ebbe a rétegbe kerültek az alkalmazás során használt enumeráció típusú változók is.

TaxiService.Bll: A BLL rövidítés Buisness Logic Layert, azaz üzleti logikai réteget jelent. Önmagában szintén nem futtatható project, ide kerül az alkalmazás tényleges logikája, az API a kliensek kérései hatására interfészeken keresztül éri el, itt történik az adatelérés is.

4.1.2 Az adatbázis létrehozása

Az adatbázis létrehozása során az EF Core által támogatott Code First megközelítést alkalmaztam. Ennek lényege, hogy modell osztályokat hozunk létre, ezeket konfiguráljuk beállításokkal, megadjuk az egymáshoz viszonyított kapcsolataikat és ebből generáljuk az adatbázist. A konfiguráció során lehet megadni az adatbázisra vonatkozó megkötéseket is, így itt kellett felkészíteni a rendszert a több típusú adatbázis használatára. Az alábbi 4.1.2-1 képen látható például a PostgreSQL esetén történő konfiguráció.

```
if (configuration["DatabaseType"] == "POSTGRES")
{
    var connectionString = "";
    if (Environment.GetEnvironmentVariable("DATABASE_URL") == null)
    {
        connectionString = configuration.GetConnectionString("TaxiServiceContextPostgres");
    }
    else
    {
        connectionString = Environment.GetEnvironmentVariable("DATABASE_URL");
    }
    var stringBuilder = new PostgreSQLConnectionStringBuilder(connectionString)
    {
        Pooling = true,
        TrustServerCertificate = true,
        SslMode = SslMode.Require
    };
    var connectionUrl = stringBuilder.ConnectionString;
    builder.UseNpgsql(connectionUrl, builder =>
    {
        builder.EnableRetryOnFailure(5, TimeSpan.FromSeconds(10), null);
    });
}
```

4.1.2-1 PostgreSQL konfigurálása

A connection string beállítása kapcsán kiemelnék egy kódrészletet mely a kódban sok helyen előfordul majd, az Environment változó és config. Együttes használata abban az esetben hasznos, ha a kitelepített rendszer támogatja az Env. változók használatát akkor az abban szereplő bejegyzéseket a kitelepítést követően is könnyen lehet változtatni, azonban ha ez nem támogatott vagy például local környezetben futtatjuk a rendszert az alkalmazás *config.json* fájlja „backup” -ként működik.

Az entitások konfigurációja során sok kapcsolatot képes az EF Core magától felismerni, például egy-több kapcsolatokat, ám sok esetben kézzel kell konfigurálni, például az alábbi esetben a felhasználókat külön (leszármazott) modell entitásban szerettem volna tárolni ám az adatbázisban egy táblában kívántam tárolni őket, melyet az alábbi kódrészlettel oldottam meg:

```
modelBuilder.Entity<User>()  
    .HasDiscriminator(d => d.Role)  
    .HasValue<User>(Entities.Authentication.UserRoles.Administrator)  
    .HasValue<ApplicationClient>(Entities.Authentication.UserRoles.User)  
    .HasValue<Worker>(Entities.Authentication.UserRoles.Worker);
```

4.1.3 TaxiService.Web

A TaxiService.Web projekt nyújtja a backend futtatható részét. A kliensek felé szolgáltat egy Web Api-t, amin keresztül elérhetik az adatbázisból az adatokat. Az alkalmazást a *Startup.cs* fájlban kell konfigurálni, melyet a Visual Studio automatikusan generál a projekt létrehozásakor. Többek között itt lehet beállítani, hogy az alkalmazás autorizációt és autentikációt használjon, itt vehetjük fel az üzleti szolgáltatásainkat, illetve itt adjuk hozzá a Swagger szolgáltatásait, beállításait is. A konfigurációk közül kettő érdekesebbet emelnék ki ebben a részben.

Az egyik az SPA működésének konfigurálása, melyet részletesen kifejttek a 4.1.3.1 fejezetben. Két konfigurációs beállítás szükséges, az egyik a felület fájljai helyének megadása (felül), a másik pedig a web Api pipeline megfelelő konfigurálása (alul):

```

services.AddSpaStaticFiles(configuration =>
{
    configuration.RootPath = "ClientApp/build";
});

app.UseStaticFiles();
app.UseSpaStaticFiles();
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
app.UseSpa(spa =>
{
    spa.Options.SourcePath = "ClientApp";
    if (env.IsDevelopment())
    {
        spa.UseReactDevelopmentServer(npmScript: "startdebug");
    }
});

```

A másik konfiguráció, amit kiemelnék az a Barion kliens beállítása. A 2.1.4 fejezetben már említett Barion Client nuget csomag segítségével egyszerűen és egy helyen megadhatóak a különböző szükséges beállítások, ahelyett, hogy minden kéréshez külön kéne ezeket felvenni.

```

var barionSettings = new BarionSettings
{
    BaseUrl = new Uri(configuration["Barion:Url"]),
    POSKey = Guid.Parse(configuration["Barion:POS"]),
    Payee = configuration["Barion:PayeeEmail"],
};

services.AddSingleton(barionSettings);
services.AddTransient<BarionClient>();
services.AddHttpClient<BarionClient>();

```

4.1.3.1 Az SPA hozzáadása

Az előzőekben már bemutattam az SPA beállításához szükséges konfigurációs sorokat, ám ebben a fejezetben kifejtem, hogy mi szükséges még ahhoz, hogy egy React keretrendszer használatával készült forntendet a backendünk szolgáltatssa a kliensek felé. A Visual Studio alapértelmezetten a 'www' mappát generálja a frontend fájlok tárolására, az előző fejezetben bemutatott konfiguráció szükséges, hogy megváltoztassuk a használt mappát (a mi esetünkben ez a „ClientApp” mappa). A kiválasztott mappába dolgozhatunk a React alkalmazásunkkal, ebbe fog megjelenni a frontend összes fájlja. Ahhoz, hogy a frontend alkalmazásunk is megfelelően frissüljön, forduljon és fusson, be kell illesztenünk a projekt fordítási folyamatába néhány parancsot, melyeket a *.csproj* kiterjesztésű fájlba írhatunk.

```

<ItemGroup>
  <!-- Don't publish the SPA source files, but do show them in the project files list -->
  <Content Remove="$(SpaRoot)**" />
  <None Remove="$(SpaRoot)**" />
  <None Include="$(SpaRoot)**" Exclude="$(SpaRoot)node_modules\**" />
</ItemGroup>

```

4.1.3-1 forrásfájlok kivétele a telepítésből

Először is – ugyan nem kötelező, de sokat tömörít a projektünk méretén – ki kell vennünk a forrásfájlokat a telepítés útvonalából a 4.1.3-1 ábrán látható paranccsal. Következő lépésként meg kell bizonyosodnunk róla, hogy a célzott rendszer rendelkezik Node.js alkalmazással és a *node_modules* nevű a frontend dependenciákat tartalmazó mappa telepítésre került (4.1.3-2 ábra).

```

<Target Name="DebugEnsureNodeEnv"
  BeforeTargets="Build"
  Condition="'$(Configuration)' == 'Debug' And !Exists('$(SpaRoot)node_modules') ">
  <!-- Ensure Node.js is installed -->
  <Exec Command="node --version" ContinueOnError="true">
    <Output TaskParameter="ExitCode" PropertyName="ErrorCode" />
  </Exec>
  <Error Condition="'$(ErrorCode)' != '0'"
    Text="Node.js is required to build and run this project.
    To continue, please install Node.js from https://nodejs.org/,
    and then restart your command prompt or IDE." />
  <Message Importance="high"
    Text="Restoring dependencies using 'npm'.
    This may take several minutes..." />
  <Exec WorkingDirectory="$(SpaRoot)" Command="npm install" />
</Target>

```

4.1.3-2 Node.js ellenőrzése

Utolsó feladatunk, hogy a telepítés során a friss fájlokat melyeket a frontend fordítása során generáltunk, belevegyük a telepített csomagba, ezt a 4.1.3-3 ábrán látható módon tehetjük meg.

```

<Target Name="PublishRunWebpack" AfterTargets="ComputeFilesToPublish">
  <!-- As part of publishing, ensure the JS resources are freshly built in production mode -->
  <Exec WorkingDirectory="$(SpaRoot)" Command="npm install" />
  <Exec WorkingDirectory="$(SpaRoot)" Command="npm run build" />

  <!-- Include the newly-built files in the publish output -->
  <ItemGroup>
    <DistFiles Include="$(SpaRoot)build\**; $(SpaRoot)build-ssr\**" />
    <ResolvedFileToPublish Include="@(\DistFiles->'%(FullPath)')" Exclude="@(\ResolvedFileToPublish)">
      <RelativePath>%(DistFiles.Identity)</RelativePath>
      <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
      <ExcludeFromSingleFile>true</ExcludeFromSingleFile>
    </ResolvedFileToPublish>
  </ItemGroup>
</Target>

```

4.1.3-3 A generált fájlok csomaghoz tétele

A fent említett lépések mellett még egy parancsot be kellett építenem a folyamatba, mivel a frontend különböző módokon – SPA-ként és React keretrendszerben Visual Studio Code használatával – fordítottam, a `'tsconfig.json'` fájl automatikus generálása hibára futott, így minden build előtt törölni kellett azt, az alábbi ábrán látható paranccsal.

```
<Target Name="DeleteTsConfigBeforeBuild" BeforeTargets="Build">
  <ItemGroup>
    <FileToDelete Include="$(SpaRoot)tsconfig.json" />
  </ItemGroup>
  <Exec ContinueOnError="true" Command="del /F /Q &quot;@(FileToDelete)&quot;" />
</Target>
```

4.1.3-4 A tsconfig fájl törlése build előtt

A *BeforeTargets* property megadja, hogy a fordítási folyamat mely szakasza előtt futtassuk az adott szakaszt, az *Exec* tag *Command* property-je pedig lefuttat egy szabadon definiálható parancsot, ebben az esetben törli az *SpaRoot* változó által meghatározott mappában lévő `'tsconfig.json'` fájlt.

4.1.3.2 Controllerek

A kliensektől érkező kérések az előző 4.1.3.1 és 4.1.3 fejezetekben látható konfigurációk alapján egy kiszolgálási sorba, úgynevezett pipeline-ba kerülnek. A különböző feldolgozó egységek a *Startup.cs* kódja alapján meghatározott sorrendben futnak le. A beérkező kérések az URL-ben található útvonal alapján kerülnek kiszolgálásra, a `„/api”` kezdetű-ek a backend Controllereihez kerülnek, míg a többi kérést a frontend kísérli meg kiszolgálni. Azok a csomagok, melyek a backendhez érkeznek az URL további részei alapján kerülnek a megfelelő Controllerekhez. A Controller osztályokat az `[ApiController]` és a `[Route("api/admin")]` annotációkkal hozhatjuk létre. A `Route()` konstruktorában lévő string-ben adhatjuk meg, hogy milyen útvonalon akarjuk elérhetővé tenni az adott osztályt – ebben a példában a kontroller az `„admin”` útvonalon lesz elérhető. A Controller osztályokban függvényeket definiálhatunk melyeket a `[HttpPost]`, `[Route("assign")]` annotációkkal jelölve hozhatunk létre. Az előbbi a HTTP kérés típusára vonatkozik az utóbbival pedig a Controller utáni útvonalat adhatjuk meg hasonlóan a Controller osztály létrehozásához. Az előzőekben leírt példákban tehát a `http(s)://www.baseurl.valami/api/admin/assign` POST kérést kiszolgáló függvényt határoztuk meg. A végpontokhoz ezeken kívül bemenő paramétereket is megadhatunk. A paraméterek több helyről is érkezhettek, az alapértelmezett az elérési útvonal további

részből, például `Route('assign/{id}')` meghatározásával egy `id` nevű paramétert várunk az útvonalból. Ezen kívül még sokszor használjuk a `[FromBody]` annotációt ekkor azt mondjuk meg a rendszer számára, hogy a paramétert a kérés Body részéből olvassuk ki, ebben az esetben valamilyen sorosított formátumot használunk például JSON-t. A Controller osztályokat úgy érdemes megválasztani, hogy a karbantarthatóság, átláthatóság és az URL struktúra logikus felépítése érdekében a logikailag/ a működés szempontjából összetartozó funkcionalitásokat blokkokra bontjuk. Az én alkalmazásomban hat Controller osztályt hoztam létre.

AdminController:

Ide kerültek a kizárólag adminisztrátorok által használható funkciók, az elérési útvonala `„/admin”`. Két függvényt határoztam meg: Az `„assign”` útvonalon elérhető funkció meghívásával rendelhet az adminisztrátor a beérkezett rendelésekhez dolgozót, a `„workers”` végpont pedig kilistázza a dolgozókat.

PaymentController:

A `„/payment”` URL alatt találhatóak a fizetéssel kapcsolatos funkciók. Jelenleg két függvényt tartalmaz, a `„{reservationId}/barion”` és a `„barionCallback”` a Barion működéséről az üzleti logikai réteg kifejtésénél fogok beszélni. Ebbe a Controllerbe kerülhetnek a későbbiekben beépíthető további fizetési lehetőségek végpontjai.

PreferenceController:

Ebben a Controllerben a szolgáltatásokhoz kapcsolódó extra igényekkel, preferenciákkal kapcsolatos funkciók vannak. Mivel ezek betöltése jelenleg konfigurációs fájlból történik nem adminisztrációs felületről, ezért az osztály egyetlen függvénye az alap `„/preferences”` útvonalon elérhető GET hívás mely kilistázza a választható elemeket.

ReservationController:

A foglalási funkciók nagy része ebbe a Controllerbe érkeznek, mely a `„/reservation”` útvonalon található. Az összes foglalás közötti keresés ugyan csak az adminisztrátorok érhetik el, ám logikailag a foglalásokhoz tartozik ezért ide került az *AdminController* helyett, az alap útvonalon érhető el POST kéréssel. A POST kérések esetén általában a `[FromBody]` attribútummal a HTTP kérés body részébe sorosított objektumban továbbítjuk a használt adatokat. A különböző funkciók korlátozásáról és

védelméről a következő 4.1.3.3 fejezetben fogok írni. Egy adott foglalás részleteit az „{id}” útvonalon lehet elérni. A „price” URL-ről lehet lekérdezni egy foglalás árát. A hibás kérések korai kiszűrése érdekében már a Controller rétegben validáljuk a kéréseket.

```
[HttpPost]
[Route("price")]
0 references | Krisz, 202 days ago | 2 authors, 2 changes
public async Task<double> GetReservationPrice([FromBody] ReservationPriceDto reservation)
{
    if (String.IsNullOrEmpty(reservation.FromAddress))
    {
        throw new BusinessException("From address cannot be empty.");
    }
    if (reservation.Duration == null && String.IsNullOrEmpty(reservation.ToAddress))
    {
        throw new BusinessException("Destination address cannot be empty.");
    }
    if (reservation.ReservationType == Dal.Enums.ReservationType.ByTheHour
        && (reservation.Duration == null || (reservation.Duration < 0 || reservation.Duration > 12)))
    {
        throw new BusinessException("Reservation duration out of bounds.");
    }

    return await reservationService.GetPrice(reservation);
}
```

4.1.3-5 Price funkció validálása

A fenti 4.1.3-5 ábrán például leellenőrizzük a kitöltött mezők helyességét mielőtt meghívnanánk az üzleti logikát. A „make” URL meghívásával véglegesíthetünk egy foglalást, az „{id}” URL GET helyett DELETE hívásával mondhatunk le egy rendelést.

UserController:

Ebben az osztályban kaptak helyet a felhasználói profilokkal kapcsolatos funkciók végpontjai. A végpontok „/user” útvonal alatt érhetőek el. A felhasználó regisztrálás a „register” URL-en érhető el. A felhasználókezelést az ASP.Net Core Identity nuget könyvtár segítségével végzem, erről a 4.1.3.3 fejezetben részletesen is beszélek, itt csak annyit említenék meg, hogy a validálás során a lábbi 4.1.3-6 ábrán látható módon állítható be, hogy milyen típusú jelszavakat és felhasználóneveket állíthatnak be a regisztrálók.

```

services.Configure<IdentityOptions>(options =>
{
    // Password settings.
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 0;

    // User settings.
    options.User.AllowedUserNameCharacters =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = true;
});

```

4.1.3-6 Felhasználó regisztráció konfigurálása

Regisztráció után a felhasználók a „login” URL alatt található végponttal léphetnek be. A „reservations” végpont használatával listázhatóak az adott felhasználó foglalásai, a „changePassword” hívással megváltoztatható a felhasználó jelszava, hasonló módon a „changeData” végponton változtathatóak a további adatok. Az „emailNotifications” hívásával a hírlevelekről való le/feliratkozást lehet változtatni. A GDPR szempontjából rendkívül fontos, hogy a felhasználói fiókok törölhetőek legyenek a hozzájuk kapcsolódó adatokkal együtt. Ezt a Controller DELETE hívásával lehet megtenni.

WorkerController:

A „/worker” útvonalon érhetőek el a dolgozókkal kapcsolatos végpontok. „jobs/{resId}/update” URL hívásával állítható a dolgozóhoz rendelt munkák állapota, a „jobs” pedig kilistázza ezeket a munkákat.

4.1.3.3 Biztonság

Az alkalmazás és a felhasználók bizalmas adatainak védelme érdekében fontos végig gondolni a biztonság kérdéseit. Elsősorban ahogy a 3.3 fejezetben már említettem a kliens és az API közötti kommunikációra HTTPS üzeneteket használunk. A felhasználói adatok tárolására az Identity Core csomagot használom (lásd 2.1.2 fejezet), melyet JWT tokennel kombinálva azonosítom a felhasználókat. A token használatához két konfigurációt kell elvégeznünk.

```

JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear(); // => remove default claims
services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(cfg =>
{
    cfg.RequireHttpsMetadata = false;
    cfg.SaveToken = true;
    cfg.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(configuration["Jwt:Key"])),
        ValidateIssuer = true,
        ValidIssuer = configuration["Jwt:Issuer"],
        ValidateAudience = false,
        ClockSkew = TimeSpan.Zero // remove delay of token when expire
    };
});

```

4.1.3-7 JWT Autentikáció konfigurálása

A fenti 4.1.3-7 ábrán látható kódsorokkal állítható be az alkalmazáson a JWT tokenek használata. A *TokenValidationParameters* property-ben állítható be, hogy milyen paramétereket figyeljen a tokenben a program. A fenti kódban a kiállító kulcsát és magát a kiállítót adjuk meg validálandó elemként. A második konfigurációs lépés a token generálása, amit a sikeres bejelentkezés esetén adunk át a kliensnek, hogy továbbiakban ezzel azonosítsa magát. A generálás kódja az alábbi 4.1.3-8 ábrán látható

```

public static string GenerateJwtToken(User user, IConfiguration configuration)
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
        new Claim(ClaimTypes.Email, user.Email),
        new Claim(ClaimTypes.Role, user.Role),
    };

    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration["Jwt:Key"]));
    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512);
    var expires = DateTime.Now.AddDays(Convert.ToDouble(configuration["Jwt:ExpireDays"]));

    var token = new JwtSecurityToken(
        issuer: configuration["Jwt:Issuer"],
        audience: configuration["Jwt:Issuer"],
        claims: claims,
        expires: expires,
        signingCredentials: creds
    );

    return new JwtSecurityTokenHandler().WriteToken(token);
}

```

4.1.3-8 JWT token generálása

A tokenbe a készített kulcson kívül elhelyezzük a kiállítót a lejárat dátumot és az úgynevezett „claimeket”, ezek extra információk melyeket átadhatunk a kliensnek, hogy könnyebbé tegyük a későbbi azonosítást. Az alkalmazásomban a név azonosításon kívül, melyet a felhasználó ID-jára állítottam, berakom még az email címet és a felhasználó szerepkörét is a tokenbe. A Role alapú azonosítás előnye, hogy ahelyett, hogy a kódban kéne manuálisan ellenőrizni a szerepköröket, meg lehet határozni Controllerekre és azon belül külön-külön a végpontokra szerepköröket melyekkel elérhetőek. Azt, hogy az adott végpont csak bejelentkezve érhető el az [Authorize] attribútummal jelezhetjük, ha szerepkört is szeretnénk hozzá rendelni akkor a [Authorize (Roles = UserRoles.Administrator)] konstruktort használhatjuk. A fenti példa csak adminisztrátor felhasználókat engedélyez. Amennyiben egy Controller legtöbb függvényére igaz egy autorizációs szabály, de szeretnénk egy adott metódusnál kivételt tenni azt az [AllowAnonymous] attribútummal tehetjük meg. Kitérőként megemlíteném még, hogy a JWT token használatát külön be kell állítani Swagger használatakor az alábbi 4.1.3-9 ábrán látható módon.

```

services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "TaxiService API", Version = "v1" });
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        Description = @"JWT Authorization header using the Bearer scheme. \r\n\r\n
            Enter 'Bearer' [space] and then your token in the text input below.
            \r\n\r\nExample: 'Bearer 12345abcdef'",
        Name = "Authorization",
        In = ParameterLocation.Header,
        Type = SecuritySchemeType.ApiKey,
        Scheme = "Bearer"
    });

    c.AddSecurityRequirement(new OpenApiSecurityRequirement()
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = "Bearer"
                },
                Scheme = "oauth2",
                Name = "Bearer",
                In = ParameterLocation.Header,
            },
            new List<string>()
        }
    });
});

```

4.1.3-9 JWT konfigurálása Swagger használatakor

A véletlen/ rossz konfigurálásból adódó adatszivárgások elkerülése érdekében – és a biztonsági kérdéseken kívül a kód könnyebb kezelése érdekében – a hibákat nem engedjük ki ellenőrizetlenül a válaszokban. Ennek megvalósítására egy hibakezelő middleware-t használok, melynek lényegi kódjá alább látható.

```

public async Task Invoke(HttpContext context)
{
    try
    {
        await this._next(context);
    }
    catch (ArgumentException e)
    {
        await this.WriteAsJsonAsync(context, (int)HttpStatusCode.BadRequest, new
        ErrorDto
        {
            Message = e.ParamName ?? e.Message,
            StackTrace = this._hostingEnvironment.IsDevelopment()
                ? e.StackTrace : null,
        });
    }
    catch (BuisnessLogicException e)
    {
        await this.WriteAsJsonAsync(context, (int)HttpStatusCode.BadRequest, new
        ErrorDto
        {
            Message = e.Message
        });
    }
    catch (Exception e)
    {
        await
        this.WriteAsJsonAsync(context, (int)HttpStatusCode.InternalServerError, new
        ErrorDto
        {
            Message = _hostingEnvironment.IsDevelopment()
                ? e.Message : "Internal error",
            StackTrace = this._hostingEnvironment.IsDevelopment()
                ? e.StackTrace : null,
        });
    }
}

```

A kódrészlet elkapja a beérkező HTTP üzeneteket, és ha a kezelésük során hibát észlel, azokat hiba típus alapján különböző módon kezeli, ezen kívül a *this._hostingEnvoronment.IsDevelopment()* hívás segítségével megállapítja, hogy éles vagy teszt környezetben /konfigurációban fut az alkalmazás és ha nem teszt alatt fut nem engedi ki az esetlegesen érzékeny információkat tartalmazó hibarészeket.

4.1.4 TaxiService.Dal

Ahogy az elnevezésből is következik, a Data Access Layer felel az adatbázis eléréssel kapcsolatos feladatokért. Ilyen az adatbázisban megjelenő adatosztályok, az adatbázist leíró context és az adatbázis nyomon követéséért felelős migrációk.

4.1.4.1 Context

Az adatbázis leírását a *TaxiServiceContext.cs* fájl tartalmazza. Az EF Core segítségével a legtöbb kapcsolat típus elég az entitások szintjén meghatározni, azonban a bonyolultabb kapcsolatokat, különleges konfigurációkat, úgynevezett „seed„ adatokat ebben a fájlban lehet beállítani.

```
modelBuilder.Entity<User>()
    .HasDiscriminator(d => d.Role)
    .HasValue<User>(Entities.Authentication.UserRoles.Administrator)
    .HasValue<ApplicationClient>(Entities.Authentication.UserRoles.User)
    .HasValue<Worker>(Entities.Authentication.UserRoles.Worker);

modelBuilder.Entity<Discount>()
    .HasIndex(d => d.DiscountCode)
    .IsUnique();
```

4.1.4-1 Adatbázis konfiguráció

A fent látható (4.1.4-1) ábrán az adatbázis konfiguráció egy részlete található. A fenti kódrészlettel a három különböző szerepkörű felhasználó osztályokat egy táblába „sűrítjük” úgynevezett Discriminator segítségével, ez azt jelenti, hogy beállítunk egy közös őssztállyal rendelkező osztályok halmazára egy olyan property-t, amivel az adatbázisból való beolvasáskor meg tudjuk különböztetni a különböző osztályokat – ez a esetünkben a szerepkör. A lenti kódrészlet definiál egy indexet a *DiscountCode* property-re és kiköti, hogy két ugyanolyan nem lehet az adatbázisban.

A *Migrations* mappa tartalmazza az adatbázis-változások során keletkezett migrációkat, ezek Code First megközelítéskor automatikusan generálhatóak az adatbázis aktuális állapota és a kód állapota közötti különbségek alapján. Ennek célja, hogy az entitások változása esetén frissíteni lehessen egy már élesben működő adatbázist anélkül, hogy az adatokat elveszítenénk.

4.1.4.2 Entities

Az adatbázisban használt osztályok kerültek ebbe a mappába. Az entitások felépítése látható a 3.5 Az adatbázis felépítése fejezetben, így itt csak a lényegesebb pontokat emelném ki. Egyes kapcsolatot a másik tábla kulcsára mutató kulcs felvételével, illetve egy navigation property felvételével lehet definiálni, erre az alább látható kapcsolótáblán két példát is látni.

```

3 references | Szalkai Krisztián Farkas, 262 days ago | 1 author, 1 change
public class ReservationPreference
{
    0 references | Szalkai Krisztián Farkas, 262 days ago | 1 author, 1 change
    public Guid Id { get; set; }
    8 references | Szalkai Krisztián Farkas, 262 days ago | 1 author, 1 change
    public Preference Preference { get; set; }
    2 references | Szalkai Krisztián Farkas, 262 days ago | 1 author, 1 change
    public int PrefId { get; set; }
    0 references | Szalkai Krisztián Farkas, 262 days ago | 1 author, 1 change
    public Reservation Reservation { get; set; }
    0 references | Szalkai Krisztián Farkas, 262 days ago | 1 author, 1 change
    public Guid ReservationId { get; set; }
}

```

4.1.4-2 Foglalások és preferenciák közötti kapcsolótábla

Egy-több kapcsolat a másik oldalról egy lista meghatározásával lehetséges, több-több kapcsolat pedig egy a 4.1.4-2 ábrán látható kapcsolótábla létrehozásával lehetséges.

Ebbe a rétegbe kerültek még az alkalmazás során használt enum típusú változók, ezek szigorúan nem tartoznak az adatbázishoz ezért kerülhettek volna például az üzleti logikai rétegbe is, ám mivel az entitásokban is használjuk az osztályokat itt is logikus lehet az elhelyezésük.

4.1.5 TaxiService.Dto

A DTO-k szerepéről és hasznosságáról már beszéltem a 4.1.1 fejezetben, így itt csak kiemelnék néhány érdekesebb részletet.

```

12 references | Szalkai Krisztián Farkas, 262 days ago | 1 author,
public class PagedData<T>
{
    3 references | Szalkai Krisztián Farkas, 262 days ago | 1 aut
    public List<T> Data { get; set; }
    3 references | Szalkai Krisztián Farkas, 262 days ago | 1 aut
    public int ResultCount { get; set; }
}

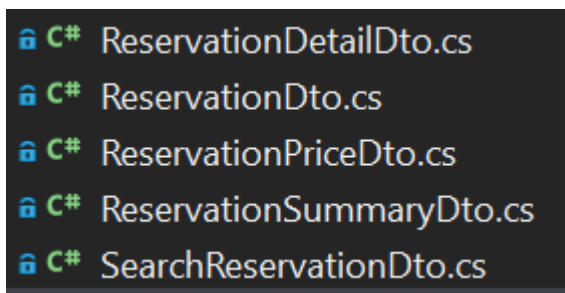
0 references | Krisz, 189 days ago | 1 author, 1 change
public class PagerDto
{
    0 references | Krisz, 189 days ago | 1 author, 1 change
    public int PageNumber { get; set; }
    0 references | Krisz, 189 days ago | 1 author, 1 change
    public int PageSize { get; set; }
}

```

4.1.5-1 Pager DTO-k

A fenti képen láthatóak a lapozáshoz használt osztályok. A lapozás azért előnyös, mert nem kell egyben megkeresni és átküldeni több száz, vagy akár több ezer rekordot egy kereséskor, hanem csak egy előre meghatározott mennyiséget töltünk be egyszerre, erről az üzleti logikai résznél részletesen is írni fogok.

Az alábbi képen jól látszik, hogy ugyanahhoz az adatbázis entitáshoz rengeteg féle DTO-t készíthetünk felhasználási esettől függően.



4.1.5-2 Foglalásokhoz kapcsolódó DTO osztályok

A *ReservationDto* a listázáskor megjelenő adatok megjelenítésekor kerül felhasználásra, az alapvető információkat tartalmazza.

```
public class ReservationDto
{
    public string FromAddress { get; set; }
    public string ToAddress { get; set; }
    public int? Duration { get; set; }
    public ReservationType ReservationType { get; set; }
    public string Date { get; set; }
    public CarType CarType { get; set; }
    public List<int> PreferenceIds { get; set; }
    public string Comment { get; set; }
    public string DiscountCode { get; set; }
}
```

Ezzel szemben a *SearchReservationDto* azokat a property-ket, amelyek a keresés szűkítésére szolgálhatnak.

```
public class SearchReservationDto
{
    public int PageNumber { get; set; }
    public int PageSize { get; set; }
    public DateTime? FromDate { get; set; }
    public DateTime? ToDate { get; set; }
    public double? MinPrice { get; set; }
    public double? MaxPrice { get; set; }
    public List<int> PrefIds { get; set; }
    public ReservationType? ReservationType { get; set; }
    public CarType? CarType { get; set; }
    public ReservationStatus? Status { get; set; }
}
```

Hasonlóan, a *ReservationDetailsDto* a foglalás részletes nézetére vonatkozó adatokat, a *ReservationSummaryDto* a rendelés leadása előtt megjelenő összegző nézet adatait, a *ReservationPriceDto* pedig kizárólag az árszámításhoz kapcsolódó adatokat tartalmazza.

4.1.6 TaxiService.Bll

A Bll réteg tartalmazza az alkalmazás üzleti logikáját – magját. Az *Exceptions* mappa tartalmazza a saját kivételeket, jelenleg egy ilyen használok a *BuisnessLogicException*-t, ezt a kivételt használtam, ha olyan „hibára” futna az alkalmazás szempontjából nem hibás működés, csak az adatok szempontjából. Ilyenek például a validációs hibák, de nem ilyen, ha egy nem bejelentkezett felhasználó próbál elérni egy csak bejelentkezéssel elérhető végpontot. A *Templates* mappában találhatók az email szolgáltatás által használt HTML sablonok, a *Services* és a *ServiceInterfaces* mappák pedig a szolgáltatásokat tartalmazzák, melyeket a controllerek meghívnak.

4.1.6.1 Service interfacek

A konkrét szolgáltatások és azok a rétegek melyek ezeket használják interfacekkel vannak elválasztva. Ennek az az előnye, hogy a tényleges logikai kód könnyen módosítható, sőt akár lecserélhető anélkül, hogy az azokat használó kódot módosítani kellene.

```
services.AddScoped<IPreferenceService, PreferenceService>();  
services.AddScoped<IReservationService, ReservationService>();  
services.AddScoped<IUserService, UserService>();  
services.AddScoped<IAdminService, AdminService>();  
services.AddScoped<IPaymentService, PaymentService>();  
services.AddScoped<IUserService, UserService>();  
services.AddScoped<IWorkerService, WorkerService>();  
services.AddScoped<IEmailService, EmailService>();
```

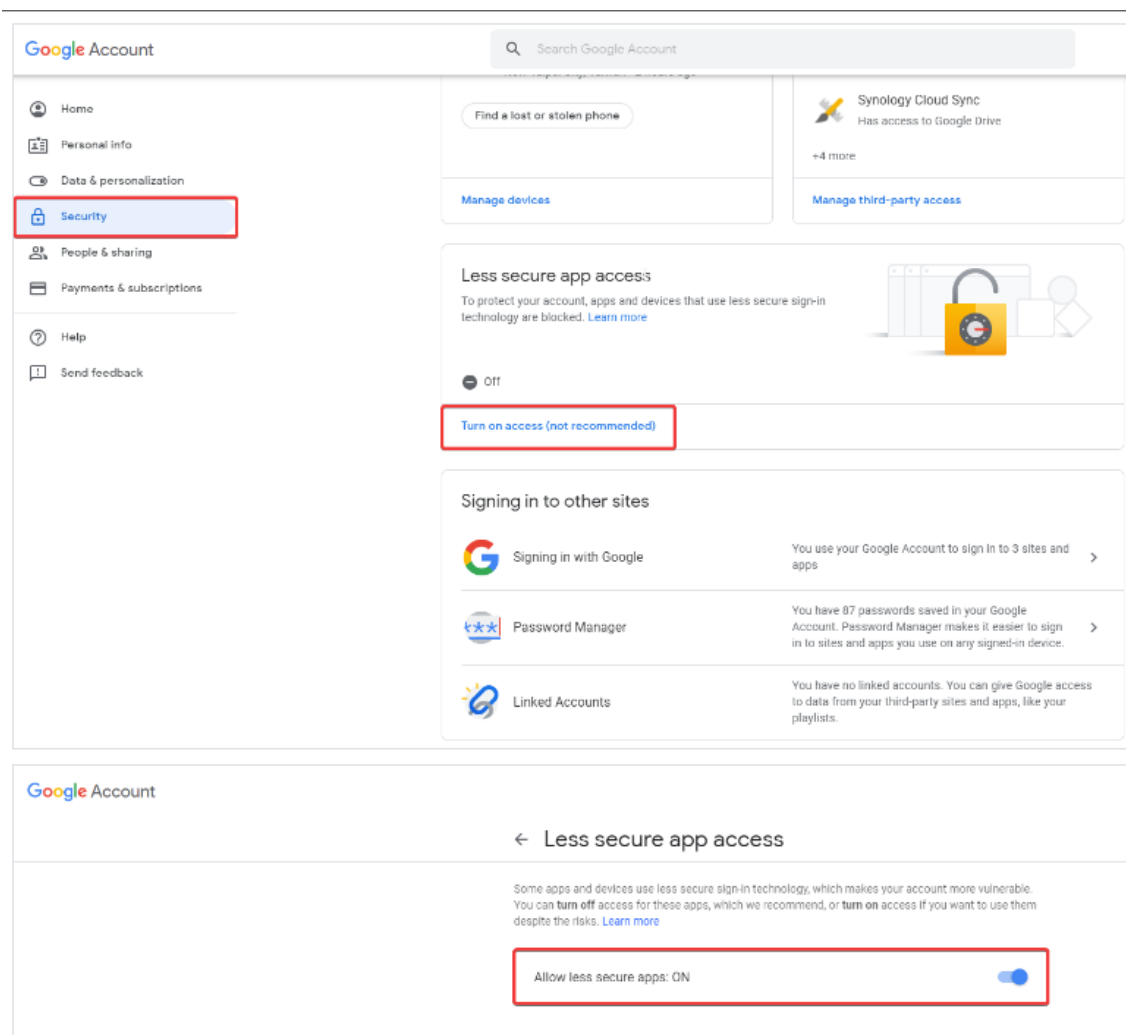
4.1.6-1 A szolgáltatások regisztrálása

A fenti 4.1.6-1 ábrán látható a Web rétegben a szolgáltatások regisztrálása. Az interface bal oldalon az előbb említett összekötő kapocs, a tényleges megvalósítás – jobb oldalon – pedig kicserélhető bármire, ami megvalósítja az adott interfacet.

4.1.6.2 Services

A szolgáltatásokat a controllerekhez hasonló módon a funkciók alapján osztottam fel, itt csak az érdekesebb kódrészleteket fogom kiemelni. Az adatbázis műveletek során a Linq könyvtárat használtam, ami segítségével típusosan és könnyebben kérdezhetőek le adatok C# nyelven.


EmailService: Az alkalmazás által automatikusan küldött emailek kezelésére a google smtp szolgáltatását használtam HTNML email sablonnal. Ennek használatához konfigurálni kell egy gmail fiókot és magát az alkalmazást. Az email fiók konfigurálásának legegyszerűbb módja egy úgynevezett alkalmazásjelszó létrehozása.



4.1.6-2 Hozzáférés megadása külső alkalmazások számára

Az első lépés, hogy a gmail fiókhoz hozzáférést kell biztosítani külső alkalmazásoknak is a fenti 4.1.6-2 ábrán látható módon. Miután ezt megtettük, be kell állítani két faktoros hitelesítést a fiókra, hogy biztosan ne történhessen visszaélés esetleges jelszószivárgás esetén, ha ezt megtettük elérhetővé válik az alkalmazásjelszavak generálása. Az ilyen jelszavak használatának előnye, hogy kizárólag az email küldési funkciókhoz fér hozzá, így az alkalmazásunk feltörése vagy a jelszó kiszivárgása esetén elegendő törölni a megfelelő bejegyzést és az alkalmazás nem tud többé visszaélni a fiókunkkal. Alább a létrehozott bejegyzések felülete látható.

Saját alkalmazásjelszavak

Név	Létrehozva	Utolsó használat ideje	
Diploma	okt. 21.	okt. 21.	

Válassza ki azt az alkalmazást és eszközt, amelyhez alkalmazásjelszót szeretne generálni.

Válassza ki az alkalmazást ▼ Válassza ki az eszközt ▼

LÉTREHOZÁS

4.1.6-3 Alkalmazásjelszavak konfigurációs felülete

Az alkalmazásban be kell állítanunk a fent generált felhasználó-jelszó párost, illetve a google által használt smtp klienst.

```
var smtp = new SmtpClient
{
    Host = "smtp.gmail.com",
    Port = 587,
    EnableSsl = true,
    DeliveryMethod = SmtpDeliveryMethod.Network,
    UseDefaultCredentials = false,
    Credentials = new NetworkCredential(
        Environment.GetEnvironmentVariable("BOOKING_EMAIL")
        ?? configuration["EmailService:User"],
        Environment.GetEnvironmentVariable("BOOKING_PASS")
        ?? configuration["EmailService:Password"]),
    Timeout = 20000,
};
```

4.1.6-4 Google Smtplib kliens konfigurálása

A HTML sablon használatához küldés előtt engedélyezni kell az *IsHtmlBody* kapcsolót, mivel alapértelmezetten biztonsági szempontokból a html kódot stringként értelmezi a kliens. A HTML sablonok használata alapos tesztelést igényelt, mivel rengeteg kliens máig nem támogat számos modern css megoldást, illetve ahány kliens annyi különböző működést tapasztaltam.

PaymentService: Fizetés jelenleg a Barion használatával lehetséges – ám kiegészíthető tetszőleges más klienssel is. A Barionhoz hasonló automatikus online fizetési rendszerek általában a következő módon működnek:

1. Az alkalmazás kliensén a felhasználó fizetést kezdeményez
2. Az alkalmazás backend összerak egy kérést a külső fizetési rendszer felé, majd elküldi azt. A Barion Client segítségével például az alább látható módon:

```
var transaction = new PaymentTransaction
{
    Items = new[] { item },
    POSTransactionId = reservation.Identifier,
    Payee = barionSettings.Payee,
    Total = Convert.ToDecimal(Math.Round(reservation.Price,0)),
};

var startPayment = new StartPaymentOperation
{
    Transactions = new[] { transaction },
    PaymentType = PaymentType.Immediate,
    Currency = Currency.HUF,
    FundingSources = new[] { FundingSourceType.All },
    GuestCheckOut = true,
    Locale = CultureInfo.CurrentCulture,
    OrderNumber = reservation.Identifier,
    PaymentRequestId = reservation.Id.ToString(),
    CallbackUrl = configuration["Barion:CallbackUrl"],
    RedirectUrl = configuration["Barion:RedirectUrl"],
};

var result = await barionClient.ExecuteAsync(startPayment);
```

4.1.6-5 Barion fizetés kezdeményezése

3. A küldő rendszer összerakja a saját fizetési felületét, majd annak URL-jét visszaküldi válaszként.
4. Az alkalmazás kliense a saját backendjétől visszakapja a külső fizetéshez tartozó URL-t és átirányítja a felhasználót.
5. A felhasználó fizet/ nem fizet/ lemondja/ hiba történik. Ezután visszairányítja a felhasználót a megadott redirect url-re.

6. A külső rendszer összeállít egy hívást a történetek alapján az alkalmazásunk által megadott redirect url-re Ezt megpróbálja elküldeni x-szer, majd, ha nem érkezik kielégítő válasz hibát jelez.
7. Amennyiben a megadott redirect URL-re hívás érkezik az alkalmazás backendje megfelelőképpen tudja kezelni a sikeres/ sikertelen fizetéseket. Fontos megjegyezni ezen a ponton, hogy amennyiben valaki ismeri ennek a végpontnak a felépítését és url-jét megpróbálhatja átverni a rendszert azzal, hogy hamis visszaigazolást küld fizetéséről, ennek elkerülése érdekében az adott tranzakciók állapotát mindig le kell ellenőrizni, egyeztetni a külső szolgáltatással ezen a ponton is.

Visszafizetés esetén is a fent leírt módon lehet eljárni, így például fizetés után a foglalás lemondása esetén automatikusan felmérhetjük, hogy a felhasználó jogosult-e visszafizetésre és azt automatikusan elvégeztethetjük a rendszerrel – rendszerekkel.

ReservationService: A foglalás árának számításakor rengeteg izgalmas kérdést kellett megoldanom.

Az adatbázisban egyedi azonosítónak GUID-ot használtam, mely tökéletesen alkalmas erre a feladatra, azonban felhasználói fogyasztásra meglehetősen alkalmatlan. Erre a célra egy egyedi azonosítót generáltam minden foglaláshoz, melyet a felhasználó megkap emailben és probléma/ kérdés esetén erre hivatkozhat.

```
1 reference | Szalkai Krisztián Farkas, 244 days ago | 1 author, 1 change
public static string GenerateAccessToken() // generates a unique, random, and alphanumeric
{
    const string availableChars =
        "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    using (var generator = new RNGCryptoServiceProvider())
    {
        var bytes = new byte[16];
        generator.GetBytes(bytes);
        var chars = bytes.Select(b => availableChars[b % availableChars.Length]);
        var token = new string(chars.ToArray());
        return token;
    }
}
```

4.1.6-6 Egyedi azonosító generálása

A módszer előnye más hasonló kódolásokkal szemben, hogy megadható hány darab karaktert szeretnénk, illetve mely karakterekből válasszon az algoritmus, mely kifejezetten hasznos, ha a felhasználói „élményt” is figyelembe vesszük. Több

választható karakter, illetve hosszabb generált kulcsok használatával csökkenthető annak az esélye, hogy kettő ugyanolyan azonosító jöjjön létre.

Az ár számításakor a két elsődleges szempont a távolság és a bérelt autó típusa. A távolság meghatározására a Google Maps Directions Api szolgáltatását használtam. Az email szolgáltatáshoz hasonlóan ehhez is konfigurálni kell egy gmail fiókot. A legtöbb API fizetős, így első lépésben egy fizetési módot kell hozzáadni a fiókhoz. Ha ezzel megvagyunk, generálhatunk egy kulcsot mellyel azonosíthatjuk az alkalmazásunkat.

API Keys

<input type="checkbox"/>	Name	Creation date ↓	Restrictions	Key	Actions
<input type="checkbox"/>	▲ API key 2	Nov 16, 2021	None	AIzaSyBli-...AbhvpRQWp8	
<input type="checkbox"/>	▲ API key 1	Oct 21, 2021	None	AIzaSyBayJ...tuikDY2G9U	

4.1.6-7 Api kulcsok felülete

Az általunk használt API-k aktiválása után érdemes az adott végpontokat lekorlátozni, ha a publikált oldalunkon keresztül támadás formájában sok kérés megy az adott API felé, jelentős összegeket fizethetünk.

Quota Name	Limit
Map loads per day	1,000
Map loads per minute	100
Map loads per minute per user ?	100

4.1.6-8 API hívások korlátozása

A fő szempontok mellett különböző alszempontokat is figyelembe vettem, például azt, hogy egy út érint-e adott helyeket (repülőterek, vasútállomások, hotelek), zónákat. Londonban például egyes területeknél behajtáskor úgynevezett dugódíjat kell fizetni. Az ilyen területek meghatározására – mivel több esetben koordinátákkal meghatározott területekről van szó, ezért nem használhattam olyan metódust amely egy ponthoz közel eső területek vizsgálja – egy poligon tartalmazás megállapítására szolgáló algoritmust használtam, mely alább a 4.1.6-9 ábrán látható.

```

/// <summary>
/// Determines if the given point is inside the polygon
/// </summary>
/// <param name="polygon">the vertices of polygon</param>
/// <param name="testPoint">the given point</param>
/// <returns>true if the point is inside the polygon; otherwise, false</returns>
8 references | Szalkai Krisztián Farkas, 269 days ago | 1 author, 1 change
public static bool IsPointInPolygon4(PointF[] polygon, PointF testPoint)
{
    bool result = false;
    int j = polygon.Count() - 1;
    for (int i = 0; i < polygon.Count(); i++)
    {
        if (polygon[i].Y < testPoint.Y && polygon[j].Y >= testPoint.Y
            || polygon[j].Y < testPoint.Y && polygon[i].Y >= testPoint.Y)
        {
            if (polygon[i].X + (testPoint.Y - polygon[i].Y)
                / (polygon[j].Y - polygon[i].Y)
                * (polygon[j].X - polygon[i].X)
                < testPoint.X)
            {
                result = !result;
            }
        }
        j = i;
    }
    return result;
}

```

4.1.6-9 Poligon tartalmazás vizsgálata

Az ár számításakor a kezdő és a végpontra is ellenőriztem, hogy belesznek-e a különböző feláras zónákba. A repülőterek leellenőrzésére például az alább látható (4.1.6-10) kódrészlet szolgál, Ahol az *airport* lista tartalmazza az előre definiált koordinátahalmazokat melyek a repülőterek területét jelölik ki.

```

var startIsInAirport = false;
var endIsInAirport = false;

foreach (var airport in airports)
{
    if (IsPointInPolygon4(airport,
        new PointF
        { Y = (float)directions.Routes.First().Legs.First().EndLocation.Longitude,
          X = (float)directions.Routes.First().Legs.First().EndLocation.Latitude
        }))
    {
        endIsInAirport = true;
        break;
    }
    if (IsPointInPolygon4(airport,
        new PointF { Y = (float)directions.Routes.First().Legs.First().StartLocation.Longitude,
                     X = (float)directions.Routes.First().Legs.First().StartLocation.Latitude
        }))
    {
        startIsInAirport = true;
        break;
    }
}

```

4.1.6-10 Repülőterek ellenőrzése

Különböző szolgáltatásoknál bevett szokás reklámozás, jutalmazás céljából kuponok használata, az én alkalmazásom esetén akciós kódokat adhat meg az adminisztrátor melyeket a felhasználók foglalás során válthatnak be. A kódok megadásánál lehet összeget, illetve százalékot is megadni, illetve egy lejárat dátumon túl a kód nem felhasználható, ezen kívül opcionálisan darabszám is megadható, hogy hányszor használható fel egy adott kód. Az alábbi kódrészleten a kupon modelljének kódja látható:

```

public class Discount
{
    public Guid Id { get; set; }
    public string DiscountCode { get; set; }
    public double? DiscountAmount { get; set; }
    public double? DiscountPercent { get; set; }
    public int? DiscountCount { get; set; }
    public DateTime ExpireDate { get; set; }
}

```

4.2 Frontend

A felhasználói felületek megalkotását a 3.4 fejezetben bemutatott wireframe-ek alapján készítettem el. Ebben a fejezetben először bemutatom a kód felépítését, majd a funkciók mentén az alkalmazás felületeit, a fejlesztés közben felmerülő problémákat, megoldásokat.

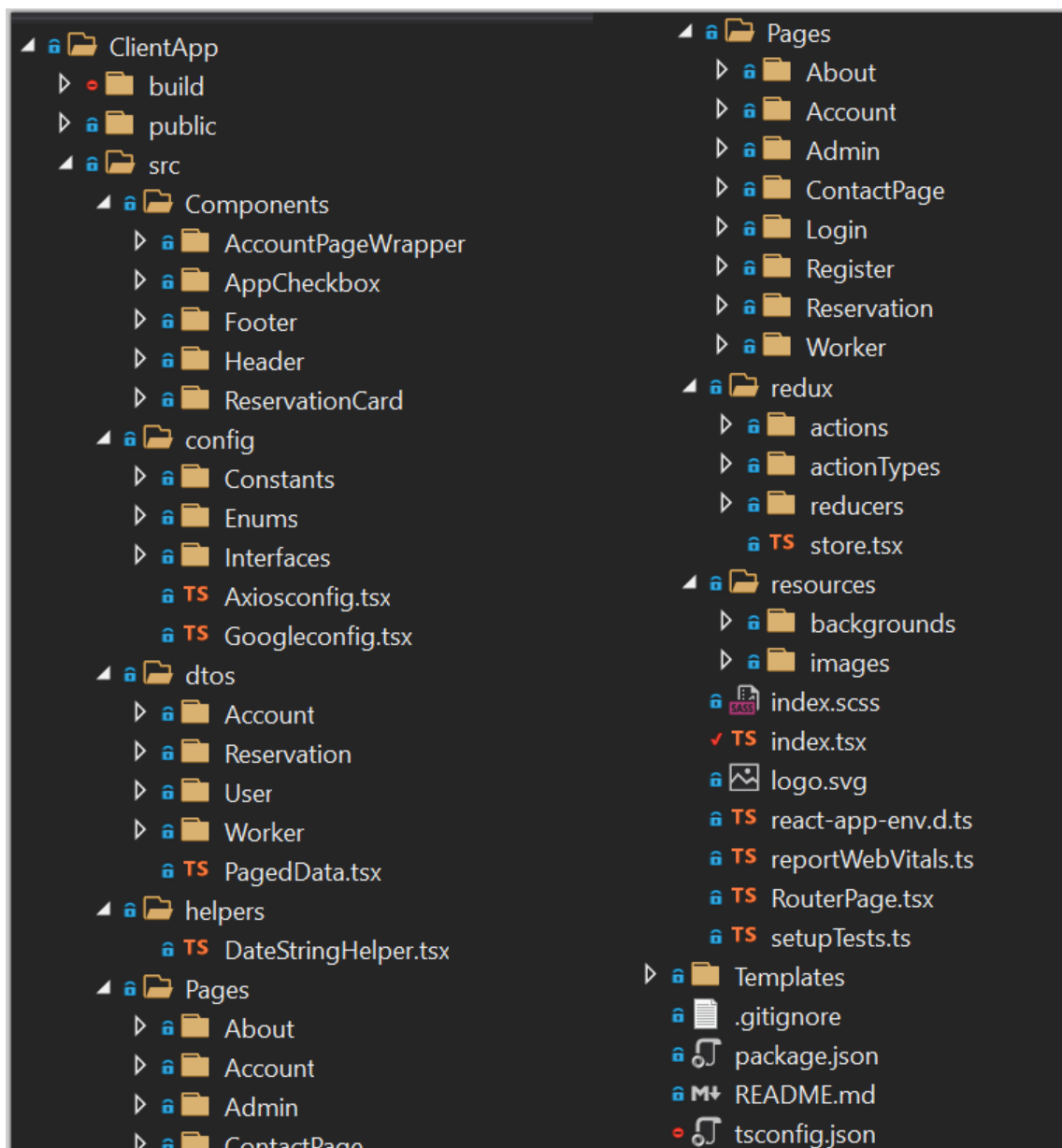
4.2.1 Felépítés

A React alkalmazást a *create-react-app* nevű csomaggal készítettem. Ennek előnye, hogy egyetlen *npm* paranccsal létrehozza a szükséges fájlokat, beállítja az alkalmazáshoz használt konfigurációkat, beállítja a build eszközöket.

```
my-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
└── src
    ├── App.scss
    ├── App.tsx
    ├── App.test.tsx
    ├── index.scss
    ├── index.tsx
    ├── logo.svg
    ├── serviceWorker.tsx
    └── setupTests.tsx
```

Az eszköz által létrehozott struktúra a fenti ábrán látható. Fentről lefelé haladva, a *node_modules* mappa tartalmazza a telepített *node* csomagok kódját, ennek tartalma minden build-nél, csomag telepítésénél automatikusan generálódik, igen nagy méretűre nőhet, így általában például a verziókövetésből is kivesszük – ezt a kivételt tartalmazza az eszköz által generált *gitignore* fájl is. A *package.json* fájl írja le, hogy milyen csomagok vannak telepítve az alkalmazáshoz, milyen verziókban kell azokat telepíteni, illetve minden más külső függőség metaadatát is itt kell megadni. A *public* mappába kerülnek azok a fájlok melyeket kívülről el lehet majd érni, a különböző méretű logók, SEO fájlok, mint a *robots.txt* vagy a sitemap, illetve itt van alapértelmezetten az oldal „belépési pontja” az *index.html*. Az *index* fájl az a *html*, amit a webszerver kiszolgál a böngészőnek, majd ennek a *root* id-vel ellátott *body* részébe kerül a megfelelő tartalom, ennek megfelelően ebben a fájlban van a *html head*, ide adhatjuk meg a metaadatokat, link tageket. Az *src* mappa tartalmazza az alkalmazás felületeinek, logikájának kódját. Az *src/index.tsx* a React kód belépési pontja, itt lehet megadni, hogy mi történjen az oldal betöltése után. Alapértelmezetten az *App.tsx* kerül betöltésre. Elméletileg ezen a ponton írhatnánk akár az egész működést ebbe (az *index*) fájlba, ám természetesen a keretrendszer használatának pont az a lényege, hogy átláthatóbb, karbantarthatóbb kódot

készítsünk, így ezen a ponton általában szétbontjuk a komponenseket funkciók alapján. Az általam készített struktúra az alábbi 4.2.1-1 ábrán látható.



4.2.1-1 A frontend kód felépítése

4.2.1.1 Navigáció

Az alkalmazásban a navigációt a React Router segítségével valósítottam meg. Ahogy a 2.2.1 fejezetben is bemutatam, ez a komponens segít az SPA-n belüli navigálásban, a navigálási logikát kiszerveztem a *RouterPage.tsx* fájlba, ezt töltöttem be az alkalmazás indulásakor az alapértelmezett *App.tsx* helyett.

```

<Router >
  <HeaderComponent/>
  <Switch>
    <Route exact path="/login">
      <LoginPage />
    </Route>
    <Route exact path="/register">
      <RegisterPage />
    </Route>
    <Route exact path="/about" component={AboutPage}/>
    <Route exact path="/services/airport"/>
    <Route exact path="/services/chaffeurs"/>
    <Route exact path="/services/events"/>
    {props.token || <Redirect to="/login"/>}
    {props.role === UserRoles.Administrator
      && <Route exact path="/users" component={UsersPage}/>}
    {props.role === UserRoles.Administrator
      && <Route exact path="/users/:id" component={UserReservationsPage}/>}
    {props.role === UserRoles.Administrator
      && <Route exact path="/home" component={ReservationsPage}/>}
    {props.role === UserRoles.Administrator
      && <Route exact path="/manage" component={ManagePage}/>}
    {props.role === UserRoles.Administrator
      && <Route exact path="/reserve" component={ReservationPage}/>}
    {props.role === UserRoles.User
      && <Route exact path="/home" component={ReservationPage}/>}
    {props.role === UserRoles.Worker
      && <Route exact path="/home" component={JobsPage}/>}}
    <Route exact path="/successfulPayment" component={SuccessfulPaymentPage} />
    <Route exact path="/account/overview" component={OverViewPage}/>
    <Route exact path="/account/pass" component={ChangePasswordPage}/>
    <Route exact path="/account/personal" component={ChangePersonalDataPage}/>
    <Route exact path="/account/reservations" component={MyReservationsPage}/>
    <Route exact path="/account/:id/details" component={ReservationDetailsPage}/>
    <Route exact path="/account/settings" component={SettingsPage}/>
    <Route exact path="/contact" component={ContactPage}/>
    <Redirect to="/home"/>
  </Switch>
</FooterComponent/>

```

4.2.1-2 A frontend URL struktúrája

A kódban szereplő – a 4.2.1-2 képen látható – Route bejegyzésekkel hozhatók létre a frontend végpontok – lapok. A komponens egy url-re történő kérés esetén fentről lefelé végig nézi a bejegyzéseket és az első egyezés esetén visszaadja a megadott komponenst, ebből látszik, hogy a *Switch* komponens utolsó bejegyzése visszairányítja a felhasználót a „/home” útvonalra, ha nem talál más egyezést (így a korábban megszokott 404 oldal sem szükséges ebből a szempontból). A lapok között többféleképpen lehet navigálni, a leggyakrabban a már említett Redirect komponenst és a lent látható Link komponenst használtam.

```
<Link href="/login" variant="body2">
  Already have an account? Sign in
</Link>
```

A Link komponens kiír egy a *variant* változóban meghatározott módon formázott szöveget és kattintásra *href* segítségével elnavigál a megadott oldalra.

4.2.1.2 Config

Ebbe a mappába helyeztem a konfigurációs fájlokat, konstansokat, interfészeket. A *Googleconfig.tsx* fájl a google api kulcsát tartalmazza, melyet már a 4.1.6-7 Api kulcsok felülete ábránál is említettem. Az *Axiosconfig.tsx* fájlba az axios szolgáltatáshoz kapcsolódó beállítások kerültek. Az axios egy singleton példány létrehozásával és konfigurálásával működik, ezt a példányt hozzuk létre ebben a fájlban és itt kerül beállításra a backend URL-je. A hálózati kommunikációhoz tartozik még egy beállítás mely nem itt kerül beállításra, hanem a fő komponensben – esetünkben a *RouterPage.tsx*.

```
useEffect(() => {
  if(init){
    axiosInstance.interceptors.response.use((response) => response, (error) => {
      if (error.response?.status === 404) {
        props.setError("Unknown error occurred");
      }
      else {
        if (error.response?.data !== undefined) {
          props.setError(error.response.data.message);
        }
        else {
          props.setError("Cannot reach server");
        }
      }
      throw error;
    });
    setInit(false);
  }
})
```

4.2.1-3 Axios interceptor beállítása

A fent látható lépén (4.2.1-3) egy úgynevezett interceptor-t állítunk be az összes HTML kérésre küldött válaszhoz. A backenden beállított hibakezelő middleware hatására, melyről a 4.1.3.3 fejezetben beszélek részletesen, minden hibára specifikus státusz jön vissza ezekre tudunk itt egységesen reagálni. Esetünkben, ha 404-es hiba jön nem kezelt probléma történt, ha nem, akkor a backend által egységesen formázott üzenet

érkezik vissza melynek kiírhatjuk a megfelelő attribútumát a felhasználónak. Abban az esetben, ha nem 404-es hiba, de nincs response beállítva nem is értük el a szerveret.

Az enum típusú változókat és interfészeket nem emelném ki, mivel megegyeznek a backend oldalon lévőkkel, ám a konstansokat megemlíteném. A konstansok használata egyszerűbbé teszi az olyan statikus tartalmak kezelését, melyek időnként változhatnak és több mint egy helyen jelennek meg a kódban, esetünkben például a kontakt információk.

```
export const mainEmail = "mainemail@smth.com";
export const phoneNumber = "06304569874";
export const lostAndFoundEmail = "lostandfound@smth.com";
```

4.2.1.3 Dtos

A DTO-k megegyeznek a backenden lévő osztályokkal így ezeket nem fejtem ki ebben a fejezetben. Azok a kódrészletek melyek közvetlenül megegyeznek a backenden lévő kóddal generálhatóak különböző eszközök segítségével, például a Swaggernek is van ilyen modulja, ám tapasztalataim szerint ezek konfigurációjával és megbízható működésével több probléma, munka van, mint amennyi időt spórol olyan kevés osztály esetén amivel jelenleg dolgoztam, így manuálisan vettem fel és tartom karban az alkalmazásban ezeket a részeket is.

4.2.1.4 Helpers

A helper osztályok olyan kódrészletek melyek talán a .net-es világból Extension-ként ismert osztálykiegészítésekhez hasonlíthatóak leginkább, gyakran használt kódrészleteket szervezünk ki statikus függvényekbe.

```
export const dateToString = (date: any) => {
  var d = new Date(date);
  return d.getFullYear() + "/"
    + (d.getMonth() + 1) + "/" + d.getDate()
    + " " + (d.getHours() % 12)
    + ":" + (d.getMinutes() < 10 ? ("0"+d.getMinutes()) : d.getMinutes())
    + (d.getHours() >= 12 ? ' pm' : ' am')
}
```

4.2.1-4 Helper dátum formázásához

A fent látható kód például egy dátumot formáz, hogy minden esetben egységesen tudjuk megjeleníteni a kívánt tartalmakat. Itt kiemelnék egy érdekes részletet mely jól mutatja miért szeretem a TypeScript nyelvet. A függvény egyetlen bemenő paramétere *any* típusúként van megjelölve, a függvényen belüli *d* paraméter pedig ebből a változóból

generálódik, még hozzá úgy, hogy a *date* lehet *Date* típusú vagy *string* típusú, mindkét módon működik a függvény, ám a *d* változó már *Date* típusú, így a későbbiekben a típusosság előnyeit élvezhetjük az IntelliSense által. A JavaScript rugalmasságát ötvözve a típusos nyelvek biztonságával rendkívül kényelmes megoldásokat hozhatunk létre.

4.2.1.5 Resources

Ide kerültek a frontend által használt erőforrások. A jelenlegi alkalmazásban ez a képeket jelenti, melyek megjelennek a felületeken.

4.2.1.6 Redux

A Redux szerepéről a 2.3.1 fejezetben beszéltem. Alapvetően két komponensre van szükség, a store a reducerekből áll össze, melyek a globálisan tárolt állapotot manipulálják, az actionök pedig azok az események melyek elsütésének hatására a reducerek meghívódnak, azonban a TypeScript típusossága miatt szükségünk van még az *ActionTypes* fájlokra is, melyek nem csinálnak mást csak a különböző eseményeket típusosan definiálják. A teljes folyamat így a következőképpen néz ki:

1. Egy esemény típusának definiálása:

```
export interface ISetErrorActionType {  
  type: string,  
  payload: string  
}
```

2. Az esemény meghívásának definiálása:

```
export const setError = (content: string): ISetErrorActionType => {  
  return {  
    type: SET_ERROR_MSG,  
    payload: content  
  }  
};
```

3. Az eseménytípus azonosítójának definiálása:

```
export const SET_ERROR_MSG = '[Error] Set Error';
```

4. Az egy reducerhez tartozó típusok egyesítése:

```
export type ErrorActionTypes =  
  ISetErrorActionType &  
  IClearErrorActionType
```

5. A reducer definiálása:

```

export default function
(state = initialState, action: ErrorActionTypes): IErrorState {
  switch (action.type) {
    case SET_ERROR_MSG: {
      return {
        ...state,
        message: action.payload
      }
    }
    case CLEAR_ERROR_MSG: {
      return {
        ...state,
        message: ""
      }
    }
  }
  default:
    return state;
}
}

```

4.2.1-5 Error reducer definiálása

6. A store bekötése egy komponensbe:

Két irányt kell definiálni, a store-ból a komponens felé menő változókat és a komponensből a store felé menő eseményeket.

```

const mapStateToProps = (state: RootState): IMappedState => {
  return {
    token: state.user.token,
    role: state.user.role,
    error: state.error.message
  }
}

const mapDispatchToProps = (dispatch: Dispatch<AnyAction>) =>
  bindActionCreators(
    {
      clearError,
      setError
    },
    dispatch
  );
const connector = connect(mapStateToProps, mapDispatchToProps)
export default connector(RouterPage)

```

4.2.1-6 Store bekötése egy komponensbe

```

interface IMappedState{
  token: string;
  role: string;
  error: string;
}

interface IDispatchedProps {
  clearError: () => void;
  setError: (msg: string) => void;
}

type Props = IMappedState & IDispatchedProps;

function RouterPage(props: Props){
  ...
}

```

Itt szintén a TypeScript típusosságának köszönhetően szükség van pár extra lépésre, hogy a storeból érkező, illetve esetlegesen a saját változókat átadhatjuk a komponens számára.

4.2.1.7 Pages

Ebbe a mappába kerültek az olyan React komponensek melyek teljes oldalakat jelenítenek meg a weboldalon. A könnyebb átláthatóság érdekében a lapokat alcsoportokra osztottam, illetve minden oldal saját mappát kapott, melyben szerepel a

TypeScript kód, illetve az scss fájl is. A React néhány más például php keretrendszerekkel ellentétben szerencsére nem tesz megkötéseket a fájlstruktúra szintjén, így szabadon helyezhetjük emberi szemmel kellemesen fogyasztható módon a fájlokat.

4.2.1.8 Components

Ide kerültek az olyan komponensek melyek oldalak kisebb részeit képzik, általában több helyen felhasználásra kerülnek. A React egyik előnye, hogy a definiált komponenseket később a kódban HTML tagekhez hasonlóan használhatjuk fel.

```
export interface AppCheckBoxProps {
  id?: string;
  label: string;
  checked: boolean;
  onChange(e: React.ChangeEvent<HTMLInputElement>): void;
}

export default function AppCheckbox(props: AppCheckBoxProps) {
  return(
    <FormControlLabel
      control={
        <Checkbox
          style={{
            color: "goldenrod",
          }}
          checked={props.checked}
          onChange={(e) => props.onChange(e)}
          id={props.id}
        />
      }
      label={props.label}
    />
  )
}
```

4.2.1-7 Checkbox definiálása

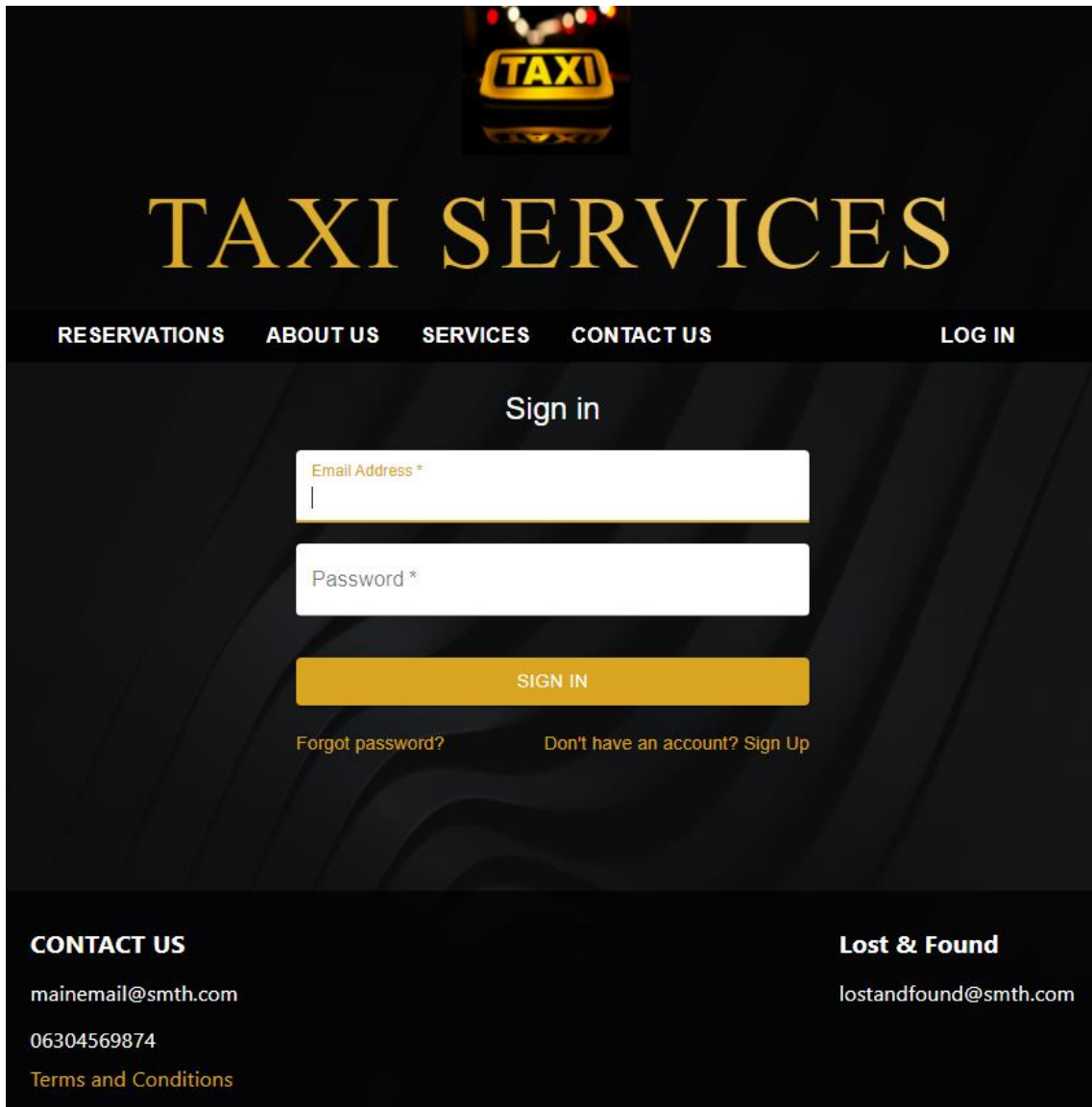
A fenti 4.2.1-7 ábrán például egy saját checkbox komponens definiálását láthatjuk. A komponenst az `<AppCheckbox />` tag használatával jeleníthetjük meg később bárhol a kódban, így minden checkbox ugyanúgy néz ki az alkalmazásban, ha pedig változtatásra szorul egy helyen tehetjük meg.

4.2.2 Funkciók

Ebben a fejezetben bemutatom az alkalmazás működését és az elkészült felületeket, azok szerepét, működését. Jelen dokumentumban a foglalás menetére koncentrálok, nem fejték ki minden oldalt, statikus felületet.

4.2.2.1 Bejelentkezés/ Regisztráció

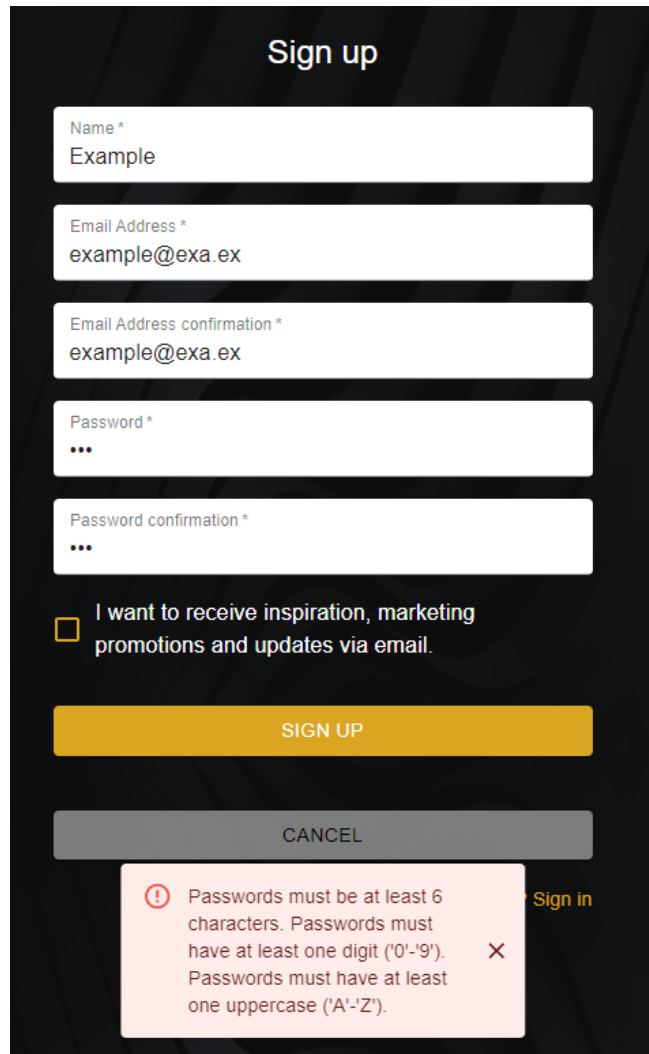
A folyamat természetesen a regisztrációval, illetve a bejelentkezéssel kezdődik.



4.2.2-1. ábra Bejelentkezési felület

A jelszó mezőben a tényleges karakterek helyett „*” szimbólumok jelennek meg. Regisztrálás során a felhasználónév, jelszó követelményeit a backenden a 4.1.3-6 ábrán látható módon konfiguráltam. Az adatok ellenőrzése frontenden is fontos, mivel a

felesleges backend forgalom csökkenthető vele, de könnyen változtatható így nem elegendő, backenden is validációt kell végezni a kritikus adatokra. A hibák megjelenítését a *material-ui* külső komponens könyvtár által szolgáltatott *Alert* modullal jelenítem meg, rossz jelszó esetén például az alábbi képen látható módon.



The image shows a 'Sign up' form with the following elements:


- Name ***: Input field with 'Example' text.
- Email Address ***: Input field with 'example@exa.ex' text.
- Email Address confirmation ***: Input field with 'example@exa.ex' text.
- Password ***: Input field with masked characters '...'. A red error message is displayed below it: 'Passwords must be at least 6 characters. Passwords must have at least one digit ('0'-'9'). Passwords must have at least one uppercase ('A'-'Z').'
- Password confirmation ***: Input field with masked characters '...'. A red error message is displayed below it: 'Passwords must be at least 6 characters. Passwords must have at least one digit ('0'-'9'). Passwords must have at least one uppercase ('A'-'Z').'
- ☐ **I want to receive inspiration, marketing promotions and updates via email.**
- SIGN UP** button (yellow).
- CANCEL** button (gray).
- Sign in** link (yellow text).

4.2.2-2. ábra Beviteli hiba megjelenítése

A külső UI könyvtárak használata rengeteg előnnyel jár, azon kívül, hogy nem nekünk kell megírni a komponenst, egy kész modult használhatunk, ami egyértelműen gyorsabb megoldás, ráadásul az ilyen könyvtárak általában követik az aktuális trendeket, tehát, ha nem érezzük magunkat biztosnak dizájn készítésben, hasonló könyvtárak használatával nem nyúlhatunk félre.

4.2.2.2 Foglалás oldal

Bejelentkezés után a felhasználó azonnal foglalhat is a specifikációban meghatározott módokon.



TAXI SERVICES

[RESERVATIONS](#)
[ABOUT US](#)
[SERVICES](#)
[CONTACT US](#)

[ACCOUNT](#)
[LOG OUT](#)
Email: taxiservicediploma@gmail.com

One-Way


By The Hour

Tatabánya, Hungary

Budapest, Damjanich utca, Hungary

Date
12/08/2021 11:10 PM


Executive



Max 4 passengers

SELECT


Luxury



Max 4 passengers

SELECT

7 seater



Max 7 passengers

SELECT

Preferences:

☒ Smoking

☐ Pets allowed

☐ Big trunk

Comment

I need to stop at a shop midway.

Discount Code

CALCULATE

173.24 -

MAKE RESERVATION

CONTACT US

mainemail@smith.com
06304569874
[Terms and Conditions](#)

Lost & Found
lostandfound@smith.com

4.2.2-3 Egy útra történő foglalás felülete

A felhasználó egy útra foglalás esetén kiválaszthatja az indulás és az érkezés helyszínét, az indulás időpontját, sofőr bérlet esetén pedig a felvétel helyét és időpontját, ezen kívül a bérlet időtartamát. A bérelt autó típusa, a preferenciák kiválasztása, a komment megadása és az esetleges promóciós kód megadása a két típusnál megegyezik. A foglalás véglegesítésekor a bejegyzés mentésre kerül a felhasználóhoz és ezzel a folyamat első lépése lezárult.

4.2.2.3 Összefoglalás és fizetés

A foglalás véglegesítésére eredetileg a bejegyzés tényleges mentése nélkül, az adatokat ideiglenesen tárolva egy összefoglaló oldalt képzeltem el, ám gyorsan beláttam, hogy ez az út sok fejfájással jár, elsősorban a külső fizető oldalra való átirányítás miatt. Bonyolult tárolna ezeket az adatok abban az esetben, ha a fizetés megghiúsul, vagy bármi más probléma lép fel az átirányítási lánc során, így végül úgy döntöttem a foglalás mentésre kerül még fizetés előtt, így a felhasználó bármikor folytathatja a folyamatot a saját profiljának adatai közül.

< My Account / Reservation Details	
Account overview	Reservation Identifier: MOUSShtdJNMROc8D
Change password	Reservation Type: Oneway
Change personal data	Car Type: Luxury
My reservations	Date: 2021/12/8 11:10 pm
Settings	Activated Discount: -
	From Address: Tatabánya, Hungary
	To Address: Budapest, Damjanich utca, Hungary
	Comment: I need to stop at a shop midway.
	Selected Preferences: • Smoking
	Price: 173.24HUF
	Reservation Status: Reserved
	Select payment option:
	Reservation Made: 2021/11/30 11:39 am
	Last Edited: -

4.2.2-4 Foglalás áttekintésének felülete

Ahogy a 4.1.3 fejezetben is említettem, jelenleg kizárólag a Barion támogatott fizetési módként, ám a felületet úgy terveztem, hogy könnyen hozzáadható legyen más fizetési mód is. A kiválasztott fizetési mód ikonjára kattintva a felhasználó átirányításra kerül a külső oldalra, majd sikeres fizetés után a rendelése véglegessé válik. A fizetés menetét a 4.1.6.2 fejezetben részletesen is bemutattam. A rendszerek integrációjának tesztelését megkönnyíti, hogy általában vannak olyan kártyaszámok melyekkel szimulálható a sikeres/ sikertelen fizetés, azonban nehezítő tényező, hogy a legtöbb rendszer esetében csak egy publikusan megosztott oldal esetében tesztelhető a vissza irányítás és a callback működésének tesztelése.

Sikeres fizetés esetén a felhasználó vissza irányításra kerül a foglalásaihoz, ahol – amennyiben a fizetés callback függvénye sikeresen lefutott – az összefoglaló felület reflektálja a sikeres fizetés tényét.

Reservation Identidier:	M0USShtxJNMR0C8D
Reservation Type:	Oneway
Car Type:	Luxury
Date:	2021/12/8 11:10 pm
Activated Discount:	-
From Address:	Tatabánya, Hungary
To Address:	Budapest, Damjanich utca, Hungary
Comment:	I need to stop at a shop midway.
Selected Preferences:	<ul style="list-style-type: none">• Smoking
Price:	173.24HUF
Reservation Status:	Payed
Reservation Made:	2021/11/30 11:39 am
Last Edited:	-

CANCEL RESERVATION

4.2.2-5 Sikeresen fizetett rendelés összefoglaló felülete

A foglalás adott időn belül lemondható, addig megjelenik a lemondást lehetővé tévő gomb a felületen, az időpont elmúltával a gomb is eltűnik. A felhasználó szempontjából ekkor vége is a foglalási folyamatnak, a további teendőket az adminisztrátorok és dolgozók végzik. A foglalás állapotának változását a fenti képeken látható oldal fogja mutatni, ám a továbbiakban ezeket már nem fogom külön kiemelni.

4.2.2.4 Foglalások dolgozókhoz rendelése

Miután egy foglalás véglegessé vált – fizetésre került – az adminisztrátori jogkörrel rendelkező felhasználók abban az időpontban szabad dolgozókhoz rendelik a bejegyzéseket.

RESERVATIONS
ABOUT US
SERVICES
CONTACT US
ADMIN
RESERVE

ACCOUNT
LOG OUT
Email: adminemail@smth.com

From Date:
To Date:
Min. Price:
Max. Price:
Car type:

Reservation type:
Reservation status:
Preferences:

Page size: 5
1/3

From:
To:
Status:
Reservation type:
Car type:

Tatabánya, Hungary
Budapest, Damjanich utca, Hungary
Paid
Oneway
Luxury

Preferences:
• Smoking

ASSIGN

Price: 173.24 .-

Date: 2021/12/8 11:10 pm

From:
To:
Status:
Reservation type:
Car type:

Budapest, Damjanich utca, Hungary
Budakeszi, Farkashegyi Repülőtér, Hungary
Reserved
Oneway
Minibus

Preferences:
-

Price: 113.49 .-

Date: 2021/3/19 3:41 am

4.2.2-6 Dolgozók foglalásokhoz rendelése

A foglalások között a könnyebb átláthatóság érdekében szűrni lehet a találatokat. A listában megjelenik az összes státuszú foglalás, ám csak a megfelelő állapotban lévőhöz lehet dolgozót rendelni.

4.2.2.5 Dolgozói felület

Miután egy foglalás egy dolgozóhoz kerül, az megjelenik a dolgozó felületén. A dolgozói jogosultsággal rendelkező felhasználó két státuszt állíthat be: *úton* és *megérkezett*.

Page size: 10
☒ Hide completed jobs.
1/1

From: Tatabánya, Hungary
Preferences: • Smoking
UPDATE

To: Budapest, Damjanich utca, Hungary

Status: Assigned

Reservation type: Oneway

Car type: Luxury

Date: 2021/12/8 11:10 pm

From: Budapest, Tölgyfa utca 26, Hungary
Preferences: • Big trunk
UPDATE

Duration: 6 hours

Status: On the way

Reservation type: By the hour

Car type: Minibus

Date: 2021/5/20 0:00 pm

From: Budapest, Daubner, Szépvölgyi Way, Hungary
Preferences: • Smoking
• Pets allowed
• Big trunk
Date: 2021/5/26 0:04 pm

Duration: 3 hours

Status: Arrived

Reservation type: By the hour

Car type: Executive

4.2.2-7 Dolgozói felület.

Amikor a foglalás átkerül a *megérkezett* állapotba a foglalási folyamat lezárul. A foglalások továbbra is szerepelnek a különböző felhasználói felületeken adminisztratív szempontból hasznos, ám az adatokat, állapotot módosítani már nem lehet.

5 Összefoglaló

5.1 Továbbfejlesztési lehetőségek

A rendszer továbbfejlesztésében az első lépés mindenképpen a flottakezelés teljesen, vagy legalább részben automatizálása. Jelenleg minden lépést manuálisan kell végezni beleértve az adott időben rendelkezésre álló sofőrök/ autók kezelését, ami egy idő után kézzel átláthatatlanná válik.

Mivel egy taxi szolgáltatásról van szó, ezért érdemes lenne egy mobilos – lehetőleg cross/ multiplatform – frontend kialakítása. Mind az autókban dolgozó sofőrök kényelmesebb adminisztrációja, mind a felhasználók kényelmesebb foglalása érdekében hasznos lehet egy ilyen alkalmazás.

Érdekes fejlesztés lehet még a teljes adminisztrációs folyamatot az oldalon intézni. A dolgozók ledolgozott munkaóráit, szabadságait, kifizetéseit, baleseteit, az autók állapotát stb. mind kezelhetné a webszolgáltatás egy adminisztrációs aloldala, bár hasonló funkciókkal számos külső szolgáltatás létezik.

5.2 Végző

A diplomamunka készítése során megismertem új technológiákat, illetve mélyítettem a tudásom a már ismertekben. Az évek során lehetőségem volt több webes keretrendszert, nyelvet megismerni és számomra egyértelműen ez a kombináció a legmegfelelőbb egy hasonló projekt készítéséhez.

A .NET Core mind funkcionalitásban, megbízhatóságban, támogatottságban, mind sokoldalúságban nyerte a versenyt tapasztalataim szerint hasonló funkcionalitású vetélytársaival szemben. Válszínű, hogy az is szerepet játszik, hogy ennek használatában van a legtöbb tapasztalatom, ám ebben a nyelvben látom a jövőt WEB API-k fejlesztése terén.

A React és TypeScript kombinációja tökéletesen ötvözi a JavaScript által nyújtott rugalmasságot az objektumorientált típusos nyelvek biztonságával, ezen kívül a más fejlesztők által készített és karbantartott komponensek nagyban könnyítik, illetve gyorsítják a fejlesztést. Egyetlen komoly hiányossága melynek felfedezésére sajnálatos módon nem marad idő és hely ezen dolgozat írásakor, ám a későbbiekben biztosan

beleásom magam a témába – a SEO támogatottság. A *Search Engine Optimization* meglehetősen hasznos bármely weboldal szempontjából, ám a React nem teszi könnyebbé a Google által előírt segédanyagok követését. A teljes keretrendszer arra épül, hogy egy SPA-t készítsünk, mely alapvetően kliensoldali alkalmazást feltételez, azonban a keresőmotorok egyelőre nem kifejezetten támogatják a teljes mértékben JavaScripttel betöltődő oldalakat.

Összességében úgy gondolom, hogy sikerült egy olyan alkalmazást létrehoznom, mellyel az új technológiákkal való önálló megismerkedés és a már ismert technológiákban szerzett tapasztalataim bővítése közben egy valós problémát tudtam effektíven megoldani.

6 Irodalomjegyzék

- Flavio Copes. (2018. April 11). *Medium*. Forrás: <https://medium.com/free-code-camp/simple-http-requests-in-javascript-using-axios-272e1ac4a916>
- Google. (2021). *Google Trends*. Forrás: <https://trends.google.com/trends/explore?cat=31&q=Vue,React,Angular>
- Jigar Mistry. (2021. March 18). *Monocubed*. Forrás: <https://www.monocubed.com/10-most-popular-web-frameworks/>
- Maki, M. (2017. December 4). *Medium*. Forrás: <https://medium.com/@marcellamaki/a-brief-overview-of-react-router-and-client-side-routing-70eb420e8cde>
- Microsoft team. (2018. 06 19). *Microsoft*. Forrás: <https://docs.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server?toc=%2Faspnet%2Fcore%2Ftoc.json&bc=%2Faspnet%2Fcore%2Fbreadcrumb%2Ftoc.json&view=aspnetcore-3.1>
- Nnamandi, C. (2018. November 28). *Medium*. Forrás: <https://blog.bitsrc.io/comparing-http-request-libraries-for-2019-7bedb1089c83>
- Patel, T. (2020. August 31). *concetto labs*. Forrás: <https://www.concettolabs.com/blog/java-vs-net-which-is-better/>
- Rawat, D. (2019. January 2). *DEV.TO*. Forrás: <https://dev.to/singhdigamber/what-is-typescript-jig>
- Rick Anderson. (2019. 07 12). *Microsoft*. Forrás: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-3.1&tabs=visual-studio>
- Scott Hanselman. (2019. September 23). *Microsoft*. Forrás: <https://dotnet.microsoft.com/learn/dotnet/hello-world-tutorial/intro>
- Stack Overflow. (2021). *Stack Overflow*. Forrás: <https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-web-frameworks-loved2>
- Unknown. (2018. April 16). *Zero to hero in TypeScript*. Forrás: <http://zerotoherointypescript.blogspot.com/2018/04/typescript-introduction.html>

