

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Frontó András Levente

Multiplatform alkalmazás fejlesztés React Native alapokon

Konzulens:

Benedek Zoltán

BUDAPEST, 2019

Tartalomjegyzék

[Összefoglaló 5](#_Toc27075433)

[Abstract 6](#_Toc27075434)

[1 Bevezetés 7](#_Toc27075435)

[2 Technológiák feltérképezése 9](#_Toc27075436)

[2.1 React 9](#_Toc27075437)

[2.2 React Native 11](#_Toc27075438)

[2.3 React Native Web 13](#_Toc27075439)

[2.4 ReactXP 15](#_Toc27075440)

[2.5 Expo és React Native CLI 16](#_Toc27075441)

[2.6 Native Base és egyéb GUI könyvtárak 17](#_Toc27075442)

[2.7 Entity Framework és ASP.NET Core 18](#_Toc27075443)

[2.8 Axios 19](#_Toc27075444)

[2.9 SignalR 20](#_Toc27075445)

[3 Alkalmazás követelmények 22](#_Toc27075446)

[3.1 Felhasználási esetek 22](#_Toc27075447)

[3.1.1 Belépés előtt 22](#_Toc27075448)

[3.1.2 Belépés után 23](#_Toc27075449)

[4 Architektúra 26](#_Toc27075450)

[4.1 Adatbázis 26](#_Toc27075451)

[4.1.1 Adatbázis Táblák 27](#_Toc27075452)

[4.2 Backend 28](#_Toc27075453)

[4.2.1 Modellek 29](#_Toc27075454)

[4.2.2 Adatelérési réteg 29](#_Toc27075455)

[4.2.3 Biznisz logika 30](#_Toc27075456)

[4.2.4 API réteg 32](#_Toc27075457)

[4.3 Kliensek 32](#_Toc27075458)

[4.3.1 Auth 33](#_Toc27075459)

[4.3.2 HomeScreen 33](#_Toc27075460)

[4.3.3 Egyéb bejelentkezett képernyők 33](#_Toc27075461)

[5 Alkalmazás megvalósítása 35](#_Toc27075462)

[5.1 Bejelentkezés és regisztráció 35](#_Toc27075463)

[5.1.1 Regisztráció 36](#_Toc27075464)

[5.1.2 Bejelentkezés 37](#_Toc27075465)

[5.2 Bejelentkezett felhasználó és navigáció 37](#_Toc27075466)

[5.2.1 Android 39](#_Toc27075467)

[5.2.2 Web 40](#_Toc27075468)

[5.3 Profil képernyő 40](#_Toc27075469)

[5.3.1 Felhasználónév és email cím 41](#_Toc27075470)

[5.3.2 Jelszó megváltoztatása 41](#_Toc27075471)

[5.4 Skillek kezelése 42](#_Toc27075472)

[5.4.1 Új Skill felvétele 43](#_Toc27075473)

[5.4.2 Skills képernyő 45](#_Toc27075474)

[5.4.3 All Skills nézet lista felépítése 46](#_Toc27075475)

[5.4.4 Készségek kezelése az All Skill nézeten 48](#_Toc27075476)

[5.4.5 My Skills nézet 49](#_Toc27075477)

[5.5 Beszélgetési szálak 50](#_Toc27075478)

[5.5.1 Kérdés feltétele 50](#_Toc27075479)

[5.5.2 Threads képernyő 51](#_Toc27075480)

[5.5.3 My Feed nézet 52](#_Toc27075481)

[5.5.4 All Threads és My Threads nézetek 54](#_Toc27075482)

[5.5.5 Chat nézet 54](#_Toc27075483)

[5.5.6 Törlés és hiba kezelés 55](#_Toc27075484)

[6 Értékelés 57](#_Toc27075485)

[6.1.1 Fejlesztési javaslatok 58](#_Toc27075486)

[7 Irodalomjegyzék 60](#_Toc27075487)

Hallgatói nyilatkozat

Összefoglaló

Manapság az alkalmazás fejlesztés világában az egyik legnagyobb kihívás a sok elterjedt egymástól különböző platform. A felhasználói igények szerint egyre kedveltebb kliens oldali megvalósítások a Web alkalmazások és a mobilos natív kliensek. Ez a két irányvonal nagyon eltérő technológiákkal dolgozik. Ráadásul a mobil platform sem egységes, legalább két jelentős ágra válik szét (iOS, Android). A React Native webes technológiák segítségével teszi lehetővé natív mobilos alkalmazások fejlesztését a két említett mobil platformra. Innen már csak egy lépés, hogy a fejlesztés során felhasznált webes technológiákkal egy tényleges web alkalmazás is elkészíthető legyen. Erre egy lehetséges megoldás a React Native Web.

A webes és mobilos technológiák régóta érdekelnek. A React Native ezt a két világot ötvözi. Ez inspirált, hogy az Önálló laboratórium keretein belül a React Native keretrendszerrel foglalkozzak. Ennek egyenes következménye a szakdolgozatom témaválasztása.

A téma bemutatásához egy alkalmazást fejlesztettem, amely mobilos és asztali webes klienssel is rendelkezik.

Az alkalmazás egy tudás gráf felhasználásával összekapcsolja a szakértőket a kérdéseket megfogalmazó felhasználókkal. A feltett kérdések a hozzájuk kapcsolt szakterületek függvényében (Tag) automatikusan jutnak el a tudás gráf alapján a szakértőkhöz, akik valószínűleg megválaszolhatják azokat.

A fejlesztés során olyan könyvtárakat vizsgáltam meg, amelyek lehetővé teszik a webes és a mobil kliens egyidejű elkészítését. Ezeknek a könyvtáraknak a feltérképezése és azok megtalálása, amelyek képesek egymással együttműködni, volt a legnagyobb kihívás.

Az alkalmazás elkészítése során igyekeztem minél több technológia lehetőségeit bemutatni. Ezek közül a legfontosabbak React Native, React Native Web, SignalR és Native Base UI.

Mindkét platformra elkészítettem a klienst. A dolgozatban bemutatott szoftverben még sok fejlesztési potenciál rejlik. Újabb funkciókkal bővíthető és a felhasználói élmény tovább növelhető.

Abstract

Nowadays one of the greatest challenges in software development is dealing with all the different platforms. The Web applications and the native mobile applications both show growing tendencies in popularity. These platforms are based on different principals and work with very different technologies. To top it all, the mobile platform is divided as well. The most significant operating systems are the Android and the iOS. The React Native technology makes it possible to create a native mobile application with technologies reminiscent of Web development. The applications created this way are both Android and iOS compatible. From here its only a minor step to create a real Web application as well using these technologies. To create a full-on Web application the React Native Web library gives support for the React Native technology.

I was interested in the Web and mobile technologies for a long time and the React Native technology combines these two worlds. I was motivated by this fact to take this technology as the subject of my Independent laboratory (Önálló laboratórium). Writing my thesis about the subject was the next logical step.

The application that I developed includes both an Android native client and a Web browser-based client.

This application is based on a knowledge graph, which connects the experts in a field with the user asking a question about said field. The questions posted in the application are automatically delivered, based on the knowledge graph, to the users, who are most likely to have the answer for them.

During the development phase I searched for libraries that made it possible to create the mobile and Web client simultaneously. The most challenging part was to find the libraries that work well together.

While developing the application I tried to cover a wide array of technologies. The most important of these were the React Native, React Native Web, SignalR and Native Base UI.

Both clients were created successfully. The software showcased in the thesis has a lots of room for improvement. Plenty of new functions can be implemented and this can improve the user experience quite a bit.

# Bevezetés

Manapság már szinte mindenkinek van okostelefonja. A webalkalmazások fejlesztésénél épp ezért figyelembe kell venni ezt az egyre szélesebb körben használt platformot is. A mobil eszközöknél viszont számos más tényezőre is tekintettel kell lenni amellett, hogy a kisebb kijelzőn megfelelő elrendezéseket és kényelmes felhasználói interfészeket alakítsunk ki. Az okostelefon a legtöbb ember életének szerves része, akik szeretnek dedikált alkalmazásokat használni adott feladatokra (natív alkalmazásokat). Az ilyen alkalmazásoknak lényeges előnyei vannak a böngészőben betölthető weboldalakkal szemben. Ezen előnyök közé tartozik az is, hogy jobban integrálódnak a mobil felhasználási élménybe és offline is használhatók, még ha csökkentett funkciókkal is. A natív alkalmazások fejlesztésénél abba a problémába ütközünk, hogy a piacot uraló két rivális operációs rendszer esetében (iOS és Android) a fejlesztés teljesen máshogy történik.

A fent vázolt problémákra az egyik célszerű megközelítés egy olyan webalkalmazás kifejlesztése, amely fut Androidon és iOS-en is natív alkalmazásként, és még a böngészőben is elindul. A szakdolgozatom keretein belül egy olyan alkalmazást valósítok meg, mely a React Native [1] és a React Native Web [2] segítségével erre mutat példát. A React Native használata során pusztán JavaScript alapokon tudunk natív mobil alkalmazásokat fejleszteni. A React Native-ra épülő React Native Web pedig arra nyújt lehetőséget, hogy böngészőben is fusson az ugyanazon, illetve némiképp bővített kódbázisra épülő alkalmazásunk. Így egy nagyrészt közös kódbázisra építve natív Android, natív iOS és böngészőkben is működő webalkalmazást tudunk készíteni.

A dolgozatban a React Native platformot és a hozzá kapcsolódó technológiákat ismertetem. A feladat során számos multiplatform alkalmazások fejlesztéséhez készült, React Native alapú megoldással, keretrendszerrel és eszközzel találkoztam. Ezek lehetőségeit és gyengeségeit is tárgyalom a későbbiekben. A célom az volt, hogy a fent vázolt technológiák segítségével készítsek egy alkalmazást, mely amellett, hogy jól szemlélteti a közös kódbázis használatának előnyeit, rávilágít a nehézségekre is.

Az általam megvalósított alkalmazás a következő problémára próbál megoldást találni: az olyan környezetekben, ahol sok egymástól független felhasználói csoport van különböző tudásterületek lefedésére, nem mindig egyértelmű, hogy ki tudhatja a választ egy szakértői kérdésre. Az alkalmazás így áll egy backendből, ami kezel egy tudásfát. Ez alapján a feltett kérdéseket képes azokhoz a szakértőkhöz irányítani, akik azt a leghatékonyabban meg tudják válaszolni. A frontend mobilra készül React Native segítségével, mivel a mobil alkalmazás a felhasználók számára praktikusabb. Az így elkészült felhasználói felület minél nagyobb részének újra felhasználásával a React Native Webre építve egy böngészőből is használható webalkalmazást is készítettem. A böngészőből elérhető webalkalmazás desktop környezetben a nagyobb képernyő miatt különböző elrendezésekkel és néhol kiegészítésekkel kényelmes alternatívát ad a praktikus hordozható mobil applikáció mellett.

Az alkalmazás legfontosabb funkciója, hogy a felhasználók kérdéseket tehetnek fel. A React Native, és a hozzá kapcsolódó könyvtárak rengeteg lehetőséget nyújtanak a kérdéseket tartalmazó nézetek és a kérdések közti navigációra. Emellett a kérdések megválaszolására szolgáló valós idejű beszélgetés kialakítására is alkalmas. Az alkalmazás másik lényegi funkciója a tudásterületek kezelése és az ezekből felépített tudásfa megvalósítása. A bevitelhez és listázáshoz használt felhasználói felületek a React Native-os elemek segítségével egy teljesen natív alkalmazás élményét keltik. A legtöbb felület egy az egyben átemelhető webre a React Native Web segítségével a natív alkalmazásból. Néhol viszont a kényelmesebb felhasználói felület érdekében teljesen web specifikus a nagy képernyőre optimalizált felületeket hoztam létre.

A továbbiakban részletesen ismertetem az alkalmazásom megvalósítása során felhasznált technológiákat, valamint bemutatom az alkalmazás megvalósításának részleteit.

# Technológiák feltérképezése

Ebben a fejezetben az alkalmazásomhoz használt technológiákat fogom részletezni. Illetve azokat a React Native és React Native Web-hez kapcsolódó technológiákat, melyeket megvizsgáltam a munkám kutatási fázisában, de végül elvetettem valamilyen okból.

## React

A React Native bemutatásához elengedhetetlen a React [3] megismerése. A React egy a Facebook által fejlesztett JavaScript könyvtár felhasználói interfészek készítéséhez. A React könyvtár 2013-ban vált nyílt forráskódúvá. [4] [5]

A felhasználói felületet React esetén Component [6] nevű elemekből épül fel. A Component-ek mellet szól, hogy így egy jól elkülönülő egységbe zárhatjuk magát a komponens viselkedését leíró logikát és a komponens kinézetéért felelős dizájnt. Minden Component újra felhasználható és egymásba ágyazható. A Component-ek rendelkeznek life cycle függvényekkel. Ezekkel lekezelhetjük például, amikor a UI elemek felkerülnek a DOM-ra (Data Object Model), vagy valamilyen elemük frissül.

Minden Component ’props’ néven vesz be paramétereket amikor létrejön és van egy render függvénye, mely visszatér azzal, hogy hogyan is fog kinézni ez a komponens. A komponenseknek van egy ’state’ változója is, ez tárolja a komponens pillanatnyi állapotát és segít a dinamikus felhasználói felület kialakításában.

A React esetében a React komponens felépítését JSX segítségével írjuk le. A JSX a JavaScript XML-t rövidíti ez egy XML alapú kiegészítés JavaScripthez, ami a leginkább a HTML-hez hasonlít, de kapcsos zárójelek közt bármilyen JavaScript kifejezést beágyazható.

Példa kód:

*//A Welcome egy React.Component*

*//A render függvénye határozza meg hogy hogyan fog kinézni*

class Welcome extends React.Component {

  render() {

*//Itt a JSX ami html szerű, de a kapcsos zárojelek közt*

*//javascriptet írunk, itt a komponens props-ból vesszük ki a név változót*

    return <h1>Hello, {this.props.name}</h1>;

  }

}

*//Elteszem az elementbe a Welcome komponenst*

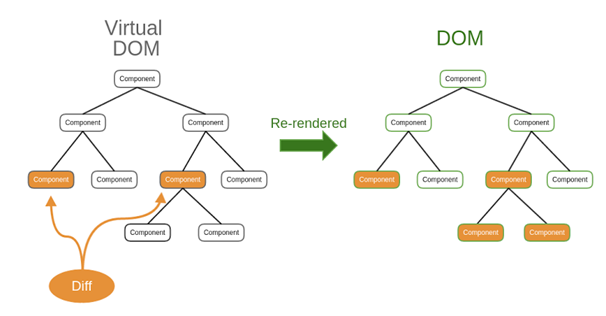
*//A komponens létrejöttekor a name propsba a 'Sara' nevet adom*

const element = <Welcome name="Sara" />;

*//Meghivoma  ReactDOM.render fuggvenyt az elementre*

*//Az oldalra kirajzolódik az element*

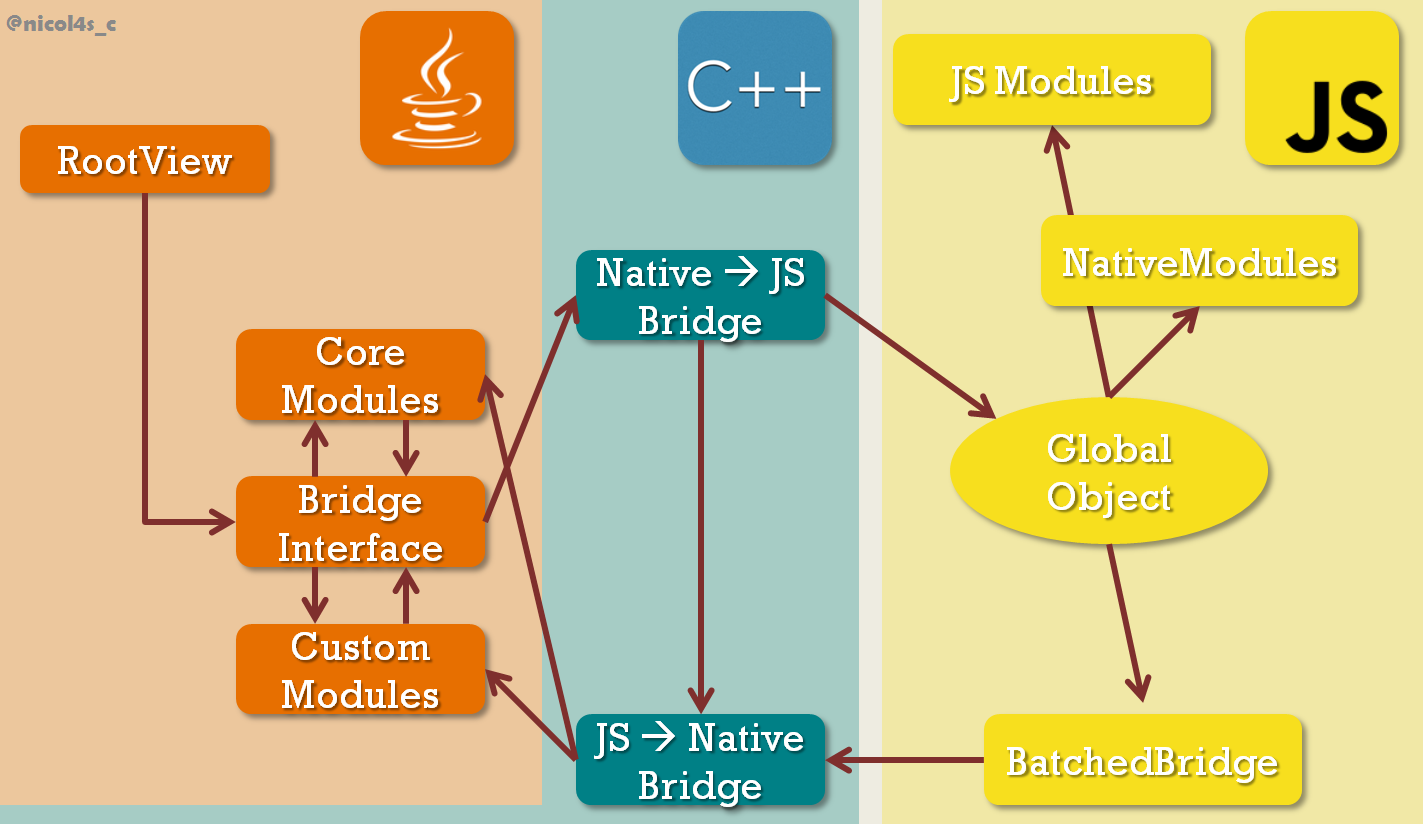
ReactDOM.render(element, document.getElementById("root"));

Ezen felül a Reactnak sok előnyös tulajdonsága van, amit sima HTML-lel nem lehetne megvalósítani. Például virtuális DOM [7] segítségével kezeli a DOM-ot, aminek előnye, hogy a memóriában felépített cache segítségével csak a tényleg szükséges részeit frissíti a DOM-nak.

1. ábra:Illusztráció a virtuális DOM különbség alapú újra renderelésére [8]

Összegezve: a React egy erős eszköz dinamikus weboldalak készítéséhez, csak JavaScript használatával.

## React Native

 A React bemutatása után rátérhetünk magára a React Native-ra ami a Reacthoz hasonlóan épül fel és hasonló koncepciókat foglal magában. A React Native szintén a Facebooktól eredő nyílt forráskódú könyvtár, amivel JavaScriptből lehet natív mobil alkalmazásokat építeni. A fő különbség a Reacthoz képest, hogy nem a DOM-ot manipulálja, hanem egy JavaScript értelmező segítségével az adott eszköz natív vezérlőivel kommunikál (natív module bridge). [9]

2. ábra: Illusztráció a React Native bridge-hez [9]

Ez lényeges különbség például a szintén népszerű Apache Cordovahoz [10] képest, amivel szintén hibrid appokat lehet gyártani mobilra webalkalmazásból, de ott még közbeékelődik egy WebView is. Tehát React Nativenál például a navigáció, gombok Androidon a natív Androidos navigációt és gombokat fogják jelenteni, és ez iOS esetében is ugyanígy működik.

A natív module bridge használatának sok előnye van teljesítmény szempontjából, de lehetőséget ad arra is, hogy könnyen teljesen natív kódot adjunk az alkalmazáshoz. A teljesen natív kód természetesen feláldozza, hogy csak egyszer kell megírni a kódot iOS-re és Androidra is, de még gyorsabbá teszi a működést, és tovább növeli a teljesítményt, mert nem kell aggódni a JavaScript értelmező és a natív komponensek közti kommunikáció miatt.

A felhasználói felület itt is Component nevű elemekből épül fel. A Reactból jól ismert JSX itt is jelen van, de az alap építőelemek itt nem a HTML-ből ismerősek, hanem sajátos React Native elemek. Például Button, View, Text és még sok hasonló komponens.

Példa kód:

import React, { Component } from "react";

import { Text, View } from "react-native";

*//A HelloReactNative egy React komponens*

class HelloReactNative extends Component {

  render() {

    return (

*//Itt a JSX nem HTML elemeket használ hanem ReactNative modulokat*

      <View>

        <Text>If you like React, you'll also like React Native.</Text>

        <Text>

          Instead of 'div' and 'span', you'll use native components like

          'View' and 'Text'.

        </Text>

      </View>

    );

  }

}

export default HelloReactNative;

A komponensek felépítését JSX-ben írjuk le, a kinézetük, stílusuk definiálásához viszont a CSS-hez hasonló szerkezetű Flexbox alapú StyleSheet komponenst használjuk a React Native-ból.

Példa Kód:

import { StyleSheet } from "react-native";

const commonStyles = StyleSheet.create({

*//CSS-hez hasonló szintaxissal soroljuk fel a tulajdonságokat*

*//fontos, hogy a CSS-el szemben itt nincs öröklés és*

*//elemek megkeresése a DOM-ból. A StyleSheet elemet át kell adni*

*//a style propba annál az elemnél, akire használni szeretnénk a stílust.*

*//A stílusok listaként felsorolva össze fűződnek*

  errorTextStyle: {

*//a flexboxból ismert módszerekkel kezeljük a Layoutot*

    alignSelf: "center",

    color: "red",

    marginBottom: 5

  },

  commonText: { color: "#FFFFFF" }

});

*//Használat*

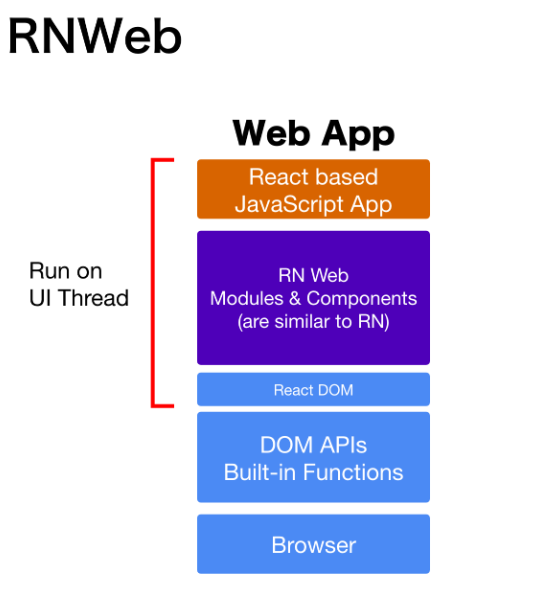
render = () => {

  return <Text style={commonStyles.commonText}>HelloWorld</Text>;

};

Összegezve: a React Native egy React alapokon működő natív alkalmazások fejlesztésére használt keretrendszer.

## React Native Web

A React Native esetében láttuk, hogy a legnagyobb különbség abban van, hogy hogyan épül fel a felhasználói felület, és ezzel hogyan kommunikál a JavaScript. A React Native ettől függetlenül erősen épül a webes Reactra, és a legtöbb elemük megfeleltethető egymásnak. Szeretnénk olyan web alkalmazásokat írni, melyek nem csak natív-an működnek mobilokon de böngészőben is futnak. A React Native Web teszi lehetővé, hogy teljesen React Native kódot írjunk az ott használt vezérlőkkel, komponensekkel, és egy böngészőben futó weboldalt kapjunk. A React Native Web ezt úgy éri el, hogy a React Native komponenseknek megfelelő saját komponenseket definiál, melyek a react DOM-mal kommunikálnak a bridge helyett. [11]

3. ábra: Illusztráció a React Native Web működési modelljéhez [11]

A web specifikus megfelelőit a React Native elemeknek a webpack.config.js-ben kell beállítani. Ez alapján a program fordítási időben, ha webre fordít a React Native Web-et használja, ha pedig mobilra akkor a React Native-ot.

Részlet a konfigurációból:

alias: {

*// Support React Native Web*

  'react-native': 'react-native-web',

  'native-base': 'native-base-web',

  'react/lib/ReactNativePropRegistry':

    'react-native-web/dist/modules/ReactNativePropRegistry'

},

A kódbázis nagyrészét újra felhasználjuk a React Native Web segítségével, de van, hogy platform specifikus kódra van szükségünk. Erre két megoldás létezik, React Native és React Native Web használatakor. Futási időben egy if elágazásban megvizsgálhatjuk, hogy milyen platformon vagyunk es annak megfelelően hajthatunk végre kódot.

Példa kód:

import { Platform } from "react-native";

import env from "../../env";

*//Itt a React Native Platform osztályát használom*

*//amivel letudom kérni a futtató környezet operációsrendszerét*

if (Platform.OS === "android") {

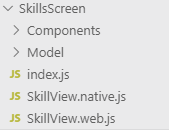
  this.server = env.ServerUrlForAndroid;

  this.props.setTitle("Skills");

} else {

  this.server = env.ServerUrlForWeb;

}

Alternatív megoldás, ha egész fájlokat akarunk csak egyik vagy másik platformon használni megadhatjuk a nevükben, hogy a natív vagy a web alkalmazáshoz tartozzanak (Android, illetve iOS platform specifikus fájlokat is lehet így megjelölni). Fordítási időben dől el, hogy melyik fájlt használjuk. Ezt a módszert akkor érdemes használni, ha például ugyanannak a komponensnek teljesen más kinézetet akarunk adni weben, illetve natív környezetben.

4. ábra: Kép a .native és .web jelölésű azonos nevű fájlokról

Összegezve: a React Native Web egy hatékony eszköz az alkalmazásunk weblappá alakítására.

## ReactXP

A ReactXP egy a Microsoft által fejlesztett keretrendszer. Ez egy absztrakciós szint a React Native felett. A React Native bár képes natív alkalmazást fejleszteni minden platformra, de a View definíciók, stílusok és animációk mind platform függőek, hiszen a natív vezérlőket használja. ReactXP használatával ezek mind egységessé vállnak. Biztosítja ezt az egységes UI felületet React alapokon webre is. Ezen felül a Microsoft tervezi, hogy az UWP-re (Universal Windows Platform) is lehet majd fejleszteni ezen az absztrakciós szinten. A ReactXP fejlesztés TypeScript segítségével történik. A ReactXP majdnem teljesen ugyanúgy működik, mint a React Native. Apró eltérések vannak csak. [12]

Példa kód:

import React from "react";

import RX from "reactxp";

interface AppProps {

  userName?: string;

}

*//Különbség, hogy nem egy React Native-os StyleSheet-et használ*

*//hanem több külön álló ReactXP stílust, így oldja meg,*

*//hogy egységesek legyenek a stílusok minden platformra*

const \_styles = {

  main\_container: RX.Styles.createViewStyle({

    justifyContent: "center",

    alignItems: "center"

  }),

  text: RX.Styles.createTextStyle({

    color: "red",

    fontWeight: "bold"

  })

};

*//A render és minden más ugyan úgy működik, mint React Native esetében,*

*//de a React Native komponensek helyett*

*//a rájuk épülő ReactXP elemeket használjuk*

export class App extends RX.Component<AppProps, void> {

  render() {

    return (

      <RX.View style={\_styles.main\_container}>

        <RX.Text style={\_styles.text}>Hello World</RX.Text>

      </RX.View>

    );

  }

}

A ReactXP, bár nagyon sok mindent leegyszerűsít és nagy segítség a felhasználói felületek egységesítésében, de megvannak a maga hátrányai. Azért, hogy az egységes kinézetet elérje csak a legfontosabb és minden platformon könnyen egységesíthető vezérlőket valósítja csak meg. Amennyiben ezek az általánosan használt komponensek elegek lennének az alkalmazás elkészítéséhez ez egy nagyon hatékony megközelítés.

Az alkalmazás fejlesztésének korai fázisaiban kiderült, hogy ezek miatt a korlátozott funkcionalitások miatt sok más külsős könyvtárral nehezen működik együtt. A kompatibilitási problémák természetesen feloldhatók megfelelő csomagolók elkészítésével, tehát ezzel a technológiával is megoldható lenne a feladat. Azért nem ezt választottam végül, mert a dokumentáció nem teszi egyértelművé mindig mi az, amit meg lehet oldani ReactXP-vel és mi az, aminél vissza kell térni a React Native gyökerekhez. A csomagolók írása miatt pedig inkább bonyolította a feladatot, mint egyszerűsítette. Ezért maradtam az alap React Native mellet, mely jóval flexibilisebb a sok hozzá írt külső komponens miatt.

## Expo és React Native CLI

A React Native alkalmazások build-jének a legegyszerűbb és legteljesebb módja az Expo használata. A hivatalos React Native dokumentációban is az Expo használatát javasolják. Az Expo két workflow-t támogat. Az egyik esetében a cél eszközöket teljesen az ExpoSDK-n keresztül érjük el. Ekkor nem kell egyáltalán a platform specifikus eszközökkel foglalkozni (AndroidStudio és XCode). A másik esetben a build sokkal inkább hasonlít az eredeti React Native CLI-al is elérhető buildre. Megkapjuk a platform specifikus projekteket és teljes kontrollunk van ezek felett, de a buildet ekkor is támogatja és gyorsítja az ExpoSDK. [13] [14]

Az Expo a build gyorsítása mellett, a cél eszközök kezelésében is segít. Egyik nagy előnye például az asset-ek kezelése. Az AndroidStudio-ban is látott módon az eszköz képernyő DPI-nak megfelelő erőforrásokat tölt be például. Kezeli a nyelvi beállításokat, fontokat és minden mást, ami az erőforrásokhoz tartozik.

Bár ez a legteljesebb build tool ami rendelkezésre áll a React Native alkalmazások készítéséhez végül én az eredeti React Native CLI használata mellett döntöttem. Ennek oka, hogy az Expo bár bizonyos szintig képes együtt működni az általam használt React Native Web-bel, de nem támogatja azt és a közeljövőben nem is tervezik támogatni.

Az alkalmazás fejlesztése során a React Native Web és az Expo platform specifikus font kezelése hibákhoz vezetett, ezért az Expo helyett a jóval egyszerűbb React Native CLI használatát választottam. Ez leginkább az Expo nem menedzselt workflowjára hasonlít. Teljes kontrollt enged a platform specifikus projectek felett, de nem rendelkezik az Expo nyújtotta előnyökkel. Ez a legegyszerűbb build rendszer React Native-hoz.

Ezeken felül még használhattam volna a CRNWA (Create React Native Web App) [15] build rendszert, de ennek feltérképezésére már nem jutott idő.

## Native Base és egyéb GUI könyvtárak

A felhasználói felület építésére már a React Native is sok eszközt nyújt, de pár komplikáltabb vezérlőhöz és egy egységes téma eléréséhez külső könyvtárakat használtam. Találni egy olyan komponens könyvtárat, mely teljes mértékben működik weben és natív alkalmazásokban is, viszont nem könnyű.

A feladatom során több ilyen könyvtárat is kipróbáltam. A react-native-material-ui [16] a webes világból jól ismert Material Theme-et adja hozzá a natív alkalmazásunkhoz. A React Native Web-bel viszont nem működött együtt. A React Native Web komponensei bár nagyon hasonlítanak a React Native komponensekhez ezért többnyire egy az egyben megfeleltethetők egymásnak, de a prop-okat nem teljesen egyformán kezelik és emiatt ez a komponens könyvtár nem működött.

A második próbálkozás a React Native Elements [17] volt. Ezt elvileg össze lehetne kombinálni a React Native Webbel. Ehhez még egy tutorial is van. Ehhez viszont a CRNWA build eszközt kellene használnom, melynek megértésére nem jutott idő és emiatt nem emellett döntöttem végül. A megfelelő build rendszer nélkül sajnos nem sikerült működésre bírni.

Harmadik próbálkozásként a Nachos UI-al [18] próbálkoztam, mely külön a React Native Webhez készült, ez működött weben és natívan is. Az volt az egyetlen gond, hogy a natív alkalmazásban nagyon specifikus stílus szabályok nélkül teljesen szétesett a felület főleg, amikor felület változás történt. Ez idővel nagyon kényelmetlenné vált tehát ezt is elvetettem.

A képen képernyőkép látható

Automatikusan generált leírásA képen monitor, képernyőkép, képernyő, mobiltelefon látható

Automatikusan generált leírásA negyedik és egyben utolsó próbálkozásom a Native Base [19] komponens könyvtárral volt. Ennek a könyvtárnak van egy külön webes forkja a Native Base Web [20]. Ez teljesen együtt működik a React Native Web-bel. A webpack.config.js-ben ezt is mappelni kell a React Native – React Native Web mappeléshez hasonlóan. Ez a komponens könyvtár egységes gomb stílusokat, Card elemeket, interaktív listákat és sok hasznos vezérlőt tartalmaz.

5. ábra: A képen a kérdéseket tartalmazó Card komponensek listája látható mobilon és weben

Ennek a komponens könyvtárnak az egyetlen komolyabb hátránya, hogy nem minden natívon működő komponens része a webes forknak ezért az egységes kinézet kialakítása néha nehezebb.

Végül emellett a könyvtár mellet döntöttem mivel jól együtt működött az alkalmazásom többi részével és az összetettebb komponensek sokat segítettek a felhasználói felület fejlesztésében.

## Entity Framework és ASP.NET Core

Mivel egy olyan rendszerről beszélünk, ami több felhasználót támogat és az ő interakciójukra épít, szükség volt egy backendre. Az ASP.NET Core nyílt forráskódú webes keretrendszert használtam fel, ami cross platform backendek készítésére képes. Ez egy moduláris keretrendszer, ami a backend minden funkcionalitását lefedi. Például a többfelhasználós rendszerhez elengedhetetlen azonosítást az ASP.NET Identity segítségével valósítottam meg. [21]

Az adatok kezelését a Microsoft ORM (Object-relational mapping) keretrendszerével az Entity Framework segítségével oldottam meg. A keretrendszer által biztosított Code First [22] megközelítést használtam.

## Axios

Az alkalmazás nagy része HTTP segítségével kommunikál a backend és a frontend között. Backend oldalról ezt az ASP.NET Core-val készített kontrollerek tökéletesen kiszolgálják. Kliens oldalról viszont az Axios promise alapú HTTP kliens könyvtárat használom. Ezzel a könyvtárral könnyen kezelhető http kéréseket lehet összeállítani. A kérések egy promis-sal térnek vissza, ami lekezelhető mikor megérkezik a http válasz. [23]

Példa kód:

import axios from "axios";

deleteSkill = id => {

*//Beállítom a JWT tokent a kéréshez*

  const headers = {

    Authorization: "Bearer " + this.props.jwt

  };

  this.View.current.setState({

    error: ""

  });

*//Felépítek egy HTTP delete kérést a megfelelő headerrel*

  axios({

    method: "DELETE",

    url: "http://" + this.server + `:5000/api/skills/${id}`,

    headers: headers

  })

*//Feloldom a Promiset ha visszatér*

    .then(response => {

      console.log(response);

    })

*//Hiba esetén lekezelem és lekezelem a hibát*

    .catch(error => {

      console.log(error);

      this.View.current.setState({

        error: error.response.data.error,

        loading: false

      });

    });

};

A JavaScript-ben alapértelmezett fetch-el szemben számos előnye van az Axios-nak. Az Axios a küldött és válaszként kapott adatokat automatikusan alakítja a kérésnek megfelelő formátumra, valamint vissza JSON-be (JavaScript Object Notation). Axios-ban könnyen állítható időkorlát a kérésekre, aminek meghaladása után az Axios eldobja a kérést. Emellett képes szimultán kérések indítására is. [24]

Összegezve: az Axios egy jól kezelhető és kényelmes könyvtár HTTP kérések indítására és HTTP válaszok kezelésére és mindenképp előre lépés az egyszerű fetch-vel szemben.

## SignalR

A HTTP API mellett a valós idejű kommunikáció is folyik a szerverrel. Például a feltett kérdésekre adott válaszok egy valós idejű chatben jelennek meg a felhasználóknál. Ezt a SignalR segítségével oldom meg. A SignalR egy a Microsoft által fejlesztett ASP.NET keretrendszerre épülő szerveroldali megoldás, mellyel hatékony kétoldalú aszinkron kommunikáció valósítható meg. [25]

Alapvetően Websocket segítségével valósít meg kétirányú kommunikációt a szerver és a kliens között. Amennyiben a Websocket nem érhető el a SignalR egyéb alternatív kommunikációs rétegen keresztül továbbít üzeneteket. Ezt viszont mind elfedi a függvénykönyvtár és megvalósításának köszönhetően skálázható környezetben is elhelyezhető. [26]

C# példa kód:

using Microsoft.AspNetCore.SignalR;

*//A MessageHub egy SignalR Hub leszármazott*

public class MessagesHub: Hub

{

*//Amikor üzenetet küldök ezt a metódust hívom a kliens oldalról*

    public async Task SendMessage(string message)

    {

*//Az össze kliensen elsütöm a 'ReciveMessage' eseményt*

*//és visszaadom nekik a messaget*

        await Clients.All.SendAsync("ReceiveMessage",message);

    }

}

A kliens oldalon az @aspnet/signalr könyvtár segítségével kezelem le a két irányú kapcsolatot.

JavaScript példa kód:

import \* as signalR from "@aspnet/signalr";

*//Megadom a kapcsolat paramétereit a szerverrel*

var connection = new signalR.HubConnectionBuilder()

  .withUrl("http://" + this.server + ":5000/api/thread")

  .build();

*//Ha elsüti a szerver a ReceiveMessage eseményt kiírom az üzenetet*

connection.on("ReceiveMessage", message => {

  console.log(message);

});

*//Felépítek egy kapcsolatot a szerverrel*

connection.start();

*//Definiálok egy függvényt arra, hogy hogyan küldök üzenetet*

sendMessage = message => {

*//Elsütöm a szerver SendMessage eseményét*

  this.connection.invoke("SendMessage", message).catch(function(err) {

    return console.error(err.toString());

  });

};

*//Elküldöm a szervernek, hogy 'Hello World'*

senndMessage("Hello World");

# Alkalmazás követelmények

Egy nagyvállalatnál vagy olyan környezetben, ahol sok ember és ezzel együtt sok szakterület, ember specifikus képesség található, gyakran készítenek kompetencia mátrixokat, hogy felmérjék milyen lehetőségekkel (kompetenciákkal) rendelkeznek. A kompetencia mátrix felhasználásával, illetve azzal a feltételezéssel, hogy a különböző területek közt kapcsolatok állnak fenn, egy tudás gráfot építhetünk. Az általam készített alkalmazás arra épít, hogy amikor kérdések merülnek fel egy ilyen környezetben nem mindig egyértelmű, hogy kit kell elérni vagy kit érintenek a kérdések. Viszont egy tudás gráfban keresve viszonylag egyszerű elvek alapján is hatékonyan lehet a kérdéshez leginkább értőket megtalálni.

Az alkalmazás tehát egy több felhasználós rendszer, ami egy tudás gráfot felhasználva próbál szakértőket találni egy kérdéshez a többi felhasználó közül, illetve lehetőséget nyújt a kérdés megvitatására chat formában. Az alkalmazás különlegessége, hogy van egy Androidos natív kliense és egy webes kliens is (bár ez nem rendelkezik minden funkcionalitással).

## Felhasználási esetek

Az alkalmazásnak két elkülönülő állapota van. A felhasználó bejelentkezése előtt és az után.

### Belépés előtt

6. ábra: A képen az alkalmazás belépés elötti Usecase diagrammja látható

A felhasználó ebben az állapotban két dolgot tehet vagy bejelentkezik a már létre hozott fiókjába, vagy regisztrál egy újat fiókot az alkalmazásba.

**Regisztráció:** A felhasználónak meg kell adnia egy szabályos email címet, egy felhasználó nevet, amivel később az alkalmazás hivatkozni fog rá, és egy jelszót. Egy email címhez csak egy fiók tartozhat és a felhasználó neveknek is egyedinek kell lenniük.

**Belépés:** A felhasználó be tud lépni egy már létező fiókkal, az ahhoz tartozó egyedi email címet, és jelszót megadva. Bejelentkezés után a felhasználót az alkalmazás bejelentkezve tartja akkor is, ha bezárjuk egészen addig amíg a felhasználó ki nem lép.

### Belépés után

A képen szöveg látható

Automatikusan generált leírásA felhasználó miután bejelentkezett az alkalmazásba képes a különböző beszélgetési szálakat menedzselni, a különböző készségeket kezelni és a saját profilját módosítani.

7. ábra: A képen az készségek kezelésének Usecase diagrammja látható

**Készségek kezelése:** A készségek alatt valamilyen tudást kell érteni, ezekből építenek a felhasználók tudás gráfot melyet az alkalmazás később fel fog használni. A készségeket a *Mobilos felhasználók* tudják csak teljes mértékben kezelni mivel a webes kliens számára a felhasználói felület bizonyos részei készültek csak el. A *Mobilos felhasználók* egy listában látják a tudásfák gyökereit, ezeket képesek *kibontani*, így elérik azok gyerekeit.

A készségeket mobilról *fel lehet venni* a felhasználónkhoz, illetve *le lehet adni* őket. A felhasználóhoz felvett készségek alapján fogja megtalálni őket később az alkalmazás a rájuk vonatkozó kérdésekkel. Mobilos felületről ezen felül lehet *törölni* készségeket a listából, ekkor a készségek végleg törlődnek, a fából is és az őket ismerő felhasználóktól is.

A képen szöveg, térkép látható

Automatikusan generált leírásEzen felül lehet új készségeket felvenni viszont ez a *webes és mobilos felhasználók* számára is egyaránt elérhető. Új készség felvételénél megadhatók a készség szülei. Ez azt jelenti, hogy az új készség a szüleinek egy specifikusabb része esetleg épít rájuk valamilyen módon. Innentől kezdve a tudás gráfban lesz egy kapcsolat a szülők és az új készség között.

8. ábra: A képen a beszélgetési szállak kezelésének Usecase diagrammja látható

**Beszélgetési szálak kezelése:** Az alkalmazás legfontosabb funkciója az adott készségekre vonatkozó kérdések feltétele és ezek megvitatása. Az alkalmazásban szálnak nevezünk egy feltett kérdést és a rá érkező válaszüzenetek összességét. Minden felhasználó számára a legfontosabb a *személyre szabott nézet* melyben a képességeik alapján hozzájuk szóló kérdésekkel induló szálak szerepelnek. Ide azok a szálak kerülnek be melyek vagy pontosan olyan a készséggel vannak megjelölve, amivel a felhasználó is rendelkezik vagy elég magas a prioritásuk és a rajtuk lévő készségek elég közel helyezkednek el a tudás gráfban a felhasználó valamelyik készségéhez.

Azokat a szálakat, amiket a felhasználó hozott létre ki is *törölheti*, amikor úgy érzi, hogy választ kapott a kérdésére. A szálak megnyithatók egy *chatszerű nézetben*. Itt valós időben zajlik beszélgetés a kérdésről, ami elindította a szálat. A felhasználó *tehet fel* bármikor új kérdést. Az új kérdéseknél be kell állítani a készségeket, amiket a felhasználó szerint érint a kérdése. Ezen felül be kell állítani egy prioritást a kérdéshez, ami minél magasabb annál több emberhez jut el a kérdés. Az alkalmazás magasabb prioritás esetén nem csak a beállított készségek birtokosait, de a tudás gráfban közel A képen szöveg látható

Automatikusan generált leíráselhelyezkedő készségek birtokosait is értesíti.

. ábra: A képen a profil kezelésének és a kilépésnek a Usecase diagrammja látható

**Profil kezelése:** A felhasználó képes a profilja alatt a személyes adatait kezelni. Az *email címét* képes átírni természetesen csak egy másik szabályos és eddig még másik fiók által nem foglalt címre. A *felhasználó nevet* is át tudja írni, illetve a *jelszavát* is megváltoztathatja.

**Kilépés:** A felhasználó kiléphet ezzel az alkalmazás vissza kerül a belépés előtti állapotba.

# Architektúra

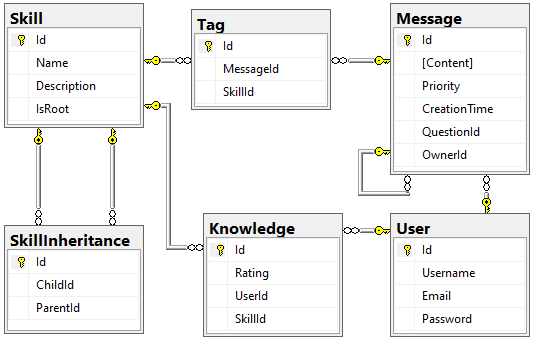
A képen képernyőkép látható

Automatikusan generált leírásA szakdolgozatban bemutatott szoftver áll egy adatbázisból, egy backendből, és egy frontendből. Az alkalmazás egy több felhasználós rendszer a felhasználók adatait, a feltett kérdéseket és a tudás gráf alapjául szolgáló készségeket mind egy közös adatbázisban tárolja. A felhasználók azonosítását, egymással, és a rendszerrel való interakciójukat a backend végzi. A frontend pedig egy közös kódbázisra épülő Natív Android kliens alkalmazás és egy Web alapú kliens alkalmazás. Ez a két kliens működés szempontjából egymástól teljesen független.

10. ábra: A képen a szoftverem absztrakt architechturális rajza látható

## Adatbázis

Az adatbázis elkészítésénél a Microsoft Entity Framework objektum-relációs leképzés részeként bemutatott Code-First megközelítést alkalmaztam. A Code-First lényege, hogy nem egy előre megtervezett adatbázishoz illesztünk C# osztályokat. A C# osztályok készülnek el először és ezekhez automatikusan legenerálja az adatbázis sémát.



11. ábra: A képen az alkalmazás adatbázisának entitás-kapcsolat diagrammja látható

A Code-First megközelítésnek hála az adatbázist könnyen le lehet cserélni az alkalmazás alatt. Én egy Microsoft SQL szervert használtam adatbázisként.

### Adatbázis Táblák

Az alkalmazásnak három fontos résztvevőből áll: a felhasználók, az üzenetek és a készségek. Ezeket és a köztük fenn álló kapcsolatokat tároljuk az adatbázisban.

**Felhasználók:** Az alkalmazás felhasználói a *User* táblában vannak eltárolva. A felhasználók rendelkeznek egy felhasználónévvel és emailcímmel, melyek egyediek minden egyes felhasználó esetén. A jelszavaikat hash-elve tároljuk.

**Üzenetek:** Az alkalmazás egyik fő funkciója az üzenetek váltása. Ezek az üzenetek a *Message* táblában vannak tárolva. A tartalmukat, prioritásukat és a létrehozási idejüket tartalmazza az adatbázis. Az üzenetek kérdések vagy kérdésekre adott válaszok lehetnek. A kérdések és a hozzájuk rendelt válaszok közt egy a többhöz kapcsolat van. Az üzenetek több az egyhez kapcsolattal kapcsolódnak az őket létrehozó felhasználókhoz.

**Készségek:** Az alkalmazás egy tudás gráfra épül és ennek alkotó elemei a készségek. Ezek a *Skill* táblában vannak tárolva. Mindegyik rendelkezik egy névvel, leírással és egy flag-vel, ami azt hivatott jelezni, hogy gyökérelemről beszélünk-e. A készségek lehetnek gyerek vagy szülő készségek. A gyerekek és a szülők közt több a többhöz kapcsolat van. Ezt a kapcsolatot a *SkillInheritance* tábla tárolj. A felhasználók által feltett kérdések megjelölhetők készségekkel. Ez egy több a többhöz kapcsolat az üzenetek és a készségek között, amelyet a *Tag* tábla tárol. A készségeket a felhasználók magukhoz rendelhetik. Ez egy több a többhöz kapcsolat, amit a *Knowledge* tábla tárol.

## Backend

A képen képernyőkép látható

Automatikusan generált leírásA backend kezeli a felhasználókat és a köztük és a rendszer között lezajló interakciókat. A Microsoft ASP.NET Core webes keretrendszerével készült a szerveroldali kód. A szerver elkészítésekor törekedtem a DDD (Domain-Driven design) [27] követésére. Ennek következtében a kódot három külön álló részre bontottam. Ez a három réteg az adatelérési, bizniszlogikai, és a kliensekkel való kommunikációért felelős API réteget jelenti. A DDD elveinek megfelelően az alkalmazás többi rétege mind a bizniszlogikától függ. [28] [29]

12. ábra: A képen a backend Package diagrammja látható

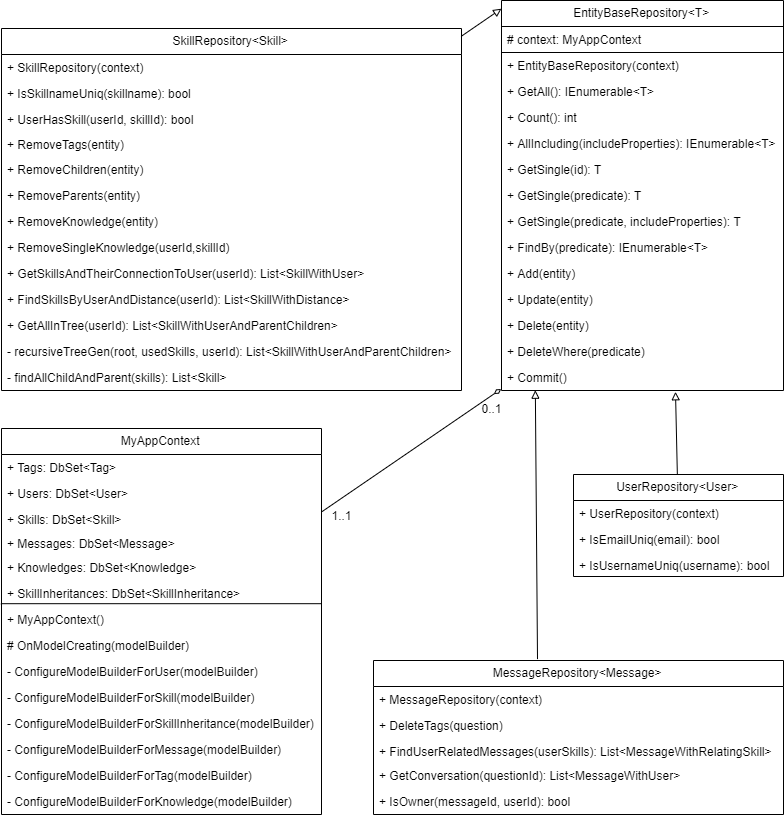
### Modellek

A képen szöveg, térkép látható

Automatikusan generált leírásA DDD filozófiájával egyedül a Model csomag megy szembe, amely a backend összes többi része által használt modelleket tartalmaz. Ezek nem csak domain modellek, az Entity Framework ezekre a modellekre képezi le az adatbázis megfelelő tábláit. A Code-First megközelítés során ezeket a modelleket kellet elkészíteni és ezek alapján generálódik le az adatbázis. Az összes modellben közös, hogy egy egyedi GUID segítségével azonosítható a backenden. Ezt a közös tulajdonságot egy közös interfész implementálásával érem el.

13. ábra: A képen a Model csomagban található osztályok osztály diagrammja látható

### Adatelérési réteg

Az adatelérési réteg megvalósításához a repository mintát [30] követtem. A biznisz objektumokhoz (Message, Skill, User) készítettem saját repository-kat. Ezek egy közös általános repository-ból származnak le. A közös repository az adatbázis entitásokra jellemző törlés, módosítás és listázás műveletek általánosított formáit tartalmazza. A biznisz objektumok egyedi adatelérési műveletei, amelyek általában két entitás közti kapcsolati táblára is építenek a biznisz objektumokra specifikus repository-kban találhatóak. Magát az adatbázis a MyAppContect osztály segítségével érem el. A kontextusra mutató változót a repository-k közösősében vettem fel és a specifikus repository-k öröklésen keresztül érik el.

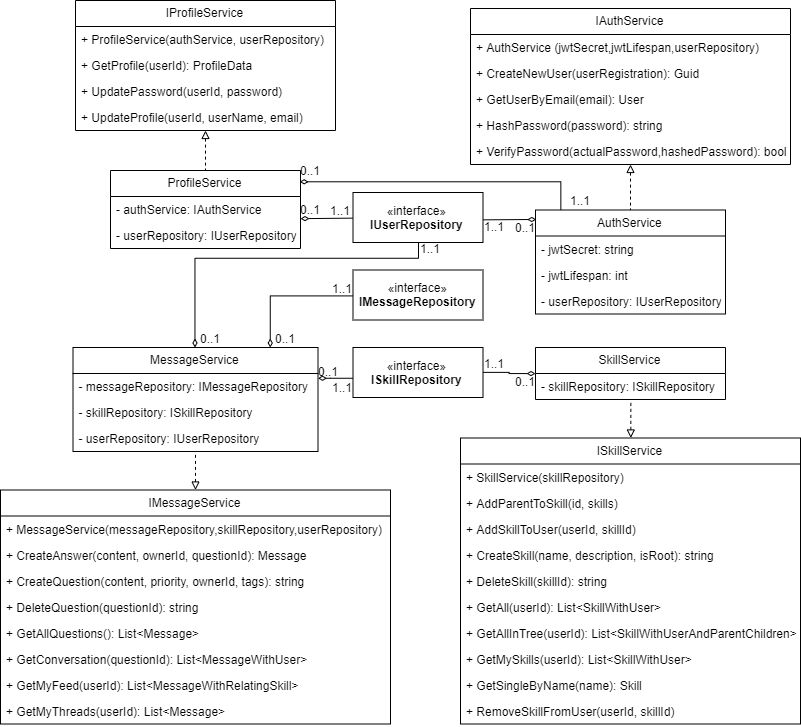
. ábra: A képen a Data csomag osztály diagrammja látható

### Biznisz logika

Az alkalmazás lényege a biznisz logika ettől a rétegtől függ az összes többi réteg. Itt találhatók a szolgáltatások, amelyek a repository-kból kinyert adatokon végeznek műveleteket, vagy több repository-ból szerzett adatokból készítenek egy biznisz objektumot. Ez a viselkedés feltételezné, hogy a biznisz logikai réteg függ az adatelérési rétegtől. A függés viszont megkerülhető a függőség megfordításának elvét [31] felhasználva.

Ezt a tervezési mintát követve létrehoztam a biznisz logikai rétegen belül egy RepositoryInterfaces nevű csomagot. Ettől a csomagtól függenek a szolgáltatások, így a rétegnek nem lesz kifelé mutató függősége. A csomagban definiált interfészeket implementálják a repository-k ennek köszönhetően az adatelérési réteg függ a biznisz logikától és nem fordítva.

A szolgáltatások a működésükhöz szükséges repository-kat illetve egyéb szolgáltatásokat függőség befecskendezésen [31] keresztül kapják. Ennek a tervezési mintának hála a szolgáltatás által felhasznált függőségek könnyen módosíthatók a szolgáltatás megváltoztatása nélkül.

A szolgáltatásokon és a repository interfészeken felül a biznisz logika részét képezik a különféle biznisz objektumok. Ezeket a szolgáltatások hozzák létre vagy már ilyen formában nyerik ki a repository-kból. A különálló Model csomagban megismert biznisz objektumok mellet az itt definiált formákban utazik az adat a szerveren.

15. ábra: A képen a BLL csomag részleges osztály diagrammja látható

### API réteg

Az API (Application Programable Interface) réteg a backend utolsó rétege. Ez a szerver legkülső rétege, itt találhatók a kontrollerek és a SignalR Hubok. Az API réteg elemei a szükséges szolgáltatásokat függőség befecskendezésen keresztül érik el. Ez a réteg a biznisz logikától függ és minden kérést szolgáltatások segítségével kezel le, az adat elérési rétegtől teljesen független.

A kontrollerek a Controllers csomagban találhatók. Az összes az ASP.NET Core ControllerBase ősosztályából származik le. Mindegyik rendelkezik az ApiController attribútummal ez jelzi, hogy HTTP kérések kiszolgálására hivatottak. A rendszer több felhasználós, a különböző felhasználók azonosítására és kezelésére az ASP.NET Core Identity keretrendszerét használtam. A bejelentkezésért és felhasználói regisztrációért felelős kontrolleren kívül minden kontroller rendelkezik az Authorize attribútummal. Ennek az attribútumnak hála a többi kontroller csak bejelentkezett felhasználókat szolgál ki.

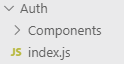
Az alkalmazásban a kommunikáció nem csak http protokoll felett zajlik. A Hubs csomagban található SignalR hub-ok az ASP.NET Core SignalR könyvtárának segítségével két irányú valós idejű kommunikációt folytatnak a klienssel.

Az API rétegben található a DTO (Data Transfer Object) csomag. Ebben a csomagban vannak definiálva azok az objektumok, amelyek a szerver felé irányuló kérésekben vagy a szerver által adott válaszokban szerepelnek. Az ilyen szállító objektumok és a biznisz objektumok közti átalakítás az AutoMapper könyvtár segítségével történik.

## Kliensek

Az alkalmazás kliens oldala két klienst jelent egy Web alkalmazást és egy Android natív alkalmazást. A két alkalmazás kódbázisa bár nagy rész közös szükség volt platform specifikus megoldások alkalmazására. A Web alkalmazás belépési pontja egy index.js fájl és rendelkezi egy index.html fájlal is. Az index.js a kiinduló index.html-re ki render-eli a kliens oldal közös belépési pontjaként szolgáló App.js már React Native-ban megírt tartalmát. Az Android alkalmazás belépési pontja az App.js. A klienskód a különböző megjelenítendő képernyők alapján van tagolva.

### Auth

Az Auth képernyő két egymást váltó ablakból áll egy bejelentkező ablakból és egy regisztrációs ablakból. A fő képernyő alképernyői egy a csomagban található Components csomag alatt helyezkednek el. Az Auth képernyő minden eleme közös a Webes és az Android kliens között.

16. ábra: A képen az Auth képernyőhöz tartozó csomag látható

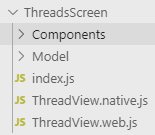
### HomeScreen

A HomeScreen routerrel ellátott képernyő. Ez a router tartalmazza az összes bejelentkezés után elérhető képernyőt és teszi lehetővé a köztük való navigációt. A router implementációja különbözik a Web és az Android kliensek között. Webnél a react-router-dom könyvtár oldja meg a navigációt. A natív esetben a React Native saját react-router-native navigációs könyvtárat használ az alkalmazás.

A felhasználó egy oldalsó menü sáv segítségével tud navigálni. Ez a HomeScreen része és a Components csomag alatt található.

17. ábra: A képen az HomeScreen képernyőhöz tartozó csomag látható

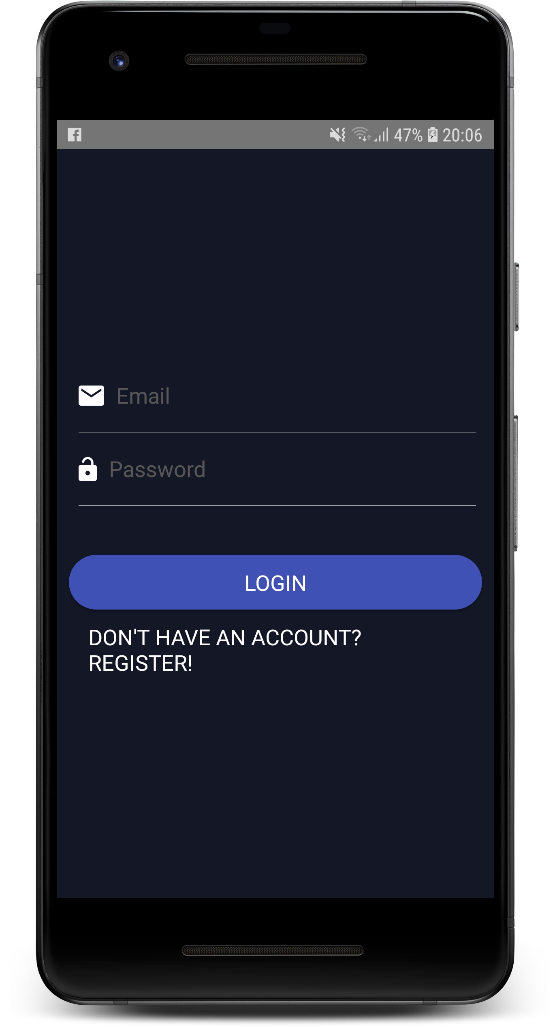
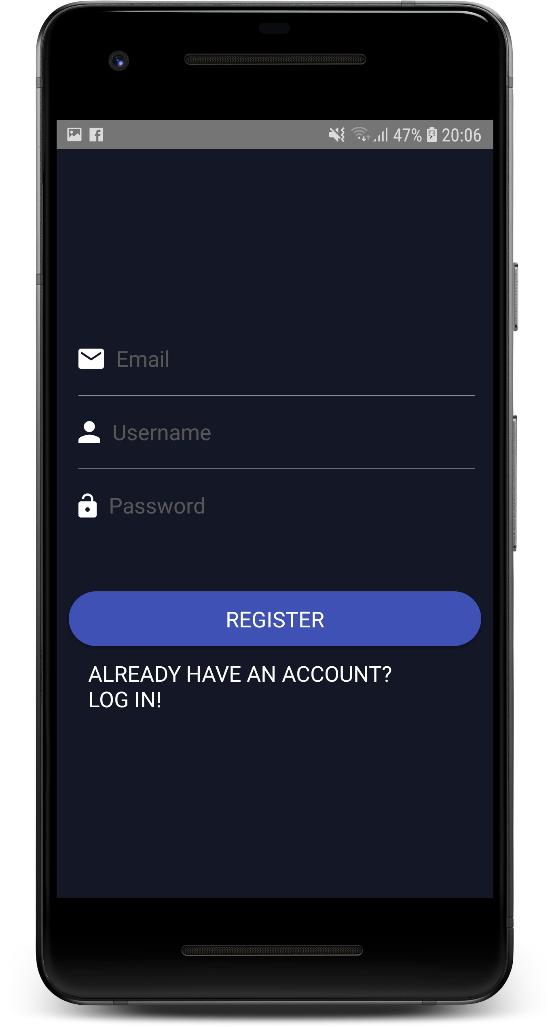
### Egyéb bejelentkezett képernyők

A kliens oldali alkalmazás többi bejelentkezett képernyője mind ugyanaz alapján a minta alapján épül fel. Minden csomagban található egy index.js ez tartalmazza a képernyő kliens oldali logikáját. Itt történik a szerverrel való kommunikáció és az adatok átalakítása a kliensnek, illetve szervernek megfelelő alakokra. Az index.js mellet van egy a képernyő kinézetéért felelős nézet (View). Ezek a nézetek lehetnek platform specifikusak, mivel a képernyők megjelenítése nem egységes a Web és az Android kliensek között. A bonyolultabb komponensek további rész komponenseket tartalmazhatnak ezek a Components csomag alatt találhatók. A komponens specifikus enumok pedig a Model csomagban kapnak helyet.

. ábra: A képen az ThreadScreen képernyőhöz tartozó csomag látható példaképp az egyéb bejelentkezett képernyőkre

# Alkalmazás megvalósítása

## Bejelentkezés és regisztráció

A felhasználó Androidon és Weben is először egy bejelentkező képernyővel találkozik. Itt be tud jelentkezni, ha van már fiókja. Amennyiben nincs még fiókja regisztrálhat egy újat.

19. ábra: A képen a belépő (bal) és regisztrációs (jobb) felületek láthatóak

A felhasználó a fenti két nézet között a komponensek alján található linkkel válthat. A beviteli mezőket mindkét képernyő esetén egy KeyboardAvoidingView-val vettem körbe. Androidnál ennek hála a billentyűzet nem takarja ki őket. Ez a komponens nincs hatással a Web kliensre. A kliens mind regisztrációnál, mind belépésnél egy HTTP hívást kezdeményez a szerver felé az Axios segítségével.

### Regisztráció

Regisztráció esetén a kliens a felhasználó nevet, jelszót és az email címet egy POST kérés formájában elküldi a szervernek. Szerver oldalon ezt az AuthController fogadja.

Példa kód a modell ellenőrzéséről DTO-val:

*//Ezt a modellt várja a szerver és az attribútumok alapján ellenőrzi is.*

public class RegisterViewModel

{

    [Required]

    [StringLength(60, MinimumLength = 2)]

    public string Username { get; set; }

    [Required]

    [EmailAddress]

    public string Email { get; set; }

    [Required]

    [StringLength(60, MinimumLength = 8)]

    public string Password { get; set; }

}

A modell validációja után a kontroller áthív az AuthService szolgáltatásba és megpróbál létrehozni egy felhasználót a megadott adatokkal. A beérkezett adatokat az AutoMapper segítségével UserRegistration biznisz objektummá alakítja.

Az AuthService CreateNewUser függvénye hozza létre az új felhasználót. A felhasználó elkészítése során ellenőrzi, hogy van-e már felhasználó a megadott névvel vagy email címmel. Már létező felhasználó esetén hibát dob, amelyet a kontroller kezel le. Ellenőrzés után létrehoz egy új felhasználót és elmenti a UserRepository segítségével az adatbázisba. A függvény a létrehozott felhasználó egyedi azonosítójával tér vissza.

A kontroller elkapja a felhasználó készítése során keletkező hibákat és hibát jelez vissza a kliens felé. Sikeres felhasználó generálás esetén az AuthService segítségével az új felhasználó azonosítója alapján készít egy JWT (JSON Web Token) tokent és visszaküldi a kliensnek. Innentől kezdve az alkalmazás ezt fogja használni a felhasználó azonosításához.

Kliens oldalon az Axios segítségével elkapja és lekezeli az alkalmazás a szervertől érkező hibákat. A sikeres felhasználó létrehozás után a deviceStorage szolgáltatást használva kimenti a kliens a JWT tokent. A deviceStorage az AsyncStorage segítségével, kulcs érték párként menti le az adatokat lokálisan. Ha nem jelentkezett ki a felhasználó, az alkalmazás ismételt indításakor az elmentett JWT token alapján bejelentkezve nyílik meg a kliens.

### Bejelentkezés

A bejelentkezés esetén egy POST kérés formájában a felhasználónevet és a jelszót küldjük fel a szerverre. A szerver ezeket a regisztrációhoz hasonlóan ellenőrzi. Az AuthService segítségével lekéri a felhasználót az email címe alapján. Ha nem létezik felhasználó ezzel az email címmel akkor hibával tér vissza a szerver. Létező felhasználó esetén ellenőrzi a jelszóhelyességét. A jelszavakat szerver oldalon hash-elve tárolom az adatbázisban. Az AuthService VerifyPassword függvénye a Crypto VerifyHashedPassword függvényével ellenőrzi a jelszót. Helytelen jelszó esetén hibát jelez a kliens számára. Helyes jelszónál a bejelentkező felhasználó egyedi azonosítója alapján generál egy JWT tokent.

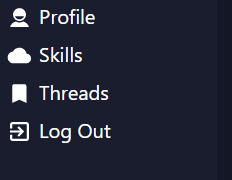
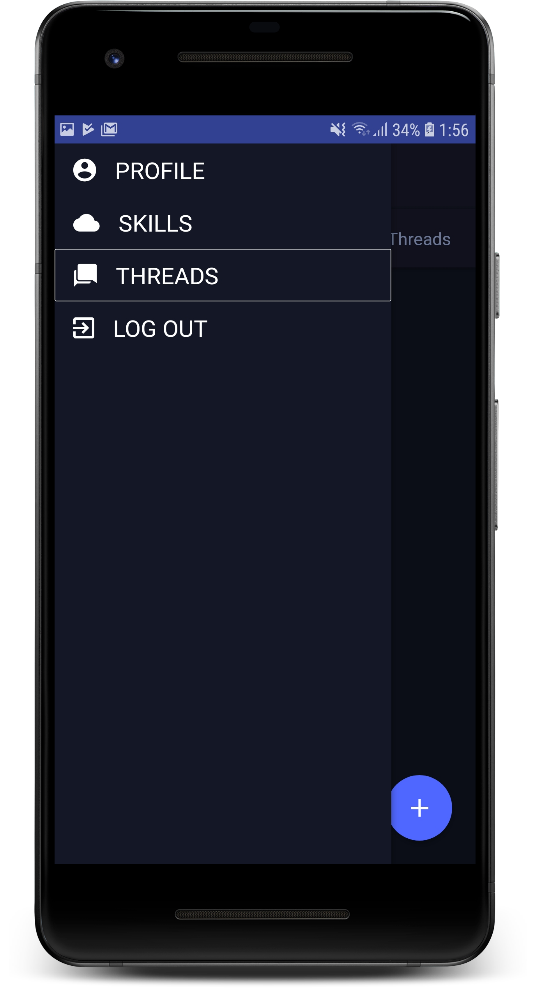
A kliens helyes bejelentkezés esetén a regisztrációhoz hasonlóan elmenti a JWT tokent és amíg nem jelentkezik ki a felhasználó addig azt használva bejelentkezve tartja.

## Bejelentkezett felhasználó és navigáció

20. ábra: A képen a bejelentkezett alkalmazás látható Androidon

A kliens indításakor Android és Web esetén is megpróbálja betölteni a tárolt JWT tokent. A lokális tárolást az Android a SharedPreferences segítségével valósítja meg. Ez egy olyan API Androidon, amely lehetővé teszi, hogy egy fájlban kulcs-érték párok formájában tároljunk adatokat. A SharedPreferences fájlokat az Android keretrendszer kezeli és garantálja, hogy csak a mi alkalmazásunk írhassa és olvashassa. Web esetében a felhasználó által használt böngésző Local Storage-ának segítségével tároljuk el a JWT tokent.

Sikeres token betöltés után az alkalmazás a kezdő képernyőn indít. A kliensben ezek után egy navigációs sáv segítségével tud a felhasználó navigálni. A navigációt egy Router komponens teszi lehetővé. A navigáláshoz használt könyvtárak bár különböznek a két kliens között a Router komponens felépítése nagyon hasonló mindkét kliensben.

A két kliens fő komponensében az alapvető különbség az oldalsó navigációs sáv viselkedése és az Androidon meglévő felső fejléc.

21. ábra: A képen a navigációs oldalsáv látható Androidon (bal) és Weben (jobb)

### Android

Androidon a navigációs sáv megvalósításához a Native Base komponens könyvtárból a Drawer komponenst használtam. A Native Base ezen komponense az Androidos Navigation Drawer komponensre épít. Ezt a menüt a bal felső sarokban található ikon segítségével lehet elérni. Az animációi és megnyitási, bezárási viselkedése mind teljesen megegyezik a Navigation Drawer eredeti viselkedésével. A bal oldalról előhúzás szintén alapértelmezett viselkedés volt, de ezt lekellet tiltani mert összeférhetetlen volt egyéb komponensekkel, akik ugyanezt a gesztust használják.

A fejléc egy gombbal a bal felső sarokban szintén Android specifikus felhasználói felületi elem. A fejléc tartalma a navigáció során minden új képernyővel változik. A változások kezelésére a fő komponensben két callback függvényt definiáltam. A Router komponensen keresztül props-ként átadhatók ezek a callback függények a többi képernyőnek. Mert az összes bejelentkezés után elérhető képernyő a fő komponens alatt helyezkedik el, mint gyerek elem a DOM-ban (ez igaz a Web kliensre is).

Példa kód a callback-ek átadására:

*// Egy elem a routerből*

<Route

  path="/skills"

  render={props => (

*// A kirajzolandó képernyő*

    <SkillsScreen

      {...props}

*// A JWT token prop-ként van átadva*

*// az alkalmazás minden komponensének, aki használja*

      jwt={this.props.jwt}

*// A callback függvények átadása props-ként*

      setTitle={this.setTitle}

      setRunOnClick={this.setRunOnClick}

    />

  )}

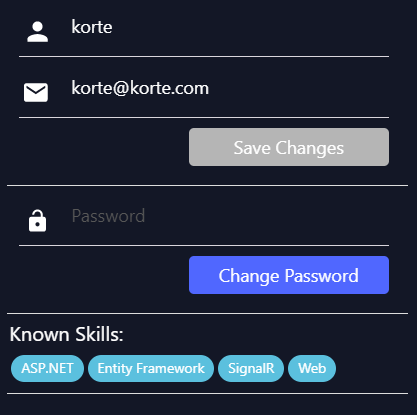
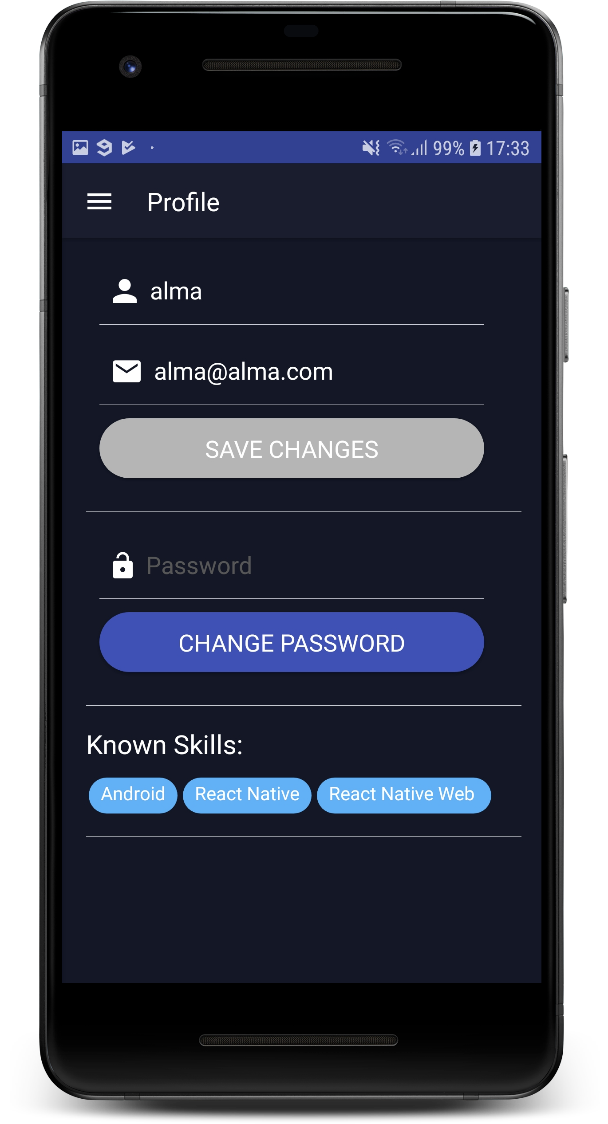
/>;

A setTitle függvény azt állítja be, hogy mi legyen a fejlécre írva képernyő címként. Ezt a függvényt minden képernyő meghívja, amikor létre jön. A setRunOnClick függvény pedig azt állítja be, hogy milyen függvény fusson amikor a bal felső sarokban lévő ikont megnyomják. Alapértelmezetten a navigációs menüt nyitja meg, de bizonyos esetekben egy vissza navigáló nyíllá alakul az ikon ilyenkor a futtatandó függvényt is lecseréli az adott képernyő.

### Web

A web alkalmazás nem rendelkezik fejléccel és az oldalsó navigációs sáv mindig láható. Az oldalsó sáv elrendezése kissé eltérő. Az Android specifikus ikonok helyett a program olyanokat használ, amelyek a Weben is megjeleníthetők.

## Profil képernyő

A kliens képernyői közül, amelyek a navigációs sávból elérhetőek az első a Profile képernyő. Ezen a képernyőn a bejelentkezett felhasználó a profiljának az adatait módosíthatja, illetve kap egy áttekintést az általa felvett készségekről. A megjelenített adatokat a kliens a szerverről nyeri tehát szerver kapcsolat hiányában minden Profil mező üres marad. Ez a képernyő két form-ból áll. Ezek külön tölthetők ki és tartalmuk külön változtatható a profilban.

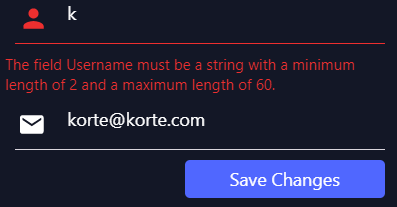
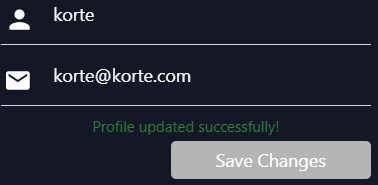
22. ábra: A képen a Profil képernyő látható Androidon (bal) és Weben (jobb)

### Felhasználónév és email cím

A felhasználónév és email cím módosítása együtt történik. A Save Changes gomb megnyomásával lehet felküldeni a módosított felhasználónevet és email címet a szerverre. Ez a gomb alapvetően ki van kapcsolva, amíg a felhasználó nem módosítja vagy az email, vagy a felhasználónév mezőt. Ezzel a kliens jelzi, hogy tényleges változtatások történtek és ezek elküldhetők a szervernek.

A kliens egy PATCH kérés formájában küldi el a felhasználónevet és az email címet a szervernek. A kérést a ProfileController fogadja. Ez egy Authorize attribútummal rendelkező kontroller ezért csak bejelentkezés után elérhető. Először a regisztrációhoz hasonlóan ellenőrzi a modellt a DTO attribútumai alapján. Helytelen modell esetén hibát küld vissza a kliensnek. Helyes modellnél lekéri a bejelentkezett felhasználó egyedi azonosítóját az Identity keretrendszer segítségével. Megpróbálja a ProfileService segítségével frissíteni a felhasználói profilt.

A ProfileService UpdateProfile függvénye a UserRepository segítségével lekéri az aktuális felhasználót az egyedi azonosítója alapján. Ellenőrzi az új email cím és felhasználónév egyediségét, ha már létezik felhasználó valamelyikkel akkor hibát dob. Egyedi értékek esetén frissíti a felhasználót és elmenti a változtatásokat az adatbázisba a UserRepository segítségével.

Sikeres frissítés után a szerver egy üres válasszal tér vissza 204-es sikeres státuszkóddal. A frissítés során történő hibákat a szerver továbbítja a kliens felé.

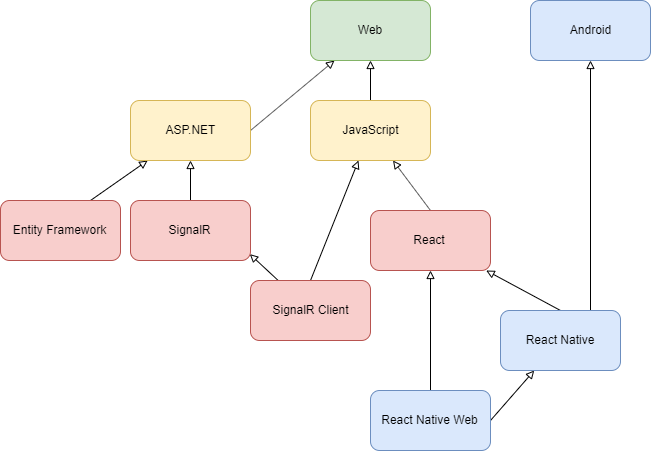
23. ábra: A képen a felhasználónév és email cím sikeres (bal) és sikertelen (jobb) frissítése látható

### Jelszó megváltoztatása

A jelszó megváltoztatása nagyon hasonlóan működik a profil többi adatának módosításához. A lényegi különbség, hogy a profil betöltésekor biztonsági okokból a szerver nem tölti be a jelenleg használt jelszót. Szöveges formában nem is rendelkezik a jelszóval csak a hash-elt jelszót tárolja a szerver. Ezen felül kliens oldalon sima beviteli mező helyett egy biztonságos beviteli mező található, amely elfedi a bevitt karaktereket.

A jelszó frissítése ezen túl ugyan úgy zajlik. Egy PATCH kérést intéz a kliens a szerver felé, aki ellenőrzi a jelszó követelményeit és elmenti a változásokat, ha minden megfelelt.

## Skillek kezelése

Az alkalmazás egyik fő funkciója a készségek kezelése. A készségek azokat a szakterületeket jelentik, amiket a felhasználók ismerhetnek (Knowledge) illetve amikkel az üzenetek meg lehetnek jelölve (Tag). Az alkalmazás a készségeken keresztül köti össze a szakértőt a szakértelemre vonatkozó kérdéssel. A készségek között is léteznek kapcsolatok (SkillInheritance). Ezen kapcsolatok alapján felépíthető egy tudás gráf. A tudás gráf egy irányított aciklikus gráf. A továbbiakban a **24. ábrán** látható tudás gráf segítségével fogom bemutatni az alkalmazást.

24. ábra: A képen egy általam felhasznált példa tudás gráf látható a kapcsolatok mindig a leszármazottból mutatnak a tágabb készség felé

Az alkalmazás bár egységesen kezeli a készségeket szerver oldalon, de a Webes és Android alapú kliensek között itt tettenérhető a legnagyobb különbség. Mivel a fejlesztés során a Mobil First tervezést követtem a nézetek és dizájnok mind Androidra készültek el elsőnek. A Webes kliensen a készségeket megjelenítő nézete nem készült el és az Androidos nézet egy az egybeni átemelése nem passzol a Web kliensbe. Ezek következtében az alkalmazás Weben csak csökkentett funkcionalitással rendelkezik a készségek kezelésének tekintetében.

### Új Skill felvétele

25. ábra: A képen a készség létrehozó képernyő látható

Az új készségek felvétele implementálva van Androidon és Weben is. Weben ez az egyetlen készség kezelési lehetőség, ezért amikor a Skills képernyőre navigálunk ide irányít át az alkalmazás. Androidon ez a képernyő a Skills nézetről érhető el egy lebegő akció gomb megnyomásával. Az új készség felvételére szolgáló nézet egy form-ot tartalmaz, ez a komponens azonos mind két kliensben. Meg lehet adni a készség nevét, leírását és egy legördülő listából hozzá lehet adni a hozzá tartozó szülő készségeket. A legördülő lista tartalmát az alkalmazás az összes addig már létrehozott készség lekérésével állítja elő. Mivel csak a szülő készségek állíthatók be a form segítségével a tudás fát fentről lefelé kell építsük. A form alján található egy Cancel és egy Add gomb. A Cancle visszalép a Skills képernyőre. Web esetén ez azt jelenti, hogy a form újratöltődik.

Az új készség alap esetben egy gyökér készség mivel nem rendelkezik szülő készségekkel ez az állapot akkor változik meg, ha hozzá adunk egy szülő készséget. Az Add gomb megnyomásakor a kliens egy POST kérést intéz a szerver felé. Felküldi az új készség nevét, leírását és hogy gyökér készség e.

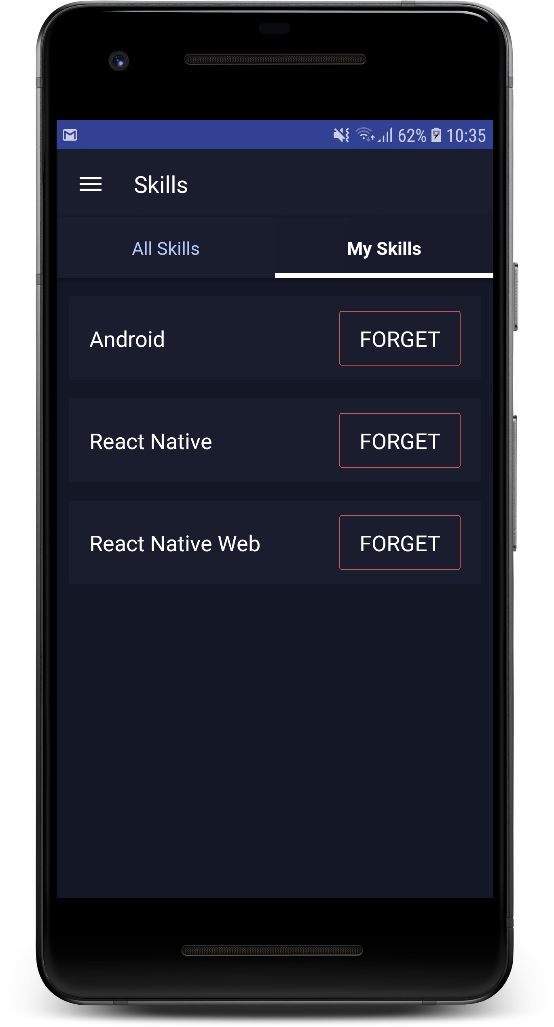
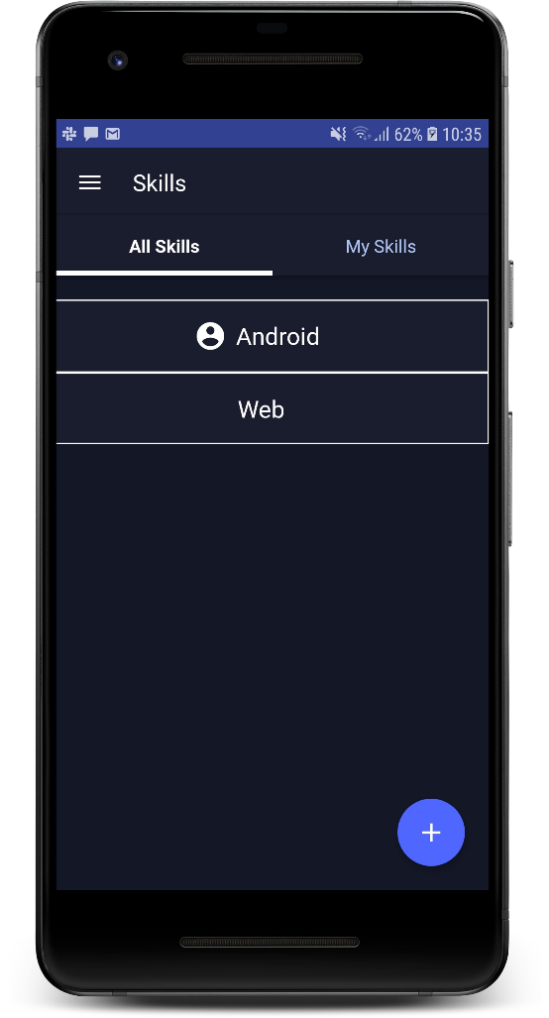
A szerveren a SkillsController kezeli le a kérést, ez egy Authorize attribútummal ellátott kontroller. A kontroller először ellenőrzi a modell helyességét, majd megpróbál létrehozni egy új készséget a SkillService segítségével. A SkillService CreateSkill függvénye ellenőrzi a készség nevének egyediségét a SkillRepository segítségével. Minden készségnek egyedi neve kell legyen, hogy ne váljon zavarossá a tudás gráf. A felküldött adatok alapján létrejön az egyedi készség és a SkillRepository segítségével bekerül az adatbázisba is. A készség egyedi azonosítójával tér vissza a függvény sikeres készség generálás esetén. Sikertelen készség létre hozás esetén a hibát továbbítja a kliens felé a kontroller. Siker esetén pedig visszaküldi az újonnan létrejött készség azonosítóját.

A kliens elkapja az új készség azonosítóját, ha az sikeresen legenerálódott. Az azonosítóval indít egy új PATCH hívást, amely az új készséghez hozzá rendeli a szülőket, amelyeket a form-ban hozzá adtunk.

A szerver SkillsControllere szolgálja ki ezt a hívást is. A SkillService segítségével veszi fel a szülőket az új készséghez. Új SkillInheritance objektumokat hoz létre minden szülő készséghez és módosítja ezekkel az frissen létrejött készséget az adatbázisban. A sikeres szülő felvétel után a szerver 204-es státuszkóddal tér vissza.

A kliens alkalmazás egy sikeres készség létrehozás után átnavigál a Skills képernyőre.

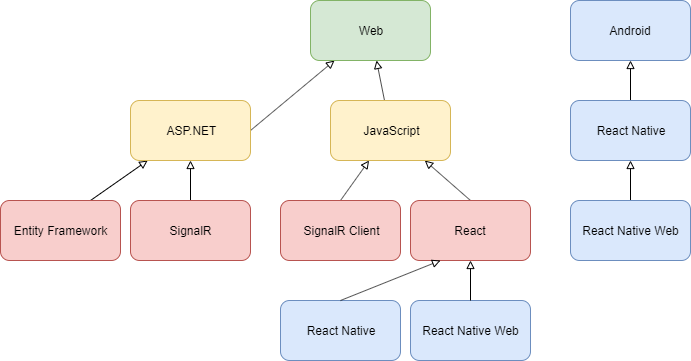
### Skills képernyő

A Skills képernyő további funkciói csak Androidon érhetők el. Ez a képernyő Tab Based Navigation segítségével jeleníti meg a tartalmát. Két fül között váltogatva lehet elérni a készség kezelés két fő nézetét. A két nézet két teljesen független komponens a kliens kódjában és a navigáció aktuális állapotát egy enum segítségével követi az alkalmazás. A fülek között a fülekre nyomva lehet váltani ilyenkor az adott nézet szerver oldalról betölti a listát, amelyet az adott nézet megjelenít.

26. ábra: A képen a két fül látható

Mind két fül egy lista nézetet tartalmaz, amelyeken készségek vannak felsorolva. A nézetek görgethetők és az Androidról jól ismert módon, ha a lista tetején a felhasználó lehúz akkor a lista tartalma be frissül a szerverről.

### All Skills nézet lista felépítése

A készségek kezelésének ez az oldal a fő nézete. Ez a nézet jelenik meg először, mikor Androidon a Skills képernyőre navigálunk. A képernyő jobb alsó sarkában található egy lebegő akció gomb ennek megnyomásával új készséget vehet fel a felhasználó. A listában a felhasználó eredetileg csak a gyökér készségeket látja. A lista elemek megnyomásával egy új lista töltődik be a lista elem gyerek elemeivel feltöltve. A bal felső sarokban található ikon átalakul egy visszalépő nyíllá, amivel a szülő elem szintjére lehet visszatérni. Ezt a kibomló lista nézetet úgy éri el az alkalmazás, hogy még a szerveren fákká alakítja a tudás gráfot és az elemek meg érintésével ezekben a fákban navigál a kliens.

27. ábra: A képen az a két fa látható, amelyet a szerver készít a 24. ábrán látható tudásfából

A **27. ábrán** látható, hogy egy fában egy elem csak egyszer van jelen. A két egymástól független fa viszont tartalmazhat olyan elemet, amely a másikban is előfordul.

Mikor az All Skill fülre navigálunk vagy újra töltjük az adatait a szerverről a kliens egy GET kérést intéz a szerver felé. A szerver a SkillsController segítségével kapja el ezt a kérést. Lekéri az aktuálisan bejelentkezett felhasználó egyedi azonosítóját. A felhasználói azonosítót a SkillService GetAllInTree függvényének adja tovább. A SkillService a SkillRepository segítségével lekéri az összes készséget fák formájába rendezve. Azért a repository rendezi fákká a készségeket mert ehhez szükség van a SkillInharitance kapcsolatokra is. A SkillRepository kikeresi az adatbázisból a gyökér készségeket. A Skill táblának van egy oszlopa, amely azt tárolja, hogy az adott elem gyökérelem e. A gyökérelemekből SkillWithUserAndParentChildren típusú biznisz objektumokat gyárt a szerver. Az ilyen objektumok gyakorlatilag a később épülő fa gráfok egy-egy csomópontját reprezentálják. A felhasználói azonosító alapján azt is eltárolják ezek az objektumok, hogy a felhasználó ismeri e az adott készséget. Ez az információ később fontos lesz a UI szempontjából. Ezek után minden gyökérelemből fát generál egy rekurzív algoritmussal a SkillRepository.

A recursiveTreeGen privát függvény egy gyökérelemet, az eddig már felhasznált készségek listáját és a felhasználói azonosítót veszi be paraméterként. A gyökérelem alapján kilistázza annak azokat a gyerek elemeit, amelyek nincsenek a már felhasznált készségek között. A gyerek elemekből SkillWithUserAndParentChildren objektumokat készít a felhasználói azonosító segítségével és beállítja szülőként a gyökérelemet. A gyerek elemeket felveszi a már használt készségek közé. Ennek köszönhetően egy fában minden készség maximum egyszer fordulhat elő. Ezután az összes gyerek elemmel meghívja a rekurzív függvényt és az eredményt az adott gyerek elem gyerekeiként veszi fel. A függvény visszatérési értéke a gyerek elemek listája.

Példa kód:

private List<SkillWithUserAndParentChildren> recursiveTreeGen(

    SkillWithUserAndParentChildren root,

    List<SkillWithUserAndParentChildren> usedSkills,

    Guid userId )

{

    List<SkillWithUserAndParentChildren> children = context.SkillInheritances

                            .Where(sI => sI.ParentId == root.Id)

                            .Where(sI => !usedSkills.Any(uS => sI.ChildId == uS.Id))

                            .Select(sI => new SkillWithUserAndParentChildren

                            {

                                Id = sI.Child.Id,

                                IsRoot = sI.Child.IsRoot,

                                Description = sI.Child.Description,

                                Name = sI.Child.Name,

                                UserKnows = sI.Child.SkillUsers

                                                    .Any(k => k.UserId == userId),

                                Parent = root

                            }).ToList();

    if (children.Count > 0)

    {

        usedSkills.AddRange(children);

    }

    foreach (var child in children)

    {

        child.Children = new List<SkillWithUserAndParentChildren>(

                                                    recursiveTreeGen(child,

                                                                     usedSkills,

                                                                     userId ));

    }

    return children;

}

Miután az összes gyökérelem elkészült a SkillController az AutoMapper segítségével leképezi a fák csomó pontjait SkillViewModel adatátviteli objektumokra. A SkillViewModel rendelkezik a JsonObject(IsReference = true) attribútummal. Ennek hála minden készséget csak egyszer küld le a szerver és ha több helyen hivatkozik rájuk (pl.: Childre, Parent tulajdonságok) azt referenciával oldja meg.

A kliens a szervertől visszakapott fákat kimenti a state-be és csinál egy ideiglenes listát skillsShown néven ebben csak a megjelenítendő készségek vannak tárolva. Ezek először a gyökérelemek, de ez a lista frissül minden készség gyerekeinek kibontásakor is.

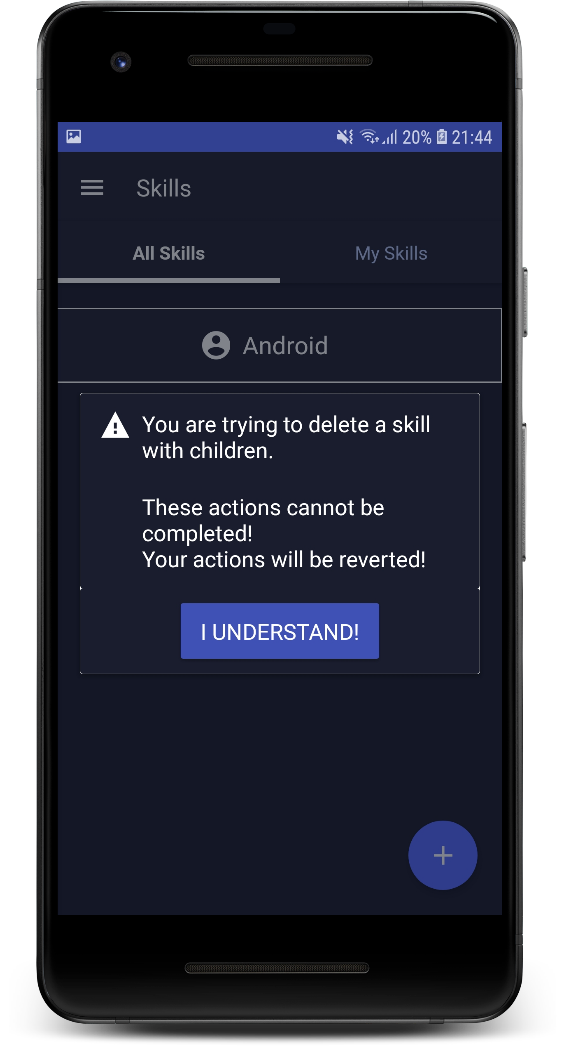
### Készségek kezelése az All Skill nézeten

A készségek kezeléséhez a Native Base Swipeable Row komponensét használja a kliens. Az alkalmazás kirajzol egy felső listát a megjelenítendő adatokkal és alatta egy rejtett listát nyomógombokkal. A felső lista elemei elhúzhatók és felfedik az alattuk lévő opciókat. Egyszerre egy elem lehet elhúzva a listában.

Minden lista elemnek két opciója van. A bal oldalt rejtett az a készség megtanulása vagy, ha már meg van tanulva akkor az elfelejtése. A jobb oldali az a készség törlése.

28. ábra: A képen a készség megtanulása, a készség elfelejtése és a készség törlése látható (balról jobbra)

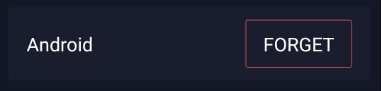
A készségek megtanulása és elfelejtése is egy-egy PATCH formájában történik. A szerver felvesz egy Knowledge kapcsolatot a felhasználó és a készség közé megtanulásnál, elfelejtésnél pedig törli ezt a Knowledge elemet az adatbázisból. A kliensen azok a készségek, amelyeket ismer a felhasználó egy profil ikonnal vannak jelölve az egyértelműség kedvéért.

A készségek törlése esetén a kliens egy DELETE kérést intéz a szerver felé. A szerver ellenőrzi, hogy a készségnek vannak e gyerek elemei. A tudás gráfból csak levél elemet lehet törölni, mert olyan elem törlése esetén, amely rendelkezik gyerekekkel fennáll a gráf szét hullásának veszélye. Levél elem törlése esetén a szerver törli az összes készséghez tartozó kapcsolatot (Tag, Knowledge, SkillInheritance), majd a készséget is törli. Nem levél elem törlésénél dob egy hibát a szerver, amelyet egy felugró figyelmeztetéssel kezel le a kliens.

29. ábra: A képen a készség törléskor fel ugró figyelmeztetés látható

### My Skills nézet

A My Skills nézet egy egyszerűbb készség nézet, ahol egy listába vannak rendezve a felhasználó által ismert készségek. Ez a nézet megkönnyíti a felhasználó által ismert készségek leadását. A készség kártyákon szereplő Forget gomb ugyan azt az API végpontot hívja, mint a komplexebb All Skills nézetben a leadás.



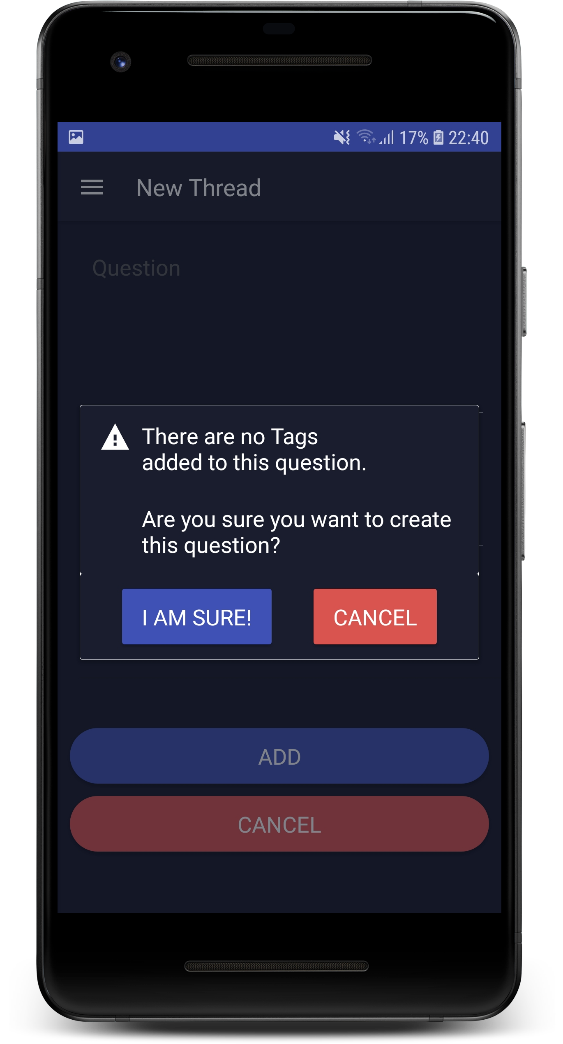
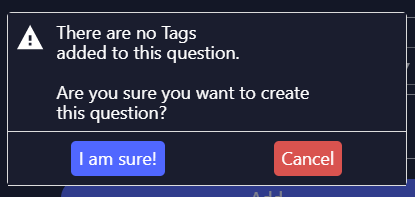
30. ábra: A képen egy készség kártya látható a My Skills nézetről

## Beszélgetési szálak

Az alkalmazás legfontosabb funkciója a szakértőt igénylő kérdések eljuttatása a megfelelő szakértőhöz és a kérdés megvitatása a szakértővel. Az alkalmazásban tehát kezelünk **üzeneteket**. Az üzenetek egyik speciális formája a **kérdések.** Ezek olyan üzenetek, amelyek rendelkezhetnek szakterület címkékkel (Tag) és rendelkeznek egy prioritással. A prioritás fogja eldönteni, hogy mekkora részét nézi át a szerver a tudás gráfnak, hogy találjon egy szakértőt a kérdéshez. A kérdésekhez tartozhatnak válaszok ezek egyszerű üzenetek a legfontosabb őket definiáló tulajdonság, hogy melyik kérdéshez tartoznak. Egy kérdés és a hozzá rendelt üzenetek sokasága együtt a **beszélgetési szál.**

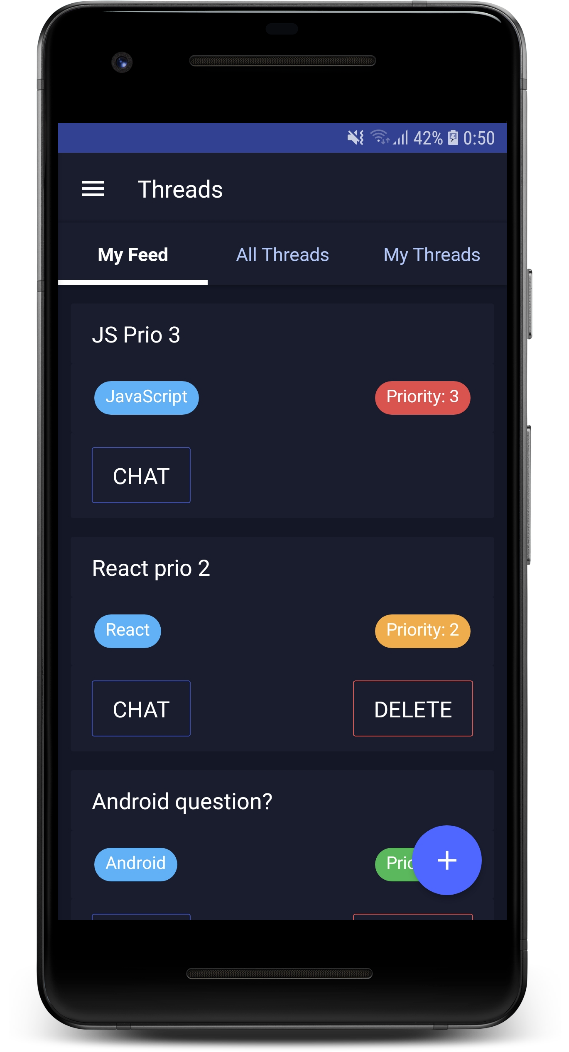
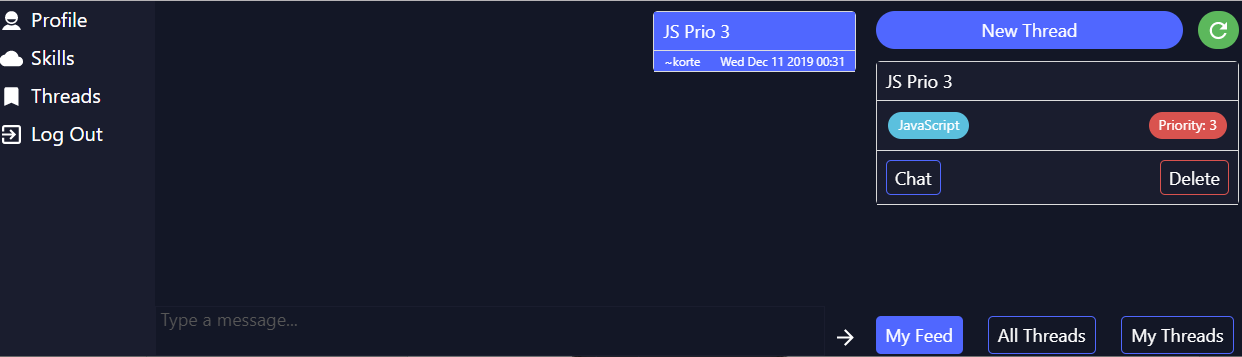
### Kérdés feltétele

31. ábra: A képen a kérdés létrehozó ablak látható Androidon (bal) és Weben (jobb)

A felhasználó tehet fel kérdéseket mindkét kliensben. A kérdések feltételéhez ugyan azt a komponenst használja az alkalmazás Weben és Androidon is. A kérdés feltétel a készség létrehozáshoz hasonlóan egy form. A kérdés szövegén felül a kérdés prioritása és a kérdéshez kapcsolódó szakterületek állíthatók. Lehet menteni kérdést szakértői területek nélkül is. Ilyenkor a kliens figyelmeztet, hogy nincsenek szakértői területek beállítva és emiatt nem fog senkihez közvetlenül eljutni a kérdésünk. Az ilyen kérdéseket is meg lehet találni az összes kérdést kilistázó All Threads alatt.

32. ábra: A képen a felugró ablak látható, amely figyelmeztet szakterület nélküli kérdés létrehozásakor Androidon (bal) és Weben (jobb)

### Threads képernyő

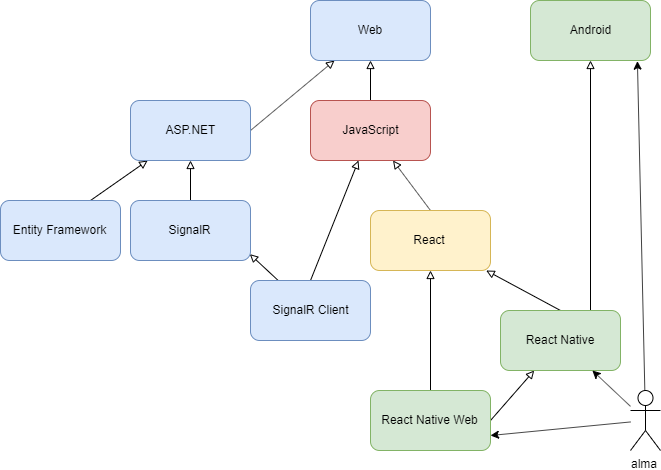
A Threads képernyő a kezdő képernyő az alkalmazásban. Ez a képernyő Androidon a Skills képernyőhöz hasonlóan egy Tab Based Navigation komponensre épül. Weben a szélesebb képernyő miatt a szállak listáit tartalmazó navigációs nézet ki került jobb oldalra. A Webes kliensen a képernyő középső részén az épp megnyitott szál chat ablakja kapott helyet. A chat ablak Androidon egy külön álló képernyő. A navigációs nézet Androidon és Weben hasonló, de Androidon ez egy beépített komponens míg Weben teljesen meg kellet valósítani.

. ábra: A képen a Threads Képernyő látható Androidon (felül) és Weben (alul)

### My Feed nézet

A három nézet közül, amely ezen a képernyőn megjelenik a My Feed a legfontosabb. Ez a lista a felhasználó készségei alapján a felhasználó számára releváns szállakat tartalmazza. Ez a nézet egy GET kérést indít a szerver felé, amelyet a MessagesController szolgál ki. A kontroller lekéri a bejelentkezett felhasználó egyedi azonosítóját. A MessageService GetMyFeed függvénye az azonosító alapján a SkillRepository segítségével lekéri a tudás gráfból a felhasználó közvetlen készségeit és az összes többit, amely egy bizonyos távolságra van.

Az üzenetek prioritása minél nagyobb annál több felhasználóhoz jut el a kérdés. Ezt az alkalmazás úgy éri el, hogy nem csak azokat értesíti ki, akik a kérdésben megjelölt szakterületekkel rendelkeznek. Magasabb prioritásnál azokat is kiértesíti, akik egy vagy akár két gráf távolságra lévő készségekkel rendelkeznek. A gráfban a távolság oda-vissza ugyan úgy értelmezhető. Ezért nem építünk fel minden üzenetnél egy a prioritásának megfelelő részgráfot a tudás gráfból. Helyette a felhasználótól indulva épít fel az alkalmazás egy kettő gráf távolság mély részgráfot a felhasználó készségei alapján. Ez kevés kérdésnél felesleges overhead-del jár, de sok kérdés esetén sokkal hatékonyabb ezzel összehasonlítani minden kérdés szakértelmeit.

Példa:

34. ábra: A képen az alma által elért tudás tér látható

A **34. ábrán** az látható, hogy az alma felhasználó milyen készségekkel megjelölt kérdéseket kaphat. A **zöld** azokat a készségeket jelöli, amelyeket az alma felhasználó birtokolja. Ezekkel a szakterületekkel megjelölt kérdések prioritástól függetlenül mindig megjelennek az alma My Feed nézetében. A **sárga** az alma készségeitől egy gráf távolságra lévő készségeket jelöli. Az alma felhasználó akkor látja az ilyen szakterülettel megjelölt kérdést, ha az legalább kettes prioritással rendelkezik. A hármas prioritású kérdéseket látja az alma, ha már akár egy **piros** szakterület is van hozzájuk rendelve.

A fentiek alapján a SkillRepository FindSkillsByUserAndDistance által visszaadott felhasználó specifikus részgráf segítségével a MessageRepository FindUserRelatedMessages függvénye megtalálja a felhasználó számára releváns üzeneteket.

A kliens a My Feed nézeten felsorolt szálakat kártyák formájában jeleníti meg. A felhasználó a saját szálait törölheti. Minden szálnál megnyithatja a chatet. A kártyákon látszanak a szakterületek, amelyeken keresztül a felhasználót megtalálta a rendszer és a szál prioritása is látszik.

### All Threads és My Threads nézetek

Az All Threads és a My Threads nézetek teljesen ugyan úgy működnek. Az All Threads az összes létező kérdést kilistázza még azokat is, amelyek egyáltalán nem rendelkeznek kapcsolódó szakterülettel. A My Threads pedig a felhasználó által létrehozott szálakat listázza ki. Az összes szál nézetre általánosan igaz, hogy készítési idő szerint mindig a legfrissebb szálak vannak legfelül.

### Chat nézet

35. ábra: A képen a Web kliens chat nézete látható

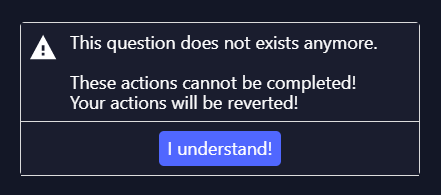
Minden szál kártyán található egy chat gomb. A chat gomb megnyomása Web kliensnél betölti középre a kiválasztott chatet. Androidon egy új nézet nyílik meg a bal felső sarokban visszanyíllal. Ez az új nézet a GiftedChat segítségével kerül megjelenítésre. A Web klienshez teljesen új chat nézetet kellet létrehozni.

A chat betöltése először SignalR segítségével felcsatlakozik a MessagesHubra, majd csatlakozik a szál azonosítója alapján elnevezett csoporthoz (amennyiben még nem létezik létre jön). Ezek után egy GET kéréssel lekéri a szálhoz tartozó üzeneteket időrendi sorrendbe rendezve. A sorba tett üzeneteket megjeleníti a kliens a chat nézetben és innentől kezdve, ha új üzenet érkezik azt valós időben kapja meg a felhasználó.

36. ábra: A képen a Web kliens chat nézete látható

### Törlés és hiba kezelés

A felhasználó bármikor törölheti az általa létre hozott szálakat. A törlés egy DELETE kéréssel történik, amely kitörli a kérdést a hozzá tartozó válaszokat és az összes kérdéshez kapcsolódó Tag-et.

A többi felhasználó nem értesül a törlésről egyből. Egy másik felhasználó megpróbálhat üzenetet írni már törölt szálra vagy megnyithat már törölt szálat. Ilyenkor a szerver hibát dob a kliensek pedig egy felugró figyelmeztetéssel jelzik a felhasználónak, hogy amit csinált többé már nem lehetséges. A felugró ablak ugyan úgy viselkedik, mint a nem levél készség törlésénél megjelenő.

37. ábra: A képen a Web kliensben felugró figyelmeztető ablak látható

# Értékelés

A dolgozat elkészítése során rengeteg új technológiával ismerkedtem meg. Sok tervezői döntést kellet hoznom, amelyek a későbbiekben kifizetődtek, vagy éppen hátráltatták az alkalmazás fejlesztését.

A szoftver fejlesztése során behatóan megismerkedtem a React Native keretrendszerrel. A React Native szorosan kapcsolódik a React-hoz ezért a React megoldásokról és a workflow-król is sokat tanultam. A React Native egy erős eszköz a crossplatform mobil alkalmazások fejlesztésére. Jelenleg nagy népszerűségnek örvend a keretrendszer és komoly fejlesztések várhatóak a jövőben hozzá [33]. A ReactXP és egyéb kezdeményezéseknek hála a React Native számára elérhető platformok száma is bővül. A crossplatform megoldások és a könnyen kezelhető JSX nagyban megkönnyítette a kliensek felhasználói felületének kialakítását.

A fejlesztés során az okozta számomra a legtöbb problémát, hogy a React Native rengeteg különböző könyvtárral egészíthető ki, de ezek nem mindig működnek együtt. Az egységes kinézet elérése érdekében használt közös komponens könyvtár megtalálása például komoly kihívásnak bizonyult. Az olyan esetekben, amikor elég csak hasonló funkcionalitású könyvtárakat használnunk Weben és Androidon a React Native crossplatform megoldásaival könnyen elfedhető a kód elágazása.

A kliens oldal talán legnagyobb gyengesége a globális állapotkezelés hiánya. React alkalmazásoknál a keretrendszer biztosítja a referenciák, a props és a callback függvények segítségével a komponensek közti adatáramlást. Ezek a megoldások nem skálázódnak jól és nehezen kezelhetővé teszik a kódot. A bonyolultabb React alkalmazások esetén ezért szoktak valamilyen közös állapot tárolót alkalmazni, például a Redux keretrendszert. A fejlesztés során az én alkalmazásom is elérte azt a komplexitást, ahol egy ilyen állapot tároló keretrendszer a továbbiakban már lényegesen megkönnyítené a fejlesztést.

A legnagyobb kihívást a Web platform bevezetése jelentette. A Web alkalmazások alapjaikban nagyon különböznek a natív alkalmazásoktól. A kliensek erőforrásaitól kezdve egészen a nézetek elrendezéséig számos különbség van. Ezeket a különbségeket megpróbálja elfedni a React Native Web. Ennek ellenére sok kompatibilitási problémát okozott a Web platform. A Webes kliens miatt a felhasználói felület tervezése is nehézséget jelentett.

Szerveroldalon a repository minta, a függőség befecskendezés és a DDD használata sokat segített. Ezek a tervezési minták megkönnyítették a backend kód felépítését és előre definiálták a struktúráját. Az ASP.CORE keretrendszer nyújtotta lehetőségeknek hála gyorsan és hatékonyan tudtam backendet fejleszteni.

### Fejlesztési javaslatok

Az alkalmazás még sok helyen fejleszthető. A kliens oldalon a Redux állapot tároló rendszer bevezetése sokat segítene a kliens kód kezelhetőségén. A Webes kliens felhasználói felülete tovább fejleszthető. Kisebb képernyőre jelenleg nincs optimalizálva. Az Android kliens összes funkciója jelenleg nincs implementálva Weben. A készség menedzsment összes nézetét el kell készíteni a Web kliensre is. Például a készségeket egy valódi gráfban lehetne ábrázolni.

A React Native keretrendszer egyik eleme a ListView depracted-á vált a fejlesztés folyamata során. A React Native a ListView használata helyett a Flat Listet javasolja. Az alkalmazásban le lehetne cserélni a ListView összes előfordulását a modernebb megközelítésű Flat List-re.

A kliens oldali alkalmazás jelenleg az eredeti React Native build rendszert használja. Ez egy elég egyszerű eszköz, amely nem támaszt buildelés közben szigorú követelményeket és nem támogat magasabb szintű build eljárásokat, mint például az erőforrások kezelése. Ezt a build eszközt le lehetne cserélni pl.: a CRNWA keretrendszerre.

A backenden a Model csomag objektumai jelenleg egyszerre takarnak biznisz objektumokat és részei az adat elérési rétegnek is. Ezt a kapcsolatot a két réteg között meg lehetne szüntetni, ha bevezetnénk tisztán biznisz objektumokat, amelyek a BLL csomag alatt találhatóak. Az adatelérési réteg Model objektumait pedig a Repository-k átalakítanák ilyen biznisz objektumokra, amikor visszaadják az adatokat.

Az alkalmazás funkciói is tovább gondolhatók. A rendszerben jelenleg bárki vállhat szakértővé bármilyen készséggel. Ez egy nagyobb környezetben már nem hatékony, ezért jó lenne bevezetni egy olyan rendszert, amely megerősíti, hogy a készséget felvevő felhasználó tényleg szakértő. Például a válaszokat értékelhetnék mások vagy csak pusztán azok a felhasználók maradnak szakértők, akik rendszeresen válaszolnak meg kérdést a szakterület kapcsán.

A feltett kérdések között jó lenne keresni szavak vagy szakterületek alapján. Egy olyan rendszerben, ahol sok kérdés van jelen ez elengedhetetlen. A kérdéseket jelenleg az őket feltevő felhasználó törölheti. Ha a feltett kérdések nem csak törölhetők, de egyszerűen archiválhatók is lennének, akkor az megkönnyítené a felhasználást. Az archivált kérdések nem listázódnak ki közvetlen a folyamatban lévő kérdések mellé, de ha szükséges visszanézhetők.

Jelenleg az összes felhasználó kezelheti a készségeket és ezen keresztül módosíthatja a tudás gráfot. Az alkalmazásba be lehetne vezetni egy jogosultság kezelő rendszert. A jogosultságok korlátoznák, hogy a tudás gráfot kik szerkeszthetik.

A tudás gráf jelenleg csak készségek felvételével épül. Ha a tudás gráf ki- és betölthető lenne, az nagyban növelné az alkalmazás kezelhetőségét.

Az alkalmazást a felsoroltokon túl még számtalan módon lehetne fejleszteni. Például, a felhasználók láthatósági státuszának kezelésével, a socket alapú realtime kommunikáció kiterjesztésével az alkalmazás más részeire, vagy akár csak egyszerű felugró figyelmeztetések segítségével.

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | „React Native,” Facebook, [Online]. Available: https://facebook.github.io/react-native/. [Hozzáférés dátuma: 11 12 2019]. |
| [2] | „React Native Web,” Facebook, [Online]. Available: https://github.com/necolas/react-native-web. [Hozzáférés dátuma: 11 12 2019]. |
| [3] | „React,” Facebook, [Online]. Available: https://reactjs.org/. [Hozzáférés dátuma: 11 12 2019]. |
| [4] | „React W3 School,” [Online]. Available: https://www.w3schools.com/whatis/whatis\_react.asp. [Hozzáférés dátuma: 11 12 2019]. |
| [5] | „React Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/React\_(JavaScript\_library). [Hozzáférés dátuma: 11 12 2019]. |
| [6] | „React Component,” Facebook, [Online]. Available: https://reactjs.org/docs/components-and-props.html. [Hozzáférés dátuma: 11 12 2019]. |
| [7] | K. Yusov, „Is React JS Fast?,” [Online]. Available: https://jelvix.com/blog/is-react-js-fast. [Hozzáférés dátuma: 11 12 2019]. |
| [8] | N. Engineering, „Virtual Dom,” [Online]. Available: https://medium.com/naukri-engineering/naukriengineering-virtual-dom-fa8019c626b. [Hozzáférés dátuma: 11 12 2019]. |
| [9] | N. Couvrat, „Wait… What Happens When my React Native Application Starts? — An In-depth Look Inside React Native,” [Online]. Available: https://levelup.gitconnected.com/wait-what-happens-when-my-react-native-application-starts-an-in-depth-look-inside-react-native-5f306ef3250f. [Hozzáférés dátuma: 11 12 2019]. |
| [10] | K. Nahtkasztlija, „Az idegen szavak toldalékolása,” június 2009. [Online]. Available: http://www.pcguru.hu/blog/kredenc/az-idegen-szavak-toldalekolasa/5062. |
| [11] | P. Koopman, „How to Write an Abstract,” október 1997. [Online]. Available: https://users.ece.cmu.edu/~koopman/essays/abstract.html. [Hozzáférés dátuma: 20 október 2015]. |
| [12] | W3C, „HTML, The Web’s Core Language,” [Online]. Available: http://www.w3.org/html/. [Hozzáférés dátuma: 20 október 2015]. |
| [13] | „ExpoSDKInWork,” [Online]. Available: https://docs.expo.io/versions/latest/workflow/how-expo-works/. |
| [14] | „Expo,” [Online]. Available: https://docs.expo.io/versions/latest/. |
| [15] | „Axios vs Fetch,” [Online]. Available: https://blog.logrocket.com/axios-or-fetch-api/. |
| [16] | „Axios,” [Online]. Available: https://github.com/axios/axios. |
| [17] | „ASP.NET\_Core,” [Online]. Available: https://en.wikipedia.org/wiki/ASP.NET\_Core. |
| [18] | [Online]. Available: https://cordova.apache.org/docs/en/9.x/guide/overview/index.html#page-toc-source. |
| [19] | [Online]. Available: http://www.reactnative.com/react-native-dom/. |
| [20] | „Code-First,” [Online]. Available: https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx. |
| [21] | „Domain-Driven Designe,” [Online]. Available: https://airbrake.io/blog/software-design/domain-driven-design. |
| [22] | „PackageDiagram,” [Online]. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-package-diagram/. |
| [23] | „Class Diagram,” [Online]. Available: https://en.wikipedia.org/wiki/Class\_diagram. |
| [24] | „Agregation and Composition,” [Online]. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/. |
| [25] | „Függőség megfordításának elve,” [Online]. Available: https://hu.wikipedia.org/wiki/F%C3%BCgg%C5%91s%C3%A9g\_befecskendez%C3%A9s%C3%A9nek\_elve. |
| [26] | „Dependancy Injection,” [Online]. Available: https://en.wikipedia.org/wiki/Dependency\_injection. |
| [27] | „Future React Native,” [Online]. Available: https://brainhub.eu/blog/future-react-native/. |
| [28] | „ReactXP,” Microsoft, [Online]. Available: https://microsoft.github.io/reactxp/docs/getting-started.html. [Hozzáférés dátuma: 11 12 2019]. |
| [29] | „SignalR,” [Online]. Available: https://hu.wikipedia.org/wiki/SignalR. [Hozzáférés dátuma: 11 12 2019]. |