# Bevezetés

Manapság nagyon elterjedtek a hangalapú kommunikációs szoftverek, amik jelentősen az internetes kommunikáció legjobb és legegyszerűbb formája. Ezeken a szoftvereken keletkezik a legnagyobb mennyiségű információ váltás az interneten. A szakdolgozatom témáját abból az okból választottam, mivel sokat játszom számítógépes játékokkal online és nincsen olyan voice chat alkalmazás, ami megbízható és az átlag felhasználó számára is egyszerűen kezelhető lenne a másik ok viszont az, hogy nagyon érdekelnek az üzenetek átvitelére használatos hálózati protokollok. A szoftvert leginkább a játékos közösségnek és azoknak az embereknek tervezem, akik nem tudnak gyakran beszélni egymással. A szoftver még alkalmas lesz konferencia beszélgetések lebonyolítására is, ami a használhatóságot kibővíti az üzleti életre is. Nagyon sok voice chat alkalmazással az a baj, hogy elég rosszminőségben kerülnek kiadásra és, ezért szakadozik és rossz minőségű a hang és sok olyan hibaüzenetet ír ki, amit egy átlag felhasználó nem tud értelmezni. Példának szeretném felhozni a Battle.net kliensébe beépített voice chat-et, ami értelmezhetetlen hibaüzeneteket dobál fel a kijelzőre és csak többszöri hívás után tudjuk felvenni a kapcsolatot a partnerünkkel. A másik nagy probléma, amit a Microsoft fejlesztésű Skype-al szeretnék bemutatni, hogy ha valaki a Windows 10-be eleve telepített Skype-ot akarja használni akkor, ott nagyon nehéz megtalálni,hogy kik elérhetőek az ismerőseink közül és a másik Skype-al kapcsolatos észrevételem, pedig, hogy a Windows folyamatosan felugró ablakokkal és értesítésekkel akarja a tudomásunkra hozni, hogy ez a szoftver ott van már a Windows rendszerében és minden áron azt akarja, hogy használjuk. Mondjuk ez a dolog egy üzleti megoldás lehet a Microsoft felől, de ez nagyon idegesítő és frusztráló lehet egy átlag felhasználó szempontjából. Nagyon sok voice chat alkalmazásnál nem sok energiát fektetnek bele Quality of Service-be, ami a hang minőségének a romlását eredményezi. A Quality of Service másnéven a szolgáltatás minősége olyan dolgoktól függ, mint például a csomagvesztés vagy a csomagok késése, tehát hogy mennyire megy zökkenőmentesen a szolgáltatás. Tehát, ha számban kicsi az elveszett csomagok mennyisége és kicsi a késésük, akkor egy jó minőségű szolgáltatásról beszélhetünk. Ez a fogalom manapság a legfontosabb az interneten keresztül történő szolgáltatásoknál.

A szoftverem kialakításánál az egyszerűségre és a felhasználó felület design-nál a legjobb átláthatóságra fogok törekedni. A szoftvert C# programozási nyelven .NET keretrendszer és NAudio könyvtár segítségével fogom megvalósítani, a klienst windows formos alkalmazásban és a szervert pedig console-os formában. A szerver adminok a consoles-os ablakba beírt parancsokkal vagy vizuálisan kialakított többlet funkciókkal tudják majd a szervert használókat koordinálni. A szoftvert akármilyen operációs rendszer alatt lehet majd futtatni, amire telepíthető a .NET keretrendszer, ezzel a platform függetlenséget megoldottam a szoftverem kapcsán. A szoftverhez használni fogom a NAudio könyvtárat, amivel a hanghoz köthető funkciókat fogom kialakítani, mint például a bejövő és kimenő hang kezelése vagy a mikrofon állapota(néma, nyomógombos, hangérzékelés). A bejelentkezés és a regisztráció SHA2-es hasheléssel fog történni a küldés közbeni törhetőséget elkerülvén. A hashelés során elkerülhetetlen az adatvesztés, ellenben a titkosításnál minden adat teljes mértékben visszaállítható. Két különböző tartalmú dolognak nagyon kis eséllyel lehet csak ugyan az a hash kódja. A szerver és a kliens között TCP protokoll fogja kialakítani a kapcsolatot, ezek után az UDP és az RTP protokoll veszi át a szerepet az szöveg és az aszinkron hang küldésénél. A szöveg küldésére, azért fogok UDP protokollt használni, mert az UDP egészen közel van a valós idejű adattovábbításhoz és az UDP protokoll nem vár választ a címzettől, ezért is ez a legjobb az adatküldésre. A hang küldésére viszont real-time protokollt fogok használni, hogy a felek késés nélkül tudjanak beszélgetni egymással. A kialakításnál figyelni fogok, hogy a szoftver bővíthető legyen késöbb más funkciókkal. A szoftver több felhasználós rendszerben lesz, ami annyit jelent, hogy két felhasználó típus lesz az alkalmazáson belül, az átlagos felhasználók és az admin vagy adminisztrátorok, akik moderálják a szerveren tartózkodókat. Az adminoknak külön belépési felhasználó neve és jelszava lesz amit a rendszer érzékel és az adminok által használt funkciókkal kibővített klienst fognak kapni.

# A feladat megvalósításához használt szoftverek és külső források

## 2.1 VoIP Bevezetés

A VoIP a protokollok feletti hangátvitelt jelenti. A távközlés egyik legelterjedtebb módja a mia világban. Pár éven belül, akár felválthatja a telefonok használatát is. Ezt a fajta távközlést nagyon nehéz volt kialakítani az internet csomagvesztései miatt. A kialakításához több különféle protokollra van szükségünk ezeket összegzően VoIP protokolloknak nevezzük. A VoIP elődjét már 1973-ban létrehozták Network Voice Protokoll néven. Alap esetben a szoftver össze van kötve a végkészülékekkel, amik telefon formájában vannak jelen, de a szoftverem kliense majd csak a, hozzá megírt szerverrel és más klienseivel tud majd kommunikálni.

## 2.2 Visual Studio 2017

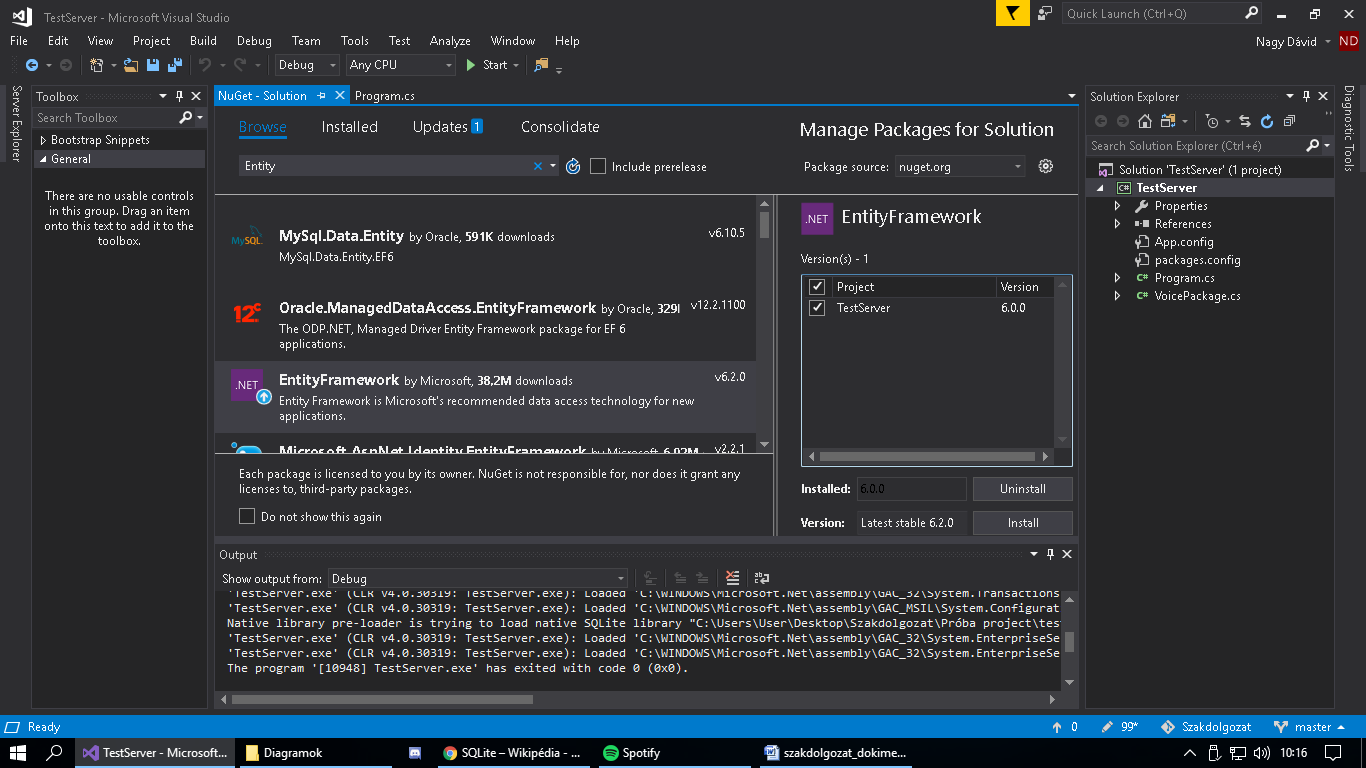
A Visual Studio legújabb verzióját fogom használni a feladat megvalósítása során, mert ez a legátláthatóbb és legtöbb segítséget nyújtó IDE, vagyis fejlesztői környezet. A telepítés ez új verzióban egy moduláris telepítővel telepíthető, ahol ki lehet választani, hogy milyen részegységeket telepíthetünk a Visual Studiohoz. A Visual Studioban rengeteg programozási nyelv támogatott, ezért szinte akármilyen project megvalósítható benne, ezen felül a szintaktikai hibákat egyből jelzi a fejlesztő számára.

A Visual Studio legújabb verziójában lehetőség van a valós idejű egység tesztekre, ami azért jó mivel ha valaki elakad egy szemantikai hiba felderítésével akkor a hiba egyből felderíthető, mivel a szoftver egyből teszteli kód írása közben és a teszt eredmények akármikor elérhetőek lesznek.

A Visual Studioban lehetőség van külső keretrendszereket és könyvtárakat csatolni a NuGet package manager segítségével. A NuGet package manager segítségével sokkal egyszerűbben csatolhatunk külső forrásokat és kódokat a projectünkhöz. NuGet package-et akár saját magunk is létrehozhatunk. A NuGet package managert konzolosan, parancsokkal is lehet kezelni.

## NuGet package-ek

A NuGet package-ek lényegében külső kódok, amiket DLL-ként tudunk a projectünkhöz csatolni. A NuGet packageknek van egy managerük a Visual Studion belül, amivel pár kattintás és hozzáadhatjuk a külső kódokat a projectünkhöz.



A telepítés a projecthez úgy történik, hogy a Browse menüpont alatt kiválasztjuk a telepítendő projectet és aztán a jobb oldalt kiválasztjuk aprojectet, amihez hozzá akarjuk csatolni és végül csak rákattintunk az Install gombra.

## 2.3 Modelio 3.6

A Modelio program egy nyílt forráskódú szoftver, amiben akármilyen UML diagramot tervezhetünk és szerkeszthetünk. A class diagramunkat még kóddá is alakíthatjuk. Mivel a szoftver nyílt forráskódú, ezért a szoftverben nagyon sok hiba van még, de a szoftverem megvalósításában ez nem hátráltat. A Modelia 3.6-os verziója a [www.modelio.org-ról](http://www.modelio.org-ról) a downloads menüpont alatt lehet letölteni. A modelio több operációs rendszerre is elérhető, mint például Ubuntu és Debian Linux.

## 2.4 Photoshop

A logo és egyéb grafikai munkákhoz a Photoshop CS6-ot fogom használni. Minden Photoshop projectemet és képemet elmentem a késöbbi módosítás érdekében. A photoshop szoftvernek az ingyenes 30 napos verzióját használom. A photsohop-ot a [www.adobe.com](http://www.adobe.com) termékek menüpont alól lehet letölteni.

## 2.5 NAudio könyvtár

A szoftveremhez a NAudio nyílt forráskódú könyvtárat fogom használni. Ez a könyvtár elsősorban azért jött létre, mivel .NET 1.0-ás verziójában nem volt a hangok lejátszására támogatás. A nyílt forrás kód annyit jelent, hogy bárki hozzáférhet a forráskódhoz vagy akár saját igényei szerint módosíthatja azt, de módosítás után is teljesen mindenki számra elérhetővé kell tenni. A NAudio könyvtárat NuGet package-ként elérhető így tehát a telepítése könnyedén elérhető Visual Studio fejlesztői környezeten belül. Ez a könyvtár képes hangok felvételére és lejátszására vagy akár meglévő hangfájlok módosítására. Jelenleg ez az egyetlen könyvtár hangok kezelésére, ami C# nyelvhez i elérhető. A NAudio könvtárat a [www.naudi.codeplex.com-ról](http://www.naudi.codeplex.com-ról) is be lehet szerezni. Az oldalon még megtalálhatóak a legismertebb projectek, amiket a NAudio könyvtár használatával valósítottak meg. A NAudio könyvtárra, azért volt szükség, mivel a .NET-ben nem volt annyi hangra vonatkozó funkció, amivel a feladatomat meg lehetett volna oldani.

## 2.6 Git verziókövető rendszer

A szakdolgozatom írása alatt GIT verziókövető rendszert fogok használni. A Git egy nyílt forráskódú verzió kezelő rendszer, amit Linus Torvalds fejlesztett ki a Linux kernel fejlesztéséhez. Nagyobb projecteknél több ágon is dolgoznak a fejlesztők, de én a projectemnél csak egy ágat fogok használni.

Azért döntöttem a Git használata mellett, mert szeretem ha van egy biztonsági mentés az eddigi munkámról és ha mondjuk elrontok valamit a kódban, amit utólag nem tudok kijavítani, akkor visszatudom állítani előző verzióra, ezenfelül Pendrive-ra is hetente lementem. A verziókövetéshez bitbucket.org-ot és GitKraken használok. A munkámat privát repository-ban tárolom. A GitKraken egy vizuális interface a Git verziókezelőhöz, ami rendkívül átlátható és friss design-al rendelkezik. A GitKraken-t a [www.gitkraken.com-ról](http://www.gitkraken.com-ról) szerezhetjük be. A GitKraken elérhető Windows-ra, Linux-ra és MacOS-re is.

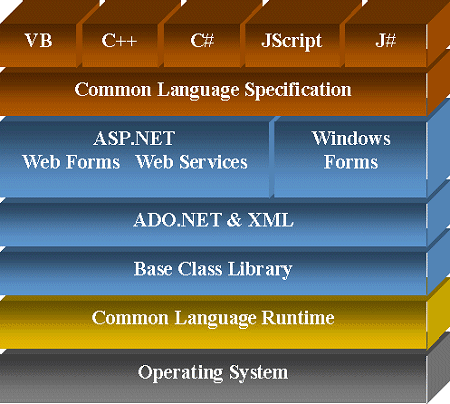
## 2.7 Mysql Workbench 6.3

Az adatbázis megtervezéséhez a Mysql Workbench 6.3 nevű szoftvert fogom használni, mert az ingyenesen letölthető verzió is több féle adatbázis motorral és a feladatom megvalósításához elegendő funkcióval rendelkezik. A Mysql Workbench -el gyorsan és egyszerűen megvalósítható akármilyen adatbázis és adatbázis terv. A Mysql Workbench 6.3 szoftvert [www.mysql.com](http://www.mysql.com) oldalról a products menüpont alatt szerezhető be ingyen.

## 2.9 C# Programozási Nyelv

A C# programozási nyelvet a Microsoft fejlesztette ki. A C# vezető fejlesztője többek között a Turbo Pascal tervezője is volt. A fejlesztés célja az volt, hogy a .NET keretrendszerhez legyen egy olyan programozási nyelv, amivel teljesen objektumorientáltan lehet .NET keretrendszer alatt fejleszteni. A C# nyelvet és a .NET keretrendszert 2000-ben mutatták be. A Microsoft először egy „saját Java” nyelvet fejlesztett ki , amit csak Microsoftos eszközökön lehetett volna futtatni, de ez ellenkezett a Sun Microsystem elveivel , mivel a Sun Microsystem a Javat platform függetlenre szerette volna megtartani. Egy per után a Microsoftnak el kellet tekintenie a Java használatától, ezért fejlesztette ki a .NET keretrendszert és a keretrendszerhez tartozóan a C# nyelvet. A C# vezető fejlesztője azt mondta, hogy inkább a C++ nyelvnek a mintájára akarták tervezni, de végeredményében leginkább a JAVA nyelvre hasonlított. A C# legelső verziója az 1.0-ás volt, most már 7.1-nél tartunk, ami 2017 augusztusában jelent meg. A C# 2002-ben már teljesen objektum-orientáltan jelent meg. A Java alkotója azt mondta a C# nyelvre, hogy szinte teljesen a Java másolata. A Java és a C# nyelv sokban hasonlít, de egy nagy különbség van köztük még pedig az, hogy a Java-nak csak virtuális metódusai vannak, ami annyit jelent, hogy használatkor kell kifejteni, hogy mit csináljanak.

## 2.10 .NET keretrendszer



A .NET egy Microsoft által fejlesztett nyílt forráskódú keretrendszer. A .NET 4.7 a jelenlegi legújabb verzió, ami elérhető. A Windows formok már a 2.0 –ás verzióban elérhetőek voltak, amit 2005-ben adtak ki. A .NET keretrendszer az előre megírt függvényekkel és osztályokkal megkönnyíti a szoftverfejlesztés folyamatát. A .NET keretrendszer alatt megírt szoftverek platform függetlenek, tehát amelyik számítógépen telepítve van a keretrendszer azokon futtathatók lesznek. A szoftverem felhasználójánál is a 4.7-es verziójú .NET-et kell telepíteni a használta elött.

A c# egy menedzselt nyelv, ami azt jelenti hogy az írt kódot a CLR az az Common Language Runtime nevezetű virtuális gép hajt végre. Szinte egy olyan szoftver sincs, amit natív c# nyelven írtak volna. Ezen felül van egy másik fordító alrendszer a .NET alatt írt szoftverek futtatására, ez a CLI vagy Common Language Infrastructure. Ez a Common Language Infrastructure CLR-t használ és az osztálykönyvtárat. A CLI programozási nyelvtől függetlenül tud .NET alatt írt szoftvereket lefordítani, tehát nem feltétlenül van szükségünk az adott nyelv alapvető fordítójára.

## 2.11 Wireshark

A Wireshark egy ingyenes nyílt forráskódú szoftver. Az alkalmazást a hálózati forgalom követésére és tanítási célokra használják. A szoftverrel szűrhetünk ip címekre, portokra és protokoll típusokra is. Az alkalmazásomnál nagyon fontos ez a szoftver, mivel sok hálózati csomaggal kell, majd dolgoznom. A teszt fázisoknál fogom alkalmazni a szoftvert. A szoftvernek a legújabb 2.4.4-es verzióját használom. A szoftver legelső kiadása 1998-ban történt. A szoftvert a [www.wireshark.org](http://www.wireshark.org) –ról a downloads gombra kattintva lehet letölteni.

# A szoftverhez használt hálózati protokollok

A hálózati protokollok írányítják az adott adatokat a kettő vagy több fél között. Helyzettől függően mindig más-más tulajdonsággal rendelkező protokollra lehet szükségünk. Az interneten a két leggyakrabban használt protokoll a TCP/IP és az UDP. A protokollok elsősorban a küldött üzenetek felépítését adják meg. Az általam használt protokollok az OSI-modell 3.-ik rétegében helyezkedik el, ami a szállítási réteg. A szállítási rétegnek nem kell ismernie a hálózati topológiát, amin a csomag halad, csak a kommunikációban részt vevő két félnél van rá szükség.

**Transmission Control Protocol TCP/IP:** A protokoll feladata, hogy a szerver és a kliens közötti kapcsolatot kialakítsa és a bejelentkezési felhasználó nevet és jelszavat elküldje a szervernek. Az adatvesztés ennél a protokollnál teljes mértékben elkerülhető. A protokoll elsősorban a kapcsolatot alakítja ki és csak aztán kezdi meg az adat küldést. A szerver mindig küld egy acknowledge üzenetet a kliens felé, ami jelzi a kliens felé, hogy a csomag megérkezett-e. A csomag ha nem érkezik meg, akkor a protokoll újra küldi ugyan azt a csomagot, de viszont ha megkapta a csomagot a szerver akkor a kliens folytatja a további csomagok küldésével. OSI modell szerint a protokoll a szállítási rétegben helyezkedik el.

**User Datagram Protocol (UDP):** Az UDP protokoll-t a szoftveremben a szöveges chat és hang üzenetek küldésénél fogom használni, mivel az üzenetek megérkezésénél fontos a gyorsaság. A gond az UDP protokollal, hogy nem garantálja az üzenetek megérkezését, de ezenfelül ez a legjobb módja a szöveges és hang üzenetek elküldésére. A Hang üzenetek küldéséhez, azért nem RTP protocolt használok, mert nagyon bonyolult a kialakítása és teljes mértékben elég az UDP annyi felhasználóra, amennyivel be lehet mutatni.

# A szoftver megvalósítása

## 4.1 Specifikációk

**Az alkalmazással szemben a következő funkciók várhatók el:**

* csoportos hang chat: Egy közös szerveren lévők egymással tudnak beszélgetni a kliens segítségével.
* csoportos szöveges chat: Egy közös szerveren lévők tudnak egymásnak írni, URL linket küldeni és hangulatjeleket küldeni.
* privát szöveges chat: A felhasználók egymással is tudnak szövegesen chatelni, ami teljesen privát formában történik.
* felhasználói profil teljes testreszabása
  1. megjelenítendő név: A többi felhasználó által látható név, akármikor módosítható az Options menüpont alatt.
  2. profil kép változtatása: A profilkép akármikor módosítható az Opciók menüpont alatt, a Windows fájlkezelő segítségével.
  3. mikrofon érzékenysége: A mikrofon felvételi érzékenysége az Opciók menüpont alatt változtatható decibel mértékegységben megadva
  4. kimeneti hang hangereje: A kimeneti hang hangereje nem csak a Windows-ban lesz állítható, hanem a szoftveren belül is lehet majd állítani.
  5. mikrofon állapota(némítva, nyomógombos, hang érzékelés): A mikrofon állapotát az Opciók menüpont alatt lehet módosítani. A mikrofonnak három állapota lesz, némított amikor a felhasználót egyáltalán nem hallják a társai, nyomógombos amikor a felhasználót csak akkor tud beszélni, ha az adott gombot lenyomja és hang érzékeléses amikor a felhasználó bármikor beszél azt egyből hallják a beszélgető partnerei.
* admin parancsok (kirúgás, kitiltás, felhasználó némítása): Az adminok a moderálás érdekében a felhasználókat ki tudják majd rúgni, azaz a felhasználót lecsatlakoztatja a szerverről, de a felhasználó egyből vissza tud majd jönni. Az adminok ki tudják majd tiltani a felhasználókat a szerverről, ami több a kirúgással ellentétben több időre szól. Az adminok még némítani tudják majd a felhasználót, adott időre vagy amíg az admin újra vissza nem kapcsolja.

**A szoftver nem funcionalitás beli specifikációi:**

* Az adatokat teljesen titkosítva kell küldeni, a felhasználók biztonsága érdekében.
* Teljesen érthetőnek és használhatónak kell lennie egy háttértudással nem rendelkező felhasználónak is.
* Érthető hibaüzenetek kiírása a leggyorsabb későbbi segítségnyújtás érdekében.

Szerintem ezek az alapvető funkciók egy voice chat alkalmazáshoz, hogy megfelelően, mindenki számára élvezhetően működjön. A hanggal kapcsolatos funkciókat a NAudio könyvtár használatával fogom megvalósítani. A szoftvert a továbbiakban majd újabb funkciókkal lehet majd kibővíteni.

## 4.2 Class diagram

A osztály diagram mutatja meg a szoftver részei közötti kapcsolatot. A osztály diagram-ot teljesen az objektum orientált elveknek megfelelően alakítottam ki. A jelenlegi class diagram a szoftverem írása közben még megváltozhat helyenként.

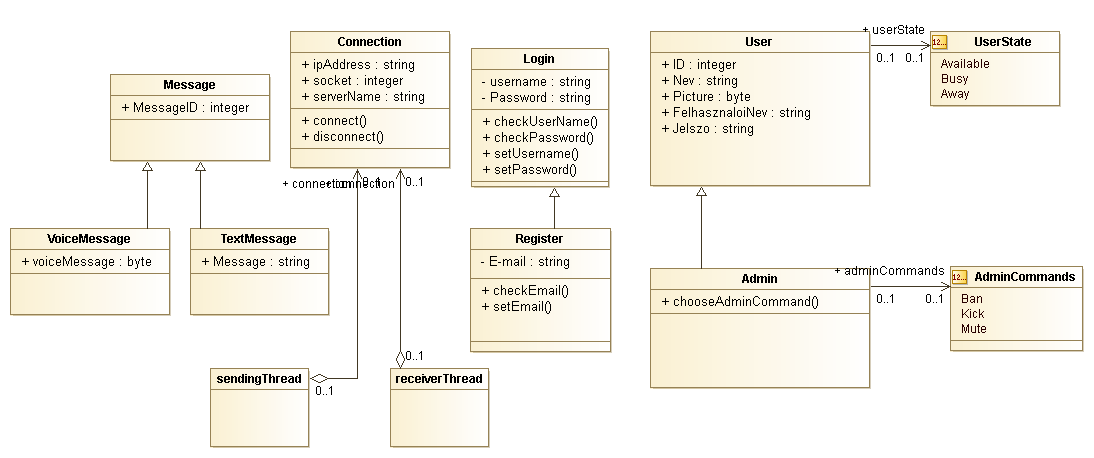
A User osztály fogja tárolni a felhasználó ID-jét integer formátumban, a megjelenítendő nevét string formátumban, a profil képét byte tömbben és a felhasználó nevét és jelszavát. A User osztályból leszármazik majd az Admin osztály, ami egy metódussal bővül ki chooseAdminCommand(), ami majd megvizsgálja az Admin által beírt parancsot és aszerint végrehajtja majd. A User osztály használ majd egy UserState enumerációt, amiben három érték lesz: Available, Busy, Away. Az Admin osztálynak is lesz egy enumerációja ami az admin parancsokat tartja majd számon.

A Message osztályban lesz egy message ID, ami integer típusú. A message osztályból leszármazik a VoiceMessage abban lesz egy byte tömb, ami átmenetileg tárolja majd a mikrofonból bejövő hangot. A Message osztályból leszármazik még a TextMessage osztály, aminek lesz egy string tömbje, amiben majd a szöveges üzeneteket fogom tárolni.

**TODO//: Connection-nél aggregáció helyett composition fordítva**

A Connection osztálynak 3 attribútuma lesz egy ip cím, egy port szám vagy socket és a szerver neve, amit majd a connect() metódus hatására kap meg a szinkronizáláskor. A disconnect() metódus hatására megszakad a kapcsolat a szerverrel és az attribútumok kinullázódnak.

A Login osztályból leszármazik a Register osztály, ez azért fontos mert A Register osztálynak szüksége lesz a Login osztály metódusaira. A Login osztálynak lesz két privát attribútuma a username és password, de ez még nem végleges. A Register osztálynak lesz egy privát attribútuma az Email, ami string formátumú.



## 4.3 Használati eset diagram

A használati eset vagy use case diagramot a felhasználók számára elérhető funkciókat mutatja meg. A használati eset diagramokat a szoftverfejlesztés azon fázisában használjuk, amikor az ügyfél meghatározza a specifikációkat és ezekből az ügyfél számára is érthető ábrát hozunk létre.



**Aktorok:**

* Átlagos felhasználó
* Admin

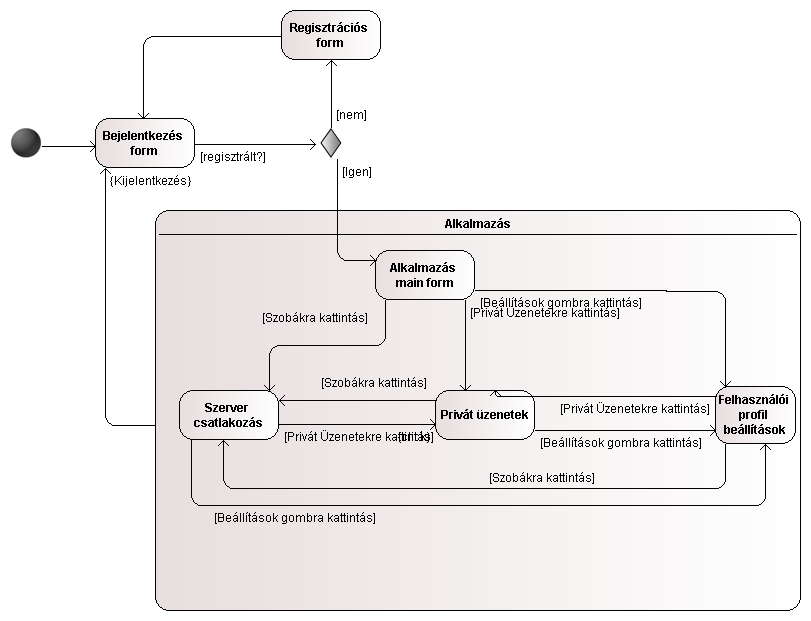
Az aktorok a szoftver felhasználóit jelenti.

Az ábrán a specifikációk cím alatt meghatározott funkciók láthatók. Minden funkció eléréséhez bejelentkezés szükséges, amit az include kifejezés és a nyíl jelenít meg. A nyíl mindig afelé mutat, amit az adott funkció magában foglal. Az Admin aktor leszármazik az Átlagos felhasználóból, ami azt jelenti hogy az Admin minden funkciót tudni fog amit az Átlagos felhasználó is tud, de viszont az átlagos felhasználó nem éri el az admin által használható funkciókat. A leszármazást a vastagabb fejű nyíl jelez. A nyíl mindig annak az irányába mutat, amiből leszármazik az adott aktor. Az aktor által elérhető funkciókat sima vonallal kötjük az aktorhoz.

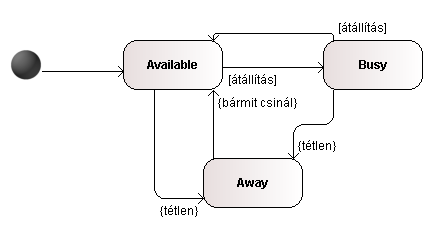
## 4.4 Állapotgép Diagram

### Kliensre vonatkozó Állapotgép Diagramok

**Az első állapotgép diagramon** aformok közötti váltásokat mutatja be. Az egész szoftver a bejelentkezés formnál kezdődik el, azután ha a felhasználó még nem regisztrált, akkor a szoftver a regisztrációs formra irányítja át. Ha a felhasználó már regisztrált akkor bejelentkezés után az összes funkciót eléri a main formból a bal oldalra kihelyezett gombok megnyomásával. A main formon a kijelentkezés gomb megnyomásának hatására a szoftver a bejelentkezés form-ra irányít.

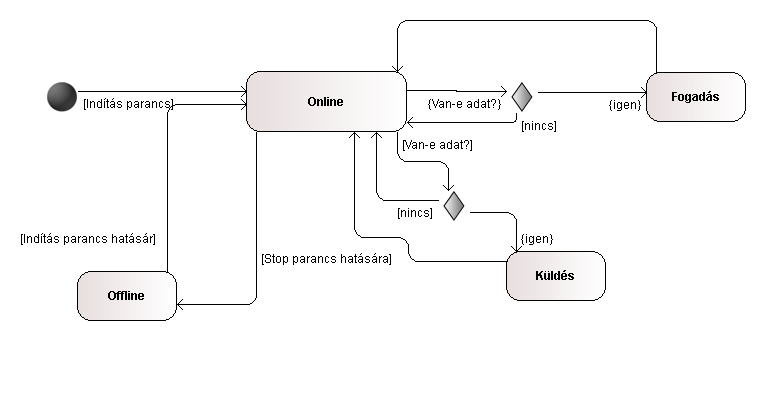


**A második állapotgép diagram** az átlagos felhasználó állapotait mutatja be. A programba belépéskor minden felhasználó elérhető állapotba kerül. Ha, a felhasználó elfoglalt be tudja állítani magának az elfoglalt állapotot és ezután vissza is tudja állítani, ha ismét elérhető. A szoftver attól függetlenül hogy milyen állapotban van a felhasználó egy bizonyos idő elteltével átállítja az állapotát távolira, azaz nincs a gép közelében állapotra.



### Szerverre vonatkozó állapotgép diagramok

A szervernek két fő állapota lesz az Online állapot és az Offline állapot. Online állapotból az admin által kiadott stop parancs hatására lehet átlépni az Offline állapotba. Az Offline állapotból viszont az Indítás parancs segítségével lehet majd újra Online állapotba kerülni.

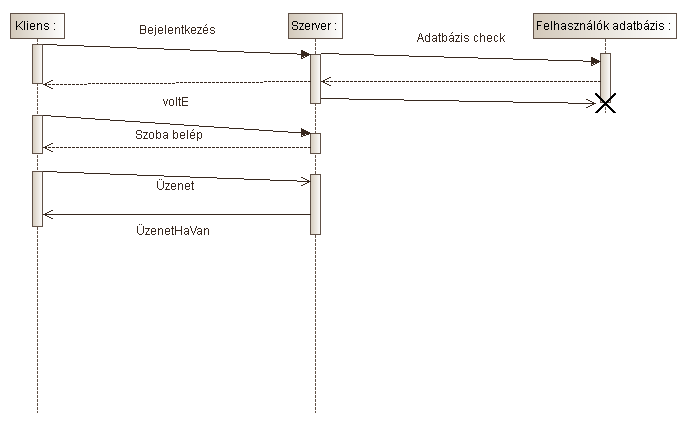


Az Online állapotból még két állapot lehetséges a Fogadás és a Küldés állapot. A Fogadás állapot folyamatosan várja az adatot és alapszinten kezeli azt. Minden fogadás után a szerver visszakerül az Online állapotba. A Küldés állapotba úgy léphetünk, ha a szervernek van küldésre előkészített adata, ha a Küldés megtörtént, akkor ugyanúgy visszatér a szerver az Online állapotába. A szerveren, ha éppen nincs adat küldésre, akkor egyből visszatér a fő Online állapotába. A Fogadásnál, pedig ha nincs bejövő adat, akkor visszatérünk, ugyanúgy Online állapotba.

## 4.5 Szekvencia diagram

A szekvencia diagram a kliens, a szerver és az adatbázis közötti időbeni üzenet váltásokat fogja szemléltetni. A diagramon a legelső üzenet a klienstől a szerver felé egy szinkron üzenet lesz, ami bejelentkezési adatokat(felhasználói név, jelszó) fog elküldeni, amint a szerver ezeket megkapta egyből az adatbázishoz fordul egy lekérdező szinkron üzenettel, hogy az adott felhasználói név és jelszó létezik-e az adatbázisban. A szerver ezután válasz üzenetet kap és azt egyből továbbítja a kliens felé. A felhasználó, ha beakar lépni egy szobába, akkor a kliens egy szinkron üzenettel felveszi a kapcsolatot a szerverrel, ami visszaküld egy válasz üzenetet,hogy a felhasználó be tud e lépni. Miután a felhasználó belépett a szobában a kliens és a szerver is csak akkor küld üzenetet, ha éppen van beérkező, ezek jelen esetben aszinkron üzenetek lesznek.

Az ábrán teli nyilak mutatják a szinkron üzeneteket, a nem teli nyilak pedig az aszinkron üzeneteket. A folyamat futását a vékony téglalapok szemléltetik.



## 4.6 Adatbázis Tervezés

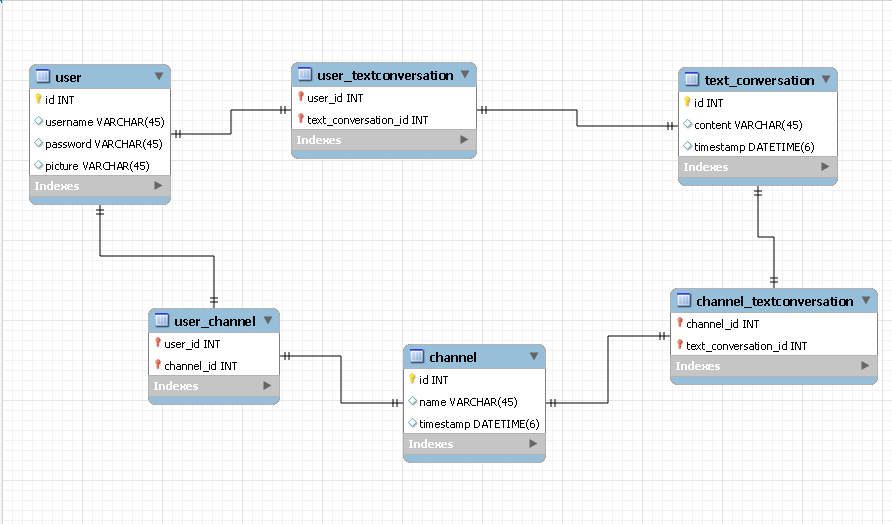
Az adatbázis kialakításánál figyelembe vettem a most felmerülő szoftver szerkezeti elemeket és a szerint készítettem el az adatbázis tábláit és kapcsolatait. Az adatbázis magyar utf-8 kódolású lesz tehát magyar ékezetes betűket is tudok bevinni majd. Az adatbázis későbbi fejlesztés során változhat, ha egyéb igények merülnek fel. Az adatbázisban hat táblát vettem fel:

1. Users tábla: a későbbiekben a felhasználók felhasználói nevét, jelszavát, nevét és a felhasználó képének az elérési útvonalát fogja tárolni. A felhasználói név megegyezik az e-mail címmel. Minden felhasználó el lesz látva egy egyedi azonosítóval is, hogy a többi táblában könnyebben lehessen beazonosítani őket.
2. text\_conversation tábla: a késöbbiekben egy egyedi azonosítót, az üzenet tartalmát és az üzenet elküldésének az idejét fogja tárolni.
3. Channel tábla: a későbbiekben a tábla a szerveren létrehozott szobákat fogja tárolni. A tábla tartalmaz egy egyedi azonosítót, a szobának a nevét és a létrehozás dátumát DateTime formátumban.
4. user\_textconversation tábla: a text\_conversation és a user táblát köti össze a több a többhöz kapcsolat érdekében. Mindkét táblából az elsődleges kulcs kerül bele a táblába. A több a többhöz kapcsolat abból az okból kell, mert a felhasználók több üzenetet tudnak küldeni és egy text\_conversation-höz viszont minimum kettő user tartozhat.
5. channel\_textconversation tábla: ez a tábla a channel és a text\_conversation táblát kapcsolja össze a két táblából az elsődleges kulcs alapján. Itt, azért döntöttem a kapcsoló tábla mellett, mert így könnyeben be lehet azonosítani, hogy melyik üzenet, melyik channelhez tartozik.
6. user\_channel tábla: ez a tábla a user és a channel táblát kapcsolja össze az elsődleges kulcsok alapján. A lényeg ennél, hogy egy felhasználó több szobát is létrehozhat egy szerveren.

A Users és a TextConversation tábla között egy az egyhez kapcsolat van, mivel a felhasználó egyszerre csak egy üzenetet küld el egy másik felhasználónak.

A Users és a Channel tábla között is egy az egyhez kapcsolat lesz, mivel a Channel táblának csak a user\_id kell a Users táblából, ami majd az owner\_id-t fogja jelenti.

Minden táblában az egyik id sem lehet null és nem ismétlődhet, ezért a mező létrehozásánál beállítottam az auto incrementet, ami minden mező felvétele után növekszik automatikusan és kitölti az id mezőket.

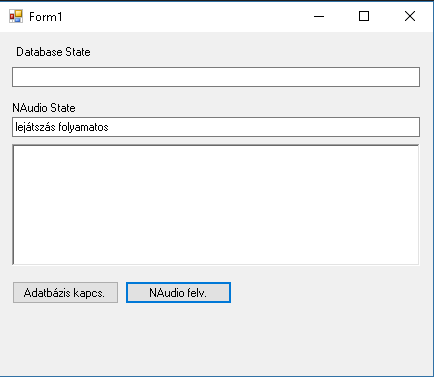


## 4.6.1 SqLite

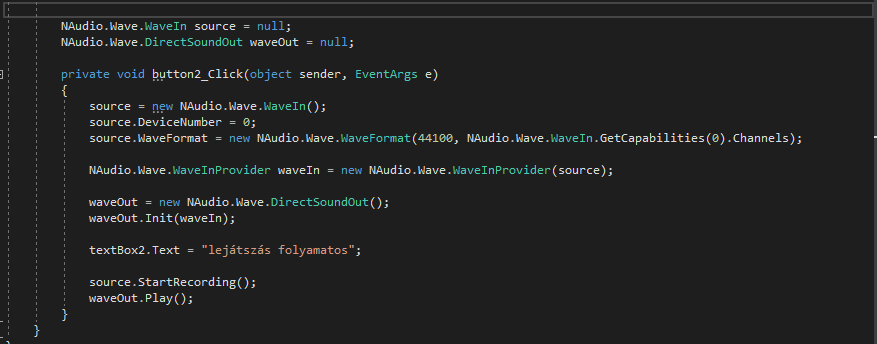
Az Sqlite egy adatbázis kezelő rendszer és adatbázis motor, ami könyvtárként lett megvalósítva. Ezt a könyvtárat NuGet packageként hozzá lehet kapcsolni a projecthez. Az Sqlite nyílt forráskódú, tehát a fejlesztésben akárki közreműködhet. Mivel, az Sqlite megvalósítja az adatbázis szerver és kliens szerepét is, ezért nincs szükségünk külön adatbázis szervert telepíteni. Az SqLite-ot egyre több szoftver használja, ilyen szoftver például a Mozilla Firefox és a Skype.

Az Sqlite adatbázist a szerver fogja kezelni az én szoftverem esetében. Regisztráció esetében „INSERT INTO” SQL parancs használatával fogok beleírni az adatbázisba. Login és felhasználói adatok lekérésénél pedig „SELECT” SQL parancsal fogom teljesíteni a lekérdezést. Az SQL parancsot egy példányosított SqLiteCommand() objektum paraméteréül kell át adni, amit lekérdezés esetén SqLiteDataReader() objektum által lesz végrehajtva. A táblába való beillesztésnél viszont a SqLiteDataReader() objektumra meghívott ExecuteNonQuery() metódus fogja végre hajtani az INSERT parancsot.

## 4.7 Adatbázis kapcsolat és NAudio keretrendszer tesztelése

Létrehoztam egy teszt projectet, amivel tesztelem a szoftveremhez szükséges erőforrásokat. Egy windows formos projectet hozok létre, amiben először felrakom a teszteléshez szükséges windows formos eszközöket, mint például gombok és szövegdobozok.Hozzáadtam a teszt projecthez a NAudio keretrendszert NuGet package manager segítségével. A következő lépésben megírom a kódban az adatbázis kapcsolatot. Valami probléma merült fel a csatlakozáskor, a szoftver futása leáll amikor az sqlcon.Open() metódussal meg szeretném nyitni az adatbázis kapcsolatot. A probléma egy *system.invalidoperationexception* aminek utána néztem a dokumentációban és azt írta, hogy nem adtam meg adatforrást vagy szerver vagy a másik hiba, hogy a kapcsolat már fent áll, de ezek közül egyik sem igaz, mivel a szervert elindítottam és adatokkal is feltöltöttem.

A továbbiakban a NAudio keretrendszert fogom letesztelni, olyan módon, hogy a mikrofonommal fogok felvenni hangot, amit azonnal lejátszok a hangfalakon, ezzel is szimulálva majd a voice chat funkcionalitását. Amikor a NAudionak készített gombbal elindítottam a megírt NAudio keretrendszerben kódot, azt tapasztaltam, hogy teljes mértékben visszahallom a hangomat. A funkciót megírását egy WaveIn() osztály példányosításával kezdtem egy source nevű változóba, ami arra szolgál, hogy megkeresse az alapértelmezett mikrofont és használatba helyezze. a WaveIn() devicenumber-nek meg kellet adni egy integert, ami ebben az esetben nullás érték, mivel a tesztelés miatt elég volt aaz alapértelmezett mikrofont bekérni. A WaveIn-nek a WaveFormatjának a két paraméter kell, egy hogy hány kilohertzen szeretnénk a hangot felvenni a másik hogy melyik az a mikrofon, amelyikkel fel szeretnénk venni. Ezek után a waveinprovider() osztály indítja el az egész hang felvételt. Ezek után miután a hangot felvettük átadjuk a felvett hangot a waveout.Init() metódusnak, ami csak akkor adja vissza a hangot, amikor meghívjuk a waveOut-ra a Play metódust. A két textboxot csak a tesztelés miatti kiíratásra használtam.

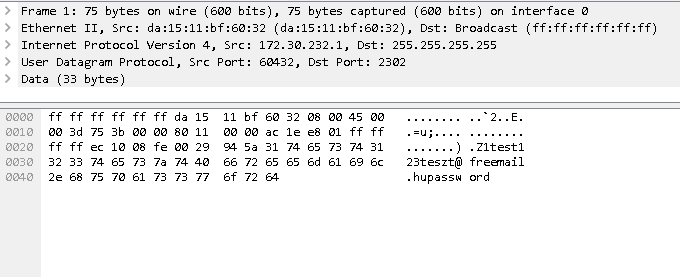


**Adatbázis kapcsolat és kiíratás teszt:**

Az adatbázis kapcsolatot a szerver és az adatbázis között Sqlite-al fogom kialakítani. A legelső, amit meg tettem, hogy megcsináltam egy adatbázis táblát, amivel szemléltetni fogom az adatok bekérését az adatbázisból és a hálózaton átküldését. Első lépésként kialakítom az adatbázis kapcsolatot a using kulcs szóval és bemenő paraméterként megadom a connection string-et ami megmondja az adatbázis nevét és a motor verzióját. Ebben a usingban megnyitom a kapcsolatot az SqLiteConnection változójára meghívott Open() metódussal. Ezen belül még létrehozok egy string változót, amiben az SQL lekérdezést fogom tárolni. Ezek után létrehozok egy SqLiteCommand nevű usingot aminek átadjuk bemenő paraméterként a stringként eltárolt commandot és az SQL kapcsolatot. Ezek után, ha az adatbázisból kiolvasni szeretnénk, akkor létrehozunk még egy SqLiteDataReader usingként és amiben a datareadert egyenlővé tesszük a commandnak az ExecuteReader() metódusával, ami majd végrehajtja az SQL parancsot. A végső mozzanat az az, hogy kiíratjuk az adatbázis tartalmát a képernyőre Console.WriteLine() metódus és benne a data readerre meghívott GetString() metódus aminek paraméterként megadjuk, hogy melyik oszlopot szeretnénk oda írattatni.

**Adatbázis adatok átküldése UDP kapcsolaton a teszt kliensre:**

Az UDP kapcsolat kialakításánál először is létrehoztam egy UdpClient-et a szervernél és a kliensnél is. A szerverről való küldésnél először az UTF-8-as kódolású adatot, amit már az adatbázisból kiszedtünk, azt átalakítjuk byte-tá és belerakjuk egy byte tömbbe, ami majd elküldésre kerül az UdpClient-re meghívott Send() metódussal, aminek át kell adni paraméterként a byte tömböt, a tömb hosszát, az ip címet és a port számot amerre küldjük. Jelen esetben az ip címet Broadcast-ként adjuk meg.

A teszt kliensnél az adatbázis gomb nyomásra történik a fogadás. A fogadásnál először byte tömbbe rakjuk a bejövő adatot, majd a GetString() metódus meghívásával visszaalakítjuk UTF-8 kódolású string-é. Mivel, az UDP porton történő hallgatózást egy végtelen ciklusba tettem, hogy folyamatosan tudja fogadni az adatokat az elvárt működés az , hogy a listbox-ba folyamatosan íródnak be az adatok. A probléma ott merült fel, hogy egy szálon lévő adatot nem tudunk közvetlen egy user interface-en lévő dologra kiíratni, tehát ott, ahol át akarjuk adni az a adatot a vizuális felületnek ott meg kell hívni az Invoke metódust, aminek meg kell adni egy delegate metódust, ami átadja a Main thread-nek az adatot. Ez a sor megállítja egy pillanatra a szál futását, amíg a szál átadja az adatot. 

A képen látszik az elküldésre került UDP csomag. A csomagból kiolvasható, hogy broadcast címre küldtem és a saját ip címem is. A csomagból kiderülnek az adatbázisból kivett és küldött adatok. A portot is le lehet olvasni a fenti blokkból, ami jelen esetben a 2302, amit azért választottam, mert tudomásom szerint ezt a portot egy régi játék használta UDP csomagok küldésére.

A szervernél a küldésnél külön példányosítottam az IPEndpoint-ot és nem helyileg hoztam létre, mivel ha a Send() metódus paraméterében hozzuk létre, akkor minden egyes küldésnél létrehoz egyet és ez leterhelné a gyengébb gépeket.

A következő fázisban egy másik számítógépen tesztelem a kapcsolatot. Az elvárt működés az az, hogy a másik számítógépen megjelenik majd a szerver által küldött adatbázisból kiszedett adat. A másik számítógépet egy tp-link routerbe kötöttem ethernet kábellel.

Egy hiba merült fel az alkalmazás futtatása során, nem indult el az alkalmazás és kiírta a 0x000007b számú hibakódot, amire az a megoldás, hogy letöltöm a .NET 4.6-os verziójú keretrendszert és a Microsoft Visual c++ 2010 Redistributable package-et, tehát a rendszer újraindítása után zökkenőmentesen, kéne működni a szoftvernek. A frissítések és telepítések után a teszt szoftver a kliens és a szerver is tökéletesen működik az elvárt módon.

**Hang átküldés UDP kapcsolaton keresztül két kliens között kialakítása és tesztelése:**

A legelső lépésben kialakítottam egy osztályt ALawChatCodec néven, aminek a feladata az, hogy a wavein() bufferjét átalakítsa UDP számára küldhetővé és az UDP kapcsolaton bejövő byte tömböt visszaalakítsa olyan byte tömbbé, amit majd a waveout() metódusnak át lehet adni. Ennek az osztálynak van egy ősosztálya AcmChatCodec, ami megvalósít egy INetworkChatCodec interface-t. Az INetworkChatCodec meghatároz két virtuális metódust, az Encode() metódust és a Decode() metódust, ez a két metódus byte[] tömb visszatérési értékű, ami az alosztályoknál lesz fontos. Ezen felül négy darab property-t is definiál, amik visszaadják a kiválasztott kodek elérhetőségét, a nevét, a felvétel formátumát és hogy hány bit-et mintavételez másodpercenként.

Az ALawChatCodec-et egy a NAudio könyvtárhoz tartozó kodek osztály, ami az interface osztályból kiindulva három darab metódusa lesz, az Encode(), a Decode() és a Dispose(), ami abból az okból került bele ebbe az osztályba, mert az interface osztály az IDisposable interface osztályból származik le, ami megvalósít egy Dispose() metódust. A Dispose metódusra, azért lehet szükségünk, mivel, ha a kódolási folyamatot leállítjuk, akkor a folyamatban bent ragadhatnak byte-ok, amik a memóriában vannak eltárolva ideiglenesen, ami egy kevesebb memóriájú gépnél probléma lehet, és ezzel a metódussal teljes mértékben kitisztíthatjuk a memóriából az adatokat.

Az osztályok beüzemeléséhez a formon először is létrehoztam egy új gombot és lekötöttem az onClick eseményét. A gomb lenyomására az elvárt működés az az, hogy egyszerre kezdi el fogadni a hangot egy másik kliensből és küldeni hangot arra a kliensre. A gomb onClick metódusában egy elágazás levizsgál a boolean változót, ami alaphelyzetben false értéket ad vissza, ezért a program első indításánál mindig az igaz ágba fog belépni, ami majd az UDP kapcsolatot, a bemeneti eszközt, ami jelen esetben az alapértelmezett mikrofon, fogja meghatározni. Ezen belül lesz egy Connect() metódus, aminek paraméterként át kell adni egy IPEndPoint-ot, az bemeneti eszköz számát, ami jelen esetben a nullás, mivel a a Windows által alapértelmezett mikrofonnal akarunk hangot küldeni és át kell még adni a kodek osztályból példányosított objektumot, ami majd elvégzi a hang átalakítását.

A Connect() metódusban legelőször létrehozunk egy új példányt a waveIn objektumból. Ez a waveIn fogja a hang felvételt biztosítani. A waveIn-nek meg kell adni, hogy a bufferjében hány milliszekundum hang adatot tároljon a bufferjében, ezenfelül meg kell neki adni még, hogy mi legyen a bemeneti eszköz. A bemeneti eszközöt integer formátumban kell megadni. Ezenfelül meg kell adni egy Waveformatot, ami esetünkben az ALawChatCodec osztályból létrehozott objektum lesz. A következő lépésben hozzá kell adni a wavein DataAvailable event-nek egy metódust, amiben belerakjuk a kimenő byte[] tömbbe kodek osztály által kódolt mikrofonból bejövő byteokat és aztán a küldő szálon elküldi UDP csomagként. Ezután a wavein-re meg kell hívni a StartRecording() metódust, ami elindítja a mikrofon hangfelvételét.

Ezután létrehoztam egy egy új példány a sendVoice és a receiveVoice UdpClient objektumból. Ezután beállítom a fogadó UdpClient Socket beállításait olyan módon, hogy a fogadó UdpClient Client –ra meg hívom a SetSocketOption() metódust, aminek át kell adni paraméterként egy socket számot, egy ip címet és egy boolean változót, ami meghatározza a SocketOption állapotát. Ez után meghívtam az UdpClient Client-re a Bind() metódust, aminek átadtam azt az IpEndPoint-ot, amit a Connect() metódus paramétereként adtam át. Ezután már meg lehet hívni az UdpClient-re a Connect() metódust, aminek semmi köze a jelenlegi Connect() metódusunkhoz és ennek is átadjuk a paraméterként bejött IpEndPoint-ot, amit az újonnan létrehozott gomb onClick-ben definiálunk.

Miután kész lett a teljes küldés része a Connect() metódusunknak, ezután létrehoztam a fogadás részét, ami egy waveOut() példányosítással kezdődik. Aztán miután a waveOut-ot létrehoztuk, létre kell hozni egy WaveProvidert, ami majd megkapja a kodek osztályból a már dekódolt byte[] tömböt. A következő lépés az az, hogy a waveOut-ra meghívjuk az Init() metódust, aminek átadjuk a waveprovidert, ez után már csak meg kell hívni a waveOut-ra a Play() metódust, ami alapértelmezetten az alapértelmezett kimeneti audióeszközt választja.

A connected változót igazra állítom és létrehozok egy ListenerThreadState-et, amiben beállítjuk a használt kodeket és a használt UdpClient-et. Ezután a ThreadPool QueueUserItem() metódusának átadjuk azt a változót, amibe beleraktuk a ListenerThreadState-et és magát a ListenerThread() metódust.

A ListenerThread() metódus feladata, hogy fogadja a hang adatokat, amíg a connected változó igaz és belerakja egy ideiglenes byte[g tömbbe ezt a fogadott adatot, amit majd a következő lépésben dekódolunk a listenerThreadState-nek átadott kodeknek a Decode() metódusával.

Az utolsó metódus a Disconnect() metódus, ami a UDP NAudio gomb újra megnyomásakor hívódik meg, az else ágban, mivel a connected változót megváltoztattuk. Ez a metódus legelőször false-ra állítja a connected változót és megállítja a mikrofon felvételét azzal, hogy meghívja a wavein –re a StopRecording() metódust. Ezután megállítja a waveout-ot is a Stop() metódussal, ezután lezárja a fogadó és küldő UDP kapcsolatot a Close() metódussal és aztán megszünteti a wavein, waveout és a kodek objektumokat a Dispose() metódussal.

**Udp hangküldés tesztelésének a folyamata:**

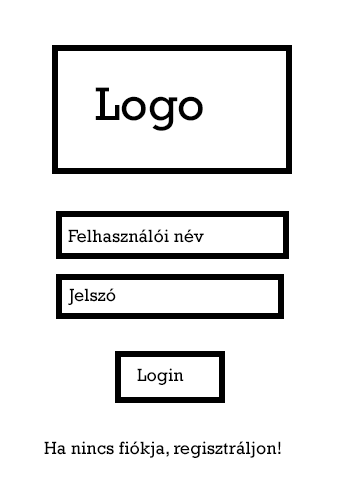
A tesztelés első fázisában az ip címben megadom a localhost ip címet, ami 127.0.0.1 és meg nézem, hogy tudja-e az alkalmazás fogadni a saját maga által elküldött hangüzenetet. Az elvárt működés az az, mint a teszt alkalmazás legelső verziójában, amikor UDP nélkül egyből lejátszottuk a mikrofonból bejövő hangot. A NAudio UDP gombra kattintva ez a folyamat megtörtént. A hangminőségen majd javítani kell még, mert elég gyenge minőségben játszotta le a hangot.

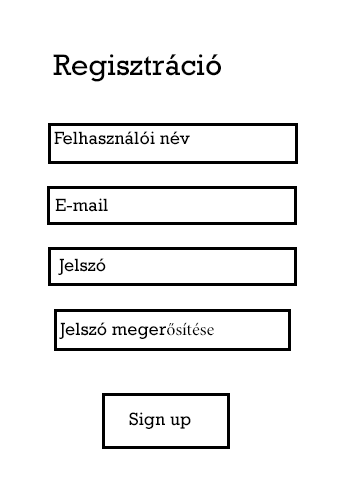
## 4.8 Design

A design kialakításánál és megtervezésénél a legjobb átláthatóságra törekszem. A designt a mai trendnek megfelelően fogom kialakítani. Manapság a designban eléggé elterjedt a flat design, ami a letisztult kinézetéről és a 3D-s formák hiányáról ismerszik meg. A design és a felület kialakításánál a windows form elemeit fogom használni. A színek kiválasztásánál az alkalmazás fantázia nevére hagyatkozom, ami a sPeach. A Peach magyarul barackot jelent, tehát a színeket is a barack színeihez fogom igazítani.

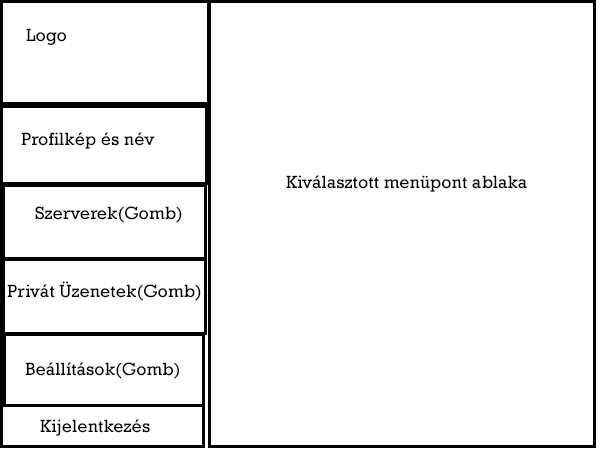
### 4.7.1 Mock-up

Design szempontból fontos itt megemlítenem a mock-up-ot ami a felhasználó által használt ablakokat modellezi le.





A felhasználó legelőször a bejelentkező ablakot látja, ahonnan Login gomb megnyomásával az alkalmazás fő ablakába kerül, de ha nincs felhasználói fiókja akkor a „Ha nincs fiókja, regisztráljon!” linkre kattintva a regisztrációs ablak ugrik fel, ahol az adatok helyes megadása után a Sign up gombra kattintva a felhasználó visszakerül a Bejelentkezés ablakhoz. A felhasználó itt megadja a bejelentkezéshez szükséges adatokat és az alkalmazás a fő ablakra visz át. Itt látható a Logo alatt a felhasználó neve és a beállított profil képe és a válaszható menüpontok. A szerverek gombra kattintva oldalt megjelennek a szerverek amikre majd csatlakozni lehet, ott a szervert kiválasztva a jobb oldali kép megváltozik és már a szerverre csatlakozott barátokat és a szöveges chatet látja a felhasználó. A privát üzenetek gombra kattintva a felhasználó az összes privátan folytatott üzenet váltását kilistázza jobb oldalra. A beállítások gombra kattintva a felhasználó a profiljával és a hanggal kapcsolatos beállításokat éri el, mint például a mikrofon állapota, mások számára látható kép és név változtatása.



### 4.7.2 Design lépésekre bontása

**Login és Regisztrációs Form kinézete:**

* Nagyobb label:
  + Font style: Lucida Sans, félkövér
  + Font size: 32,25 pt
* username/password textbox:
  + Background color: SeaShell
  + Font style: Lucida Sans, félkövér
  + Font size: 16 pt
* Sign in gomb:
  + Background color: SeaShell
  + Font style: Lucida Sans, félkövér
  + Font size: 9pt
* Linkelt label:
  + Font style: Lucida Sans, félkövér
  + Font size: 9pt
  + Link color: 0;0;225
  + Visited link color: 128;0;128

**Main form kinézete:**

* Jobb oldali panelek:
  + Background color: Web/Coral
* Logo mögötti panel:
  + Background color: Web/LightSalmon
* Felhasználó neve és állapota panel:
  + Background color: Web/LightSalmon
  + Rajta lévő Label Font: Lucida Sans
  + Rajta lévő label font size:
    - kisebb: 8,25pt
    - nagyobb: 14pt
* Gombok:
  + Background color: Web/LightSalmon
  + Font size: 14pt
  + Font style: Lucida Sans

A szoftver bal felső sarkában látható ikonhoz egy internetről ingyenesen letöltött képet körbe vágtam és transzperenssé tettem aztán a képet lekicsinyítettem 128x128 px-es méretre. Aztán <http://convertico.com/> segítségével, ami egy online konvertáló web alkalmazás png formátumból ico formátumba konvertáltam át.



A szoftverem logojához legelőször szabadkézzel rajzoltam Photoshop programban egy barackot. A baracknak az előterébe elhelyeztem az alkalmazásomnak a fantázia nevét. A barackot 5pt szélességű ecset eszközzel rajzoltam meg, a barack levelét zöld színnel, míg magát a barackot narancssárga színnel. Ez a verzió az elvártakhoz képest silány minőségű lett.

A szoftver logojának a végleges verzióját egy transparens háttérre terveztem, amin egy ingyenesen letölthető és használható barack képet körbe vágtam és egy új rétegre helyeztem. Következő lépésben létrehoztam egy új réteget, amin egy kicsit elcsúsztatva a barack képétől az alkalmazásom fantázia neve lesz. A képet 250x150 –es méretarányban mentettem ki, ami egy gyakran használt méretarány a logokhoz.

A regisztrációs és a login form között átváltást a Hide() és a Show() metódussal valósítottam meg, amik azt csinálják hogy a login formot a háttérben tartják a regisztrációs formot meg az előtérbe rakják. A regisztrációs és login formon a textboxok tartalmát mindig az adott állapothoz igazítottam és ezért a textboxok ha üresek és máshová kattintunk akkor egyből vissza írja az adott mező kezdő tartalmát írja vissza. A jelszó mezőknél a jelszót kitakarva mutatja, de ha üres a mező és átkattintunk máshova, akkor a Password szó vagy Confirm Password szavak vissza íródnak és már nem lesz rejtve a mező tartalma.

A main formon az állapotok közötti váltást a panelek közti váltással oldom meg. Mindig más gomb nyomásra más panel fog láthatóvá válni, míg a többi láthatatlan lesz.

A Main form property-ben beállítottam a border style-t, hogy magát az ablakot ne lehessen kurzor használatával átméretezni, erre azért volt szükség mivel a gombok nem külön panelen helyezkednek el és a tartalom paneleket nem tudtam kitöltésre állítani.

A gombok mellett egy vékony panel fog elhelyezkedni csík formában, ami a menüpontok aktív állapotát fogja jelezni. Ezt úgy alakítom ki, hogy a kis panel magasságának mindig megadom az adott menüpont gombjának a magasságát a windows formon, ezért amikor rányomunk egy gombra akkor a kis panel oda fog ugrani ezzel jelezve a menüpont aktív állapotát.

A main formon nem volt jó a logo nagysága, ezért Photoshop-ban átállítottam 212x112-es méretre. Először Paint-ben próbáltam, de ott nem lehetett úgy menteni hogy megtartsa az átlátszóságát. Visual Studioban minden windows form elemnek egyedi nevet adtam, hogy a későbbi munkám során könnyebben tudjak rájuk hivatkozni. Már mindenhol beállítottam az alkalmazás iconját. A későbbiekben meg kell még valósítani a panelek közti váltást és kialakítani a paneleken a szükséges elemeket.

## A szoftver megvalósítása

Az adatbázist a jól ismert „CREATE TABLE” paranccsal, benne a táblákat meg „CREATE TABLE” paranccsal létrehoztam. Az ID mezőket auto incrementre állítottam, ami azt jelenti, hogy ha beszúrunk oda egy NULL értéket akkor az adatbázis motor magától a sorban következő értékre fogja állítani az ID értékét. A legelső lépésben az adatbázist áttervezem olyan módon, hogy a channelre vonatkozó táblákat eltávolítom az adatbázisból, mert újragondolva az egészet, maga a szerver lesz a channel. Miután a channelekre vonatkozó táblákat eltávolítottam, azután ki mentem az SqLite parancssoros felületéből a kész adatbázist. A kimentés során egy olyan hiba merült fel, mivel az SqLite először a memóriába menti a kész adatbázist , hogy az adatbázist semmilyen féle képpen nem tudtam kimenteni .db kiterjesztéssel, tehát új megoldást kellett találnom. Internetes böngészéseim során rátaláltam egy Sqlite-hoz használható vizuális szoftverre, amivel könnyedén lehet táblákat és a táblának a recordjait módosítani. Ez a szoftver azonban nem tud speciálisabb mezőkre vonatkozó beállításokat, mint például, hogy a mező típusa legyen DATETIME, ezért egy harmadik megoldásra van szükségem, ami úgy működik, hogy a Mysql Workbench programból reverse enginer segítségével Phpmyadmin-ba berakom az adatbázist, és a reverse enginer által lekreált adatbázist kiexportálom és aztán az SqLitehoz használt programba meg betöltöm a kiexportált sql kódot. A kiexportált sql kódból már csak ki kell venni a mysql-re vonatkozó kódrészleteket és már nincs más dolgom, mint használni az adatbázist a szerver oldalon.

# Szerver oldal

A szerver oldalon a legelső lépésben létrehoztam egy új console application projectet. Ebben a projectben először kialakítottam a két legfontosabb classot, a Connection class-ot és az SqLite-ot kezelő class-ot.

## Connection.cs

A Connection class lényegében egy TcpListenert hoz létre, amikor létrehozunk belőle egy példányt. A connection classnak két mezője van, egy privát integer port mező és egy TcpListener típusú tcpListener mező. A TcpListener mindig akkor jön létre, amikor egy példányt létrehozunk, ez azért van, mert a classnak a konstruktorában hozzuk létre az új TcpListenert. Minden új onnan létrehozott TcpListenernek meg kell adni paraméterként ezt a portot, amin majd hallgatózni fog.

## SqLite.cs

Az Sqlite nevű classnak a feladat, az adatbázishoz csatlakozás és a parancsok végrehajtása lesz. Ebben a classban három privát mezőt vettem fel az SQLiteConnection-t, az SQLiteCommand-ot és az SQLiteDataReadert. Ezenfelül még felvettem két public láthatóságú mezőt username és password néven. ez a két mező, azért public láthatóságú, mivel ezt a két mezőt le kell vizsgálni a login üzenet beérkezésekor. A class konstruktorában létrehozunk egy új példányt az SQLiteConnection-ből és paraméterként átadunk neki egy connection stringet, amiben meg kell adni neki hogy mi az adatbázi neve, amihez csatlakoznunk kell, és meg kell adni hányas verziójú az SQLite. Jelen esetben hármas verziójú. A classban felvettem még 3 metódust, egy integer visszatérési értékű executeInsert nevű metódust, egy void típusú loginSelect metódust és egy void típusú closeDatabase metódust.

Az exectueInsert nevű metódusnak van egy paraméter, ami egy INSERT parancsot vár. A függvényben először is létrehozunk egy új Sqlite commandot, ami paraméterként meg kell adni a lefuttatandó parancsot és az SQLiteConnection-t, amit már a konstruktorban létrehoztunk. Ezután a metódus visszatér az SQL parancs végrehajtásának azt értékével és ezzel együtt végre is hajtja a parancsot.

A loginSelect nevű metódus a bejelentkezéshez szükséges lekérdezést hajtja végre. A using kulcsszó hazsnálatával létrehozunk egy SQliteCommandot és egy SQLiteDataReadert, egymásba ágyazva mivel az SQLiteDataReader használni fogja, majd azt a parancsot, amit átadtunk neki. Ezután egy elágazással levizsgálom, hogy még olvas-e az SQLiteDataReader és ha olvas akkor kimentem a kiolvasott felhasználóinevet és jelszót. Ezek kimentése az SQLiteDataReader GetString metódusával fog megtörténni, aminek paraméterébe meg kell adni, hogy a lekérdezés melyik oszlopából mentse ki.

A closeDatabase nevű metódus feladat, hogy bezárja a kapcsolatot. A kapcsolat bezárását pedig a konstruktorban létrehozott példány Close() metódusával végzi el.

## User.cs

Ezután felvettem a User class-ot, ami a becsatlakozott kliensek tárolásában fog segítséget nyújtani. A User class három dolgot fog tárolni a becsatlakozott felhasználóról, a felhasználó nevét, a felhasználó képét byte tömbben és a felhasználó TcpClient-jét. A felhasználó nevét és a képét majd a Kliensen veló kijelzésre fogom használni.

## ServerMessageType.cs

Ezután felvettem egy enumerációt, ami segítségével, üzenetbe tudjuk írni, hogy milyen üzenetet küld a server a kliensre. Az enumerációval öt féle üzenetet tudunk választani, login\_response, register\_responsse, all\_user\_data, chat\_message és voice\_message.

## UserMessageType.cs

Ezek után felvettem egy enumerációt, ami segítségével a szerverre beérkező kliens üzeneteket tudjuk meghatározni. A kliensekről öt fél üzenet érkezhet, login\_Data, registration\_data, user\_data, text\_message és voice\_message.

## Program.cs

Legelőször a Program.cs-ben fel vettem egy Dictionaryt, amiben majd a bejelentkezett Usereket fogom tárolni, miután ez meg volt létrehoztam egy új példányt az SqLite osztályból, ami ugye kelleni fog az adatbázis műveletekhez. A main függvényben létrehoztam egy új szálat, ami majd az adatok fogadásához lesz majd szükséges. Ennek a szálnak át kell adni egy metódust, ami kezeli a bejövő üzeneteket. Ezután a main függvényben meghívtam a nem régiben létrehozott szálnak a Start() metódusát. A metódus,amit átadtam a szálnak az egy statikus metódus, azért kell statikusnak lennie, mivel a szálnak csak statikus metódust lehet átadni paraméterül. Ennek a metódusnak receiverThread a neve. A metóduson belül legelőször létrehozunk egy példányt a Connection classból, mivel ez tartalmazza a TcpListenerünket, ami segíteni fog a TcpClientek csatlakozásában. Ez után felvettem egy messageType nevű integer változót, amiben késöbb fogjuk kimenteni, hogy milyen üzenet érkezett. Ezután már elindíthajuk a létrehozott Connection példánynak a TcpListenerjét a „con.tcpListener.Start()” metódussal. Ezután felveszünk egy while ciklust, aminek a feltételébe egy true-t írunk, mert azt szeretnénk, hogy mindig, folyamatosan fogadja a Felhasználókat. A while cikluson belül legelőszörLétrehozunk egy TcpClient-et, ami értékül kapja a Listener által fogadott TcpClient-el. A csatlakozó kliensek fogadása az AcceptTcpClient() metódussal valósul meg. Ezután felvettem egy NetworkStreamet, ami majd a bejövő klienssel tartja fent a kapcsolatot. Ennek a NetworkStreamnek, ezért át kell adni az előzőleg fogadott TcpClient-nek a streamjét. A bejövő TcpClient-nek a stream-jét a acceptedClient-re meghívott GetStream() metódussal lehetséges. Ezután rájöttem, hogy nem a legjobb a BinaryWriter és BinaryReadert a Connection classban tárolni, ezért ezeket kihoztam ide ebbe a metódusba. A BinaryReaderből és BinaryWriter-ből létrehoztam egy-egy példányt. Ezeknek át kell adni az előzőleg kiszedett netwrokStreamet paraméterül. Ezután már megkezdődhet az adatok fogadása a kliensektől. A legelső dolog,amit meg kell tenni, hogy kivesszük az üzenet legelső információját, azt hogy milyen típusú az üzenet. Ezt a messageType változóba tároljuk. Ezután egy switch case-el már letudjuk ellenőrizni, hogy éppen milyen üzenet érkezett be. Azért van szükség switch-re, mert ha csak i falágazásokat írnánk egymás alá, akkor, ha több féle üzenetet fogadna a szerver mint a mi esetünkben, akkor egy idő után átláthatatlan lenne az egész üzenet fogadás. A switch casenek megadjuk a messageType integer, amit majd vizsgálni fog. Ez a messageType integer önmagában nem elég, mert fel kell castolni UserMessageType-á, mivel a switch case azt tudja csak vizsgálni. A legelső esetben a login üzenetet fogjuk levizsgálni és vizsgálat után választ küldünk vissza a kliensnek. felvettem egy string változót, amibe kiolvassuk az üzenet legelső tagját, ami a username. Ezután felvettem még egy string változót, ami kiveszi az üzenet második tagját, ami a password. Ezután az üzenetben megérkezik, hogy milyen hosszú lesz a felhasználó képének a byte tömbbje, ezt az értéket lementem egy integer változóba. Ezután már csak a byet tömbböt kell kiolvasni az üzenetből. A byte tömböt csak úgy tudjuk kiolvasni a tömbböl, hogy a binaryReader.ReadBytes() metódusnak átadjuk, hogy milyen hosszú a byte tömb. Miután kiszedtünk minden információt az üzenetből, ezután már csak le kell vizsgálni, hogy benne van-e a becsatlakozott felhasználó az adatbázisnak a user táblájában. Ezt úgy teszsük meg, hogy létrehozunk egy string változót, amiben összerakunk egy SQL parancsot. Ez az SQL parancs két mezőt kérdez le az adatbázis user táblájából, a username-et és a password-ot és csak azt fogja visszaadni, ami az üzenetben megérkezett. Ezután meghívjuk az SqLiteból létrehozott példány loginSelect metódusát, aminek paraméterként át kell adni az előzőleg stringként megalkotott parancsot. Ez a parancs, ha már lefutott, utána le kell ellenőriznünk, hogy a lekérdezésből visszakapott érték megegyezik-e az üzenetben kapott értékkel. Ezt egy if elágazással levizsgáljuk, ha megegyezett akkor a clients néven felvett Dictionary-ba felvesszük az új Usert, ezt aa clients.Add() metódussal lehet megtenni. Itt, mivel a Dictionaryknak két dolgot kell átadni, egy id nevű unsigned integer veszek fel statikusként a metódusok és függvényeken kívűl és a paraméterben még létrehozok egy új példányt a User osztályból, ahol átadom a példánynak a username-et a képét a usernek és a TcpClientet, amiről érkezett a bejelentkező üzenet. Miután hozzáadtuk a Dictionarynkhez a belépett Felhasználót azután az id nevű unsigned integert megnöveljük eggyel, amire azért van szükség, hogy minden bejelentkezett felhasználónak egyedi id-je legyen. Ezután már visszaküldhetjük a kliensnek a bejelentkezés válaszát, ami a binaryWriter-rel fog történni.

A dokumentáció írása közben rájöttem, hogy nem nyitottam szálat minden egyes beérkező kliensnek, ami rendkívűl nagy baj, mivel akkor csak egy kliens tudna bármit is küldeni a szervernek és maximum egy üzenetet, ami egy több szálas szervernél több felhasználóval eléggé lekorlátozná a szoftver használatát például a szöveges chat esetében, ezért ennél a pontnál átalakítottam a szerveren a kódot. Olyan módon tettem ezt a változtatást, hogy a main függvényben felvettem ugyanúgy a fogadó szálat és utána elindítottam. A fogadó szál függvényében csak létrehozunk a Connection osztályból egy példányt, ami létrehozza nekünk a TcpListenert. Ezután ezt a TcpListenert elindítjuk a Start() metódusával. Ezután indítunk egy while ciklust, ami egy végtelen ciklus lesz és ebbene a ciklusban fogadjuk a TcpClient-eket. Minden egyes becsatlakozott TcpClient után létrehozunk a TcpClientnek egy szálat, ami kezeli a TcpClienthez tartozó adatfogadást és esetenként a válasz adatok visszaküldését a kliensre. A TcpClienthez tartozó szálnak a függvényében, amit listenForMessage-nek neveztem el. Ez a függvény paraméterként kap egy object-et, ami jelen esetben a acceptedClient lesz, amit még a listenForMessageThread.Start() metódusnak adtunk meg paraméterként, és ezzel fogja tudni a metódus, hogy melyik kliensről van szó. A metódus legelső pillanatában létrehozunk egy TcpClientet, amit egyenlővé teszünk a paraméterként bejött object-el, mivel a bejövő paraméter típusa object, ezért felkasztoljuk TcpClient-é. Ezután létrehozunk egy NetworkStreamet, aminek megadjuk ennek a TcpClientnek a GetStream() metódusát. Ennél a pontnál elkezdem használni a using kulcsszót, ami egy láthatóságot hoz létre egy adott objektum példányra, amit a using- ban hozunk létre. A using legfőbb tulajdonsága, hogy miután vége a using blokknak azután egyből megszűnik az objektum példány. Először létrehoztam egy BinaryStreamet a using használatával, és a példánynak átadjuk a networkStream-et, amit az átadott TcpClient-ből szedtünk ki. Ezután létrehoztam a using használatával egy BinaryWritert, ami majd a networkStream-re való írást teszi lehetővé nekünk. A binaryWriter-nek is átadjuk a networkStreamet. Ezután a folyamatos üzenet fogadás érdekében az egész kódot, ami az üzenetek fogadásához szükséges, azt egy while(true) végtelen ciklusba tesszük. A while ciklus belsejében legelőször levizsgáljuk, hogy van- e TcpClient, ennek a vizsgálatát úgy végezzük, hogy az acceptedClient-nek az Available property-jét levizsgáljuk, hogy nagyobb-e mint nulla, mert ha nagyobb, akkor azt jelenti, hogy van TcpClient csatlakozva, akinek küldhetjük vissza az üzenetet. Ezután kiolvassuk a bejövő üzenetből a legelső byte-ot, ami minden üzenetben az üzenet típusa lesz. Ezután egy switch case-t veszünk fel, ami a messageType-ot fogja vizsgálni. Ahhoz, hogy letudja vizsgálni vagy legalább is jól vizsgálja le, ahhoz ezt az integer változót fel kell castolni UserMessageType-ra, utána már csak a case-eknél már csak az adott UserMessageType-ot kell beállítani. A login case ugyanúgy néz ki, mint a szálak átalakítása előtt leírtam. Ezután a regisztrációs üzenetet vizsgálom le. A vizsgálathoz először ki kell vennünk az üzenetből a három stringet, amit megkapunk. Az üzenet minden eleméhez felveszek egy string-et. Mindegyik stringet sorban a binaryReader-nek a ReadString() metódusával olvassuk ki a streamből. Ezután megcsináljuk az SQL scriptet, ami a regisztráció esetében egy INSERT INTO lesz. Ezzel az INSERT-el beszúrok az adatbázisba egy id-ét egy felhasználói nevet egy jelszavat és egy emailt. Ezután egy if elágazásnak a vizsgálati részében hívódik meg a beszúró metódus, ami vissza ad egy integert és ha ez az integer egyenlő eggyel akkor az azt jelenti, hogy sikerült beszúrni az adatot az adatbázisba. Az if elágazás igaz ágában binaryWriter segítségével összeállítom a válasz csomagot a kliensnek. A válasz csomagba legelőször be kerül, hogy ez az üzenet egy ServerMessageType.register\_response. Aztán az üzenet lényege jön, ami a kliensnek a legfontosabb, ami egy egyes szám lesz. A kliens majd ezt az egyest fogja levizsgálni. Ezután már meghívhatjuk a binaryWriternek a Flush() metódusát, ami elküldi a networkStreamen a kliensnek az üzenetet. Az üzenet elküldése után console-ra kiíratom, hogy sikerült-e regisztrálni. Az if elágazás hamis ágában ugyanúgy egy ServerMessageType.register\_respons üzenetet küldünk csak azzal a különbséggel, hogy maga az üzenet egy nullás szám lesz. Ezután a chat üzenetet fogjuk fogadni a szerveren. Itt ebben a case-ben az üzenetből kivesszük a felhasználó felhasználói nevét a GetString() metódussal, ezután a chat üzenetet is kivesszük ugyanezzel a megoldással egy string változóba. Ezután kiírattam a chat üzenetet, hogy egyáltalán átjön-e az üzenet a kliensről. Ezután felvettem egy foreach ciklust, ami végig megy a becsatlakozott klienseknek a tömbjén. Ezzel a megoldással lényegében egy mindenkinek eltudjuk küldeni az üzenetet. A foreach ciklusnak meg kell adni egy változót, ami a tömbnek egy elemét fogja jelenteni.Ezután velfeszünk egy új BinaryWritert, mivel a másik BinaryWriter csak ugyanannak a TcpClientnek tud küldeni, aki neki küldött, ezért az új BinaryWriter-nek a clients tömb adott tagjának az elmentett TcpClientjének a Streamjére fogjuk küldeni a chat csomagot. A chat csomagba először is beleírjuk, hogy ez a csomag egy chat csomag, ez egy ServerMessageType. Ezután a chat csomagba beleírjuk a küldő felhasználó nevét és aztán beleírjuk még a chat üzenetet ezek, után már csak a BinaryWriter Flush() metódusát kell meghívni.

# Kliens oldal

A kliens oldalon nem a megtervezett mockup alapján alakítottam ki a formokat mivel egy könnyebb megoldást találtam ki rá. Ez a megoldás úgy néz ki, hogy nem a jobb oldalon a gombok mellett fog megjelenni az adott menüpontnak a tartalma, hanem egy külön form-ban. A jobb oldalon a gombok mellett annyi lesz majd látható, hogy éppen ki van fent a szerveren. A gombokat is kivettem a formokról és új gombokat tettem fel. Az első gomb, amit felraktam a Chat elnevezésű gomb, ami a gomb megnyomásával megnyitja a chat-nek az ablakát. A második gomb, amit felvettem az az Options nevezetű gomb, ami egy új formot nyit meg, ahol a kép beállítását és az állapot beállítását fogja tartalmazni. Erre az átalakításra, azért volt szükség, hogy könnyebben tudjak a formokkal dolgozni és hogy jobban átlássam a szoftveremet.

## logInForm.cs

Ezen a formon fog történni a bejelentkezés. Erre a formra először is felvettem egy Picturebox-ot, amiben a szoftverem logoja fog szerepelni. Ezután felvettem egy textbox-ot, ahova a felhasználó beírja a felhasználói nevét. Ez alá a textbox alá felvettem még egy mezőt, ahova a felhasználó be tudja írni a jelszavát. Ezek alá a textboxok alá raktam egy gombot, ami elindítja a bejelentkezés folyamatát. A form legalján van egy linkkel ellátott label, amire majd ha rákattintunk a regisztrációs form-ra fog átvinni. A logInform-nak legelőször a Load eseményét kötöttem le. Erre, azért van szükség, hogy majd ebbe a metódusba építi fel a kapcsolatot a szerverrel. Ezután lekötöttem a linkelt label-nek az OnClick metódusát, ami kattintáskor átvisz a regisztrációs formra. Ezután lekötöttem a username textboxának az OnClick eseményét, ami azt fogja tenni, hogy amikor belekattintunk, akkor egy üres stringet tesz bele abba a textbox-ba, hogy tudjunk aztán bele írni. Ezután egy elágazást vettem fel, amiben azt vizsgálom le a String.IsNullOrEmpty() metódusban, hogy a jelszónak a textboxában van-e valami érték, ha nincs benne semmilyen string akkor a textboxnak a UseSystemPasswordChar property-jét false-ra kell állítani, mert utána beleírjuk a jelszó textboxába, hogy Password, abból az okból, hogy a felhasználó tudja, hogy oda a jelszót kell írni. Ezután lekötöttem a jelszó textboxának az OnClick eseményét, amire azért van szükség, hogy a textbox UseSystemPasswordChar property-jét true-ra állítsuk, azért, hogy más emberek a környezetében ne láthassák a beírt jelszót. Itt is csináltam egy elágazást, ami levizsgálja, hogy a username mezőben van- e valami érték és , ha nincs akkor beleírja a mezőbe, hogy username, ugyancsak a felhasználó érdekében. Ezután a magának a formnak és a picturebox nak is leköttöttem az OnClick eseményét, mert bárhova is kattint a felhasználó, ha üres stringek vannak a textboxokban, akkor mindig ha elkattint a felhasználó, akkor újra beírja a textbox-ba alapértelmezetten, hogy mit kell abba a textboxba írni. Ezután lekötöttem a felhasználói név textboxának a TextChanged eseményét, amire a textbox ellenőrzése miatt van szükség. Az ellenőrzésénél legelőször létrehoztam egy új RegEx-et, amit usernameRgx-nek neveztem el. A felvett RegEx-nek át kell adni paraméterül egy Regex-et string formában. Miután ez megtörtént levizsgálom, hogy a textbox-ba írt felhasználói név eleget tesz-e a Regex-ben meghatározott szabálynak. Itt egy elágazást használtam, ami levizsgálja a regex-nek az IsMatch() metódusával, hogy a felhasználói név, amit beírtunk az megegyezik-e a Regex szabályával, ha ez a feltétel teljesül, akkor egy isUsernameOk boolean változóban megváltoztatjuk az értéket true-ra, ha meg nem teljesül akkor false-re. A Regex-ben meghatározott szabály az az, hogy a felhasználói névnek kis betűt és számot kell tartalmaznia és minimum három, de maximum 15 karakter hosszúságú lehet. A jelszó textboxának is lekötöttem a TextChanged eseményét és itt is fel vettem egy Regex-et. Ennek a regex-nek a szabálya, hogy kis és nagy betűt kell tartalmaznia és egy különleges karaktert. A jelszónak ezenfelül minimum 8 karakter hosszúnak kell lennie. Itt is van egy if elágazás, ami a passwordRgx IsMacth() metódusával levizsgálja a textbox-ban lévő stringet, ha az igaz ágba lépünk bele, akkor az IsPasswordOk nevű boolean változót igazra állítjuk, ha az ellenkezője tőrtének meg akkor hamisra állítjuk.

Ezután lekötöttem a bejelentkezés gomb-nak az OnClick eseményét, amikor rányomunk a gombra először is egy if elágazás levizsgálja, hogy az isUsernameOk és az IsPassswordOk változók igazra vannak- e állítva, ha beléptünk az igaz ágba, akkor feljebb deklarált változókba kimentjük a textboxokban lévő stringet. Itt ezután létre kell hoznunk egy BinaryWritert, ami majd elvégzi a küldést a szerver felé. Mivel ennek a BinaryWriternek meg kell adni egy streamet, ezért a form konstruktora alatt deklaráltam egy Connection típusú connection nevű osztály példányt. Itt még nem hozok létre belőle példányt, mert nincs értelme, ezért ezt a Load eseményben oldottam meg. Ezek után már át tudunk adni egy streamet a BinaryWriternek. Ezután a binaryWriterként elnevezett példánynak a Write() metódusával tudjuk a stream-re írni az elküldendő adatokat. A bejelentkezés üzenetbe először beleírjuk az üzenet típusát, hogy a szerver el tudja dönteni, hogy milyen üzenetről van szó. Aztán az üzenetbe beleírjuk a felhasználói nevet, amivel be akarunk jelentkezni. Ezután beleírjuk a jelszót, de a jelszónál először meghívjuk az sha256 algoritmust, ami hash-i a jelszavunkat, ezért, ha valaki lehalgatná az üzeneteket, amit a szerver felé küldünk akkor, az az ember nem tudja visszafejteni a jelszavunkat. Ezután felveszünk egy byte tömböt, ami egy Bitmap képet fog tárolni byteokban. Ebbe a byte tömbbe egy alap képet fogok tárolni, ami minden felhasználónak ugyanaz lesz, ezt a képet a késöbbiekben a felhasználók meg tudják majd változtatni az option formon. A képet, hogy átalakítsuk byte tömbbé, kell egy függvény, ami átalakítja. Ennek a függvénynek lesz egy bemenő paraméter, ami Bitmap típusú lesz. Ehhez létre kell hozni egy példány egy ImageConvert-ből. Mivel ennek a függvénynek byte tömb visszatérési értéke van, ezért vissza kell térnünk egy byte tömbbel, amit a converter nevű ImageConverter fogja a képet átalakítani byte tömbbé. Miután átalakítottuk a képet azután, már a login üzenetbe beleírhatjuk a tömbnek a hosszát és magát a tömböt. Ezutén a login üzenetünket már csak el kell küldeni a streamen a Flush() metódussal. Ezután már csak le kell vizsgálnunk a szervertől visszakapott login response üzenetet. Itt a using kulcsszó használatával felveszünk egy BinaryReadert, aminek átadjuk a a Connection osztályból létrehozott példány TcpClient-jének a GetStream() metódusát, ami visszaad egy streamet. Ezután egy if elágazással levizsgáljuk a legelső byte-ot az üzenetből, ami a ServerMessageType, tehát szerver érkező üzenet típusa, ha ez az üzenet típus nullával egyenlő, akkor vizsgálhatjuk meg magát az üzenet tartalmát, ami ha egyes szám, akkor sikeres volt a bejelentkezés és megjelenik akkor a mainform, ami az alkalmazás fő formja.

## registerForm.cs

A regisztrációs formon legelőször felvettem egy labelt, amibe beleírtam, hogy Create an account, hogy a felhasználónak a szeme elött legyen, hogy milyen formon van. Ezután felvettem négy textboxot. Az első textboxba írhatja a felhasználó a felhasználói nevét, a másodikba, az email címét, a harmadikba a jelszavát és a negyedikben csak meg kell erősítenie a harmadikban megadott jelszót. Felvettem ezután két gombot. Az egyik gomb a regisztrációs gomba a másik meg egy mégsem gomb, hogy ha úgy döntene a felhasználó, hogy mégsem regisztrál.

Ezena formon legelőször deklarálunk egy statikus Connection típusú változót, amibe késöbb a konstruktorba egyenlővé tesszük paraméterként átadott objektumot. Mivel ezt az objektumot csak Object típusként lehetett átadni két form között, eézrt az átadott objektumot fel kell castolni Connection típusúvá. Ezután felveszünk négy darab boolean változót, amik a textboxok ellenőrzésénél fognak segíteni. A négy textboxnak és a labeleknek lekötöttem az OnClick eseményét, hogy amikor üres a textbox akkor visszaíródjon a textbox-nak a szövege. A jelszó esetében, ha a jelszónak az egyik textboxába kattintunk, akkor a textbox UseSystemChar propertyét true-re állítjuk. A regisztrációs gombra kattintáskor, legelőször leellenőrizzük, hogy minden egyes kitöltött textbox helyesen van- e kitöltve. Ezt a négy darab boolean változóval tehetjük meg. Egy if elágazásban levizsgáljuk, hogy mindegyik változó egyenlő-e nullával, ha igen akkor megkezdődik a regisztrációs folyamat. Ezután létrehozunk egy új példányt a Hash osztályból, mert kelleni fog a password átküldésénél. Ezután létrehozunk egy új BinaryWriter, hogy majd ezzel küldjük el a regisztrációs adatokat. Ennek a BinaryWriternek átadjuk a loginformról átadott objektum tcpClient-nek a GetStream() metódusát, amivel megkapja azt a streamet, amire írnia kell. Ezután a regisztrációs üzenetbe először az üzenet típusát írjuk bele, ami egy UserMessageType.registration\_data. Az üzenet típusát mindig byte-á kell castolni a ReadByte() metódus miatt. Az üzenet típusa után most már a felhasználói név textbox-nak a text-jét, az email textbox-nak a Text-ét és a password textbox-nak a Text-jét írjuk bele a Binarywriterrel. Végül a BinaryWriter Flush() metódusával elküldjük a streamen. Ezután fogadni kell a visszakapott adatot. Ehhez a using kulcsszóval létrehozunk egy BinaryReadert, amit a regisztrációs válasz fogadására használunk. Ennek a BinaryReadernek is átadjuk az átadott Connection osztályból létrehozott példány TcpClientjének a GetStream() metódusát. Legelőször az üzenet típusát olvassuk ki a binaryReader-nek a ReadInt32() metódusával, amit egy integer változóban tárolunk.egy if elágazással levizsgáljuk, hogy olyan üzenetet kaptunk-e, amire nekünk szükségünk van, szóval egy ServerMessageType.register\_respone, és ha ez akkor egy másik if elágazással levizsgáljuk az üzenet tartalmát. Ha az üzenet tartalma egyenlő egyel, akkor bezárja a formot, ha viszont mást kapunk vissza akkor egy label-be kiírja, hogy nem sikerült regisztrálni.

## mainForm.cs

A mainForm-nak a konstruktorában paraméterként négy dolgot adunk át. Egy Bitmap-et, ami majd a felhasználónak a képe lesz, egy stringet, ami a jelenlegi állapotát fogja a felhasználónak mutatni, egy Color típusú változót, ami beállítja majd a mainFormon a form betöltésekor az állapot színét, például elérhető állapotban zöld színt adunk át a formnak és utoljára átadjuk a login formban létrehozott Connection objektumot. Ezután létrehoztam egy függvényt, ami beállítja a felhasználó profilképét a mainformon. Ennek a függvénynek csak egy paramétere van, ami egy Bitmap típusú változó lesz. Ez a függvény csupán annyit csinál, hogy a picturebox-nak az Image property-vel egyenlővé teszi. Ezt a függvényt meghívjuk a mainForm-nak a konstruktorában, ahol átadjuk ennek a függvénynek a login vagy az option formtól kapott Bitmapot. Ezután létrehoztam egy függvényt, ami beállítja a felhasználó állapotát. Ennek a függvénynek két bemenő paramétere van, egy string, amit a login vagy az option formtól kapunk meg és egy Color, amit ugyancsak a login vagy az option formtól kapunk meg. Ez a függvény annyit csinál, hogy az állapotot mutató labelnek a Text-jét beállítja a paraméterként kapott állapot névre és az állapotot jelző pictureboxnak a háttérszínét beállítja a paraméterként kapott színre. Ezt a függvényt is meghívjuk a mainform konstruktorában és átadjuk neki a mainformba kapott stringre és Colorra. Ezután létrehoztam egy- egy példányt a chat\_formból és az option\_formból, mivel majd a gombok megnyomásakor ezeket kell megnyitni. Mind a kettőnek átadtam a megkapott Connection példányt. Ha a Chat gomra kattintunk, akkor meghívódik a chat\_form-nak a Show() metódusa, ami előhívja a chat\_formot és ha az Options gombra kattintunk, akkor megjelenik az Options form. Ezenkívül lekötöttem a mainForm-nak a Load eseményét, abból az okból kifolyólag, hogy kipróbáljam, hogy hogy adok hozzá listview form elemhez képet és text-et. Ehhez először létrehoztam egy új példányt az ImageList-ből, amihez először hozzá kelett adni egy képet. Ezután a listview-ban beállítottam, hogy LargeImageList property egynelő legyen a korábban létrehozott ImageList-el. Ezután a listview-nak a View propertyjét beállítottam View.LargeIcon-ra, mivel ha View.Details-re állítom, akkor nem jeleníti meg a képet. Ezt a View property még lehetne View.SmallIcons-ra is állítani, de az a rosszabb szemű felhasználóknak nem lenne jól látható. Ezután létrehozunk egy Listviewitemet, amit majd késöbb hozzáadunk a listview-hoz. Ennek a listview itemnek meg kell adni, hogy milyen indexű képet akarunk vele megjeleníteni és még hozzá kell adni a megjelenítendő szöveget.

## Connection.cs

A Connection osztály fogja kiépíteni és fenntartani a kapcsolatot a szerverrel. Ebből az osztályból csak egyetlen példányt hozunk létre a loginform-on és ezt a példány adjuk át egyik form-ról a másikra. Az osztályban legelőször felveszek egy stringet ipAddress néven, ami a szerver ip címe lesz. Ezt az adattagot privátra állítom, hogy máshonnan ne lehessen módosítani. Ezután létrehoztam egy integert, ami a portot határozza meg. A portnak az 1234 –et adtam meg. Ezután létrehoztam egy új példányt a TcpClientből, ami ebben az esetben publikusnak kell lennie ,mert a formoknak tudnia kell ennek a TcpClientnek a streamjét. Ezután létrehoztam a class konstruktorát, ami paraméterül kap egy onResponse delegate metódust. Erre azért van szükség, hogy tudjuk a kliensen, hogy sikerült-e csatlakozni a szerverre, mert ez a metódus csak akkor adódik át paraméterben, ha sikerült a szerverrre való csatlakozás. Ezután felvettem egy publikus delegate metódust, hogy feltudjak venni egy ilyen típusú adattagot, ezek után fel is vettem ezt a privát onResponse adattagot, aminek majd át tudjuk adni azt a metódust, aminek majd le kell futnia a server válaszakor. A konstruktorban ezek után a legelső, hogy a classon belüli response adattagot egyenlővé tesszük a paraméterként bejött metódussal. Ezután meghívjuk a TcpClientre a Connect() metódust, aminek paraméterben át kell adni az ip címet amire csatlakoznia kell és a portot. Ezután meghívjuk az onResponse metódust, amit paraméterben átadtunk a Connection osztálynak. Ezután létrehoztam egy publikus metódust, ami a tcp kapcsolat lezárásáért lesz a felelős. ebben a metódusban csak annyi fog történni, hogy a tcpClient-nek meghívjuk a Close() metódusát.

## Hash.cs

A Hash osztály végzi el a jelszavak hashelését. Ez az algoritmus egy sha256-os hashet hoz létre, a stringben egy kis módosítással, hogy nehezebb legyen feltörni, vagy ne lehessen két jelszónak ugyanaz a hash-e. Ebben a classban két metódus lesz. Ez a két metódus teljes mértékben megoldja a hashelés folyamatát. Az első metódus egy string visszatérési értékű metódus, abból az okból, hogy egyből lehessen ezt a hashelt stringet más részeknek is használnia. A metódusnak bemenő paraméterként bemegy egy string, ami maga a password, amit hashelnünk kell, ahogy beérkezik a password egyből hozzáadunk egy másik stringet, hogy a továbbiakban az új megalkotott stringet hasheljük meg. A :NET tartalmaz már sha256-os algoritmust. Ezt az algoritmust fogjuk használni. Legelőször létrehozunk egy SHA256 típusú példányt, amit egyenlővé teszünk SHA256Managed.Create() metódussal, ez egy alapértelmezett SHA256-os implementációt hoz létre. Ezután létrehozunk egy byte tömböt, amibe belerakjuk a jelszó stringünket byte-okká alakítva. Ezt az Encoding.UTF8.GetBytes() metódussal végezzük úgy, hogy paraméterként átadjuk az átalakítandó stringet. Ezután létrehozunk még egy byte tömböt, amiben a már hashelt jelszavat fogjuk tárolni. Ezt a byte tömböt egyenlővé tesszük az SHA256- ból létrehozott példánynak a ComputeHash() metódusával, aminek paraméterül megadjuk a byte tömböt ,amit létrehoztunk a jelszóból. A jelszónak a byte tömbjét paraméterként adjuk át ennek a metódusnak. A visszatérési értéknek pedig a metódusban megadjuk a másik metódusnak átadott hashelt tömböt, ami visszaalakítja stringé a már hashelt tömböt. Ezt a folyamatot úgy teszi meg a másik metódus, hogy megkapja a hashelt byte tömböt paraméterként és létrehoz egy StringBuildert, ami felépíti majd a hashelt stringet. Egy for ciklus végig megy a hashelt tömbön és minden egyes tömbelemnél hozzáfűzi a StringBuilder a stringhez az Append() metódussal a jelenlegi tömbelemre meghívott ToString() metódus. A metódus return-el visszaadja a StringBuilder által készített stringet ugyancsak a ToString() metódus meghívásával.

## MySettings.cs

Ez a class fogja tárolni a felhasználónak a beállításait. Három adat lesz tárolva ebben a classban a felhasználóról. Az egyik adata felhasználó felhasználói neve, amit a loginformon állítunk majd be. A másik adat az a felhasználó képe, amit ugyancsak a loginformon állítjuk be egy alapértelmezett képet mivel a bejelentkezésnél muszáj valamilyen képet küldenünk. Ha, ez a kép megváltozik vagy bármelyik adat, akkor újra küldjük a szervernek a felhasználónak az adatait. Ezenfelül lesz még egy adattagja ennek a class-nak, ami a felhasználó állapotát fogja eltárolni integer formátumban. Ehhez majd fel kell venni egy enumerációt, amivel majd meg tudjuk határozni mondjuk a mainFormon, hogy milyen állapot legyen beállítva. Mind a három adattag láthatósága privát, ezért csak metódusokkal lehet elérni őket kívülről.

## INetworkChatCodec.cs

Az INetworkChatCodec egy interface class. Ez a class a felelős a chat kódek adatainak a tárolására, mint példáúl a chat codec neve, elérhető-e a codec, hány bit másodpercenként a mintavételezés az analóg hangból, maga a felvételnek a formátuma és beállításai, az encode és decode során létrejövő byte tömbök. Ez az interface class, akkor lenne fontos leginkább, ha több mint egy codecünk lenne, de nekünk csak egy van, ezért a fontossága elenyésző, de objektum orientáltság szempontjából nagyon is fontos. Ez a class megtalálható magában a NAudio könyvtárban, de kiszedtem egy külön classba, hogy jól látható hgelyen legyen.

## ALawChatCodec.cs

Ez a class is megtalálható magában a NAudio könyvtárban. Ez a class végzi lényegében a hang mintavételezését és decode-olását. A codecekre, azért van szükség, mivel amikor a WaveIn() metódus veszi fel a hangot, akkor a hang felvétele úgy történik, hogy minél jobb legyen a hangminősége. Ez, hogy nagyon jó a hang minősége, ez azt jelenti, hogy a hálózat is nagyon le lesz terhelve, ezért van szükség a decode folyamatára, hogy a hálózat ne legyen leterhelve. A legelső lépésben, amit a classban meg kell valósítani, az az hogy beállítjuk az interface Name property-jét, ezt a get kulcsszóval és a return kulcsszó segítségével tesszük meg. Ezután beállítjuk a BitsPerSecond propertynek, hogy milyen gyakran történjen meg a mintavételezés. Itt a NAudionak az alapmintavételezését felszorozzuk nyolccal, ezzel is javítva egy kicsit a késöbbi hangminőségen. A RecordFormat propertynek átadunk egy WaveFormat-ot, ami majd magát a felvételt fogja szabályozni, a WaveFormatnak meg kell adni paraméterül, hogy mennyi legyen a bit ráta, mennyi legyen a bit mélység és, hogy mono vagy stereo legyen a hang felvétele. Ezután létrehozunk egy Encode metódust, aminek három paramétert kell megadni. Az első paraméter, az az a byte tömb, amit encodeolni kell, egy offsetet kell átadni még, ami meghatározza, hogy hányadik byte-ol kezdődik a hang (a hang általában nullától kezdődik, de lehetnek, ettől eltérő esetek) és harmadik paraméterbe, pedig a tömb méretét kell átadni. A metódusban legelsőzör létrehozunk egy új byte tömböt, ami már az encodeolt hangot fogja tárolni byteokban. Ennek a tömbnek meghatározzuk a méretét, fele akkorára, mint a paraméterként bejött tömböt. Fel kell venni egy integer változót, ami a kimenő tömbnek lesz az indexe. Ezután szükségünk van egy for ciklusra, ami a bejövő paraméterként átadott hosszig megy. A ciklus változót mindig kettővel kell növelnünk. Az encode-ra kialakított tömbnek mindig a plussz egyedik elemébe rakunk bele byte-ot. Ez a byte, ami belerakunk mindig a bemenő byte tömböl jön elemenként eltolva offsettel plusz egyel, a hang kavarodását elkerülve. A for ciklus után return kulcsszóval visszaadjuk az encode-olt tömböt. A következő a Decode() metódus, amivel a kliensen a beérkező byte tömböt fogjuk hanggá átalakítani, ehhez bemenő paraméterként szükségünk van egy byte tömbre egy offset-re, ami általában nulla szokott lenni és a tömbnek a hosszúságára. A metódusban legelőször felveszünk egy byte tömböt pont úgy, mint az Encode() metódusnál, csak itt a tömb hosszúságát felszorozzuk kettővel. Itt is szükségünk van egy indexre, ami a tömbnél fog segíteni nekünk. Ezután kell nekünk egy for ciklus, ami megint csak a tömb végéig megy és egy short változóba mindig kivesszük a bejövő tömb offset plussz n-edik elemét és ezt az elemet adjuk hozzá a dekódolt a tömbhöz olyan módon, hogy byte-á kasztoljuk és megadunk neki két határértéket. A for ciklus után visszatérünk a dekódolt byte tömbbel, ami a waveout már tud majd kezelni.