5. melléklet

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK	Hallgató neve:	**Ugró Dániel Szilárd**
2022	Hallgató törzskönyvi száma:	**T/007145/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Kiberfizikai Rendszerek Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Ugró Dániel Szilárd**
Törzskönyvi száma: T/007145/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Titkosított kommunikáció megvalósítása
Encrypted communication methods

Intézményi konzulens: Légrádi Gábor
Külső konzulens:

Beadási határidő: 2022. május 15.

A záróvizsga tárgyai: Számítógép architektúrák
Big Data és üzleti intelligencia specializáció

A feladat

Készítsen el egy olyan alkalmazást, amellyel a felhasználók (ügyfelek) az alkalmazást (szolgáltatás) biztonságos módon, titkosított üzenetek küldésével tudnak kommunikálni! Térképezze fel, hogy mely területeken, és milyen titkosítást alkalmazva érhet el elfogadható eredményt! Elemezze az elkészítendő rendszer esetleges biztonsági problémáit! Vizsgálja meg, a felhasználható technológiai lehetőségeket! Tervezze meg, és valósítsa meg az alkalmazást! Készítse el a felhasználói és fejlesztői dokumentációt, amelybe beletartozik a felhasznált technológiák és az alkalmazott titkosítás(ok) dokumentációja, valamint továbbfejlesztési tervek az elkészült szolgáltatással kapcsolatban! Az elkészült alkalmazást tesztelje le, és dokumentálja a tesztelési eredményeket!

A dolgozatnak tartalmaznia kell:

- a titkosítás(ok) pontos menetét,
- a követelmények specifikációját,
- az adatbázis technológiák összehasonlító elemzését,
- rendszertervet,
- egyéb választott fejlesztői eszközök ismertetését és azok választásának indoklását,
- a tesztelés terveit és annak eredményeit,
- továbbfejlesztési lehetőségeket.

ÓBUDAI EGYETEM Neumann János Informatikai Kar 1. Budapest

Fleiner R.

Dr. Fleiner Rita
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

Jegyzői Gábor
.....
intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar
7. sz. melléklet

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat / diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban / diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 20 22. 05. 10.

Ugró Dániel
hallgató aláírása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar
9. sz. melléklet

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Ugró Dániel Szilárd

Neptun kód: [redacted]

Tagozat: Esti

Telefon: [redacted]

Levelezési cím (pl: lakcím): [redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Titkosított kommunikáció megvalósítása

Szakdolgozat / Diplomamunka[2] címe angolul:

Encrypted communication methods

Intézményi konzulens: Légrádi Gábor

Külső konzulens:

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2021.09.23.	Téma és cím megbeszélése	Légrádi Gábor
2.	2021.10.07	Feladatlap kitöltése, szakdolgozat vázlatos felépítésének átbeszélése	Légrádi Gábor
3.	2021.11.18.	Szakirodalmi áttekintés megbeszélése	Légrádi Gábor
4.	2021.12.02.	Rendszerterv megbeszélése, szakdolgozat véglegesítése	Légrádi Gábor

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2021. dec. 3

Légrádi Gábor
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar
9. sz. melléklet

KONZULTÁCIÓS NAPLÓ

Hallgató neve:	Neptun kód:	Tagozat:
Ugró Dániel Szilárd	[redacted]	Esti

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Titkosított kommunikáció megvalósítása

Szakdolgozat / Diplomamunka[2] címe angolul:

Encrypted communication methods

Intézményi konzulens: Külső konzulens:

Légrádi Gábor

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.02.24.	A fejlesztés irányának a megbeszélése	Légrádi Gábor
2.	2022.03.17.	A felhasznált technológiák ismertetése és az aktuális alkalmazás bemutatása	Légrádi Gábor
3.	2022.04.28.	Működő alkalmazás bemutatása, a fejlesztői dokumentáció megbeszélése	Légrádi Gábor
4.	2022.05.10.	Az elkészült dokumentáció megbeszélése, szakdolgozat véglegesítése	Légrádi Gábor

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4[3] tantárgy követelményét teljesítette, beszámolóra / védésre[4] bocsátható.

Budapest, 2022. 05. 10.

Légrádi Gábor
intézményi konzulens

[1] Megfelelő aláhúzandó!
[2] Megfelelő aláhúzandó!
[3] Megfelelő aláhúzandó!
[4] Megfelelő aláhúzandó!

TARTALOMJEGYZÉK

1. BEVEZETÉS

Az online kapcsolattartás napjainkra az elsődleges kommunikációs formává vált. A legtöbb kommunikációs platform ugyan elég hangsúlyt fektet a biztonságos adatkommunikációra, azonban maga a szolgáltató is hozzáfér az adatokhoz és rengeteg telemetrikus adatot gyűjtenek a felhasználóikról. Mindezek csökkentik az emberek bizalmát az adatszolgáltatókkal, valamint a használt kommunikációs platformokkal szemben.

A szakdolgozat témája egy olyan alkalmazás (szolgáltatás) megvalósítása, amelyet a felhasználók (ügyfelek) biztonságos módon, titkosított kommunikáció megvalósításával tudnak használni.

1.1. Motiváció

A szakdolgozat elkészítéséhez a motivációt mind a hétköznapjaimból, mind a munkámból merítettem. Emellett mindig is érdekelt az adatok titkosítása és biztonságos kezelése, illetve a szakmai fejlődés lehetőségét látom abban, hogy egy olyan szolgáltatást hozzak létre, ahol a szolgáltató sem fér hozzá az ügyfelek adataihoz.

A környezetemben sokan bizalmatlanok az általuk használt szolgáltatások üzemeltetőivel szemben, manapság már közhely, hogy az általunk szolgáltatások megfigyelnek és profiloznak minket. Ezt a bizalmatlanságot érzem én is.

A munkám során gyakran szóba kerül az adatbiztonságosság, különösen, amikor bizalmas adatok kezeléséről van szó. Az ügyfelek mindig próbálják az üzleti adataikat minél inkább védeni és újra és újra felvetődik, hogy a ránk bízott adataikat milyen módon tároljuk, titkosítjuk.

1.2. Célok

A szakdolgozat keretében egy olyan kommunikációs szolgáltatás elkészítése a célom, ahol a szolgáltató ténylegesen nem fér hozzá a felhasználók adataihoz. Emellett szeretnék hangsúlyt fektetni a rugalmas architektúrára, mind a kód, mind a kialakítandó infrastruktúra tekintetében, hogy mind a szolgáltatást, mind a hozzá kapcsolódó alkalmazást fejleszteni lehessen a jövőben, akár a dokumentumok könnyebb (és jobb) kezelése érdekében.

Végsősoron egy olyan platformot szeretnék létrehozni, amiben a felhasználók megbízhatnak és amely nem figyeli meg őket.

1.3. Felépítés

A szakdolgozatom első felében először egy történelmi áttekintéssel szeretném bemutatni a kommunikáció, majd a titkosítás fejlődését, valamint néhány eljárás – az szakdolgozat terjedelmének figyelembe vételével – részletezését. Ezen szakirodalmi áttekintés során próbáltam a dolgozat szempontjából releváns témákat és megoldásokat előtérbe helyezni.

A történelmi áttekintés után egy hasonló szolgáltatás, a Telegram bemutatásával szemléltetem a téma gyakorlati megvalósítását. Ezután egy tervet készítettem a konkrét egyedi megvalósításomra.

2. SZAKIRODALMI ÁTTEKINTÉS

2.1. A kommunikáció

Napjaink egyik leggyakrabban használt és legáltalánosabb fogalma a kommunikáció. A kommunikáció az információcsere folyamatát jelöli. Az ember az ősidők óta késztetést érez, hogy kommunikáljon a külvilággal. A kommunikáció alapja a közös jelrendszer, ez biztosítja, hogy a kommunikáló felek megértik egymást.

Az emberi beszéd fejlődése hozzávetőleg Kr.e. 100.000 körül indult el [1] és az olyan kommunikációs szimbólumok, mint például a barlangrajzok nagyjából Kr.e. 30.000 körül kezdtek el kifejlődni. [2] A barlangrajzok utáni emberi kommunikáció fejlődéséről a világszerte megtalálható sziklarajzok árulkodnak, amelyek körülbelül Kr.e. 8.000-10.000-ig meghatározó formái az emberi kultúrának. [3]

A kommunikáció fejlődésének következő pontja a piktogramok megjelenése volt, amelyek már rajzolt szimbólumok voltak és az írás elődjének tekinthetők és céljuk a gondolatok közlése volt. A legjelentősebb különbség a korábbi (képi) kommunikációs módokhoz képest, hogy míg azok egy-egy jelenet és esemény megörökítésére szolgáltak, a piktogramok a jelenetek mögött meghúzódó történet közlésére szolgáltak. [4] Később a piktogramokból fejlődtek ki az olyan piktografikus vagy logografikus írásrendszerek is, mint az egyiptomi hieroglif írás. Ezeket nevezzük képírásnak. [5]

Később a képírásos írásrendszerek fejlődtek tovább olyan, a mai napig is meghatározó írásrendszerekké, mint például az ábécé. Az első ábécé a föníciai volt, ebből fejlődött ki a görög majd a görögből a római ábécé. [6]

Az ábécé megjelenését követő fontos esemény volt a könyvnyomtatás megjelenése, még inkább annak elterjedése. A nyomtatás – és ehhez kapcsolódóan az írni-olvasni tudók számának a növekedése a történelem során – lehetővé tette, hogy az információkat megőrizzük és továbbadjuk. A nyomtatás végül a nyomtatott sajtó megjelenésével és elterjedésével megjelent a tömegkommunikáció, aminek a hatására tömegekhez lehetett eljuttatni az információt. [7]

A nyomtatás utáni következő technológiai áttörésnek a távírók megjelenését tekinthetjük. A távírók elektromos jeleket továbbítanak. Ez lehetővé tette, hogy az információt nagy távolságokra nagyon gyorsan eljuttassák. [8] A vezetékes távírót Samuel Morse mutatta be 1838-ban, aki a távírók által elterjedt Morzekód feltalálója és szabadalmaztatója. [9][10]

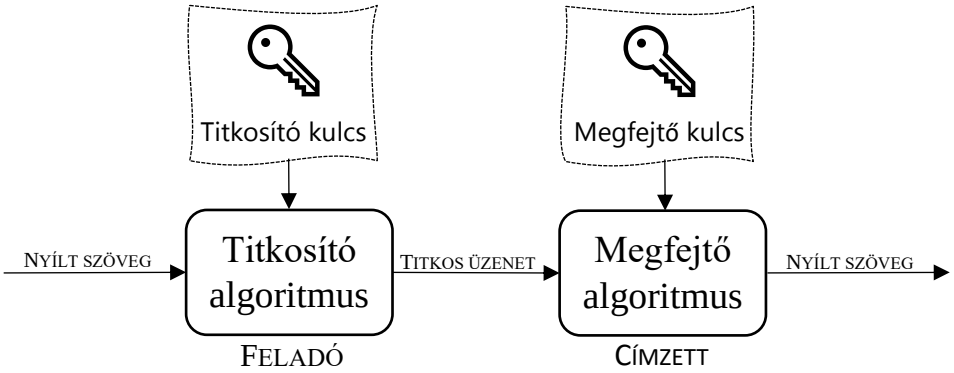
A telefonok a távírók utódjának is tekinthetők, itt azonban már nem pusztán Morzekóddal történő elektromos jelek továbbítása történik, hanem hang továbbvitelére is alkalmas. [11]

A televízió és a rádió egy egyirányú kommunikációs csatorna, vagyis van egy adó és egy vevő, és a vevő nem tud üzenni az adónak. A televízió vált a második világháború utáni világ meghatározó tömegkommunikációs készülékévé.

Az internet 20. század végi, 21. század eleji elterjedése egy egészen új kommunikációs csatornát hozott létre és megváltoztatta a világot. Az interneten népszerű oldalaknak közösség és társadalomformáló erejük van. A felgyorsult információáramlás, a közösségi médiák megjelenése és az adathalászat megjelenése új kihívás elé állította az embereket. Manapság a kommunikációnk jelentős része az interneten keresztül történik és ez rengeteg visszaélésre ad lehetőséget, éppen ezért nagyon fontos az adatok védelme, titkosítása. [12]

2.2. A titkosítás

A titkosítás az az eljárás, aminek célja a bizalmas információk elrejtése az illetéktelen személyek elől. A titkosítás során az információt (nyílt szöveg) egy algoritmus (titkosító eljárás) segítségével átalakítjuk titkosított információvá (titkosított szöveg). Ezután a titkosított információ egy kulcs segítségével válik olvashatóvá. [13][14]



2.2. 1. ábra: a titkosítás folyamata

A titkosítás folyamata:

1. A beérkező információt (nyílt szöveg) egy kulccsal (titkosító kulcs) titkosítjuk (titkosító algoritmus)
2. Az így keletkezett titkosított információt (titkos üzenet) továbbítjuk egy fogadónak
3. A fogadó a beérkező titkosított információt visszafejti (titkos üzenet) egy kulcs (megfejtő kulcs) segítségével
4. Az eljárás végén megkapja az eredeti, titkosítatlan információt, amit így elolvashat vagy feldolgozhat

A kriptográfia a titkosítási rendszerek megtervezésének tudománya, ugyanakkor a kódfejtés (cryptanalysis) az a folyamat, amely során a titkosított információt megpróbáljuk visszafejteni az eredeti, titkosítatlan információra vagy nyílt szövegre a titkosító kulcs hiányában. A kriptológia (cryptology) egy gyűjtőfogalom mind a kriptográfia (cryptography), mind a kódfejtés (cryptanalysis) számára. [15]

Fontos leszögezni, hogy a kódfejtés nem csupán azt jelenti, hogy a támadó hozzáférést szerez a titkosítatlan információhoz. [16]

Ha például a titkosított adathordozóra ráírjuk a visszafejtéshez használt kulcsot, akkor a támadó az adathordozó megszerzése során hozzáfér a kulcshoz is. Ez az eset példázza, hogy maga a titkosítás többről szól, mint pusztán az adatok titkosítására használt algoritmusokról. A kulcsok biztonságos módon történő kezelése ugyanolyan fontos, mint az adatok titkosítása.

2.2.1. A Kerckhoffs-elv

A titkosítási modellek alapelveit Auguste Kerckhoffs fektette 1883-ban a La criptographie militaire című könyvében. [17]

Az elv lényege, hogy a titkosított üzenet biztonsága a kulcs integritásán múljon és ne az eljárás módján. Vagyis, ha magát az eljárás módszertanát megszerzik (vagy netán publikus) akkor se lehessen az üzeneteket visszafejteni a kulcs ismerete nélkül. Manapság ez már alapvetés, amikor a titkosító eljárások és protokollok publikusan ismertek és mindenki által használtak. [18]

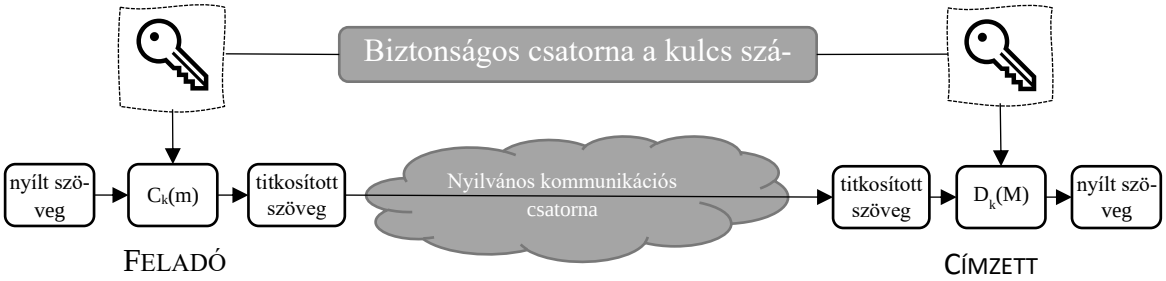
Kerckhoffs a fenti gondolatát hat pontban foglalta össze, melyek a következők: [19]

1. Ha a rendszer matematikailag nem is feltörhetetlen, a gyakorlatban annak kell lennie.
    - Ha a rendszer fel is törhető például brute force használatával, amennyiben az lehetetlenül sok időt vesz igénybe, adott esetben több millió évet, akkor a gyakorlatban feltörhetetlennek minősül.
2. Az eljárásnak nem kötelező titkosnak lennie, nem lehet probléma, ha illetéktelenek kezébe kerül.
    - Tehát a rendszer biztonsága a kulcs bizalmasságtól függ.
3. A felhasznált kulcsnak megjegyezhetőnek és kommunikálhatónak kell lennie anélkül is, hogy azokat feljegyeznék, valamint akármikor cserélhetőnek is kell lennie.
4. A kulcsnak táviratban is továbbíthatónak kell lennie.
5. A titkosító rendszernek hordozhatónak és csupán pár személy által üzemeltethetőnek kell lennie.
6. A rendszernek egyszerűnek kell lennie, ne igényelje listányi szabályok betartását.

2.2.2. Szimmetrikus kulcsú módszerek

A szimmetrikus titkosítás a legegyszerűbb titkosítási módszer. A szimmetrikus titkosítás esetében ugyanaz a kulccsal lehet visszafejteni a szöveget, amivel az titkosítva lett. Ezek a legrégebbi módszerek és kulcsfontosságú, hogy a kulcs titkos maradjon, éppen ezért nevezik még *titkos kulcsú titkosításnak* vagy *hagyományos titkosításnak*. [20]

Ennél a fajta titkosításnál a módszer legkritikusabb része, hogy a titkosításhoz használt kulcsot valamilyen biztonságos, nem lehallgatható csatornán juttassuk el a másik félhez, tehát ne ugyanúgy, ahogy az üzenetet.



2.2.2. 1. ábra: A szimmetrikus kulcsú titkosítás [21]

Amennyiben *m* a nyílt szöveg, *k* a titkosításhoz felhasznált kulcs, *M* pedig a titkosított szöveg, akkor a fenti ábrában szereplő egyenletek a következők:

$M = C_k(m)$, ahol C_k a titkosítást végző algoritmus

$m = D_k(M)$, ahol D_k a visszafejtést végző algoritmus

2.2.2.1. A Caesar módszer

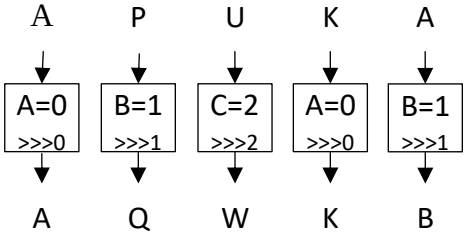
Az egyik legrégebbi és legegyszerűbb titkosítási eljárás a Caesar-kód vagy Caesar-rejtjel. A nevét onnan kapta, hogy egy római történész, Suetonius szerint Julius Caesar használta. A módszere egyszerű, minden betűt k mértékben kell eltolni az ABC-ben. Hagyományosan k=3, tehát 3 betűnyi eltolást kell használni. Ebben az esetben az a szó, hogy APA arra a szóra lesz transzformálva, hogy DSD. Mert a latin ábécében a D három betűnyi távolságra található az A-tól, ahogy az S is a P-től. [22]

Később ezt a titkosítást felhasználták más titkosításoknál is, például a Vigenère vagy az annál modernebb ROT13 titkosításba. [23]

2.2.2.2. A Vigenère módszer

A következő fontos fejlesztés a titkosításban a 16. században következett be. Ezt az olasz Giovan Battista Bellaso találta ki, de később tévesen a francia Blaise de Vigenèrenek tulajdonították és ezért azóta is az ő nevét viseli.

A módszer nagyon hasonló a Caesar kódoláshoz, de ezesetben a betűk nem hárommal vannak eltolva, hanem ezt az eltolást egy kulcs határozza meg, ami egy betűhalmaz. Ha például a kulcs ABC, akkor a szöveg eltolására használt kulcsok az 0, 1, 2, mert az A az nulla távolságra van az A-tól, a B egy távolságnyira, a C pedig kettőre. Ez a minta addig ismétlődik, amíg a teljes szöveg titkosítva nem lesz. Például az a szó, hogy APUKA a következőre lesz átalakítva: AQWKB. [24]



2.2.2.2. 1. ábra: a Vigenère módszer [25]

2.2.2.3. One-Time Pad módszer

A One-Time Pad egy mai napig rendkívül biztonságos titkosítási eljárás, ami az adat titkosításához egy ugyanakkora méretű kulcsot használ, mint maga az adat mérete. Ez azt jelenti, hogyha van például 1 GB adatunk, akkor annak a titkosításához egy 1 GB méretű kulcsra lesz szükségünk. [26]

A One-Time Pad elve a kizáró vagy, vagyis a XOR műveleten alapul. Ha a titkosítandó szöveg „P" és a véletlenszerű kulcs a „K" és a titkosított szöveg a „C", és magát a XOR műveletet a „⊕" szimbólum jelöli, akkor a titkosítás a következő formában írható fel:

$$C = P \oplus K$$

A One-Time Pad a többi szimmetrikus titkosításhoz hasonlóan ugyanazt a kulcsot használja a titkosított üzenet visszafejtéséhez, amit a titkosításhoz is, ennek megfelelően a visszafejtés a következő formában írható fel:

$$P = C \oplus K$$

A One-Time Pad esetében a legfontosabb, hogy a titkosító kulcsot csak egyszer szabad felhasználni. [27]

2.2.2.4. AES

Az AES (Advanced Encryption Standard) jelenleg az egyik legnépszerűbb, szimmetrikus kulcsú titkosítás. Az AES-re a titkosítási módszereket pályázaton választotta ki a U.S. National Institute of Standards and Technology. A győztes a Rijndael titkosítás lett. Ez egy helyettesítő és lineáris transzformációkat használó módszer. Ezt a pályázatot az akkor már több, mint 20 éves DES (Data Encryption Standard) kiváltására írták ki. [28]

Az AES titkosítás elve, hogy 128, 192 vagy 256 bit méretű blokkokat titkosít. Amennyiben a titkosítandó adat kisebb, megnöveli a blokkmérethez és úgy titkosítja. [29]

Az blokkot először egy 4×4 méretű, elemenként 1 byte méretű (összesen 16 byte) mátrixba rendezi. Ez a tömb az úgynevezett „*internal state*" vagy „*state*". Az eljárás ezeken a byte-okon, sorokon és oszlopokon végez műveleteket. [30]

Az eljárás lépései: [31]

- **AddRoundKey**
  - XOR művelet végrehajtása a round key-vel az internal state változón
- Ciklusonkénti lépések
  - **SubBytes**
    - Minden byte helyettesítése a Rijndale S-Box szerint
  - **ShiftRows**
    - Az internal state sorainak körkörös eltolása egy meghatározott értékkel
    - Amennyiben ez az érték 1, úgy a sorvégi elem a sor elején fog megjelenni

- **MixColumns**
  - Lineáris transzformáció végrehajtása a state mind a négy oszlopán

2.2.3. Aszimmetrikus kulcsú módszerek

A szimmetrikus kulcsú módszereknél alapvetés, hogy van egy közös titka a küldőnek és a fogadónak, ehhez azonban szükség van egy biztonságos csatornára, amin keresztül meg tudják egymással osztani a kulcsot és mellette pedig egy nyilvános csatornán kerül közlésre a titkosított üzenet. [32]

Az aszimmetrikus kulcsú módszereknél viszont ugyanazt a csatornát lehet használni a kulcs közzétételére, mint az üzenetére. Ehhez két kulcsot kell használni. Az egyikkel az üzenet titkosításra kerül, ez az úgy nevezett *nyilvános kulcs*. A másikat pedig az üzenet visszafejtésére kell használni, ez az ú.n. *titkos kulcs,* vagy *privát kulcs* és a fogadó fél feladata, hogy ez titkos maradjon. [33]

A nyilvános kulcs megosztásra kerül mindenkivel, aki üzenetet szeretne küldeni. A nyilvános kulcs a titkos kulcsból generálódik, azonban a titkos kulcsot nem lehet a nyilvános kulcsból előállítani. Tehát míg a nyilvános kulcs könnyedén előállítható a titkos kulcs ismeretében, ez fordítva nem igaz, a titkos kulcsot nem lehet előállítani a nyilvános kulcsból. [34]

2.2.3.1. PKI - Nyilvános Kulcsú Infrastruktúra

A PKI (*Public Key Infrastructure*) célja az információ biztonságos közlésének a megkönnyítése olyan módon, hogy az megfelelően hiteles, bizalmas, letagadhatatlan, teljes legyen. [35]

A publikus kulcsú infrastruktúra alapfeltétele az elektronikus aláírásnak és a kulcsok menedzselésének.

„Nyíltkulcsú infrastruktúra alatt az olyan rendszert vagy rendszereket értjük, amelyeknek szolgáltatásai – és eszközei – az aszimmetrikus kriptográfián alapulnak.” [36]

2.2.3.2. Diffie-Hellman eljárás

A Diffie-Hellman nyilvános kulcscseréjű eljárást 1976-ban publikálta Diffie, Hellman és Merkle.[37] Bár voltak erre korábban is megoldások, ezek mind katonai titokként voltak kezelve.[38]

Az alapelv az, hogy legyen a küldő és fogadó félnek egy közös titka anélkül, hogy előre egyeztetniük kéne. Ehhez használniuk kell egy publikus kulcsot, amellyel kommunikálni fognak. A kulcscsere eljárás végére mindketten rendelkezni fognak ugyanazzal a titkosító kulccsal, amit titokban kell tartaniuk. [39]

Az eljárás a következőképp működik két kommunikáló fél esetén:

- *Alice* és *Bob* kommunikál

- *Alice* előállít két nagy speciális prímszámot: **p** és **g**
  - p = 19
  - g = 23
- Mindketten előállítanak egy-egy hasonló méretű véletlenszámot: **a** és **b**
  - a = 6
  - b = 7
- *Alice* üzen *Bobnak*:
  - $A = g^a \bmod p$
  - $A = 23^6 \bmod 19 = 11$
- *Bob* üzen *Alice-nak*:
  - $B = g^b \bmod p$
  - $B = 23^7 \bmod 19 = 6$
- *Alice* kiszámolja **s**-t:
  - $s = B^a \bmod p$
  - $s = 6^6 \bmod 19 = 11$
- *Bob* kiszámolja s-t:
  - $s = A^b \bmod p$
  - $s = 11^7 \bmod 19 = 11$
- *Alice* és *Bob* megegyeztek egy közös titokról, 11-ről

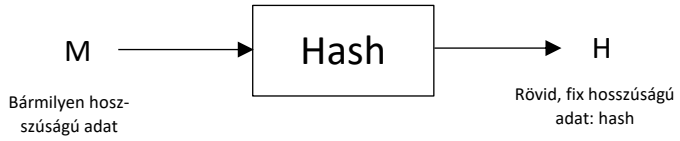
2.2.3.3. RSA

Az RSA (Rivest-Shamir-Adleman) 1977-ben került publikálásra, és ez volt az első eljárás, amely képes volt a publikus kulcsú titkosításra. Az RSA esetében két kulcsa van mindkét kommunikáló félnek: egy titkos és egy publikus. [40]

Az RSA az aszimmetrikus titkosításon túl használható még üzenetek hitelesítésére digitális aláírások elkészítésével. A digitális aláírás esetében a küldő a privát kulcsával aláír egy üzenetet, majd ezt az aláírást a publikus kulcs ismeretében mindenki validálhatja.[41]

2.2.4. Hash függvények

A hash függvények feladata a titkosítással szemben az integritásvédelem, az adatok hitelesítése. Feladata, hogy egy bármilyen hosszú adatot adott hosszúságúra képezzen le. Az így kapott adat neve a hash. [42]



2.2.4. 1. ábra: A hash függvények általános működése [43]

2.2.5. Hibrid módszerek

A korábbiakban a szimmetrikus és aszimmetrikus titkosítási módszerek kerültek kifejtésre. Ez a két féle megoldás viszont egymás kiegészítésére is használható. Az aszimmetrikus módszerek rendszerint lassabbak, mint a szimmetrikus módszerek. Emiatt az aszimmetrikus módszert lehet arra használni, hogy a szimmetrikus módszerrel titkosított üzenethez szükséges kulcsot titkosítsa, majd az üzenettel együtt kerüljön elküldésre. [44]
Így működik többek között a TLS, SSH, vagy az OpenPGP.

2.3. Tanulságok

A szakirodalmi áttekintés több tanulsággal is szolgált a szakdolgozat elkészítése közben. Egyrészt sokat tanultam magukról a használandó technológiákról, valamint a hiteles források használatáról. Éppen ezért fontosnak tartom kiemelni, hogy a helyesen megválasztott szakirodalom meghatározó a dolgozat szempontjából. Az áttekintés elkészítését nagymértékben megkönnyítette volna, ha már az elején rátalálok a jól megírt, közérthető szakirodalomra. Ezen tapasztalatok miatt törekedtem végig az átgondolt struktúra felépítésére és a közérthető fogalmazásra.

3. HASONLÓ SZOLGÁLTATÁSOK

3.1. Telegram Messenger

A Telegram egy végpontok közötti titkosítást használó kommunikációs szolgáltatás, amely képes videó és hanghívások (VoIP) lebonyolítására. A szolgáltatás 2013-ban jelent meg. A rendszer elsősorban azzal az ígérettel nyerte meg a felhasználóit, hogy nem lehet őket megfigyelni az alkalmazás használata közben. [45]

A platform alapítója Nikolai Durov, aki 2006-ban megalapította az orosz közösségi oldalt, a VKontakte-t, majd 2013-ban a Telegram Messengert.

Az alkalmazás használatához telefonszámmal lehet regisztrálni. A regisztráció megerősítéséhez SMS-ben kap egy kódot a felhasználó. Új eszköz aktiválása esetén alapértelmezés szerint a Telegramon belüli üzenetben érkezik meg az aktiváló kód.

Az alkalmazás elérhető:

- Böngészőből
- Asztali alkalmazásként Windowsra, Linuxra és macOS-re
- Androidra
- iPhone-ra és iPad-re egyaránt

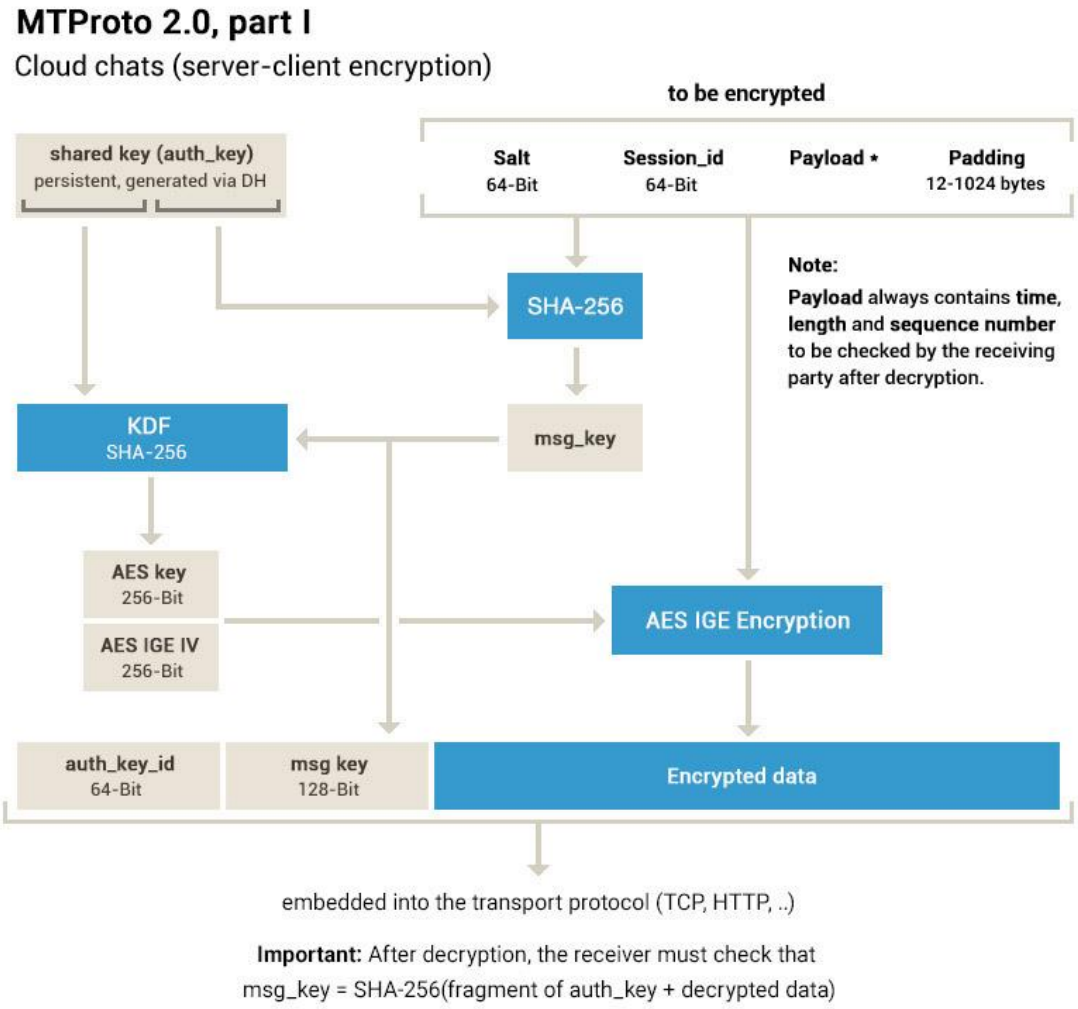
3.1.1. Funkcionalitás

A teljesség igénye nélkül a fontosabb funkciók: [46]

- Egyszerű szöveges üzenetek küldése
- Videó- és hanghívás
- Csoportos üzenőszobák létrehozása
- Üzenetek közötti keresés
    - Szöveges keresés
    - Dátum szerinti keresés
- Különböző médiaformátumok elküldése
    - Képek
    - Videók
    - Egyéb bináris állományok, fájlok
    - Matricák
    - GIF-ek
    - Hangüzenetek
- Az elküldött állományok csoportosítva visszanézhetők
    - Az elküldött képek az adott csetnél a képek között jelennek meg
    - Az elküldött videók az adott csetnél a videók között jelennek meg
    - Stb.

3.1.2 Titkosítási eljárás – MTProto

A szolgáltatás egy egyedi, az alapító, Nikolai Durov által kifejlesztett titkosítási protokollt használ. Eleinte ez az MTProto volt, később ezt lecserélték az MTProto 2.0-ra. Ez egy szimmetrikus titkosítási eljárás, amely egy 256 bites AES titkosítást, egy 2048 bites RSA titkosítást és Diffie-Hellman publikus kulcscserét alkalmaz. [47]



3.1.2. 1. ábra: Az MTProto 2.0 titkosítási eljárás sémája [48]

3.1.2.1. Terminológia

A legfontosabb elnevezések: [49]

- **Authorization Key (auth_key)**
  - Egy 2048 bites kulcs, amin a kliens eszköz és a szerver osztozik.
  - Regisztrációkor generálódik a kliensnél a Diffie-Hellman kulcscserével
  - Nem kerül elküldésre a hálózaton
- **Server Key**

    - 2048 bites RSA kulcs, amivel a szerver digitálisan aláírja a saját üzeneteit a regisztrációs folyamat és az authorizációs kulcs generálása során
- **Key Identifier (auth_key_id)**
    - A 64 alsó bit (LSB vagy least significant bit) az auth_key SHA1 hashéből
    - Arra szolgál, hogy a használt authorizációs kulcsot azonosítsa
- **Session**
    - Véletlenszerűen generált 64 bites szám, ami a különböző sessionök megkülönböztetésére szolgál
- **Server Salt**
    - Egy véletlenszerű 64 bites szám, ami 30 percenként cserélődik (minden session esetében külön-külön)
- **Message Identifier (msg_id)**
    - Egy 64 bites szám, amivel egyénileg azonosíthatóvá válik az üzenet a sessionön belül
- **Message Key (msg_key)**
    - A középső 128 bit az üzenet SHA-256 hashéből
- **Content-related Message**
    - Az üzenetek „explicit” nyugtázást igényelnek
    - Ebbe beletartozik minden felhasználói üzenet és a service üzenetek egy része is
    - Tulajdonképpen minden, leszámítva a konténereket és nyugtázásokat
- **Message Sequence Number**
    - Egy 32 bites szám, ami egyenlő a content-related üzenetek számosságának a duplájával (azok, amik nyugtázást igényelnek és nem konténerek)
- **Multipart message vagy Container**
    - A konténer több üzenetet is tartalmaz és több kliens-szerver hívást csinál egyszerre http kapcsolaton keresztül
- **Internal (cryptographic) Header**
    - 16 byte méretű header, ami az üzenet vagy konténer titkosítása előtt kerül hozzáadásra
    - Tartalmazza a server saltot (64 bit) és a sessiont (64 bit)
- **External (cryptographic) Header**
    - 24 byte méretű header, a már titkosított üzenet vagy konténer elé kerül
    - Tartalmazza az auth_key_id-t (64 bit) és a msg_key-t (128 bit)
- **Payload**
    - External header és a titkosított üzenet vagy konténer

3.1.3. MTProto transport

Az MTProto transport egy olyan szállítási rétegbeli protokoll, amely a meglévő szállítási rétegben létező protokollokra épül. Küldés előtt a csomag egy másodlagos szállítási

rétegbeli headert kap. Az ilyen módon küldött csomagokat a szerver megkülönbözteti a többi protokollt használó csomagoktól, mint például a HTTP-től. [50]

Az MTProto transport protokollok listája: [51]

- Abridged
- Intermediate
- Padded Intermediate
- Full

4. TERVEZÉS

A tervezés és megvalósítás során igyekeztem olyan technológiákat választani, amelyeket a munkám során és az egyetemen is kipróbáltam és alkalmaztam. Emellett fontos szempont volt az egyes technológiák népszerűsége, vagyis a mögötte álló közösség mérete.

Az alkalmazást alapvetően a webes kiszolgálókra vagy szerverekre (backend) és a kliensekre (frontend) lehet elkülöníteni. A szerverek feladata a kérések kiszolgálása, a klienseké pedig a kérések elküldése. A kommunikáció HTTPS-en és WebSocketekkel fog történni.

4.1. Backend technológia

Backend technológiánk a Microsoft által fejlesztett ASP.NET Core 6.0-t választottam. Ezt több szempont alapján választottam ki.

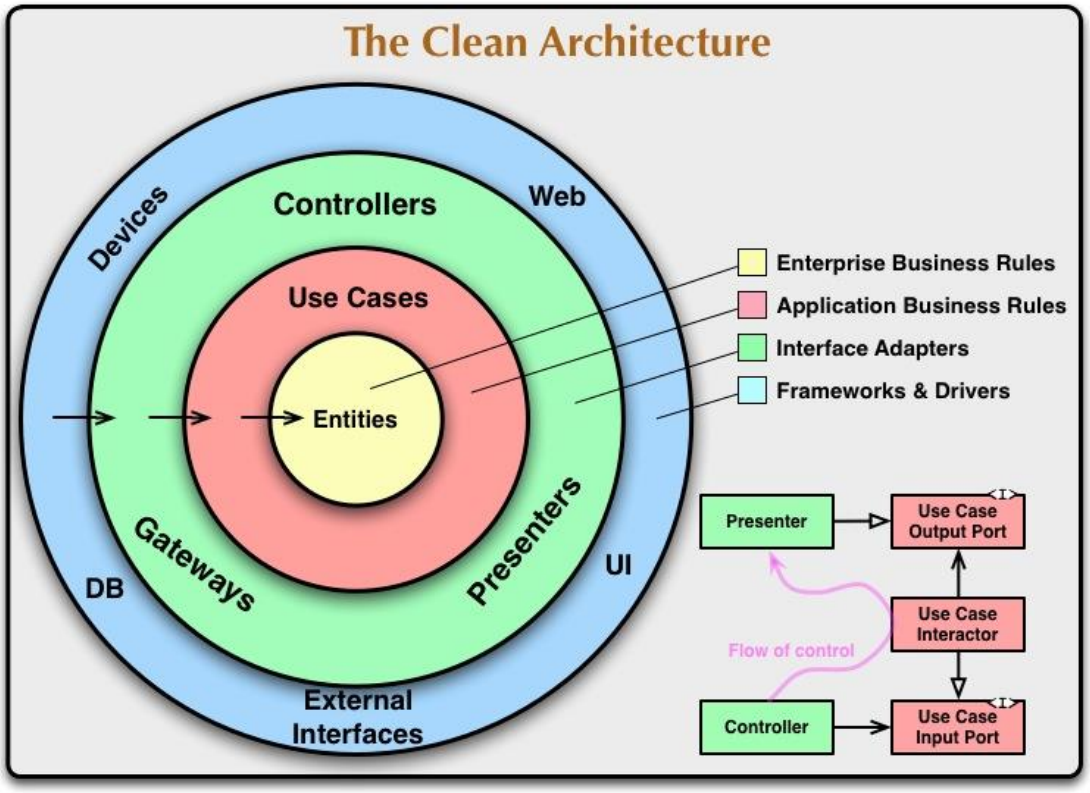
- A .NET 6.0 az úgynevezett Long Time Supported (LTS) verzió, és tovább lesz támogatott, mint a .NET 7 [52]
- Programozási nyelvek közül a C#-ot találtam a számomra leginkább megfelelőnek
- Kényelmesen és könnyen tudok API-t építeni ezzel a technológiával
    - Emellett az API-ból könnyű generált állományt létrehozni a klienshez, ami megkönnyíti a fejlesztést, mert tartalmazza az API hívásokhoz a kódot és a hozzájuk tartozó modelleket

4.1.1. Architektúra

A backend architektúrához az úgynevezett Clean Architecture felépítést fogom követni. A Clean Architecture-t egyfajta hagyma-szerű felépítésű architektúraként is értelmezhetjük.

Az architektúra lényege többek között, hogy a függőségeket minél jobban fel lehessen számolni a különböző rétegek között. Az absztrakció C#-pal jellemzően interfészek és implementációjuk által történik. Az architektúra továbbá növeli a tesztelhetőséget. Például a „Use Cases” réteg csupán az alkalmazás-specifikus üzleti logikákat tartalmazza, nem rendelkezik hozzáféréssel az adatokhoz, mert azok az „Entities” rétegből, egy absztrakción keresztül vannak elérve. [53] Így az alkalmazás-specifikus üzleti logikát külön lehet tesztelni az adatelérési logikától, aminek az implementációja lehet akármi: Entity Framework, MongoDB ORM, szöveges fájl beolvasása, API kérések eredményei, vagy a tesztelés során az absztrakció (interfész) mockolt (vagyis „hamis adatokkal feltöltött”) verziója.

Az előbb tárgyalt okok miatt az adatelérési réteg bármilyen implementációt kaphat, így, ha egy korábban adatbázisból érkező adat a továbbiakban egy HTTP kérés eredményeként egy API-ból származik, akkor sem kell a „Use Cases” réteget frissíteni.



4.1.1. 1. ábra: A Clean Architecture sémája [54]

4.1.2. Adattárolás

Az üzenetek és az adatok tárolását NoSQL-ben szeretném megvalósítani. Bár a szakdolgozat keretein belül ennek nem feltétlenül lesz jelentősége, de a NoSQL horizontálisan skálázható, így a jövőben nagyobb terhelést is képes lenne elviselni. Ehhez a MongoDB-t szeretném használni. A MongoDB-s ORM-nek a hivatalos MongoDB drivert szeretném használni. [55]

Ennek egy lehetséges alternatívája lehetne például a Microsoft SQL Server. Ehhez ORM-nek a Entity Framework Core lenne használva. [56]

A MongoDB és Microsoft SQL rövid összehasonlítása: [57]

	Microsoft SQL	***MongoDB***
Adattárolási modell	Táblák, fix sorokkal és oszlopokkal	JSON dokumentumok
Sémák	Fix	Rugalmas
Skálázhatóság	Vertikális	Horizontális
Hivatalos ORM	Entity Framework	MongoDB Driver

4.2. Frontend

A kliensoldali megvalósítás Angularral[58] fog történni. A felülethez az Angular Materialt [59] szeretném használni. Az Angular egy JavaScript framework, amit a Google fejleszt és tart karban, az Angular Material pedig az általuk az Angularhoz készített UI könyvtár, ami tartalmazza a leggyakrabban használt komponenseket: gombok, checkboxok, gridek, listák, dátumválasztó, stb.

Kliensoldalon az úgynevezett „flat architektúrát” [60] szeretném követni. Ez tulajdonképpen a kliensoldali állományok strukturálását jelenti olyan módon, hogy az minél inkább skálázható és átlátható legyen. Ez nem egy konkrét felépítés, inkább egyféle útmutató, amihez minél közelebb szeretném tartani az állományok strukturálását.

Végsősoron azért döntöttem az Angular mellett, mert támogatja a kétoldalú adatkötést, van benne routing és tapasztalatom által már magabiztosabban tudom használni, mint más technológiákat, mint például a Reactet.

Egy esetleges asztali változathoz egy Electronos alkalmazást hoznék létre, amihez a frontend oldali kód jelentős része újrahasznosíthatóvá válna.

4.3. Verziókezelés

A verziókezelés mára már mondhatni iparági sztenderddé vált. A verziókezelés segít a kód karbantartásában és a változások nyomon követésében. Ez manapság a szoftverfejlesztés integráns része, enélkül aligha képzelhető el a szoftverfejlesztés. Az egyik legnépszerűbb verziókezelő a Git.

A Git egy nyílt forráskódú, decentralizált, másképpen elosztott verziókezelőszoftver, vagyis van egy központi szerver (ez a remote) és egy local verzió. Nemlineáris fejlesztés támogatására hozták létre és támogatja a brancheket, vagyis ágakat. Ezek az ágak a létrehozásuk pillanatában az ősük pillanatképeként foghatók fel, lehetővé téve, hogy a kód olyan módon történő fejlesztését, hogy azzal a fő ágban (régen *master* újabban *main*) lévő kódot egészen addig ne kelljen módosítani, amíg az a kód elfogadásra nem került. [61]

Verziókezelőnek Gitet választottam, amit az Azure Devops Services[62] platformon fogok használni.

Az Azure DevOps Services egy Microsoft termék, amely verziókezelés mellett többek között projektmenedzsmentet CI/CD build folyamatok kezelését, követelménykezelést, tesztelést stb. tesz lehetővé. Tulajdonképpen az Azure DevOps Services funkcionalitása lefedi egy alkalmazás teljes életciklusát.

Azért esett erre a szolgáltatásra a választásom, mert támogatja a Git verziókezelőt, az automatizált buildeket és – jelenleg – mindezt ingyen nyújtja 5 felhasználóig. [63]

Felhasznált funkcionalitás:

- Azure Pipelines
    - Az automatizált buildeket, valamint telepítéseket szeretném ezzel végezni
- Azure Boards

    - A feladatok menedzselésére fogom használni
- Azure Artifacts
    - Az alkalmazás függőségeinek tárolására, vagy egyedi csomagok (pl.: saját nuget library létrehozásához) tárolására lehet használni

4.4. Hosting

Az alkalmazást hostingjához konkrét szolgáltatót még nem választottam.

4.4.1. Docker

A Docker egy olyan konténerizációs technológia, ami operációs rendszer szintű virtualizációval képes futtatni a szoftvereket konténerekben. Minden konténert egyetlen operációs rendszer kernel működted, emiatt alacsonyabb az erőforrás igényük, mint a virtuális gépeknek. A konténerek egymástól elkülönített „csomagok" és minden konténer a saját maga szoftverét tartalmazza. [64]

A Docker esetében egy képet (image) hozunk létre a konténerizált alkalmazásból, és az abból futtatott példányokat nevezzük konténereknek. A konténerek az éppen futó alkalmazást jelölik és amikor az alkalmazás futása véget ért, akkor törlődnek. Ezáltal törlik a hozzá tartozó fájlokat is.

A konténerizált technológiák esetében fontos, hogy az alkalmazások úgynevezett „*stateless*" alkalmazások legyenek, vagyis az állapotuk ne változzon. Ez azt is jelenti, hogy az alkalmazásnak nem ajánlott semmiféle lokálisan végrehajtott fájlműveletekre támaszkodnia. Éppen emiatt a konténer a felhőben, helyi gépen, bárhol futtatható a megfelelő konténerizációs technológia jelenlétében. Ennek egyik előnye, hogy az alkalmazás már a fejlesztés során ugyanolyan környezetben futhat, mint ahol a végleges változat futtatása során is jelen lesz. Egy másik fontos előnye, hogy bármilyen hiba esetén jelentősen egyszerűbb a visszaállás egy korábbi változatra.

Egy életszerű példa a Docker előnyére:

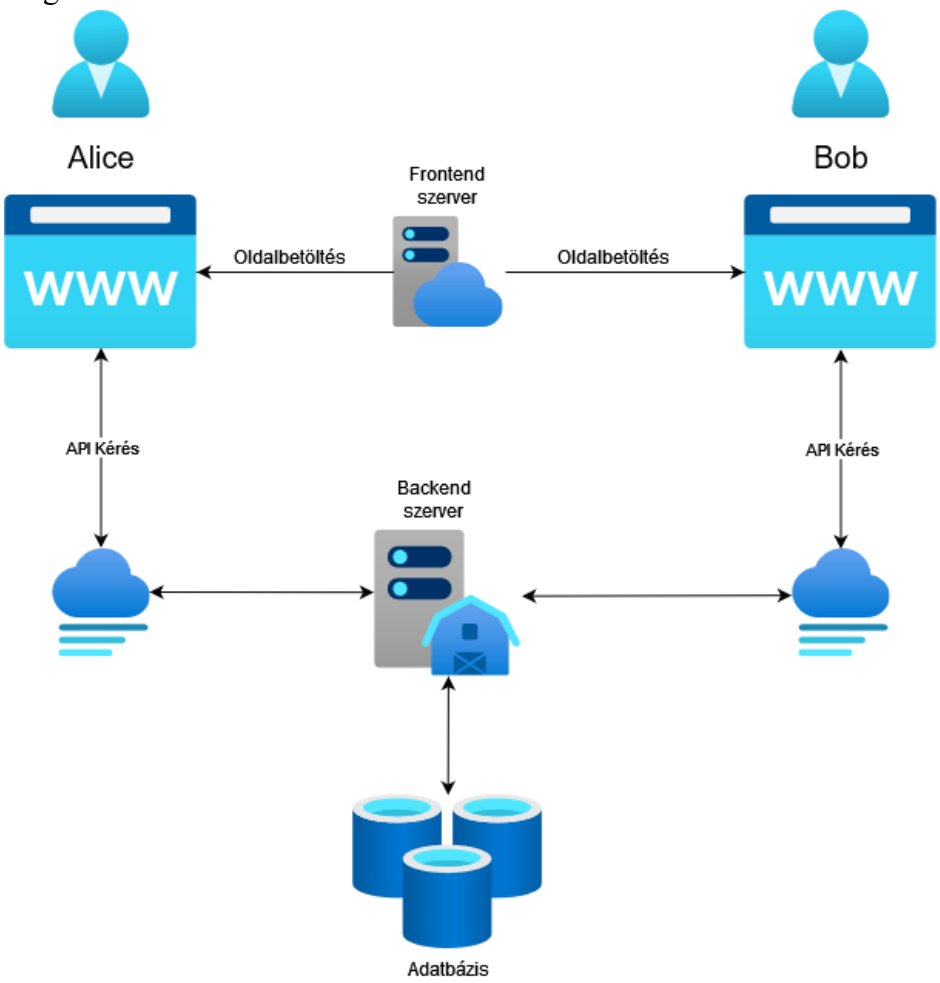
- A v1.1.1-es verzióról tudjuk, hogy stabil
- A v1.2.1-es verzió telepítése után hibát jeleznek
- Ezután az ismert stabil verzióra történő visszaállás (ideális esetben, például azonos adatbázis sémát feltételezve) gyorsabb és könnyebb, hiszen a *képek* úgy vannak létrehozva, hogy bárhol futtathatók legyenek, tartalmaznak a futásukhoz szükséges minden komponenst

4.5. Vázlatos rendszerterv

A már korábban említett módon szeretném kialakítani a szolgáltatást. Lesz egy backend és egy frontend szerver. A backend az API, WebSocket kérésekkel fog foglalkozni, míg a frontend feladata, hogy egy statikus oldalt adjon a böngészőnek. A Single Page

Application lényege, hogy a frontend egyszeri oldalbetöltés után nem töltődik be, csupán a szükséges adatok módosulnak.

Éppen ezért a frontend szerver nem kommunikál a backend szerverrel, sem semmi mással a kialakítandó infrastruktúrával. Egyetlen feladata, hogy a statikus állományokat a böngésző számára szolgáltassa. Ezután minden kérés a backend szerver irányába a böngészőből fog indulni.



4.5. 1. ábra: A fenti ábrán Alice kommunikál Bobbal.

Az ábrán azt látható, ahogy *Alice* kommunikál *Bobbal*. Ehhez a böngészőjük betölti a statikus oldalt a *frontend szerverről*. Innentől a *backend szerverrel* a böngészőjük fog kommunikálni. Minden *API kérés*, *WebSocket kérés* az interneten keresztül jut el a *backend szerverhez*, és az továbbítja az üzeneteket a másiknak.

Tehát Alice böngészője küld egy üzenetet a szervernek, ami továbbítja azt Bob számára.

4.6. Időterv, módszer

A szakdolgozat elkészítéséhez a SCRUM módszert választottam, mivel a munkavégzésem során már hozzászoktam ehhez a metodológiához. A módszer alapja, hogy a munkavégzés sprintekben történik és a feladatokat sztoripontok alapján van felosztva.

Jelen esetben ez a módszertan egy kissé módosulni fog, én leszek a SCRUM master és a fejlesztő, a konzulens pedig a product owner. Ehhez a Azure DevOps Server Boards oldalát fogom használni.

A feladatot a tervezés során az előre megjelölt célokat epicekben terveztem meg és sztoripontok helyett csupán a sprinteket jelöltem meg rá. A sprintek kéthetesek lesznek.

A fejlesztést ezen kívül a *Test Driven Development vagy TDD* szoftverfejlesztési folyamattal tervezem végig vinni. Ez egy olyan szoftverfejlesztési folyamat, amely könnyen beilleszthető a SCRUM módszertana mellé, mert egy rövid fejlesztési ciklus ismételgetésén alapul. A *TDD* esetében a teszteseteket hamarabb fogalmazzuk és írjuk meg, mint az adott funkciót végrehajtó kódot magát. A fejlesztési módszer előnye, hogy segíti a program megtervezését, valamint a tesztesetek nagy száma segít az új, hibás kódok kiszűrésében. [65]

Az alábbi időterv még csupán egy vázlatos időterv, amihez képest csúszások előfordulhatnak. Valamint a feladatok prioritása a fejlesztési folyamat során változhat.

1. Sprint	Fejlesztői környezet kialakítása, Felhasznált libraryk kijelölése, Proof of Conceptek elkészítése
2. Sprint	Backend létrehozása, Frontend létrehozása, Tesztesetek elkészítése
3. Sprint	Authentikáció Titkosítás megvalósítása a proof of conceptek tapasztalatai alapján
4. Sprint	Cset alapfunkciók létrehozása: Szobák létrehozása, üzenetküldés Smoke tesztelés
5. Sprint	Smoke tesztelés
6. Sprint	Hibajavítás
7. Sprint	Hibajavítás

4.7. Alkalmazhatóság és továbbfejlesztési lehetőség

A szolgáltatás a szakdolgozat kereteiben elsősorban a független – semmilyen nagyvállalathoz nem kötődő – kommunikációs platformot keresők számára nyújthat megoldást.

A megoldás során végig próbálom szem előtt tartani a szolgáltatás bővíthetőségét és továbbfejlesztési lehetőségeit. Egy megfelelő dokumentum-kezelő megoldással, valamint egy testreszabható munkafolyamat-kezelő funkcionalitással a kommunikációs platformmal integrálva már nagyvállalatok számára egyféle Virtual Data Room[66] megoldásként is használhatóvá válna, aminek a segítségével menedzselni tudnák az ügymeneteiket.

Ezen kívül az egyik legfontosabb továbbfejlesztési lehetőséget a keresés megvalósításában látom.

Az alkalmazás a későbbiekben konténerizálásra kerül, tehát úgy tervezem továbbfejleszteni, hogy minden egysége fusson egy Docker konténerben. Lokális környezetben ehhez a Docker Compose-t lehet használni, ami a Dockerhez tulajdonképpen egy eszköz. A Docker Compose YAML fájlokat használ a konténerek leírásához és ebben a fájlban tudunk specifikálni különböző szabályokat, mint például hálózati beállításokat (pl.: port mapping), fájlrendszeri beállításokat (pl.: shared volumes – vagyis megosztott permanens tárhely, ami a konténer futása után is megmarad), stb.

4.8. Fejlesztői eszközök

A dolgozat három részre lehet bontani és mindegyikhez a hozzá leginkább megfelelő fejlesztői eszköz került kiválasztásra.

- Adatbázis:
  - Technológia: MongoDB
  - Fejlesztő eszköz: Studio 3T
    - Grafikus felületű eszköz MongoDB adatbázisokhoz
    - Támogatja az adatok hozzáadását, megtekintését, szerkesztését, törlését
- Backend kód
  - Technológia: ASP.NET Core
  - Fejlesztő eszköz: Visual Studio 2022
    - Microsoft által fejlesztett integrált fejlesztői környezet (IDE)
    - Támogatja a kód debuggolását, szerkesztését, illetve beépített tesztelői környezettel is rendelkezik
    - Grafikus felületű Git kliensnek is megfelelő
- Frontend kód
  - Technológia: Angular
  - Fejlesztői eszköz: Visual Studio Code
    - Microsoft által fejlesztett forráskód szerkesztő eszköz
    - Hivatalosan nem IDE, de a megfelelő kiegészítők telepítésével IDE-ként kezelhető
    - Kiegészítők:
      - Angular Language Service: az Angular sablonokhoz (template) szolgáltat kódellenőrzést és segítséget
      - Beautify: automatikusan olvashatóvá rendezi a kódot

5. FEJLESZTŐI DOKUMENTÁCIÓ

A szakdolgozat elkészítése során igyekeztem tartani magam a tervezés során megállapított alapelvekhez, követni az ott ismertetett architektúrát és fejlesztési modellt.

5.1. Fejlesztési dokumentáció

Az alkalmazás alapvetően két részből áll. Az egyik a backend (szerver oldali kód), a másik a frontend (a kliens böngészőjében futó kód). A backend a már ismertetett ASP.NET Core 6 keretrendszerben íródott, míg a frontend Angular 13-ban.

5.1.1. Adattárolás, adatmodell

Az alkalmazás használata során keletkezett adatok MongoDB-ben kerülnek tárolásra. Az adatbázis neve „InsomniaChat”, az adatbázishoz pedig két úgynevezett collection jön létre: „ChatRooms” és „ChatUsers”.

A ChatRooms collection szolgál a csetszobákhoz kapcsolódó adatok tárolására. Mivel a MongoDB egy NoSQL adatbázis, ezért az adatok egymásba vannak ágyazva, így például az üzenetek a hozzájuk tartozó „ChatRoom” dokumentum alatt tárolódnak. A továbbiakban ismertetésre kerülnek az eltárolt adatok, azok típusa és szerepe.

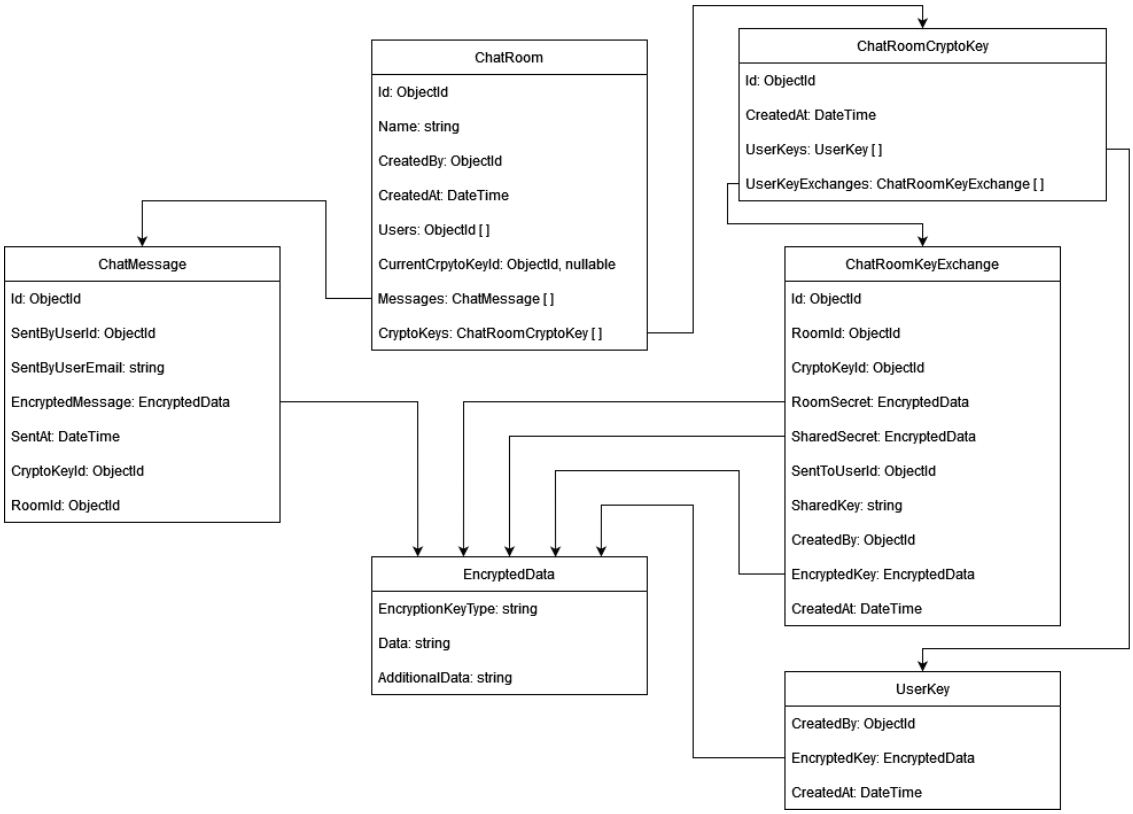
- **EncryptedData**
  - Titkosított adatok tárolását szolgálja
  - Mezők:
    - *EncryptionKeyType*:
      - Szöveges típusú.
      - Ebben a mezőben tárolódik el a titkosítás során felhasznált algoritmus neve.
    - *Data*:
      - Szöveges típusú.
      - Ebben a mezőben tárolódik el a titkosított adat.
    - *AdditionalData*:
      - Szöveges típusú.
      - Kiegészítő adat, amire szükség lehet a visszafejtés során.
- **UserKey**
  - Egy felhasználóhoz köthető titkosított kulcsok.
  - Mezők:
    - *CreatedBy:*
      - ObjectId típusú.
      - A készítő felhasználó azonosítója.
    - *EncryptedKey:*
      - EncryptedData típusú.
      - A titkosított kulcs tárolásáért felelős.

        - *CreatedAt:*
            - Dátum típusú.
            - A készítés időpontját tárolja.
- **ChatRoomKeyExchange**
    - A kulcscserék tárolásáért felelős objektum. A kulcscsere az a folyamat, aminek a során az egyes felek megállapodnak egy közös kulcsról, amit a titkosítás során felhasználhatnak.
    - A UserKey típus az ősosztálya.
    - Mezők:
        - *Id:*
            - ObjectId típusú.
            - A kulcscsere egyedi azonosítója.
        - *RoomId:*
            - ObjectId típusú.
            - A csetszoba azonosítója, ahol a kulcscsere történik.
        - *CryptoKeyId:*
            - ObjectId típusú.
            - Annak a kulcsnak az azonosítója, aminek a cseréjéről megállapodnak a felek.
        - *RoomSecret:*
            - EncryptedData típusú.
            - A szobához tartozó titkosító kulcs.
        - *SharedSecret:*
            - EncryptedData típusú.
            - A kulcscsere folyamat során létrejött közös kulcs.
        - *SentToUserId:*
            - ObjectId típusú.
            - A kulcscsere folyamat során azon felhasználó azonosítója, akivel a kulcscsere kezdeményezve lett.
        - *SharedKey*:
            - Szöveges típusú.
            - A kulcscsere modellt létrehozó felhasználó által generált publikus kulcs titkosítatlanul.
        - A többi mező: CreatedBy, CreatedAt, EncryptedKey a UserKey osztályból származik és az ott ismertetett célt szolgálják.
- **ChatRoomCryptoKey**
    - A létrehozandó, üzenetek titkosítását végző, ú.n. „szoba kulcs” kommunikációjának a tárolásáért felelős. A modell felelős a kulcscserék, és a kulcscserék eredményeként létrejött, minden felhasználó között megosztott kulcs tárolásáért.

- Mezők:
  - *Id*
    - ObjectId típusú.
    - A kulcs azonosítója.
  - *CreatedAt:*
    - Dátum típusú.
    - A létrehozás időpontja.
  - *UserKeys:*
    - UserKey tömb típusú.
    - Az első kulcscsere folyamat során jön létre az ú.n. szoba-kulcs. Ez a kulcs minden felhasználónál megegyezik és mindenki saját magának titkosítja.
  - *UserKeyExchanges:*
    - ChatRoomKeyExchange tömb típusú.
    - A kulcscserék lebonyolítását és tárolását végző modell. Egy meghívóhoz több is tartozhat, viszont minden meghívotthoz csak egy darab.

- **ChatMessage**
  - Az elküldött üzenetek tárolásáért felelős.
  - Mezők:
    - *Id:*
      - ObjectId típusú.
      - Az üzenet azonosítója.
    - *SentByUserId:*
      - ObjectId típusú.
      - Az üzenetet küldő azonosítóját tárolja.
    - *SentByUserEmail:*
      - Szöveges típusú.
      - Az üzenetet küldő e-mail címének az azonosítóját tárolja.
    - *EncryptedMessage:*
      - EncryptedData típusú.
      - A szöveg üzenetét tárolja titkosítva.
    - *SentAt:*
      - Dátum típusú.
      - A küldés időpontja.
    - *CryptoKeyId:*
      - ObjectId típusú.
      - Az üzenet titkosításához használt kulcs azonosítója.
    - *RoomId:*
      - ObjectId típusú.

        - A szoba azonosítója.
- **ChatRoom**
    - A csetszoba adataiért felelős típus.
    - Mezők:
        - *Id:*
            - ObjectId típusú.
            - A szoba azonosítója.
        - *Name:*
            - Szöveges típusú.
            - A szoba neve.
        - *CreatedBy:*
            - ObjectId típusú.
            - A szobát létrehozó felhasználó azonosítója.
        - *CreatedAt:*
            - Dátum típusú.
            - A létrehozás dátuma.
        - *Users:*
            - Szöveges tömb típusú.
            - A szobában résztvevő felhasználók azonosítója.
        - *CurrentCryptoKeyId:*
            - ObjectId típusú.
            - A modell úgy került kialakításra, hogy több titkosító kulcs tárolására is alkalmas legyen, habár jelenleg csak egy kulcs jön létre. Így egy felhasználó kilépése vagy belépése esetén a felhasználók később megállapodhatnak egy újabb titkosítókulcsról, a régi megtartásával. Ezért az éppen aktuális, használatban lévő kulcs azonosítója beállításra kerül a szobán.
        - *Messages:*
            - ChatMessage tömb típusú.
            - Az elküldött üzenetek.
        - *CryptoKeys:*
            - ChatRoomCryptoKey tömb típusú.
            - A létrejött kulcsok és az azokhoz tartozó kulcscsere folyamat tárolásáért felelős.



5.1.1.1. ábra, a csetszobához tartozó adatmodell diagramja.

A „ChatUsers” collection alatt tárolódnak az egyes felhasználók. A továbbiakban ismertetésre kerülnek a felhasználókhoz tárolt adatok, illetve azok típusa és célja.

- **UserCryptoKeyPairEntity**
  - Egy felhasználóhoz kötődő titkosító kulcspár tárolása.
  - Mezők:
    - *KeyType:*
      - Szöveges típusú.
      - Az eltárolt kulcspár típusa.
    - *EncryptedPrivateKey:*
      - EncryptedData típusú.
      - A privát kulcs titkosítva.
    - *PublicKey:*
      - Szöveges típusú.
      - A publikus kulcs titkosítatlanul.
- **ChatUser**
  - A felhasználó adatai.
  - Mezők:
    - *Username:*
      - Szöveges típusú.
      - A felhasználónév.

- *FirstName:*
  - Szöveges típusú.
  - A felhasználó keresztneve.
- *LastName:*
  - Szöveges típusú.
  - A felhasználó vezetékneve.
- *Email:*
  - Szöveges típusú.
  - A felhasználó e-mail címe.
- *Password:*
  - Szöveges típusú.
  - A felhasználó jelszava.
- *DateOfBirth:*
  - Dátum típusú.
  - A felhasználó születésének dátuma.
- *PublicEncryptionKeyPair:*
  - UserCryptoKeyPair típusú.
  - Bár jelenleg nincs használva, a továbbfejlesztés szempontjából előnyös lehet, ha a már létrejött felhasználóknak le van generálva egy nyilvános titkosítású kulcspár.
- *SigningKeyPair:*
  - UserCryptoKeyPair típusú.
  - Jelenleg ez sincs használva, azonban minden felhasználó kap egy nyilvános aláíráshoz használható kulcspárt, amely a későbbiekben a hitelesítést fogja végezni.

5.1.2. A titkosítási modell

Az alkalmazás az úgynevezett jelszóalapú titkosítást végzi. Az elv lényege, hogy a jelszóból generálódik egy folyamat során egy titkosító kulcs és minden további, privát felhasználásra megőrzendő titok ezzel a kulccsal kerül titkosításra. A szakdolgozat folyamán ez a kulcs a *mesterkulcs*.

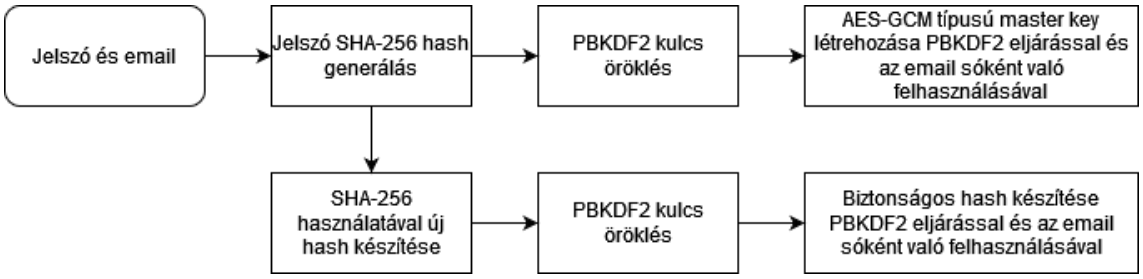
Ebben a részben a jelszóból generálandó mesterkulcs és a kétoldalú titkosítás menete lesz ismertetve.

5.1.2.1. A jelszó felhasználása a mesterkulcshoz

A kétoldalú titkosításhoz a jelszót és az e-mailt használom fel a titkosított kulcs, illetve a szerver irányába elküldött hash létrehozására. Mivel a szerver nem tudhat a felhasználó jelszaváról, sem pedig a kliensoldalon létrehozott kulcsról, ezért a jelszó két ágon kerül felhasználásra.

Az első ág a kliensoldali úgynevezett „mesterkulcs” (master key) létrehozásáért felelős. A folyamat során a jelszóból készül egy SHA-256 hash. Ez a hash lesz felhasználva egy kulcs létrehozására. Ez a kulcs egy PBKDF2 kulcsöröklтetési eljárás során jön létre. Ezután a létrehozott kulcs az e-mail sóként történő a felhasználásával egy AES-GCM típusú kulcs örökléstésére kerül felhasználásra. Az így létrehozott mesterkulcs nem kerül továbbításra senkinek és ez a kulcs kerül felhasználásra a privát kulcsok titkosítására.

A szerver irányába elküldött, úgynevezett biztonságos hash a jelszóból generált hashből kerül generálásra, első lépésként a létrejött hashből egy új hash generálódik az SHA-256 algoritmus felhasználásával. Ebből egy kulcs jön létre, majd az e-mail sóként való felhasználásával létrejön egy ú.n. ArrayBuffer. Ebből az ArrayBufferből készül egy szöveg. Az így létrejött szöveg ezután biztonságosnak tekinthető, ebből már nem állítható elő a felhasználóhoz tartozó mesterkulcs és így a szerver számára elküldhető.



5.1.2.1.1. ábra, a jelszó és az e-mail felhasználása a mesterkulcs és a szerveroldali, biztonságos jelszó hash előállításáról

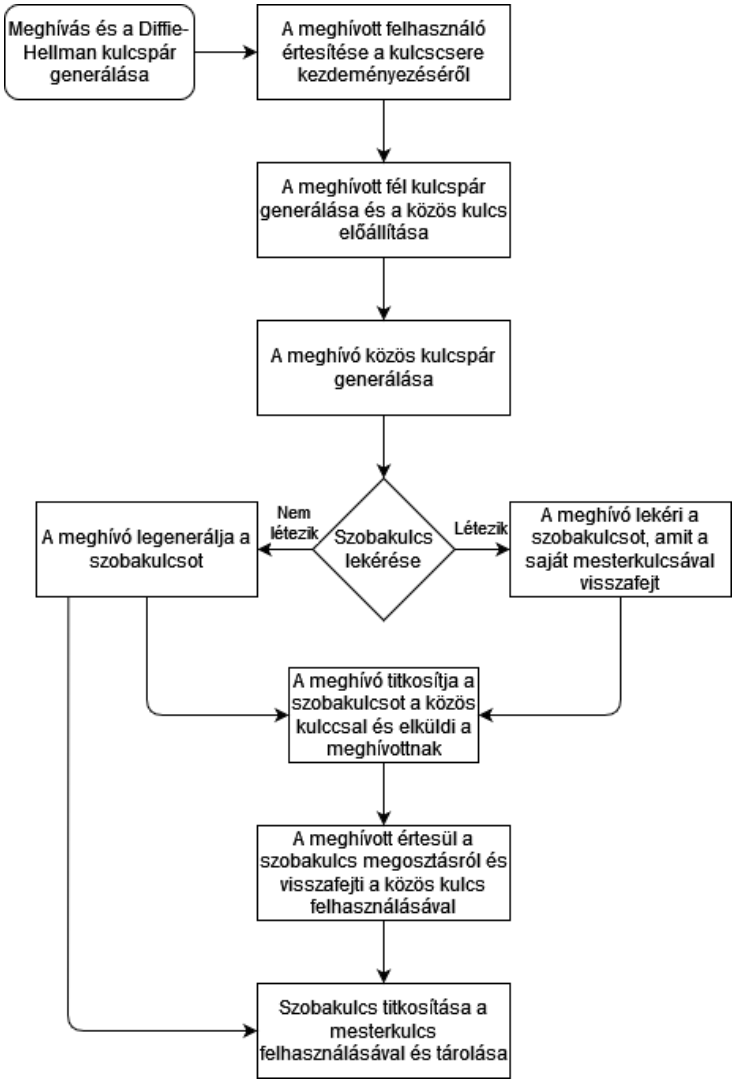
5.1.2.2. A titkosítás menete

A kétoldalú titkosítás során a felhasználók közötti kulcscsere az Elliptic Curve Diffie-Hellman (ECDH) algoritmus felhasználásával történik. A közös kulcs (sharedSecret) és a szobakulcs (roomSecret) AES-GCM típusú kulcsok.

A kulcscsere folyamata:

1. A szobába csak a szobát létrehozó felhasználó hívhat meg felhasználót. A meghívó elküldése során a *meghívó* fél generál egy kulcspárt. A kulcspár privát kulcsát a mesterkulcs felhasználásával titkosítja, a publikus kulcsot pedig json web key formátumban titkosítatlanul exportálja. Ezután jön a meghívás és a kulcscsere beállítása az ismertetett értékekkel
2. A *meghívott* ezután értesül a meghívásról és a kulcscsere létrehozásáról. A *meghívott* létrehozza a saját kulcspárját az ECDH algoritmussal. A privát kulcsot ő is titkosítja a maga mesterkulcsával, majd létrehozza a közös titkot. A közös titkot titkosítja a saját maga mesterkulcsával és beállítja a kulcscsere modellen a megosztott kulcsot.
3. A *meghívó* értesül a *meghívott* kulcscsere válaszáról és így a *meghívott* publikus kulcsához is hozzáfér. Ekkor legenerálja a közös titkot, ami ugyanaz mind a két felhasználónál. Ez a Diffie-Hellman elv működéséből következik.

4. A *meghívó* lekéri a szobához tartozó kulcsot, ha nincs ilyen, akkor az első közös kulcs esetében létrejött kulcsot használja fel a szobához tartozó kulcs létrehozásához. Ekkor beállítja a szobakulcsot, amit a saját mesterkulcsával titkosít.
5. Ezen a ponton már létezik a szobakulcs. A szobakulcsot a *meghívó* lekéri és visszafejti a saját mesterkulcsával. A kulcsot ezután a korábban létrejött közös kulcs felhasználásával beállítja a *meghívott* kulcscsere adatán.
6. A *meghívott* értesül az új szobakulcs létrehozásáról és visszafejti a közös kulcsot a mesterkulccsal. A közös kulccsal ezután visszafejti a szobakulcsot, amit titkosít a saját mesterkulcsával. A titkosított szobakulcsot pedig beállítja a szobán.
7. Ezután mind a meghívó, mind a meghívott rendelkezik egy közös szobakulccsal, amit így tudnak használni az üzenetek titkosításához és visszafejtéséhez.



5.1.2.2.1. ábra, a szobakulcs generálása és megosztása

Az így létrejött szobakulcsot a felhasználók a saját mesterkulcsukkal bármikor vissza tudnak fejteni és fel tudják használni az üzenetek titkosításához. Mivel a szobakulcs minden

felhasználónál ugyanaz, ezért mindegyik fél rendelkezik vele. A modell kettőnél több felhasználóval is működik, ezért a szobáknak több felhasználója is lehet.

5.1.3. Szerveroldal, backend

A szerveroldali kód a már korábban is ismertetett Clean Architecture követésével került kialakításra.

5.1.3.1. Chat.Core

Modellek

- Általános, a csethez szükséges modellek.
  - ChatRoom, ChatMessage, stb. modellek.
  - Ezen kívül use-case specifikus modellek definiálása, például a RegisterModel, ami a regisztráció során kerül felhasználásra és nem tartalmazza a felhasználó azonosítóját, mivel ez a regisztráció előtt még nem létezik.
- OperationResult: egy művelet végrehajtásának az eredménye
  - IsSuccess tulajdonság: sikeres volt-e a művelet vagy sem.
  - Error: a művelet végrehajtása során keletkezett hiba és a hiba oka és üzenete.

Stores

- Az adateléréshez úgynevezett Store interfészek kerültek kialakításra.
- A Store interfészek az adatelérési réteget fedik el. Nem tartalmaznak implementációt. Máshol (például a managerekben) konstruktorparaméterként vannak megadva, amiknek az implementációját a .NET Core dependency injection motorja példányosít.
- A Store interfészek egyes metódusainak a bemeneti értékei többek között a Core rétegben definiált modellek.
- A kialakítás előnye:
  - Az interfészekkel bármilyen adatelérési módot el lehet fedni, így később könnyű áttérni egy új megoldásra az üzleti logika módosítása nélkül.
  - Az üzleti logika tesztelése egyszerűbbé válik, ha nincs függősége semmilyen specifikus adateléréstől.
- A kialakítás hátránya:
  - Az absztrakció miatt use-case specifikus metódusokra van szükség. Ebben az esetben ahhoz, hogy az üzleti logikában megkapjuk a számát például a csetszobáknak, fel kell venni egy külön GetCount() metódust az adatelérési rétegben.
  - Ez azzal is jár, hogy az üzleti logikát fel kell bontani, az úgynevezett „adatelérési” logika (enterprise business rules) ez esetben a Store-ok által kerül elfedésre. Erre jó példa korábban ismertetett GetCount().

A Core réteg lényegében egy átfedő réteg, amelyre minden projekt referenciával rendelkezik és így mindenhol el lehet érni az itt definiált modelleket.

ICancellationTokenWrapper

Az interfész feladata, hogy az aktuális CancellationToken dependency injection használatával elérhető legyen bárhol, ahol szükség van rá.

ICurrentUserWrapper

Az interfész feladata, hogy az aktuális felhasználó dependency injection használatával elérhető legyen bárhol, ahol szükség van rá.

5.1.3.2. Chat.Infrastructure.DataAccess.Mongo

Az elnevezési metodika szerint a DataAccess réteg jelenti az adatelérési réteget, illetve a Mongo végződés azt, hogy MongoDB-n keresztül éri el az adatokat. Ha az adateléréshez Entity Framework Core lenne használva, akkor a réteg neve EFCore-ra végződne.

- A Clean Architecture megközelítésben ez a réteg feleltethető meg az „Enterprise Business Rule”-nak.
    - Ennek a lényege, hogy vannak bizonyos általánosan felhasználható logikák, mint például egy összetettebb lekérdezés, több tábla összekapcsolása SQL esetén, vagy aggregálás MongoDB esetében.
    - Pl.: ha le akarunk kérdezni egy specifikus kulcscserét, akkor meg kell mondani, hogy ki a kulcscsere létrehozója és a címzettje. Ilyenkor mind a két felhasználó azonosítója egy bemeneti paraméter lesz és aszerint lesz lekérdezve a kulcscsere. Az erre épülő üzleti logika viszont már alkalmazás specifikus, így ez egy felsőbb rétegben kerül megvalósításra.

Entities

- Ebben a mappában találhatók azok a típusok, amik a konkrét MongoDB megvalósítását tartalmazzák az egyes adatoknak.
- Például az entitások egyes tulajdonságaihoz (property) tartozó MongoDB specifikus attribútumok ezeken az osztályokon találhatók.
    - Pl.: BsonElemet, BsonId
- Így a MongoDB specifikus tulajdonságok elrejtésre kerülnek.

Mapping

- Ebben a mappában található a MappingProfile, ami az egyes Core modellek és az entitások közötti mappelést végzi az AutoMapper nuget package használatával. [67]

Stores

- A Stores mappa alatt vannak a Store interfészek implementációi.
- Ezen implementációk a MongoDB-re vonatkozó specifikus kódot tartalmaznak, a beérkezett adatokat átalakítja, „átmappeli” az adott Core modellhez tartozó entitásra.

IoC

- Az IoC osztály végzi a kontroll megfordítását, tehát az egyes implementációk regisztrációját a függőség befecskendezéshez, vagyis „dependency injection”-höz.

5.1.3.3. Chat.Infrastructure

Ez a réteg a menedzsereket és az üzleti logikát tartalmazza. Ez tekinthető a „Application Business Rules” rétegnek.

Itt is az IoC végzi a függőség befecskendezéshez a menedzser osztályok regisztrációját. A menedzser osztályoknál a Store interfészek a konstruktorparaméterek és a dependency injection használatával kapják meg a konkrét megvalósítást.

5.1.3.4. Chat.WebApi

Ez a réteg a backend oldali alkalmazás legfelsőbb rétege. A szerver egy RESTFUL Apin és egy SignalR hubon keresztül érhető el. A http kérések kiszolgálására három kontroller szolgál. A SignalR feladata a websocket kérések kiszolgálása. A socketek többek között az üzenetek küldésében, kulcscserék értesítésében játszanak szerepet.

Controllerek

- IdentityController
    - Ennek a feladata a felhasználókhoz kötődő kérések kiszolgálása. Ilyen a bejelentkezés, a regisztráció, a felhasználó adatainak frissítése és lekérése.
- ChatRoomsController
    - Ennek a feladata a csetszobákhoz tartozó kérések kiszolgálása. Például a szobák listájának lekérése, illetve az azokhoz tartozó műveletek, mint például a törlés és a létrehozás.
    - Minden ehhez kötődő erőforrás csak bejelentkezett felhasználók számára érhető el.
- ChatRoomCryptoController
    - Ennek a feladata a csetszobákhoz kötődő titkosító kulcsok lekéréséhez kötődő kérések kiszolgálása.

SignalR

- A SignalR egy ASP.NET keretrendszerű Websocketre épülő könyvtár, ami lehetővé teszi az aszinkron értesítéseket a webalkalmazások között. Amikor például a szerveren történik egy *esemény*, például Alice küldött egy üzenetet Bobnak, akkor Bob kliensét értesíti a szerver és így megkapja az üzenetet.
- SignalRUserIdProvider: a felhasználó azonosítóját szolgáltatja. A Json Web Tokenből fejti ki a felhasználói azonosítót.
    - Így egy felhasználót lehet értesíteni az azonosítója alapján.
- Groups:
    - Felhasználói csoportok.
    - Jelenleg minden szoba egy csoportot jelent és egy felhasználó több csoporthoz is tartozhat.
- NotificationHub

  - Feladata a felhasználók értesítése, a szobákhoz tartozó SignalR groupokhoz történő regisztrációja, valamint a kliens kapcsolódása esetén annak értesítése a félbemaradt kulcsgenerálásokról.
- Példa:
  - Amikor egy felhasználó küld egy üzenetet, akkor az egy http kérés lesz a szerver irányába. Az üzenet tárolása után a szerver a SignalR-en, vagyis egy websocketen keresztül értesíti a többi felhasználót az új üzenet létrehozásáról, amit így megkap a böngészőjük az oldal újratöltése nélkül.

Bejelentkezés

A szerveroldali authorizálását úgynevezett Json Web Tokenekkel (JWT) végzi a kliens. A kliens a JWT-t minden kérés mellé csatolja, amit a szerver ellenőriz és meggyőződik róla, hogy valóban ő állította ki a tokent. Ha sikeres volt, akkor a felhasználó bejelentkezettnek tekinthető. A tokenek rendelkeznek egy lejárati idővel, így a tokenek egy idő után már nem használhatók fel. [68]

Minden Json Web Token három részből áll:

- Header:
  - A token típusa: „typ”
    - pl.: JWT
  - A token aláírásához használt algoritmus: „alg”
    - Pl.:
      - HMAC-SHA256 esetén az érték HS256.
      - RSA SHA256 aláírás esetén RS256.
- Payload
  - A payload tartalmazza az információkat. Ezek általában nincsenek titkosítva, ezért csak a felhasználó nyilvános adatait, például az e-mail címét és azonosítóját érdemes itt tárolni, vagy egyéb értékeket, minthogy adminisztrátor-e a felhasználó vagy sem.
  - Itt tárolódik például a token lejárati ideje: „exp”.
- Signature
  - A szerver készít aláírást a tokenről.
  - Egy kiválasztott algoritmussal készít egy aláíró hasht a „headerről” és a „payloadról” illetve egy szerveroldali titkos kulcsról.
  - Ezután ezt a hash-t a szerver minden kérésnél előállítja és összehasonlítja a beküldött tokenben lévő aláírással.
  - Ha a két aláírás egyezik, akkor a tokent a szerver állította ki és a felhasználó be van jelentkezve.
  - Tehát az aláírás a következőképpen áll elő:
    - A bemeneti értékei a hashnek:
      - A szerver oldali titok
      - A base64url enkódolt header

- A base64url enkódolt payload

- Ezután a kiválasztott algoritmussal a token adatairól készül egy aláírás és összehasonlítja a tokenbe ágyazott aláírással. Ha egyezik a kettő, akkor tényleg a szerver állította elő a tokent.

AspNetCancellationTokenWrapper

A Core rétegben létrehozott ICancellationTokenWrapper megvalósítása. Konstruktorparamétere az IHttpContextAccessor ami a Microsoft.AspNetCore.Http névtérben található. Ezzel elérhetővé válik az adott request CancellationTokenje. Mivel a lehető legtöbb lekérdezés aszinkron, ezért ezek megszakíthatók is. A dependency injection ezt az osztályt fogja injektálni, amikor egy osztály konstruktorparamétere az ICancellationTokenWrapper.

CurrentUserWrapper

A Core rétegben létrehozott ICurrentUserWrapper megvalósítása. Az aktuális felhasználó beállításához egy saját middleware került kialakításra, ennek a neve CurrentUserMiddleware. A middleware minden beérkező kérés esetében lefut a kontroller metódusa előtt. Az ASP.NET Core keretrendszer ilyenkor az „Invoke” metódust hívja meg, ahol bemeneti paraméterként az IdentityManager és az ICurrentUserWrapper lett megadva. Mivel ezek „scoped”-ként lettek beállítva a dependency injectionben, ezért ezekből minden kéréshez egy példány tartozik. A middleware ezután eldönti, hogy a lekéréshez szükséges-e a bejelentkezés. Amennyiben igen, a Json Web Tokenből kifejti a felhasználó azonosítóját és lekérdezi az IdentityManageren keresztül. A lekért felhasználót ezután beállítja a CurrentUserWrapper osztályon, amely a következő injektálások esetében már tartalmazni fogja a jelenlegi felhasználót.

Így a lekérdezésekhez tartozó felhasználó injektálható és hozzáférhető az egyes rétegekben. Ez a technika felhasználható, hogy egyedi kontextust lehessen megfogalmazni egy-egy adott kérésre. A kontextus jelen esetben a felhasználó.

NSwag

Az alkalmazás használata során a kliensoldali (tehát a böngészőben futó vagy frontend kódnak) el kell érnie a szervert. A kontrollerek eléréshez a frontend számára az NSwag elnevezés nuget package-dzsel generálódik egy „kliens”. Ez egy TypeScript nyelvben és Angular keretrendszerhez generált „kliens”, ami tartalmazza a kontrollerek metódusait, azok bemeneti paramétereit és visszatérési értékeit.

Ahhoz, hogy az NSwag megfelelő kódot generáljon az egyes metódusokat a kontrollerben el kell látni a megfelelő attribútumokkal.

Jelen példában a ChatRoomCryptoController lesz alapul véve:

```
[Route("api/chat-rooms/{roomId}/crypto/{cryptoKeyId}/"), ApiController, Authorize]
1 reference
public class ChatRoomCryptoController : ControllerBase
{
```

5.1.3.4.1. ábra, a ChatRoomCryptoController attribútumai

A Route mondja meg, hogy mi legyen az elérési útja a kontrollernek, minden metódus saját elérési útja kiegészíti a kontroller elérési útját.

```
[HttpGet("shared-secret/{sharedWithUserId}")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(EncryptedData))]
[ProducesResponseType(StatusCodes.Status400BadRequest, Type = typeof(string))]
[ProducesResponseType(StatusCodes.Status401Unauthorized, Type = typeof(string))]
[ProducesResponseType(StatusCodes.Status404NotFound, Type = typeof(string))]
[ProducesResponseType(StatusCodes.Status409Conflict)]
0 references
public async Task<IActionResult> GetSharedSecret(string roomId, string cryptoKeyId, string sharedWithUserId)
{
```

5.1.3.4.2. ábra, a GetSharedSecret metódus attribútumai

Az NSwag innen fogja tudni, hogy a metódus visszatérési értéke „EncryptedData” lesz.

A megfelelő végpont elérése érdekében a következő kódot generálja a kliensoldali kód számára:

```
getSharedSecret(roomId: string | null, cryptoKeyId: string | null, sharedWithUserId: string | null): Observable<EncryptedData> {
    let url_ = this.baseUrl + "/api/chat-rooms/{roomId}/crypto/{cryptoKeyId}/shared-secret/{sharedWithUserId}";
```

5.1.3.4.3. ábra, a kliensoldali kód

Ezzel a módszerrel a kliensoldali kód könnyebben karbantarthatóvá válik, mert egyrészt minden olyan modell, ami a backenden használva van, automatikusan generálódik és így nem fordulhat elő olyan, hogyha az EncryptedData módosításra kerül, akkor a frontenden ennek a lekövetése elmarad.

Ha a „route” módosul, arra sem kell ügyelni, hiszen a kód generálása automatikus.

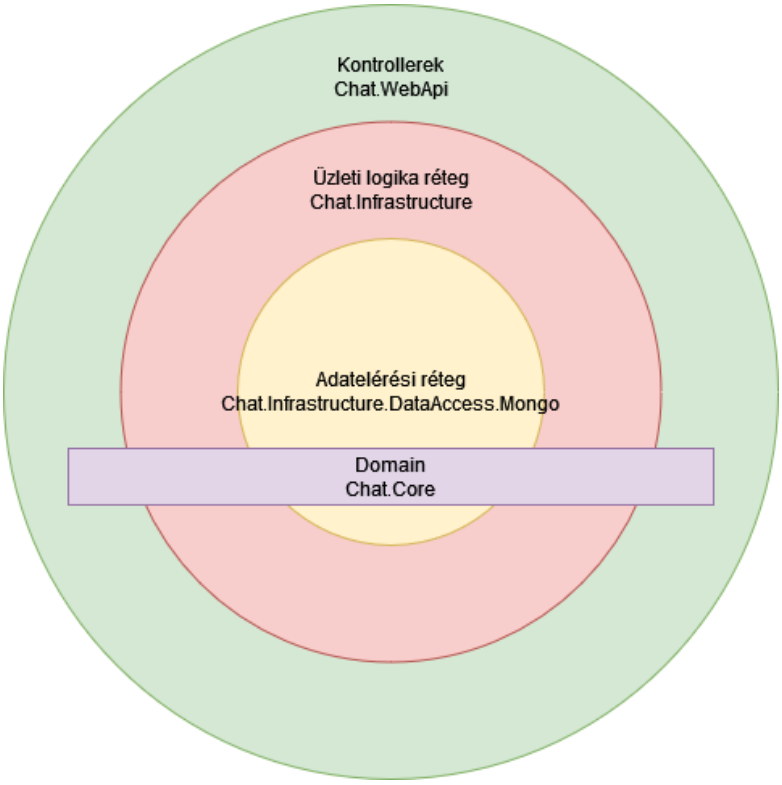
A generálás a WebApi projekt build eseményére van bekötve a Chat.WebApi.csproj fájlban. Itt az NSwag minden paramétert változóként kap meg, ezen változók értékei pedig a gyökérkönyvtár alatti Directory.Build.props nevű állományban találhatók.

Dependency Injection

A WebApi réteg végzi el a dependency injectiont vagy a függőség befecskendezést. Ennek az az oka, hogy ez indítja el az alkalmazást. A WebApi az egyes rétegekben található „IoC” osztályok „ConfigureServices” metódusait hívja meg. Ezek végzik az osztályok megfelelő regisztrációját.

5.1.3.5. Összegezve

Az egyes rétegek mindig egymásra épülnek. A WebApi projekt az Infrastructure projekt menedzser osztályait éri el a kontrollereken, illetve a SignalR hubon keresztül. Tehát a WebApi feladata a kliensoldali kérések kezelése. A menedzser osztályok az üzleti logikát tartalmazzák és nem tartalmaznak referenciát az adatelérési rétegre, csupán a Core projekt interfészeiről tudnak. A legalsó réteg a DataAccess.Mongo, ami a MongoDB eléréséért felel és megvalósítja a Store interfészeket. A Core projekt pedig egyféle átfogó réteg, amiről minden szinten tudnak, ezzel biztosítva, hogy közös absztrakciókkal és modellekkel dolgoznak.



5.1.3.5.1. ábra, az architektúra vizualizációja

5.1.4. Kliensoldal, frontend

A végpontok közötti titkosítás érdekében a folyamat minden lépése a kliensoldalon, a böngészőben történik. A szerver a bizalmas adatokat (üzenetek, privát kulcsok, szobakulcsok) oly módon kapja meg tárolásra, hogy azt egy harmadik fél már ne tudja visszafejteni.

A kliensoldali megjelenítéshez az Angular 13-at választottam, illetve a hozzá tartozó Angular Material 13-as komponens könyvtárat. Ez tartalmazza a megjelenítéshez tartozó legtipikusabb komponenseket: listát, listaelemet, ikonokat stb.

A titkosításhoz a Web Crypto API-t választottam. Ez egy interfész, amit a modern böngészők (pl.: Firefox, Chrome) támogatnak. [69]

Azért esett a választásom erre a technológiára, mert ezt támogatják a böngészők.

5.1.4.1. SubtleCrypto

A Web Crypto API alá tartozik a SubtleCrypto, ami szintén egy interfész és szintén támogatják a modern böngészők. A SubtleCrypto interfésszel érhető el az összes titkosítási eljárás, amit felhasználtam a szakdolgozat során. [70]

A SubtleCrypto interfész némely, fontosabb részletét az alábbiakban ismertetem:

- Fontosabb típusok:
    - CryptoKey: a SubtleCrypto metódusai által létrehozott kulcsok típusa, amik felhasználhatók titkosításra, örököltetésre, stb. [71]

  - CryptoKeyPair: ez a típus két CryptoKey-t tartalmaz és a nyilvános kulcsú titkosításoknál generálódik. A két kulcs a publikus és a privát. [72]
  - ArrayBuffer: az ArrayBuffer nem a SubtleCrypto része, hanem az ECMAScript nyelvben specifikált típus. Az ArrayBuffer egy generikus, fix hosszúságú nyers bináris adatbufferek reprezentálására való. Ez lehet egy Int32Array típus, Uint8Array, stb. [73]
- Fontosabb metódusok
  - generateKey: kulcs generáláshoz kerül felhasználásra. Ez a kulcs lehet AES-GCM, RSA-OAEP, ECDH stb. típusú. [74]
  - importKey: kulcsok importálásához használt metódus. [75]
    - A kulcs lehet egy létező, korábban létrehozott kulcs
      - Pl.: Diffie-Hellman kulcspár publikus, tikosítatlan kulcsa.
    - Valamint egy korábban generált hash is importálható kulcsként például a PBKDF2 kulcsörököltetési eljárás felhasználásával.
  - exportKey: egy kulcs exportálásához használható metódus. [76]
    - Az exportKey során a kulcs nem kerül titkosításra, ezért a szakdolgozat során ez publikus kulcsoknál van használva.
  - deriveKey, deriveBits: kulcsok örököltetéséhez használt eljárás. [77][78]
    - A deriveKey visszatérési értéke lehet CryptoKey vagy CryptoKeyPair.
    - A deriveBits visszatérési értéke ArrayBuffer.
  - wrapKey: egy kulcs titkosítva történő exportálását végzi el. [79]
    - A szakdolgozat során a privát kulcsok vagy a megosztott kulcsok titkosítására van használva.
  - unwrapKey: a wrapKey fordítottja, a wrapKey kimeneti értékének a felhasználásával ad vissza egy kulcsot. [80]
    - A szakdolgozat során a privát kulcsok vagy a megosztott kulcsok visszafejtésére van használva.
  - digest: egy hash-t állít elő. [81]
    - A szakdolgozat során a jelszó hashelésére kerül felhasználásra.
- Fontosabb algoritmusok: a SubtleCrypto rendelkezik több különböző, előre elkészített titkosítási, hashelési algoritmussal, a korábban ismertetett metódusok ezekkel az algoritmusokkal használható.
  - AES-GCM: egy AES alapú, szimmetrikus kulcsú titkosítási eljárás. [82]
    - Ehhez tartozik az úgynevezett inicializáló vektor, vagy iv. Az iv-t minden titkosításhoz külön kell generálni, egyazon kulcsnál egy iv-t nem szabad több titkosítás során felhasználni. Azonban az iv publikusan tárolható, nem sérti a titok integritását.
  - ECDH: a rövidítés az Elliptic Curve Diffie-Hellman eljárásból eredő mozaikszó. [83]

- PBKDF2: egy kulcs örököltetési eljárás, ami viszonylag alacsony entrópiájú bemeneteknél használt, ilyen például a jelszó. [84]

5.1.4.2. A mesterkulcs és a szerver felé küldött biztonságos jelszó létrehozása

A korábbiakban a jelszó létrehozásának a folyamata már ismertté vált koncepcionális szinten. Ebben a fejezetben a technikai részletek kifejtése történik.

- **Mesterkulcs**
  - Felhasznált adatok: jelszó és e-mail
  - Lépései:
    - Hash előállítása
      - Egy SHA-256 típusú hash állítódik elő a SubtleCrypto „digest” metódusával. A visszatérési értéke egy ArrayBuffer.
    - Kulcs előállítása
      - Egy kulcs előállítása a SubtleCrypto interfész „importKey” metódusával a PBKDF2 kulcsörököltetési algoritmus használatával.
      - Az „importKey” bemenete a hashelés során generált ArrayBuffer.
    - Mesterkulcs előállítása
      - Egy AES-GCM típusú kulcs előállítása a SubtleCrypto „deriveKey” metódusával, amely a PBKDF2 kulcsörököltetési algoritmust használja. Az előállított kulcsnál sóként a felhasználó e-mail címe kerül felhasználásra.
- **Biztonságos jelszó**
  - Jelszóból készített, a szerver irányába elküldött adat, amiből már nem lehet sem az eredeti jelszót, sem a mesterkulcsot előállítani
    - Újrahashelés
      - Ennek a hashnek a bemeneti értékeként a mesterkulcs első lépésében előállított hash lesz felhasználva. Lényegében kétszer fut le a „digest” metódus.
    - Kulcs előállítása
      - Egy kulcs előállítása a SubtleCrypto interfész „importKey” metódusával a PBKDF2 kulcsörököltetési algoritmus használatával.
      - Az „importKey” bemenete az újrahashelés során generált ArrayBuffer.
    - Biztonságos hash előállítása
      - A SubtleCrypto „deriveBits” metódusával az e-mail sóként való felhasználásával létrejön egy ArrayBuffer.

Ebből az ArrayBufferből készül egy string a generateHexString metódus meghívásával. A generateHexString metódus a Mozilla Developer Network (MDN) oldalán a „digest” dokumentációjában található példakód felhasználásával készült. [85]

- Az így létrejött string ezután biztonságosnak tekinthető, ebből már nem állítható elő a felhasználóhoz tartozó mesterkulcs és így a szerver számára elküldhető.

5.1.4.3. Architektúra

A kliensoldali kód létrehozása során igyekeztem követni a tervezés részben ismertetett flat architektúrát.

A flat architektúra miatt a kódbázist három részre osztottam:

1. Core könyvtár
    1. Itt kerültek kialakításra a service-k, providerek, base komponensek.
2. Modules könyvtár
    1. Itt kerültek kialakításra az egyes részei az alkalmazásnak. A modules alatt találhatók a főbb csoportjai az alkalmazásnak. Minden modules könyvtár alatt egy-egy külön oldal található, amire a felhasználó el tud navigálni. Minden modules alatti könyvtár saját routinggal rendelkezik.
    2. A chat alatt maga a chat komponens és az egyes részkomponensei találhatók.
    3. Az identity alatt a felhasználó kezeléséhez tartozó komponensek találhatók.
3. Shared könyvtár
    1. Itt találhatók a közös modellek, stb.

5.1.4.4. Megvalósítás

A megvalósítás során törekedtem az olvasható kódra és az újra felhasználhatóságra. A továbbiakban a megvalósítás néhány fontosabb eleme kerül ismertetésre.

A chat komponens alapvetően két elemből áll. Ebben a komponensben kerül megjelenítésre a csetszobák listája, a csetszobához tartozó komponens, amiben pedig a csetüzenetek kerülnek megjelenítésre.

A szerver oldali lekérdezések két részből és mindegyik rész két rétegből áll.

Az első rész a http kéréseket kezeli, amik a szerver RESTFUL Apiját érik el. Ehhez a legalsó réteget az NSwag szerveroldali .NET könyvtár generálja egy Angularhoz való klienssel és a hozzá tartozó modellekkel. Ez a kliens az Angular Dependency Injection felhasználásával kerül injektálásra. A http kérések legfelsőbb rétege egy service réteg, aminek a célja, hogy elfedje a generált klienst.

A másik része a szerveroldallal történő kapcsolattartásnak a SignalR. A SignalR-hez a Microsoft SignalR npm csomagja került felhasználásra. A SignalR legalsóbb rétege egy ősosztályból (BaseSignalRService) és egy abból örököltetett SignalR hub specifikus osztályból áll. Az ősosztály egy általános megvalósítás, ami nem tud az egyes hubokról. A hub elérést konstruktparaméterként kapja meg, illetve rendelkezik a register és invoke metódusokkal. Az ebből származtatott hub specifikus osztály tartalmazza a különböző SignalR eseményeket. A hub specifikus osztály egy singleton, ezzel biztosítva, hogy munkamenetenként csak egy kapcsolat jöjjön létre egy-egy hubhoz.

A SignalR-rel való kapcsolattartás legfelsőbb rétege a providerek között található. Ez egy singleton osztály, ami az RxJS könyvtár felhasználásával úgynevezett Subjectekkel teszi hozzáférhetővé a SignalR eseményeket. Ezzel a megoldással a SignalR adott esetben lecserélhető, illetve az egyes eseményekről le lehet iratkozni, ha már nincs rájuk szükség.

A kulcscserét a KeyExchangeProvider végzi. Ez szintén egy singleton osztály, ami konstruktorparaméterként a (jelenleg) SignalR-re építő providert várja és az osztály példányosítása során feliratkozik az egyes eseményekre. Minden korábban ismertetett kulcscseréhez kötődő üzleti logika ebben az osztályban kerül végrehajtásra.

A titkosítás szintén két ágból és két rétegből áll.

Az első ág a titkosítást végzi, a másik pedig a mesterkulcsot szolgáltatja. A titkosítás legalsóbb rétege az úgynevezett InsomniaCrypto ősosztály. A SubtleCrypto interfészhez tartozó metódusok az InsomniaCrypto ősosztályban kerülnek felhasználásra, illetve elfedésre. Az elfedés célja, hogy minden SubtleCrypto referencia egy helyen legyen kezelve, ami később, az interfész módosulása esetén vagy másféle megoldásra áttérés esetén, karbantarthatóbb kódot eredményez. Az egyes üzletilogika specifikus titkosítási eljárások, mint például a csetüzenetek titkosítása, visszafejtése, az egyes kulcsokkal kapcsolatos műveletek stb. az InsomniaCryptoból származtatott osztályokban kerülnek kialakításra. Ilyen például az InsomniaKeyCrypto valamint az InsomniaMessageCrypto, stb. Az erre épülő réteg a CryptoService, amely a Dependency Injection felhasználásával injektálható, például a korábban ismertetett KeyExchangeProviderben. A CryptoService már üzleti logika specifikusabb megvalósításokat tartalmaz, célja, hogy injektálni lehessen, illetve ő is megkapja az injektálható példányokat.

A másik ága a titkosításnak a CryptoProvider, ami egy singleton osztály és a dependency injection felhasználásával injektálható. A CryptoProvider tulajdonképpen a mesterkulcs előállításáért felel. Ha a mesterkulcs előállt, akkor az felhasználható például a CryptoService-ben, aminek így nem kell minden alkalommal azt előállítania és így központi helyen kezelhető a mesterkulcs.

A regisztrációhoz és a bejelentkezéshez tartozó műveleteket az IdentityService, injektálható osztály végzi. Ez az osztály fedi el többek között az automatikusan generált klienst, valamint kap egy példányt az IdentityProviderből (ez a token tárolásáért felelős) és a korábban ismertetett CryptoService-ből. Az IdentityService-ben a bejelentkezés során létrejön egy kliensoldali password hash, illetve a szerver által előállított json web token vagy JWT. A JWT minden szerveroldali kéréshez csatolásra kerül és a felhasználót ez a

token authentikálja. A szerveroldalon előállított JWT, a felhasználóhoz kapcsolódó információk, mint például az e-mail és az azonosító, valamint a kliens oldali jelszó hash a böngésző localStorage-ben kerülnek tárolásra az IdentityProviderben.

5.2. Fejlesztői környezet kialakítása

A környezet megfelelő kialakításához az egyes felhasznált technológiákból bizonyos verziójú csomagokra van szükség. Ezek, illetve a javasolt fejlesztőkörnyezetek az alkalmazás egyes részeihez kerülnek részletezésre ebben a fejezetben.

5.2.1. MongoDB

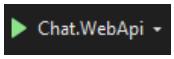
A MongoDB ajánlott indítása Docker használatával történik. A projekt könyvtárban található a docke-compose.yml fájl, amely minden szükséges paramétert tartalmaz. A fejlesztőnek a projekt gyökérkönyvtárába kell navigálnia a konzol használatával, majd pedig a „docker-compose up” parancsot kiadnia. Más teendő a MongoDB indításával nincs. Az ajánlott fejlesztő eszköz a Studio 3T.

Javasolt verziók:

- Docker Desktop 4.6.1.
- Mongo 5.0.7 (a Docker compose fájlban van specifikálva)

5.2.2. Backend

A backend ajánlott indítása Visual Studio 2022-ből történik. Az indítandó projektnek (startup project) a „Chat.WebApi”-t kell kiválasztani, majd az IDE középső részén felül az indítógombot kiválasztani. Az első indítás alkalmával importálni fogja az SSL tanúsítványokat.



5.2.2.1. ábra, az alkalmazást elindító gomb Visual Studioban

Javasolt verziók:

- .NET Core 6: a Visual Studio 2022-vel együtt települ

5.2.3. Frontend

A frontend indítása szintén konzolból ajánlott. Ehhez a fejlesztőnek el kell navigálnia a gyökérkönyvtárhoz képest ehhez a relatív útvonalhoz: *„./src/WebApp/insomnia-chat-ui/*”.

Az alkalmazás első indítása előtt futtatnia kell az „npm install” parancsot, majd utána minden alkalommal az „npm start” paranccsal lehet elindítani. Ez importálja az SSL tanúsítványt és futtatni fog egy Angular Live Servert.

Az alkalmazás fejlesztéséhez ajánlott fejlesztőeszköz a Visual Studio Code.

Javasolt verziók:

- Node v16.14.2
- Angular CLI 13.3.2:

5.3. Tesztelési dokumentáció

5.3.1. Unit tesztek

A fejlesztés menetét Test Driven Development módon kezdtem el, azonban a titkosítás koncepcióját, modelljét és így a letárolt adatokat több ízben is kénytelen voltam módosítani, nem egyszer teljesen újraírni. Mivel gyakran nem tudtam, hogy hogyan kéne működnie egy-egy metódusnak, hogy milyen adatokat szeretnék eltárolni, illetve, hogy a még formálódó adatokat milyen módon szeretném létrehozni, a tesztek írása gyakran átcsúszott a megvalósítás utánra.

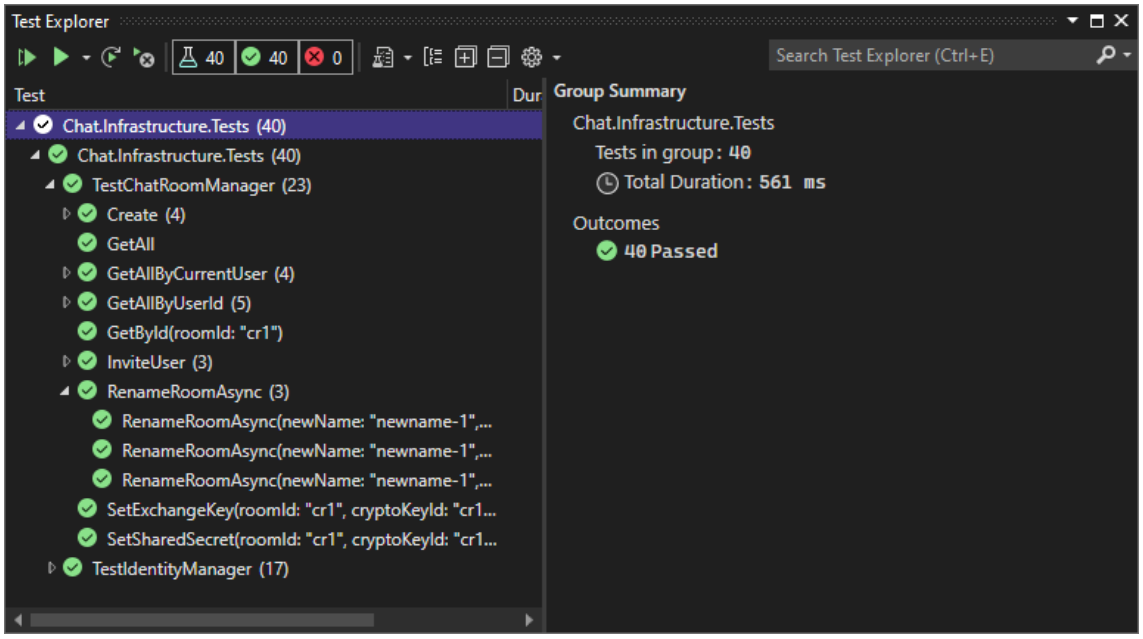
A szakdolgozat írása közben próbáltam lefedni a legtöbb tesztesetet, azonban a kiválasztott titkosítási technológia iránti tudás hiánya, valamint a gyakran módosuló koncepciók, modellek miatt a frontenden nem állt módomban teszteket írni.

A backendnél viszont az üzleti logikát tartalmazó menedzserosztályokat próbáltam a lehető legjobban lefedni.

A teszteléshez az xUnit keretrendrendszert és a Moq nevű mock könyvtárat használtam. A Moq-kal hozok létre egy példányt például az ILogger interfészből, mert ez minden menedzserosztály konstruktorparamétere.

Az egyes Store interfészeket implementáltam a teszthez megfelelően, így ezek is mockolt interfésznek tekinthetők, épp csak mock könyvtár nem került felhasználásra.

A menedzserek példányosítására factory osztályokat hoztam létre.



5.3.1.1. ábra, tesztek futásának az eredménye

A tesztek futása minden metódusra sikeres. Több olyan tesztesetet is definiáltam, ahol nem létező felhasználóval, vagy aktuális felhasználó nélkül fut le a teszt. Ezeknek a futása is sikeres és a menedzserek helyesen működnek.

5.3.2. Manuális tesztek

A fejlesztés során az alkalmazás végig tesztelve volt mind böngészőben, mind pedig Visual Studioban a kód debuggolásával.

A manuális tesztelés egy olyan esetet fog bemutatni, amikor sem a *meghívó*, sem pedig a *meghívott* nem elérhető minden időpillanatban.

A teszteléshez a Google Chrome böngésző DevTools lesz használva. Ez az F12 billentyű lenyomásával nyitható meg és többek között itt érhető el a hálózati és konzol lap. A hálózati lapon (network tab) a hálózati forgalom figyelhető meg, míg a konzol lapon (console tab) pedig az alkalmazás üzenetei láthatók.

Az adatbázisról készült adatok Studio 3T-ben készültek.

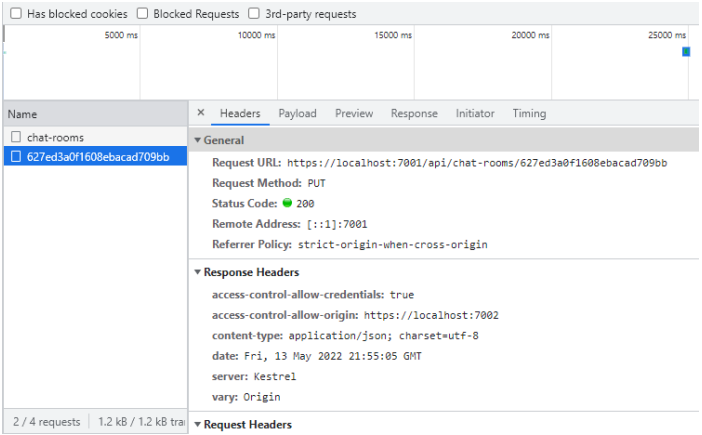
Résztvevők:

- user10: *meghívó*
  - Létrehozza az „u10-rm-01” szobát
  - Meghívja user11 felhasználót résztvevőnek
- user11: *meghívott*

User10 létrehozza a szobát. A kis lakat jelzi, hogy nincs még szobatitok.

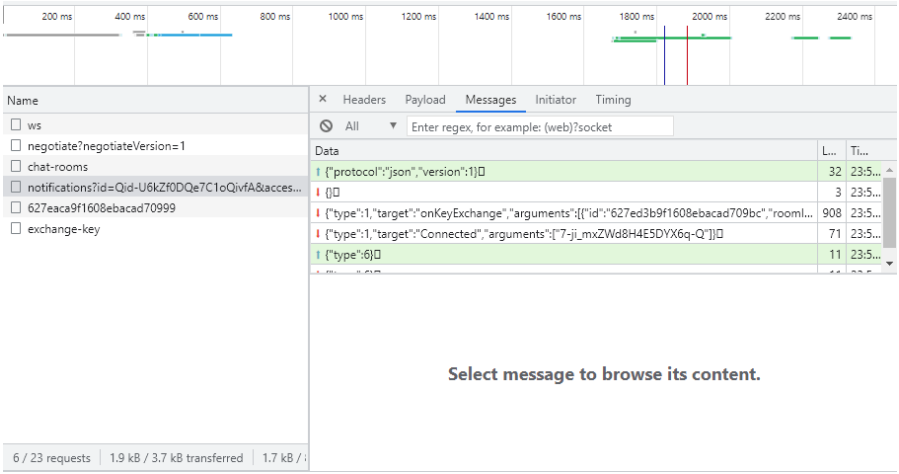
5.3.2.1. ábra, a létrejött szoba kód nélkül user10 böngészőjében

User10 meghívja user11-et. A 200-as státusz jelzi, hogy a meghívás sikeres volt.



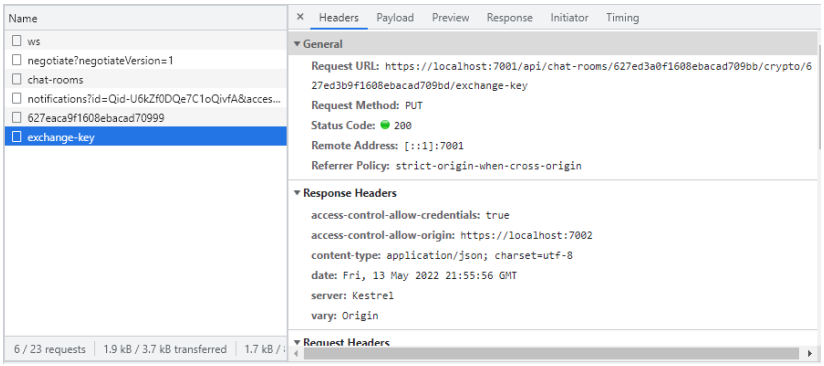
5.3.2.2. ábra, a meghívó a hálózati lapon

Ezután user10 bezárja a böngészőt és user11 belép. User11 először értesül a NotificationHubon, WebSocket protokollon keresztül a kulcscseréről az „onKeyExchange” eseménnyel.



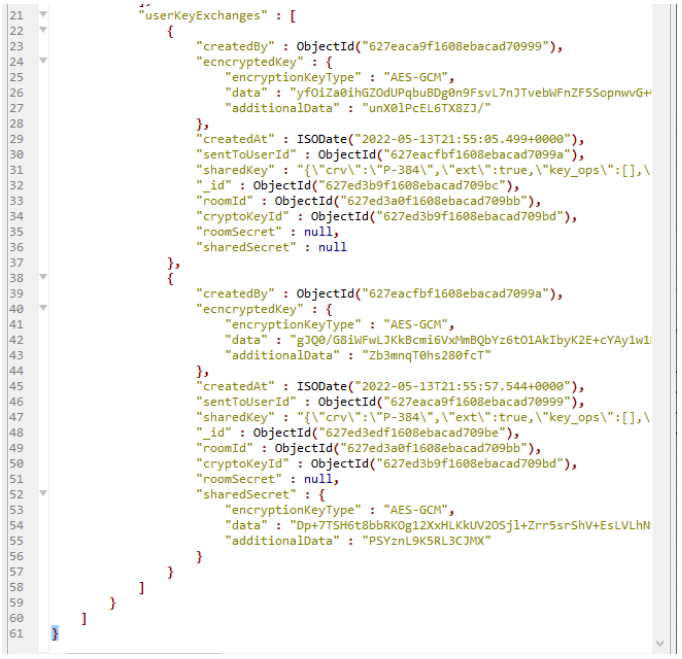
5.3.2.3. ábra, onKeyExchange SignalR esemény

Erről az eseményről értesül a KeyExchangeProvider és legenerálja a közös kulcsot user10 megosztott kulcsa (a Diffie-Hellman publikus kulcsa alapján). Ezt titkosítja a saját, vagyis user11 mesterkulcsával, majd eltárolja az „exchange-key” végpont meghívásával.



5.3.2.4. ábra, az „exchange-key” végpont meghívása

Ekkor még nem létezik a szobakulcs, csupán a közös kulcs. User11 eltárolja a privátkulcsot a saját mesterkulcsával titkosítva, a megosztott kulcsot (shared key) publikusan, titkosítás nélkül és a közös kulcsot (shared secret) a mesterkulcsával titkosítva. Ezután user11 bezárja a böngészőt.



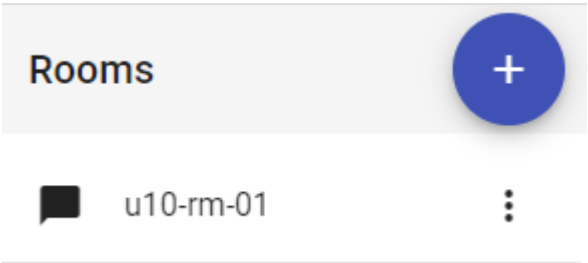
5.3.2.5. ábra, az adatbázis állapota: felül user10, alul user11 értékei

Legközelebb, amikor user10, a *meghívó* belép, akkor értesülni fog a meghívó fogadásáról és user11 megosztott kulcsával el tudja készíteni a közös kulcsot. Ezután elkészíti a szobakulcsot is, majd eltárolja azt a saját mesterkulcsával titkosítva. A szobakulcsot végül elküldi user11-nek a közös kulccsal titkosítva.

5.3.2.6. ábra, user10 szobakulcsa az adatbázisban

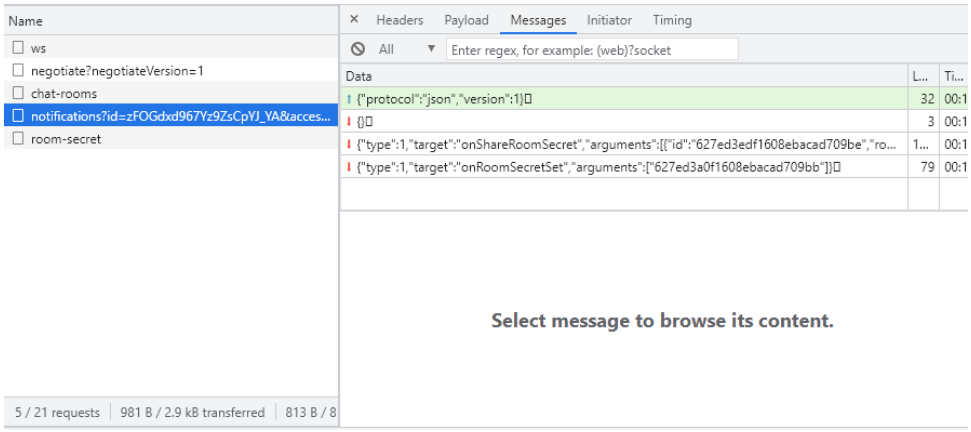
```
"userKeyExchanges" : [
    {
        "createdBy" : ObjectId("627eaca9f1608ebacad70999"),
        "ecncryptedKey" : {
            "encryptionKeyType" : "AES-GCM",
            "data" : "yfOiZa0ihGZOdUPqbuBDg0n9FsvL7nJTvebWFnZF5SopnwvG+
            "additionalData" : "unX0lPcEL6TX8ZJ/"
        },
        "createdAt" : ISODate("2022-05-13T21:55:05.499+0000"),
        "sentToUserId" : ObjectId("627eacfbf1608ebacad7099a"),
        "sharedKey" : "{\"crv\":\"P-384\",\"ext\":true,\"key_ops\":[],\
        "_id" : ObjectId("627ed3b9f1608ebacad709bc"),
        "roomId" : ObjectId("627ed3a0f1608ebacad709bb"),
        "cryptoKeyId" : ObjectId("627ed3b9f1608ebacad709bd"),
        "roomSecret" : null,
        "sharedSecret" : {
            "encryptionKeyType" : "AES-GCM",
            "data" : "mYjz1Zh3Ky8EQIzzkLYwSFlwmGs3VUfA1eZqmDfq+aVjntilF
            "additionalData" : "ADBvVokr0mJnzj/d"
        }
    },
    {
        "createdBy" : ObjectId("627eacfbf1608ebacad7099a"),
        "ecncryptedKey" : {
            "encryptionKeyType" : "AES-GCM",
            "data" : "gJQ0/G8iWFwLJKkBcmi6VxMmBQbYz6tO1AkIbyK2E+cYAy1w1
            "additionalData" : "Zb3mnqT0hs280fcT"
        },
        "createdAt" : ISODate("2022-05-13T21:55:57.544+0000"),
        "sentToUserId" : ObjectId("627eaca9f1608ebacad70999"),
        "sharedKey" : "{\"crv\":\"P-384\",\"ext\":true,\"key_ops\":[],\
        "_id" : ObjectId("627ed3edf1608ebacad709be"),
        "roomId" : ObjectId("627ed3a0f1608ebacad709bb"),
        "cryptoKeyId" : ObjectId("627ed3b9f1608ebacad709bd"),
        "roomSecret" : {
            "encryptionKeyType" : "AES-GCM",
            "data" : "VhB37+h24Lwmz2+ZNwm/uzUfjcyXvS0SyhXLj7UTv9hplWMJL
            "additionalData" : "HtOdBpeeIMT9REDV"
        },
        "sharedSecret" : {
            "encryptionKeyType" : "AES-GCM",
            "data" : "Dp+7TSH6t8bbRKOg12XxHLKkUV2OSjl+Zrr5srShV+EsLVLhN
            "additionalData" : "PSYznL9K5RL3CJMX"
        }
    }
]
```


5.3.2.7. ábra, a kulcscserék állapota: felül user10, alul user11

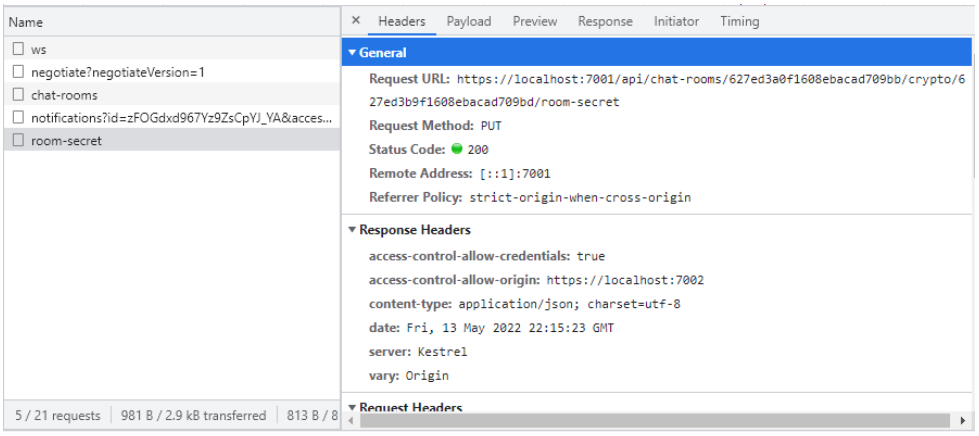


5.3.2.8. ábra, user10 rendelkezik a szobakulccsal, ezért nincs ott a lakat

User11 ezután visszajelentkezik. Először értesül az „onShareRoomSecret” eseményről, majd pedig lekéri a szobakulcsot a saját maga által létrehozott kulcscsere modellel. Ezt a kulcsot a közös kulcs felhasználásával (shared secret) vissza tudja fejteni. Majd a nyers szobakulcsot titkosítja a mesterkulccsal és elküldi a szervernek tárolásra.



5.3.2.9. ábra, az onShareRoomSecret SignalR esemény WebSocket protokollon keresztül



5.3.2.10. ábra, a room-secret végpont használata a szobakulcs tárolására PUT metódussal

5.3.2.11. ábra, a szobakulcs mindkét felhasználó számára a saját mesterkulcsukkal titkosítva

Ezután az üzenetek küldése biztonságosnak tekinthető. Ezek a szobakulccsal kerülnek titkosításra.

6. FELHASZNÁLÓI DOKUMENTÁCIÓ

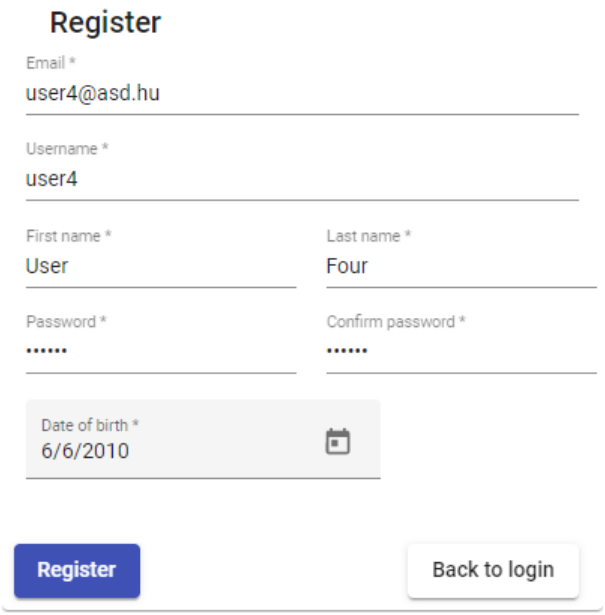
A felhasználói dokumentációban az alkalmazás használata lesz ismertetve felhasználói szemszögből és az egyes funkciók leírása található meg. Az alkalmazás használatához regisztráció szükséges, ezért a regisztrációval kezdődik, a belépéssel, majd a bejelentkezéssel elérhető funkciókkal folytatódik.

Az ajánlott böngésző a Google Chrome és a Firefox.

6.1. Regisztráció

Minden felhasználót egyedien azonosít az e-mail címe és a felhasználóneve. A regisztrációs oldalon lehetőség van megadni a következő értékeket:

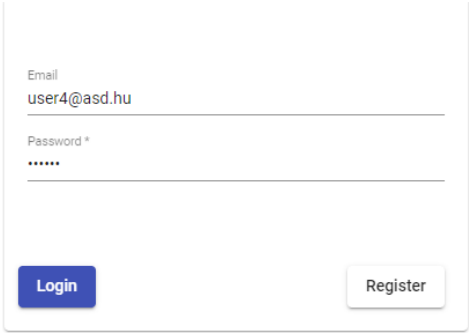
- E-mail cím (email)
- Felhasználónév (username)
- Keresztnév (first name)
- Vezetéknév (last name)
- Jelszó (password)
- Születési dátum (date of birth)



6.1.1. ábra, regisztráció

6.2. Bejelentkezés

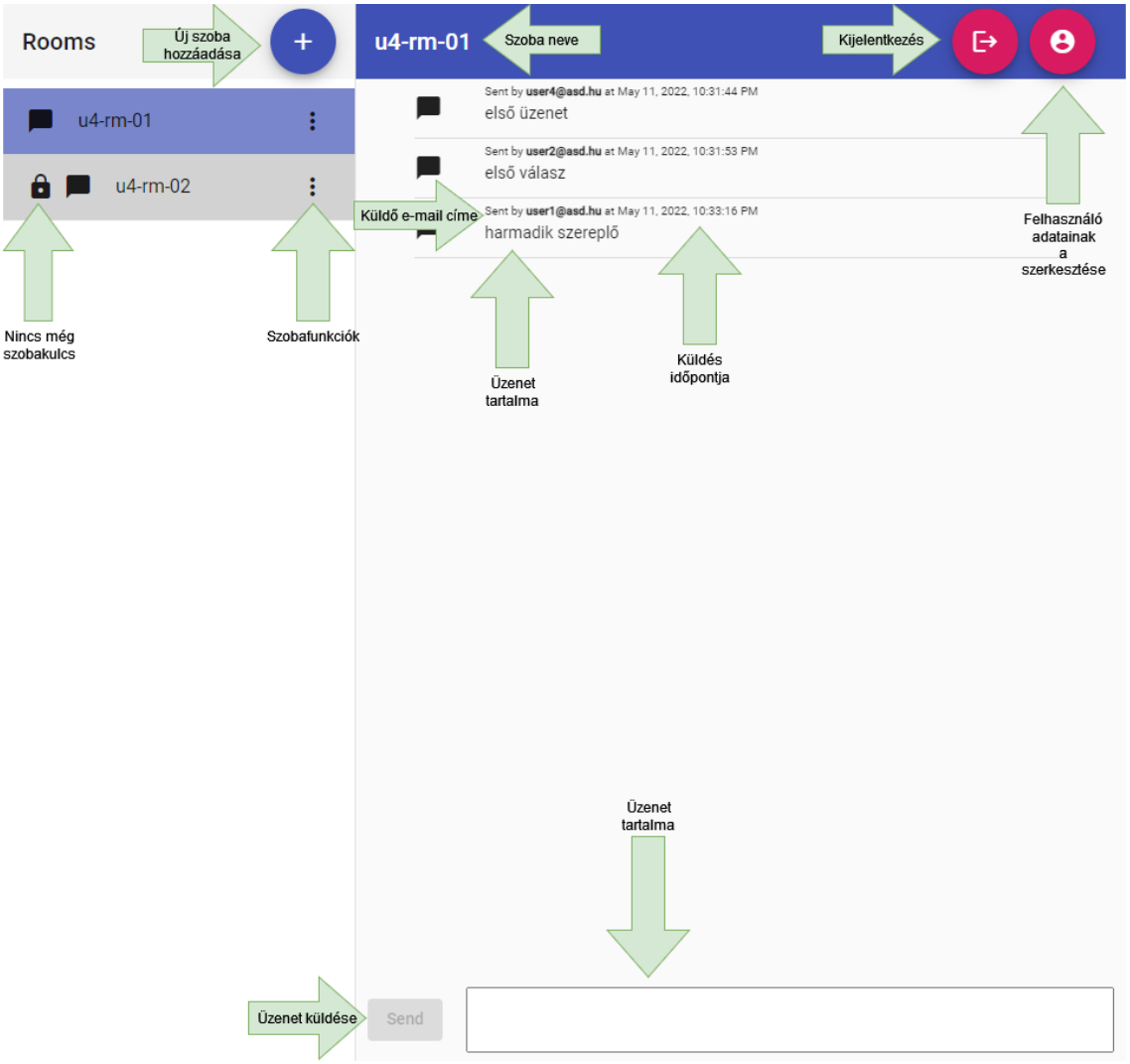
A regisztrált felhasználók bejelentkezhetnek a regisztrációs felületen megadott e-mail cím és jelszó párossal.



6.2.1. ábra, bejelentkezés

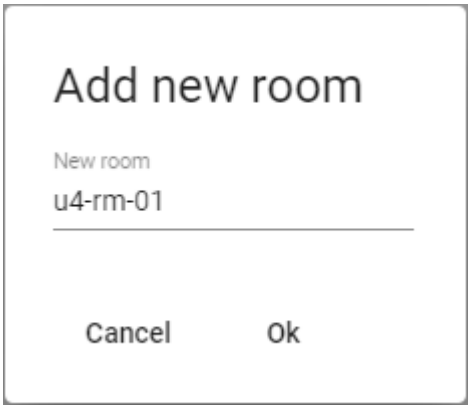
6.3. Cset nézet

A bejelentkezés után a felhasználó a csetszoba nézetre lesz irányítva. Ez a nézet csak bejelentkezve érhető el. A felhasználó minden menüpont használata során kis felugró üzenetekben értesül a műveletek eredményéről.



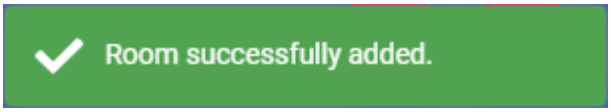
6.3.1. ábra, cset nézet

Baloldalt található a szobák listája. A szobák listájában van az új szoba hozzáadása gomb. Az új szoba létrehozásakor megnyílik egy felugró ablak, ahol csak a szoba nevét lehet megadni.



6.3.2. ábra, új szoba létrehozása

A művelet eredményéről egy felugró üzenet értesíti a felhasználót.



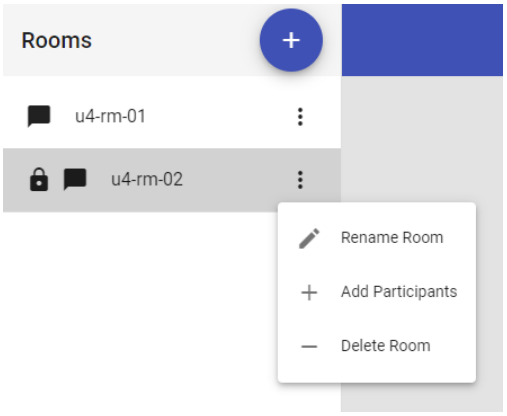
6.3.3. ábra, új szoba létrehozásának az eredménye

A szobák listájában egy kis lakat jelzi az adott szobán, ha még nem tartozik hozzá szobakulcs. Ebben az esetben a szobát nem lehet megnyitni, hiszen a felhasználó a kulcs hiányában nem tudná elolvasni az üzeneteket.

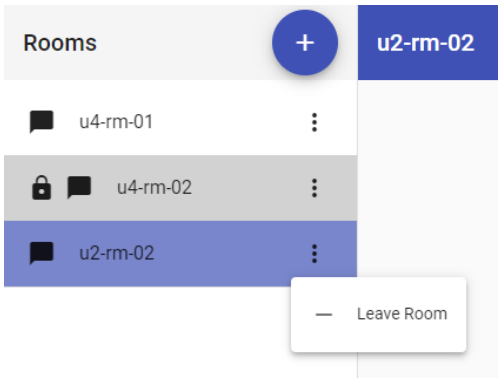
Minden szobához tartoznak beállítási lehetőségek, ehhez a menüt a szoba neve melletti függőleges három pontra kattintva lehet megnyitni.

Két féle menü létezik:

- A felhasználó a szoba létrehozója:
  - Szoba átnevezése
  - Résztvevő meghívása
  - Szoba törlése
- A felhasználó egy résztvevője – tehát nem létrehozója – a szobának:
  - Szoba elhagyása

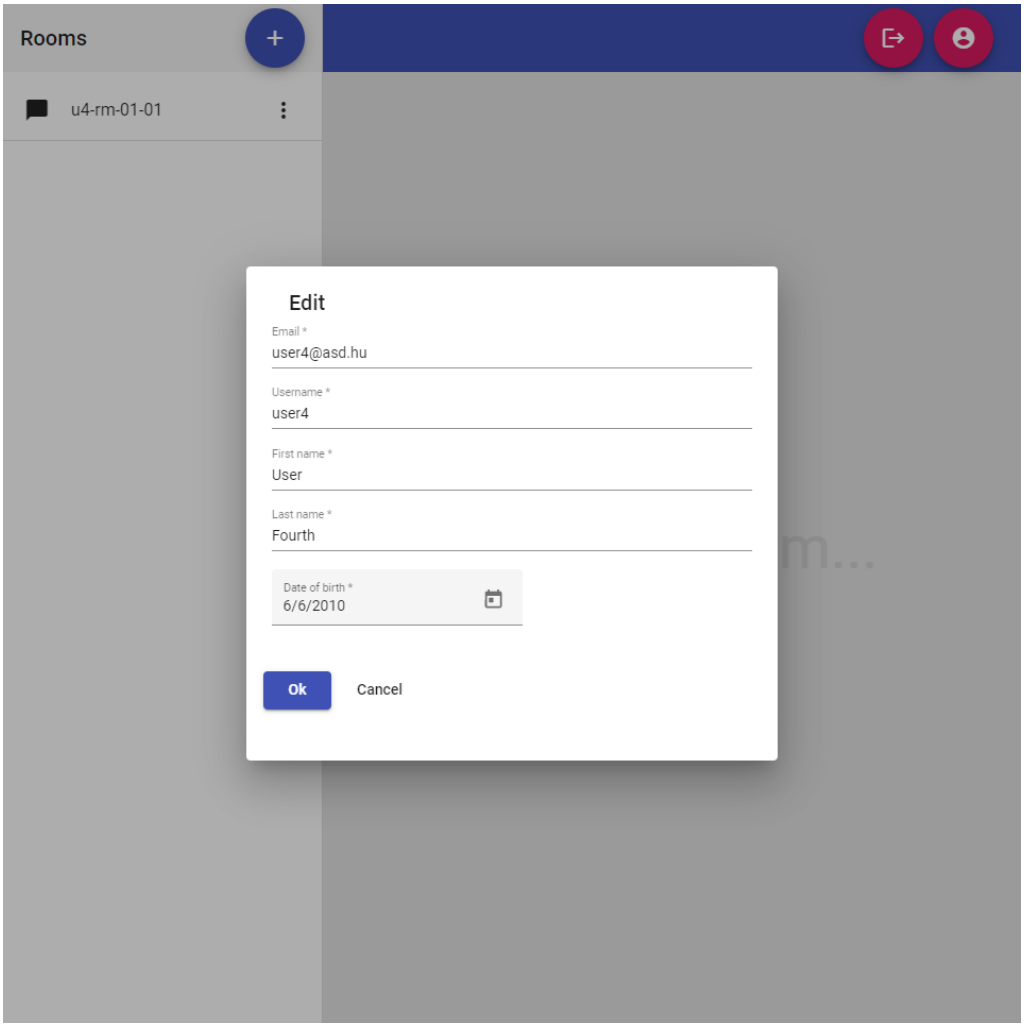


6.3.4. ábra, létrehozóként elérhető menülehetőségek



6.3.5. ábra, résztvevőként elérhető menülehetőségek

A jobb felső sarokban található felhasználó piktogramra kattintva érhető el a felhasználó adatlapja. Itt csak a felhasználó keresztnevét, vezetéknevét és születési dátumát lehet szerkeszteni.



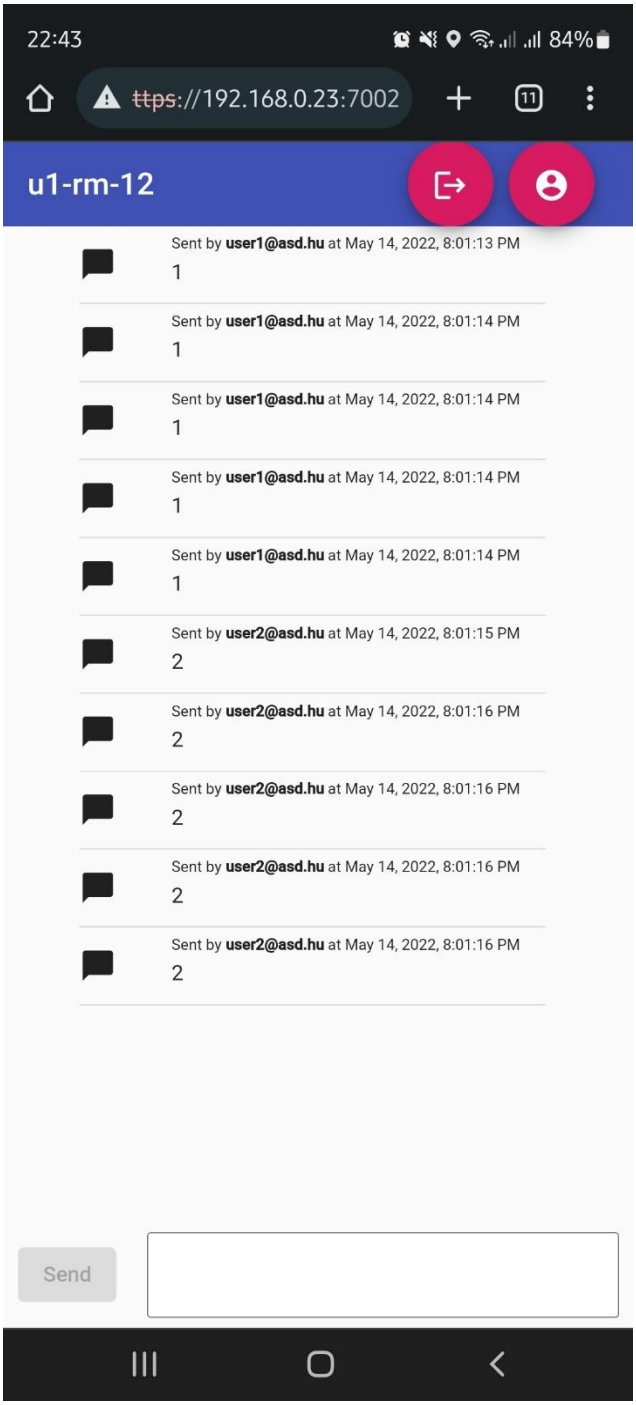
6.3.6. ábra, a felhasználó adatainak szerkesztése

A képernyő központi részén található az üzenetek listája, a beviteli szöveges mező az üzenet tartalmával és a küldés gomb. A küldés gomb csak akkor aktív, ha a felhasználó már beírt valamit.

6.4. Mobilnézet

Az alkalmazás elérhető mobilon is. Ebben a esetben, amikor egy csetszoba megnyitásra kerül, akkor a hely hiányában nem látszódnak a csetszobák.

Mobilon az ajánlott böngésző a Google Chrome.



6.4.1. ábra, mobilnézet

7. TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

Az alkalmazásban mind technikai, mind koncepcionális szinten több továbbfejlesztési lehetőséget is látok.

7.1. Koncepcionális továbbfejlesztés

A munkám során gyakran vettem részt különböző munkafolyamat támogató szoftverek, dokumentum kezelő rendszerek fejlesztésében. A dokumentumok tárolása és a munkafolyamatok menedzselése közben gyakori probléma volt az ügyfelek részéről, hogy nem volt megfelelő kommunikációs csatorna a számukra.

Például egy munkafolyamat támogató rendszer esetében a munkafolyamatok egyes lépéseiről e-mailek készülnek, amit az egyes felhasználók tovább küldenek másoknak, ám ezek hamar átláthatatlanná válnak.

Hasonló problémával szembesültem, amikor dokumentum kezelő rendszerek fejlesztésében vettem részt. Egy esetben például a projektekhez a rendszerben szobákat lehetett felvenni, a szobákba pedig dokumentumokat feltölteni. Ám az egyes dokumentumokkal kapcsolatos kommunikáció nehézkes volt.

További gondot okozott az ügyfeleknek, hogy szerették volna a dokumentumaikat úgy titkosítani, hogy még a szolgáltató se férjen hozzájuk.

Úgy gondolom, hogy egy lehetséges továbbfejlesztési irány egy Virtual Data Room létrehozása, ahol az egyes csetszobákhoz dokumentumtárat lehet rendelni és ahol a dokumentumokat akár titkosítva is lehet tárolni. Erre épülne egy következő lépés, ahol már munkafolyamat menedzsment is lenne. A különböző munkafolyamatokról és dokumentumokról a kommunikációt a cseten lehetne folytatni és az egyes lépésekhez így már egy vagy több dokumentumot is hozzá lehetne rendelni.

7.2. Technikai továbbfejlesztés

A megvalósítás során sokat tanultam a végpontok közötti titkosításról és sikerült kipróbálnom új, korábban általam csak felületesen ismert technológiákat.

MongoDB

A MongoDB hivatalos ORM-jét a szakdolgozat keretein próbáltam ki először. Még mindig nem sikerült kiismernem teljesen, de a MongoDB-re épülő adatelérési rétegnél sok helyen lehetne egyszerűsíteni a megfelelő aggregálásokkal és a megfelelő lekérdezések elkészítésével.

Titkosítás, hitelesítés

További továbbfejlesztési lehetőség egyrészt a partner, másrészt pedig minden üzenet hitelesítése az RSA algoritmus felhasználásával.

- Partner hitelesítése:
    - Ennek a folyamatnak a lényege egy session vagy munkamenet kezdetén (például kapcsolódáskor) a partnerek hitelesítése.

  - *Alice* és *Bob* kommunikálnak egymással és mindketten rendelkeznek egy-egy RSA kulcspárral és ismerik egymás publikus kulcsát, illetve titokban tartják a saját privát kulcsukat.
  - *Bob* küld egy üzenetet *Alice* számára, amit *Alice* publikus kulcsával titkosított.
  - *Alice* visszafejti az üzenetet a privát kulcsával.
  - A visszafejtett üzenetet titkosítja *Bob* publikus kulcsával és visszaküldi *Bob*nak.
  - Ha *Bob* ugyanazt az üzenetet kapja vissza, mint amit elküldött *Alice* számára, akkor *Alice* az aktuális munkamenetben egy megbízható partner.
- Üzenetek hitelesítése
  - Ennek a folyamatnak a lényege minden egyes üzenet hitelesítése egy aláírással.
  - *Alice* és *Bob* kommunikálnak egymással és mindketten rendelkeznek egy-egy RSA kulcspárral és ismerik egymás publikus kulcsát, illetve titokban tartják a saját privát kulcsukat.
  - *Bob* küld egy üzenetet *Alice* számára, aminek a publikus részéből készít egy SHA-256 hasht és azt aláírja a saját, vagyis *Bob* privát kulcsával.
  - *Alice* megkapja az üzenetet és készít a publikus részéből egy SHA-256 hasht, majd hitelesíti az aláírást *Bob* publikus kulcsával.
  - Ha *Alice* ugyanazt az eredményt kapja, mint ami az üzenet mellé volt csatolva, akkor az üzenet megbízható és fogadható.
  - Az elv kiterjeszthető a szerverre is, ebben az esetben a szerver minden kérést hitelesítene illetve aláírna.

Szobakulcsok generálása

Jelenleg a szobakulcsot mindig az adott szoba gazdája hozza létre illetve osztja meg. A szobakulcs generálása esetén előfordulhat olyan eset, hogy egyszerre több felhasználóval osztja meg a kulcsot és ilyenkor a böngészőben többször is lefut a szobakulcs létrehozása. A probléma ellen a szerveroldalon lehetne védekezni egy *lock* bevezetésével. A problémát az okozza, hogy több backend szerver is lehet, így a hagyományos értelemben vett *lock* nem működne, hiszen az teljesen véletlenszerű lenne, hogy az adott felhasználó melyik szerverhez csatlakozik és adott esetben a két szobakulcs beállítás két különböző szerveren futna le.

Ezért létre kéne egy elosztott cache megoldás, ahol az adott szoba beállítására lockot lehetne felvenni, amennyiben már elindult a szobakulcs generálás folyamata. Így ha az első szobakulcs generálása és tárolása még folyamatban van, akkor arról lehetne értesülni a többi böngészőben/szerveren. Erre szerveroldalon a Redis egy jó megoldás lenne, a klienst pedig WebSocketen keresztül, például SignalR-rel lehetne értesíteni.

8. TAPASZTALATOK

A szakdolgozat elkészítése közben több, számomra ismeretlen technológiát és saját szoftvertervezési elképzelést is kipróbáltam és alkalmaztam. Az egyik ilyen a SignalR hubokkal kapcsolatos interfészek elrejtése egy providerben a „Subjectek mögé”.

A titkosítást mindig is egzotikus problémának és kihívásnak tekintettem a szoftverfejlesztői pályafutásom során. A szakirodalmi áttekintés során több könyvet is elolvastam a témában, így lett egy átfogó képem a titkosítások felhasználásáról és működéséről. A megvalósítás során pedig a korábban gyűjtött elméleti tudást ültethettem át a gyakorlatba.

Még egy egyszerű kulcsmegosztást sem könnyű skálázható és karbantartható módon megvalósítani. Az elkészítés során több buktatóval találtam szembe magam, a „SubtleCrypto” működése nem tükrözte az előzetes elvárásaimat és ezért az eleinte megírt teszteket törölni kellett. A szakdolgozat elkészítésének első néhány hete a folyamatos refaktorálásról, újraírásról szólt, valamint a különböző koncepciókkal történő kísérletezésről.

A legelső elképzelés szerint a felhasználóknak a jelszavából egy RSA kulcspár generálódott volna, ám a „SubtleCrypto” ezt nem támogatja, ezért változtatnom kellett a koncepción.

Ez a példa jól mutatja, hogy habár a tervezés során már tisztában voltam a főbb titkosítási algoritmusokkal és azok felhasználhatóságáról, az alkalmazott technológia sajátosságait ki kellett ismerni.

9. ÖSSZEFOGLALÁS

A dolgozat célja, hogy körbejárja a lehetőségét a végpontok között titkosított kommunikáció egy lehetséges megvalósításának.

A szakirodalmi áttekintés egy átfogó képet nyújt a titkosítással kapcsolatos alapfogalmakról, algoritmusokról és elvekről. Ez lehetővé teszi, hogy meg lehessen érteni egy hasonló szolgáltatás, a Telegram titkosítási protokollját, az MTProtot és a kialakított titkosítási modellt.

Ezen ismeretek fényében készült egy vázlatos rendszerterv, amely érthetővé teszi az elkészített alkalmazás felépítését és célját.

A fejlesztői dokumentáció során ismertetésre kerültek az adatmodell, a titkosítási modell, illetve a backend és a frontend kód technológia specifikus részletei. Ezen tudás birtokában az elkészült kód megérthetővé válik. Továbbá a tesztelési dokumentáció bemutatja a tesztelés részleteit.

A felhasználói dokumentáció egy laikus felhasználó számára nyújt segítséget képernyőképekkel és az egyes funkciók részletes bemutatásával.

Végül ismertetésre kerülnek a továbbfejlesztési lehetőségek. Ez két részből áll, a koncepcionális továbbfejlesztés a lehetséges üzleti szempontokat mutatja meg, amikhez az elkészült alkalmazás már egy jó kiindulási alapot jelenthet. A technikai továbbfejlesztési lehetőségek technológiai részleteket tartalmaz, bemutatja, hogy milyen módon lehet biztonságosabbá tenni a korábban ismertetett titkosítási modellt.

A szakdolgozat alulról felfelé épül fel. Ez azt jelenti, hogy minden pont a már korábban kifejtésre került fogalmakat használ.

10. ABSTRACT

This thesis is about exploring the possibilities of end-to-end encrypted communication.

The goal of the literature review is to provide a comprehensive overview of the basic concepts, algorithms and principles related to encryption. This makes it possible to understand a similar service's encryption protocol, the MTProto and the encryption model that is being developed by the end of the thesis.

This knowledge is appropriate to comprehend the schematic system design which makes the completed application understandable.

The developer documentation contains conceptional and technology specific information about the created data model, encryption model as well as the developed backend and frontend code. This knowledge is sufficient to understand the written code. The testing documentation provides information about details of unit tests.

The user documentation was written to be understandable by an average user. It contains screenshots and detailed descriptions about the various functions of the application.

Finally, the possibilities for further development are described. It has two parts; the conceptual development provides information from a business point of view while the technical development possibilities describe the details of how to change the encryption model to be safer.

The thesis is structured from the bottom up. This means that each point uses concepts that have already been explained earlier.

11. IRODALOMJEGYZÉK

[1] https://en.wikipedia.org/wiki/History_of_communication#cite_note-:2-1, utoljára megtekintve: 2021.12.08.

[2] https://en.wikipedia.org/wiki/History_of_communication#cite_note-Paul_Martin_Lester-2, utoljára megtekintve: 2021.12.08.

[3] https://en.wikipedia.org/wiki/History_of_communication#Petroglyphs, utoljára megtekintve: 2021.12.08.

[4] https://en.wikipedia.org/wiki/History_of_communication#Pictograms, utoljára megtekintve: 2021.12.08.

[5] https://hu.wikipedia.org/wiki/Hieroglif_%C3%ADr%C3%A1s, utoljára megtekintve: 2021.12.08.

[6] https://hu.wikipedia.org/wiki/%C3%81b%C3%A9c%C3%A9, utoljára megtekintve: 2021.12.08.

[7] https://hu.wikipedia.org/wiki/T%C3%B6megkommunik%C3%A1ci%C3%B3, utoljára megtekintve: 2021.12.08.

[8] https://hu.wikipedia.org/wiki/T%C3%A1v%C3%ADr%C3%B3, utoljára megtekintve: 2021.12.08.

[9] https://hu.wikipedia.org/wiki/Samuel_Morse, utoljára megtekintve: 2021.12.08.

[10] https://hu.wikipedia.org/wiki/Telefonk%C3%A9sz%C3%BCl%C3%A9k, utoljára megtekintve: 2021.12.08.

[11] https://hu.wikipedia.org/wiki/Morzek%C3%B3d, utoljára megtekintve: 2021.12.08.

[12] https://hu.wikipedia.org/wiki/Internet, utoljára megtekintve: 2021.12.08.

[13] https://hu.wikipedia.org/wiki/Titkos%C3%ADt%C3%A1s, utoljára megtekintve: 2021.12.08.

[14] Piper, F., Murphy, S.: Cryptography - A Very Short Introduction. *Oxford*, 2002

[15] Piper, F., Murphy, S.: Cryptography - A Very Short Introduction. *Oxford*, 2002

[16] Piper, F., Murphy, S.: Cryptography - A Very Short Introduction. *Oxford*, 2002

[17] Virasztó, T.: Titkosítás és adatrejtés - Biztonságos kommunikáció és algoritmikus adatvédelem. *Netacademia*, 2004, 9. oldal

[18] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 40. oldal

[19] Virasztó, T.: Titkosítás és adatrejtés - Biztonságos kommunikáció és algoritmikus adatvédelem. *Netacademia*, 2004, 9. oldal-10. oldal

[20] Virasztó, T.: Titkosítás és adatrejtés - Biztonságos kommunikáció és algoritmikus adatvédelem. *Netacademia*, 2004, 49. oldal

[21] Virasztó, T.: Titkosítás és adatrejtés - Biztonságos kommunikáció és algoritmikus adatvédelem. *Netacademia*, 2004, 49. oldal – „16. ábra”

[22] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 28. oldal

[23] https://hu.wikipedia.org/wiki/Caesar-rejtjel, utoljára megtekintve: 2021.12.08.

[24] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 29. oldal-30. oldal

[25] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 30. oldal – „Figure 1-3 The Vigenère cipher”

[26] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 35. oldal

[27] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 35. oldal

[28] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard, utoljára megtekintve: 2021.12.08.

[29] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 107. oldal

[30] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 107. oldal - 108. oldal

[31] Virasztó, T.: Titkosítás és adatrejtés - Biztonságos kommunikáció és algoritmikus adatvédelem. *Netacademia*, 2004, 81. oldal-82. oldal – 3. Szimmetrikus kulcsú módszerek

[32] Virasztó, T.: Titkosítás és adatrejtés - Biztonságos kommunikáció és algoritmikus adatvédelem. *Netacademia*, 2004, 101. oldal – 4. Nyilvános kulcsú módszerek

[33] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 47. oldal -48. oldal

[34] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 47. oldal -48. oldal

[35] https://hu.wikipedia.org/wiki/Nyilv%C3%A1nos_kulcs%C3%BA_infrastrukt%C3%BAra, utoljára megtekintve: 2021.12.08.

[36] Virasztó, T.: Titkosítás és adatrejtés - Biztonságos kommunikáció és algoritmikus adatvédelem. *Netacademia*, 2004, 328. oldal – 15. Kislexikon

[37] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 300. oldal – 11 Diffie-Hellman

[38] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 301. oldal – 11 Diffie-Hellman

[39] Virasztó, T.: Titkosítás és adatrejtés - Biztonságos kommunikáció és algoritmikus adatvédelem. *Netacademia*, 2004, 101. oldal-102. oldal – 4. Nyilvános kulcsú módszerek

[40] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 273. oldal – 10 RSA

[41] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 273. oldal – 10 RSA

[42] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 170. oldal – 6 Hash Functions

[43] Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 170. oldal – 6 Hash Functions – „Figure 6-1”

[44] Virasztó, T.: Titkosítás és adatrejtés - Biztonságos kommunikáció és algoritmikus adatvédelem. *Netacademia*, 2004, 117. oldal – 4.5. Hibrid kriptorendszerek

[45] https://en.wikipedia.org/wiki/Telegram_(software) , utoljára megtekintve: 2021.12.08.

[46] https://telegram.org/faq#q-what-is-telegram-what-do-i-do-here, utoljára megtekintve: 2021.12.08.

[47] https://core.telegram.org/mtproto/description, utoljára megtekintve: 2021.12.08.

[48] https://core.telegram.org/mtproto/description, utoljára megtekintve: 2021.12.08.

[49] https://core.telegram.org/mtproto/description#terminology, utoljára megtekintve: 2021.12.08.

[50] https://core.telegram.org/mtproto#mtproto-transport, utoljára megtekintve: 2021.12.08.

[51] https://core.telegram.org/mtproto#mtproto-transport, utoljára megtekintve: 2021.12.08.

[52] https://dotnet.microsoft.com/platform/support/policy, utoljára megtekintve: 2021.12.08.

[53] https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html, utoljára megtekintve: 2021.12.08.

[54] https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html, utoljára megtekintve: 2021.12.08.

[55] https://docs.mongodb.com/drivers/csharp/, utoljára megtekintve: 2021.12.08.

[56] https://docs.microsoft.com/en-us/ef/core/, utoljára megtekintve: 2021.12.08.

[57] https://www.mongodb.com/nosql-explained/nosql-vs-sql, utoljára megtekintve: 2021.12.08.

[58] https://angular.io/, utoljára megtekintve: 2021.12.08.

[59] https://material.angular.io/, utoljára megtekintve: 2021.12.08.

[60] https://itnext.io/choosing-a-highly-scalable-folder-structure-in-angular-d987de65ec7, utoljára megtekintve: 2021.12.08.

[61] https://en.wikipedia.org/wiki/Git, utoljára megtekintve: 2021.12.08.

[62] https://azure.microsoft.com/en-us/services/devops/, utoljára megtekintve: 2021.12.08.

[63] https://azure.microsoft.com/hu-hu/pricing/details/devops/azure-devops-services/, utoljára megtekintve: 2021.12.08.

[64] https://en.wikipedia.org/wiki/Docker_(software) , utoljára megtekintve: 2021.12.08.

[65] https://hu.wikipedia.org/wiki/Tesztvez%C3%A9relt_fejleszt%C3%A9s, utoljára megtekintve: 2021.12.08.

[66] https://en.wikipedia.org/wiki/Virtual_data_room, utoljára megtekintve: 2021.12.08.

[67]https://www.nuget.org/packages/automapper/, utoljára megtekintve: 2022.05.08.

[68]https://jwt.io/introduction, utoljára megtekintve: 2022.05.08.

[69]https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API, utoljára megtekintve: 2022.05.08.

[70]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto, utoljára megtekintve: 2022.05.08.

[71]https://developer.mozilla.org/en-US/docs/Web/API/CryptoKey, utoljára megtekintve: 2022.05.08.

[72]https://developer.mozilla.org/en-US/docs/Web/API/CryptoKeyPair, utoljára megtekintve: 2022.05.08.

[73]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer, utoljára megtekintve: 2022.05.08.

[74]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey, utoljára megtekintve: 2022.05.08.

[75]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/importKey, utoljára megtekintve: 2022.05.08.

[76]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/exportKey, utoljára megtekintve: 2022.05.08.

[77]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/deriveKey, utoljára megtekintve: 2022.05.08.

[78]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/deriveBits, utoljára megtekintve: 2022.05.08.

[79]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/wrapKey, utoljára megtekintve: 2022.05.08.

[80]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/unwrapKey, utoljára megtekintve: 2022.05.08.

[81]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/digest, utoljára megtekintve: 2022.05.08.

[82]Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 233. oldal

[83]Aumasson, J.-P.: Serious Cryptography - A Practical Introduction to Modern Encryption. *No Starch Press*, 2018, 334. oldal

[84]Laurens Van Houtven.: Crypto101. *Magánkiadás*, 2018, 139. oldal

[85]https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/digest#converting_a_digest_to_a_hex_string, utoljára megtekintve: 2022.05.08.