Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Nagy Viktor

KÁRTYAJÁTÉK MEGVALÓSÍTÁSA

ASP.NET Core és Angular platformon

KONZULENS

Dr. Kővári Bence András

BUDAPEST, 2021

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2021. 12. 09.

...…….
Nagy Viktor

Összefoglaló

A szórakozás egyik fontosabb eleme lehet a közösségi élmény, nem véletlen, hogy a társasjátékok már évezredek óta a köztudatban vannak. A 21. század az informatikának köszönhetően elhozta azt a lehetőséget, hogy ezt az élményt már úgy is átélhetjük, hogy nincs szükség élőben találkozni társaiddal. Emiatt a többszereplős internetes játékok nagy népszerűségnek örvendenek.

Célunk az volt, hogy ezt az igény kielégítve elkészítsünk egy olyan alkalmazást, mely nem csak a társasjátékok közösségi élményét, de az online játékok kényelmét is kielégíti. Ezért készítettük el a BANG! nevezetű társasjátéknak az online verzióját. Ez egy körökre osztott kártyajáték mely célja, hogy az általad játszott karakter nyerjen a többiekkel szemben.

A szoftvert a lehető legújabb és legfrissebb technológiák felhasználásával próbáltuk elkészíteni. Ezáltal nem csak program minőségén emelni, de a saját szaktudásunkat is fejleszteni.

A többes szám használata nem véletlen ugyanis, hogy minél több mindent el tudjunk készíteni a játékból ezalatt a rövid idő alatt, így társammal úgy döntöttünk, hogy ketten fogjuk elkészíteni azt. A közös munka gördülékenyen ment, ugyanis a tervezési folyamatot leszámítva szét tudtuk választani egymástól a munkát, így elegendő volt a minimális mennyiségű kommunikáció.

Végeredményében sikerült is elkészíteni az alkalmazást játszható szintre, azonban ez nem jelenti azt, hogy ne lehetne rajta fejleszteni. Ez egy viszonylag komplex társasjáték így rengeteg opciót kellett leimplementálni, de fel voltunk rá készülve, hogy sok munkaidőt kell beleraknunk.

Azonban egy-két hónap sajnos nem elég arra, hogy egy ilyen nehézségű társasjáték teljesen hibátlan szinten működjön, de megtettünk minden tőlünk telhetőt, hogy játszható legyen a játék, melyet később könnyű továbbfejleszteni és javítani.

Abstract

One of the most important elements of entertainment can be a social experience, and it is no coincidence that board games have been around for thousands of years. The 21st century, thanks to information technology, has brought the possibility of having this experience without the need to meet your peers in person. This is why multiplayer online games are so popular.

Our goal was to meet this need by creating an application that not only offers the social experience of board games, but also the convenience of online gaming. That's why we created an online version of the board game BANG! It's a turn-based card game where the goal is for the character you play to win against the others.

We tried to make the software using the latest and most up-to-date technologies. In this way, we not only improve the quality of our software, but also our own skills.

The use of the plural is not a coincidence, because we wanted to make as much of the game as possible in such a short time, so my partner and I decided to make it together. Working together went smoothly, as we were able to separate our work apart from the design process, so that a minimum amount of communication was sufficient.

In the end, we managed to get the application to a playable level, but that doesn't mean that it can't be improved. It's a relatively complex board game so there were a lot of options to implement, but we were prepared to put in a lot of work.

However, a month or two is not enough to get a board game of this difficulty to a completely flawless level, but we have done our best to make the game playable and easy to develop and improve later.

1 Bevezetés

1.1 Általános bevezető

Ma már szinte elképzelhetetlen lenne az élet informatika nélkül. Ez az, ami meghatározza az életünk rengeteg aspektusát. A mindennapjaink részévé váltak nem csak szórakozás téren, de akár megélhetési szinten is, ugyanis ma már ritka egy olyan munkahely, ahol ne használnának minimális szinten se informatikai eszközöket. Akár említhetem a szakdolgozat írását is, mert annak is konkrét követelményi vannak, hogy az adott szövegszerkesztőben mit, hogyan kell beállítani.

Ahogy halad az idő előre, úgy fejlődnek nem csak az informatikai eszközök, mint például számítógép, játékkonzolok, telefonok, hanem az informatikai szoftverek is. Erre szükség is van ugyanis mivel egyre jobban behálózza a napjainkat ezért azok a szoftverek fognak fennmaradni, amelyek a legfelhasználóbarátabbak és leginkább kielégíti az adott társadalom igényeit.

Az elmúlt időszakban kifejezett figyelmet kaptak a szórakoztatás céljából készült szoftverek is. Hatalmas iparágaknak az egyik fő profilja is a játékszoftverek/konzolok fejlesztése és készítése (Microsoft, Sony, Nintendo), így ez megkerülhetetlen egy átlagember számára is. Egy komolyabb játék elkészítése felérhet akár egy mozifilm költségeivel, például a 2020-ban készül „Cyberpunk 2077” nevű játéknak a fejlesztési és marketing költségei meghaladták a 300 millió dollárt is [1]. Azonban nem csak a konzolos/számítógépes játékoknak van hatalmas népszerűsége, hanem az online böngészőn keresztül játszható játékoknak is.

1.2 Internetes játékok, társasjátékok

Kiskoromban is rengeteg weboldal épült arra, hogy a fejlesztők által készített ügyességi/logikai és egyéb internetes játékokat összefogja. Ilyenek voltak például az „Y8” és a „Startlap” weboldalak. Ezek mind Adobe Flash technológiára épültek, amely egy az internetes grafikát megjelenítő szoftver volt. Azonban biztonsági hibái miatt ennek a szoftvernek 2021 elején megszűnt a támogatása [2].

Azonban itt kitérnék a társasjátékok népszerűségére is és a kapcsolatukra az internetes játékokkal. A társasjáték fogalma nagyon régre nyúlik vissza, feltárások bizonyítják, hogy már időszámításunk előtt is léteztek. Így nem csoda, hogy manapság is

hatalmas népszerűségnek örvendenek. Eléggé gyakoriak azok a társasjátékok is melyek kaptak internetes megfelelőt is (pl.: Honfoglaló, Monopoly, Activity, Uno).

Ezért úgy gondoltuk, hogy mi is kihasználjuk ennek a két területnek a népszerűségét és megalkotjuk egy bonyolultabb társasjáték internetes megfelelőjét. A többesszám használata nem véletlen, erre a következő fejezetben ki is térek.

1.3 Közös munka

A projektet alapvetően ketten csináltuk Markovics Gergellyel, ugyanis szerettünk volna egy minél nagyobb és részletesebb szoftvert alkotni, ami később akár ténylegesen használható lesz bárki számára. Az önálló laboratórium tárgynál már közösen dolgoztunk és könnyen tudtunk együtt haladni, így célszerűnek gondoltuk, hogy a szakdolgozathoz készült munkánkat is közösen hozzuk létre.

Azonban a korábbi munkánkkal ellentétben, ahol nem azon volt a hangsúly, hogy minél inkább szétszedjük a megoldandó problémákat, idén úgy gondoltuk célszerűnek, hogy jobban leválasztjuk magunkat a másik munkafolyamatáról és jobban elhatárolható részeken dolgozzunk, ne függjünk attól, hogy a másik, hogy halad és mit csinál éppen. Ez az elején még nem kivitelezhető, ugyanis a tervezési folyamat alatt kénytelenek voltunk az alapvető felépítést, specifikációt és kinézeti elemeket megbeszélni és elkészíteni.

Kérdéses volt, hogy hogyan szeretnénk szétosztani a feladatköröket. Az első ötlet az volt, hogy backend-frontend vonalon választjuk el azonban ezt elvetettük. Végül azt a megoldást választottuk, hogy ugyan mindketten fogunk backend és frontend feladatokat is készíteni, de inkább funkció szerint osztjuk ketté a munkát.

Ez azt jelentette, hogy míg a társam készítette magát a játék logikáját és kinézetét (belesegítettem a tervezés részénél és a felhasznált képek elkészítésénél), én foglalkoztam minden olyan feladatkörrel, ami nem a konkrét játékmenethez tartozik. A specifikáció rész menü rendszer pontja (2.1.1) alatt részletezem, hogy mi volt a konkrét feladat, úgyhogy itt csak felületesen térek ki rá felsorolás szerűen.

1.4 Saját feladatrészek

Az elején a közös munka részeként foglalkoztam a program alapvető tervezésén, ideértve az adatbázis kezdetleges megtervezését, a használati esetek elkészítését és a

felülethez használt skiccek felrajzolását. Emellett átbeszéltük, hogy milyen architektúrát lenne érdemes használnunk. Későbbi munkafolyamat során besegítettem társamnak a felületéhez tartozó kártyák képeinek elkészítésével (46 darab kártya azonos méretűre vágása lekerekített széllel, a kártyákhoz tartozó franciakártya jelölések elkészítése és egyéb kisebb képek).

A következőnek felsorolt feladatok közül a backend és frontend rész implementálása is a feladatom volt. Elsőként létre kellett hoznom a projekt alapját, tehát működnie kellett az adatbázisnak, a mikroszolgáltatásoknak és az azokon belüli architektúrális felépítésnek. Ezek után a bejelentkezés és regisztráció implementálása volt a legfontosabb feladat, ugyanis később minden fontosabb fejlesztés erre az alapra épült. Egyéb feladataim voltak még például a barát rendszer kialakítása, a váró rendszer és egyéb ezekhez kapcsolódó feladatok megoldása. (pl. felhasználóhoz kötött eredmények mutatása, chat rendszer kialakítása)

Úgy próbáltuk szétosztani, hogy míg nálam a technológiai megoldások legyenek a hangsúlyosak addig társamnál inkább logikai megoldások implementálása. Azonban az én feladatkörömben is próbáltunk minél több hasznos funkciót meghatározni, ám ezt nem lehet olyan bonyolultsági szintre hozni logikai funkció téren, mint egy társasjáték logikát.

1.5 A dolgozat szerkezete

A bevezető részt leszámítva alapvetően 4 nagyobb egységre lehet bontani a dolgozat felépítését. Az első részben ugyanis egy általános képet adtam arról, hogy mi az alap koncepció, miről fog szólni az elkészített program.

A következő pont egy részletesebb követelményspecifikáció lesz, ahol beszélek a rendszer kialakításáról, az adatbázis felépítéséről, a program architektúrájáról és a használati esetekről.

Majd ezt követi az önálló munka minél szélesebbkörű és részletgazdagabb bemutatása. Ez egy hosszabb blokk lesz, ahol bemutatom a program szerkezetét a backendtől a frontendig haladva. Tehát ennek a résznek a felépítése nem funkciók szerint fog szétosztásra kerülni, hanem az architektúrában lentről felfelé haladva kerül bemutatásra. Kitérek minden számomra érdekesebbnek vagy bonyolultabbnak talált programelemre és logikai megoldásra. Alapvetően akartam egy külön technológia fület, ahol leírom az általam felhasznált technológiák működését, azonban feleslegesnek

éreztem, ugyanis amikor a használatáról van szó ott röviden alapból is kitértem volna a jelentésére. Úgyhogy arra jutottam, hogy inkább akkor részletesebben leírom az adott résznél.

Végül zárásként megpróbálom objektívan értékelni az elért eredményt, összegezni, hogy mire jutottunk, mik voltak a nehézségek, mit tanultam a feladat megoldása során, mire mekkora hangsúlyt fektettem. Ezen felül leírom, hogy miben lehetne még továbbfejleszteni a programot, mi az, amit terveztünk bele, de nem sikerült időben megvalósítani.

2 Tervezés, architektúra

2.1 Specifikáció

A program mivel szétoszthatóra két részre (játék logika és minden más) ezért különválasztom őket, azonban a második részéről nem fogok nagyon pontos specifikációt adni, ugyanis abban csak a tervezés szintjén vettem részt, nem én finomítottam és implementáltam. Ahogy azt a bevezetőben (1.3) említettem én feleltem a játék egyéb működtető elemeiért, ami nem maga a játszható játék. (pl.: authentikáció, várórendszer, menü, chat, barátrendszer stb.)

2.1.1 A menü rendszer

A szoftver elindításakor egy bejelentkező felület fogad minket. Itt meg tudjuk adni a felhasználónevünket (aminek muszáj egyedinek lennie) és a jelszavunkat (kötelező tartalmaznia legalább egy nagy betűt és kisbetűt, és legalább egy számot). Azonban, ha még nem regisztráltunk nem fogunk tudni belépni, ilyenkor a „Bejelentkezés” gomb alatt át tudunk lépni a regisztrációs felületre. Itt ugyanúgy meg kell adni a felhasználónév-jelszó párost, azonban meg kell erősíteni a jelszót, ugyanis, ha a két jelszó bemenet nem egyezik nem engedélyezi a regisztrációt. Sikeres regisztráció esetén visszakerülünk a bejelentkező felületre, ahol már helyes adatok megadása után be fogunk tudni jelentkezni. Innen fogunk átkerülni a menübe.

A menüt alapvetően két részre lehet osztani, a bal oldali fő részén jelennek meg azok az opciók, amelyeket végrehajthatunk a menüre vonatkozóan. Ilyen például a váró létrehozása, a váróba becsatlakozás jelszó alapján, a korábbi játékok eredményeinek megtekintése, a kijelentkezés és a profil törlése. A kijelentkezés az visszadob a bejelentkező felületre. a felhasználó törlése ugyanezt csinálja, csak mellette kitörli a felhasználót így később ugyanazokkal az adatokkal nem lehetséges újra bejelentkezni. A jobb oldalon pedig egy barátlista található.

A barátlista két felületen jelenik meg, a menüben és a váróban. Annyi a különbség a kettő között, hogyha a felhasználó egy varóban van akkor tud küldeni meghívást egy barátjának, hogy ő is csatlakozzon be, míg a menüben nincs ilyen opció. A lista három részre van osztva. Van egy felsorolás azokról, akik ténylegesen a barátaink, van alatt egy kisebb lista rész, amely azokat az embereket tartalmazza, akik bejelöltek minket

ismerősnek azonban még nem jelöltük vissza őket. Végül van legalul egy kis rész annak, hogy barátnak tudjunk jelölni új embereket név alapján (ezért is szükséges az egyedi név). Egy barátjelölést el lehet fogadni, ilyenkor bekerül a legfelső bartálistába, azonban el is lehet távolítani a jelölés kérelmet. A már bejelölt barátokat is lehetséges törölni. Ha egy váróban lévő barátunk küldött nekünk egy meghívást akkor mi azt el tudjuk fogadni, azonban, ha mi is váróban vagyunk erre nincs lehetőség. Ugyanez a helyzet, ha azonos váróban vagyunk, ott le is van tiltva a meghívás küldése.

Egy váróba többféleképpen is eljuthatunk. Ha létrehozunk egyet magunknak, ha elfogadjunk egy barátunk meghívását vagy ha a váró saját jelszavát használva belépünk oda. Minden várónak van egy tulajdonosa, ő az, aki létrehozta az adott várót, az ő jogában áll elindítani a játékot. Ha elhagyja a várót akkor ez a jog véletlenszerűen át fog kerülni más, a váróban tartózkodó játékosra. Ha minden játékos elhagyta a várót, akkor az törlődik és nem lehet oda visszalépni újra. A felületén láthatjuk az adott váró jelszavát, a benne tartózkodó emberek nevének listáját, egy indító gombot (csak a birtokos számára) amely csak akkor megnyomható, ha legalább 4-en tartózkodnak benne (maximum 7), és egy váró elhagyása gombot, melyet megnyomva el lehet hagyni a várót és vissza lehet kerülni a menübe.

Az eredmények egy egyszerű felület, itt egy listában fel van sorolva az, hogy az utolsó tíz játékban melyik karaktertípussal és hányadik helyezést értünk el.

A váróban és magában a játékban van egy beszélgető felület. Itt az adott váróban, majd később a játékban résztvevő játékosok tudnak egymással beszélgetni. Egy beszélgetés bejegyzés tartalmazza azt, hogy ki írta az üzenetet és magát az üzenetet kettősponttal elválasztva.

2.1.2 A játék rendszer

A váróból való indítást követően megjelenik mindegyik benne szereplő felhasználó számára maga a játéktér, amely több fő elemet is tartalmaz. A középen elhelyezkedő asztalt, ahonnan a lapokat kell húzni és visszarakni, A saját táblánkat és a többi játékos tábláját. Emellett egy beszélgetés funkciót.

A játék egy körökre osztott kártyajáték. Minden játékos véletlenszerűen megkap egy szerepet az elején, azonban senki nem ismeri a másik szerepét (sheriff, renegát, bandita, sheriffhelyettes), kivétel a sheriff, mert róla mindenki tudja kicsoda. A szerep mellett mindenki kap egy karaktert, ezt már a többi játékos is láthatja. Minden karakternek

különböző képességei vannak, például több életpont, több kártyát húzhat stb. A játék célja, hogy te és a veled azonos szerepeket kapó játékosok maradjanak az utolsók, akik életben maradnak. A banditák célja úgy megölni a sheriffet, hogy nem buknak le, mert ha lebuknak akkor a sheriffhelyettesek tudják, hogy kik elől kell megvédeniük a sheriffet, így könnyebben ki tudják őket iktatni. A renegát mindenkin kívül áll, neki csak annyi a feladata, hogy egyedül maradjon. A sheriff és a sheriffhelyettesek csak akkor nyerhetnek, ha életben tartják a sheriffet és megölik mind a banditákat, mind a renegátot.

A játékot segítik elő fegyverek és egyéb lapok, amiket nem fogok részletesen bemutatni. A fegyverekkel távolabbra tudunk lőni, ugyanis a két legfőbb lap a „Bang" és a „Nem talált". Ezekkel tudjuk a többi játékos életét csökkenteni, vagy épp a saját életünket megmenteni. Olyan kártyák színesítik még a j

Egy adott körben miután középről húztál két (vagy több karaktertől függően) kártyát, annyit játszhatsz ki amennyit szeretnél. Azonban arra figyelni kell, hogy maximum az aktuális életerőddel azonos mennyiségű kártya lehet a kezedben, mielőtt továbbadnád a kört. Ha esetleg nem sikerült ennyi kártyát kirakni, akkor a kör vége előtt kénytelen vagy őket eldobni.

2.2 A rendszer architektúrájának felépítése

2.2.1 Adatbázis, SQL és NoSQL összehasonlítás

Ebben a részben nem a projekt felépített adatbázisáról fogok írni, ugyanis arról a 3.1.3 pontban fogok kitérni, amikor a program részletes bemutatását prezentálom. Itt inkább egy összehasonlítás lesz különböző adatbázis típusok között és hogy melyiket és miért választottuk.

A program készítése elején ez volt a legelső kérdések egyike, hogy milyen adatbázist szeretnénk használni. Ugyanis elgondolkoztunk, hogy esetleg egy NoSQL-es adatbázist választunk, mint mondjuk a MongoDB [7] vagy a Elasticsearch [8].

A következő részben alapvetően két fontosabb különbségre fogok kitérni, ami számunkra meghatározó volt. Az egyik alapvető különbség a két adatbázis típus között az adatok kezelésénél és tárolásánál jön elő, a másik a két típus skálázódásánál.

Az SQL-es adatbázisok esetében az adatok táblákban vannak tárolva és céljuk, hogy egy adatot ne tároljunk redundánsan, vagyis ne jelenjen meg feleslegesen többször

az adatbázisban. Ezért a táblák között külső kulcsok segítségével jönnek létre a kapcsolatok, ami a lekérdezéskor elég hosszú művelet lehet, ugyanis nem csak a számunkra lényeges adatot kapjuk meg egy tábla lekérdezésekor és több táblát is bejárhatunk egy adott információ megszerzéséhez. Az adatok karbantartása is egyszerűbb feladat, ugyanis mivel kulcsokkal hivatkoznak egymásra táblák, ezért, ha megfelelő a beállítás akkor például törléskor vagy módosításkor nem kell rájuk külső kulccsal hivatkozó táblákat is manuálisan lefrissíteni, megoldja a rendszer.

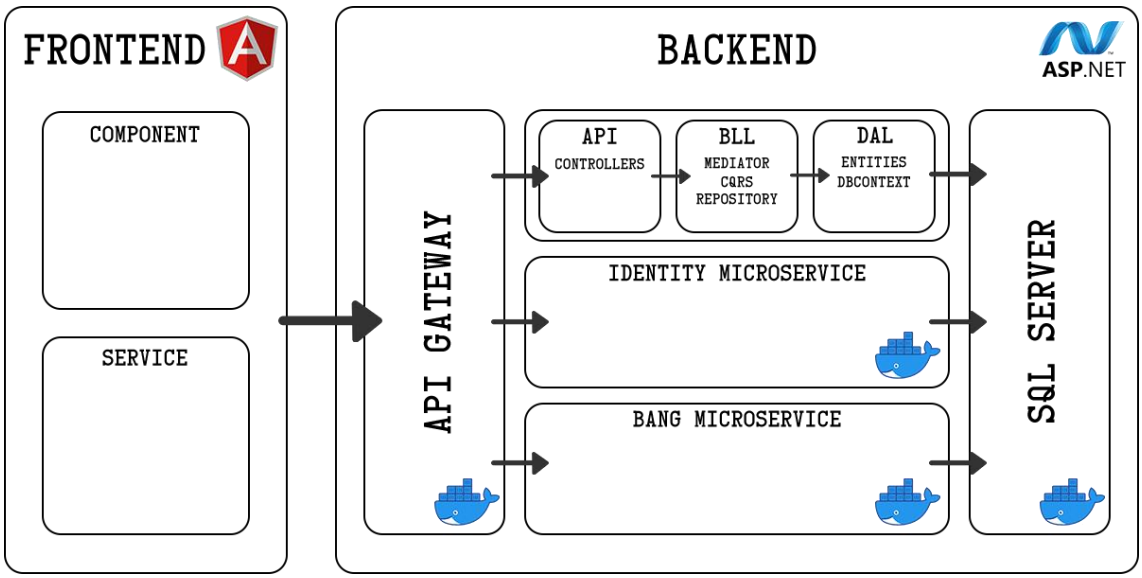
A NoSQL-es adatbázisok azonban teljesen másként kezelik az adatokat. Mivel itt nem számít az, hogy ha egy bizonyos adat többször is el van tárolva, ugyanis célunk egy ilyen adatbázis felépítésekor, hogy az aktuális tábla lekérdezésekor csak a számunkra értékes adatot kapjuk vissza. Azonban ilyenkor nekünk kell tudnunk lekezelni, azt, ha módosítás történik az egyik táblát (MongoDB-ben a táblákat kollekcióknak hívják), akkor azt mindenhol máshol, ahol el van tárolva módosítsa. Erre nyújtana megoldás a RabbitMQ például, amelynek feladata, hogy az adatokat egységesen tudjuk kezelni. Miután feliratkozunk egy adott eseményre, akkor az utána arra az eseményre kiküldött adatváltozásokat el fogjuk tudni kapni és véghez vinni. Itt mivel az adatok JSON formátumban vannak eltárolva, ezért nincsen megszabva szigorúan, hogy az adott érték az milyen típusú ellentétben a relációs adatbázisokkal szemben szemben.

A két adatbázis típus skálázódásában is különbség van. Mindkettő típus jól skálázódik vertikálisan, azonban horizontálisan csak a NoSQL-es adatbázisok skálázódnak jól. Ez azt jelenti, hogy a míg az egyik akkor fog tudni gyorsabban működni, ha az adott szervert fejlesztjük akár több memóriával vagy gyorsabb processzorral (vertikális). A másik pedig azt jelenti, hogy a bejövő kérések lesznek szétosztva.

Összességében úgy döntöttünk, hogy a relációs adatbázis sémát fogjuk választani, ugyanis azzal korábban is volt tapasztalatunk. Igaz ez egy sokkal jobban megkötött és szigorúbb elvek betartását követő módszer, azonban számunkra nem volt például szükség arra, hogy az táblák felépítését rendszeresen módosítsuk, ami itt migrációval érhető el. Ha a MongoDB-t választottuk volna, akkor egy tényleg gyorsabban működő módszerrel dolgoztunk volna azonban mindkettőnk számára hátrányt jelentett volna a kissé ismeretlenebb adatstruktúra, amit külön kell menedzselni is. Mert, ugye, mint említettem szükség lett volna mellé a RabbitMQ használatára is az adatok inkonztisztenciájának elkerülése végett.

2.2.2 Architektúra

Két nagy részre lehet szétbontani a projektet, egy frontend részre, amely Angular technológián alapul és egy backend részre mely .NET-ben íródott. A két rész hálózati hívásokon keresztül, úgynevezett REST API (Representational State Transfer) használatával kommunikál egymással. Erről részletesebben a 3.2 pontban fogok említést tenni, ahol a szerver és a kliens kommunikációját fejtem ki.



1. ábra: Architektúra

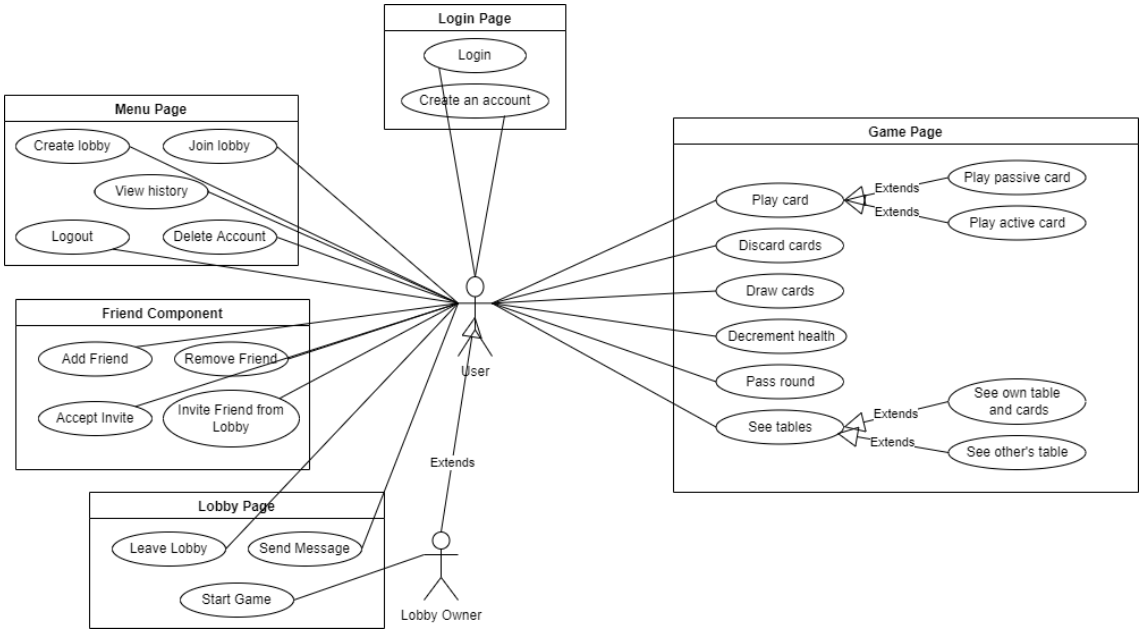
A frontend komponensei felhasználják a szervízeket, amelyek intézik a kommunikációt a szerver oldallal. Ezek a szervízek ráhívnak egy átjáróra, amely továbbítja a megfelelő mikroszolgáltatásnak a kérést.

A három rétegű struktúra legfelső rétegében lévő kontroller fogadja a hívást, majd az üzleti logikai réteg megfelelő függvényére továbbítja azt. Ebben a rétegben zajlanak az adatbázishívások. Ezt segíti az adatelérési réteg, ahol definiálva vannak az entitások és az adatbázisok kódbéli megfelelője. Miután sikerült kinyerni az adatot az adatbázisból ez a folyamat visszafele lejátszódva visszaadja a megfelelő adatot a frontend számára. Annyival még kiegészíteném, hogy a WebSocket technológia miatt nem csak egyirányú feltétlenül a kommunikáció, egy adott mikroszolgáltatás is tudja értesíteni a frontendet, ha valahol változás történt.

Részletesen nem térek ki az egyes egységek pontos működésére, erre a későbbiek folyamán fog sor kerülni.

2.2.3 Használati esetek

Az úgynevezett „Use Case diagram” egy grafikus ábrázolása annak, hogy a felhasználónak milyen interakciói lehetnek a rendszer vagy egy másik játék felé. Különböző használati eseteket mutat be akár több fajta szereplőn keresztül.



2. ábra: Használati esetek

Ahogy a képen látszódik a program több oldallal/komponenssel rendelkezik. A legegyszerűbb a bejelentkező/regisztrációs felület. Innen a felhasználó létrehozása után bejelentkezve jutunk el a menü oldalra.

A menü oldalon megjelenik az 5 különböző opció, ezek közül kettő érdekes, ugyanis a „Váró létrehozása” és a „Csatlakozás a váróhoz” az a kettő mellyel tovább tudunk menni a következő oldalra a Lobby Page re.

A barátlista megjelenik a menüben és a váróban is. Ebben el tudjuk végezni az alapvető műveleteket, mint a hozzáadás, eltávolítás, meghívás elfogadása játékra. Ha az aktuális játékos egy váróban van akkor képes meghívót küldeni oda.

A váróban pedig képesek vagyunk kilépni belőle vagy üzenetet küldeni. A váró tulajdonosa pedig el tudja indítani a játékot.

A játékban pedig a szokásos elemek jelennek meg, mint például kártyahúzás, kör abbahagyása, kártya kijátszás, életerő csökkentés.

3 Program részletes bemutatása

Ez a blokk egy hosszabb leírás lesz magáról az elkészült szoftverről, ahol bemutatom a számomra érdekesebbnek talált megoldásokat, függvényeket és ezek technológiai hátterét. A bemutatás az architektúra ábrán (2.2.2) látható iránnyal ellentétes lesz, tehát az adatbázistól haladva egészen a frontend részéig fog történni.

3.1 Backend

A .NET a Microsoft által készített keresztrendszer. A keresztrendszer a szoftverfejlesztés nagyon tág területét lefedi. Ugyanis képes szerver és kliens oldali megoldásokra, emellett a Unity [9] játékfejlesztő szoftver is a .NET keresztrendszert alkalmazza. Sok nyelvet támogat ám mindegyik közül az egyik legelterjedtebb és leghasználtabb nyelv a C#.

A programozási nyelv alapjául szolgáló C++ és Java nyelvekhez hasonlóan a C# is objektumorientált, mely azt jelenti, hogy az adott objektumok egy egységben tartják a hozzájuk köthető adatokat és az ezekhez kapcsoló műveleteket.

A 2000-es évek elején elkészült a .NET Framework 1.0-ás verziója, ám a népszerűségét csak később nyerte a 2.0-ás verzió kijövetelével. Ám ma már megszűnt a Framework fejlesztése, helyette a .NET Core fejlesztése vette át a szerepet.

Mint említettem, rengeteg szoftverfejlesztési területen használják a .NET keretrendszerét, így a szerveroldali alkalmazások fejlesztésében is nagy szerepe van. Az általunk készített szoftver is ezen alapszik. Ennek a neve ASP.NET. Ez egy nyílt forráskódú szerveroldali webalkalmazási keretrendszer. [5]

3.1.1 Mikroszolgáltatások, Docker

A Docker egy olyan szolgáltatás, amely operációs rendszer szintű virtualizációt végez el. Ez azt jelenti, hogy egy adott docker konténerben nem kell külön virtuális gépet futtatni ahhoz, hogy a szoftverünk kapjon egy saját környezetet, hanem egy jóval kisebb hardverigényű operációs rendszer kernelt futtat. Itt összegyűjti az adott szoftverhez szükséges összes konfigugrációs fájlt, könyvtárakat és mindent, ami ahhoz kell, hogy az alkalmazás futni tudjon.

A Docker egyik eszköze a Docker Compose mely számunkra is nagy segítséget nyújtott. Ez egy olyan eszköz, amely képes a több konténerből álló alkalmazások azonos idejű futtatására. A külön konténerekre vonatkozó konfigurációs beállításokat egy úgynevezett YAML fájlt segítségével lehet egységesíteni. Erre a későbbiekben példát is fogunk látni. [3]

Alapvetően az egész program konténerizált, az adatbázistól kezdve az API átjárón keresztül (frontendet leszámítva). Ehhez a Visual Studio fejlesztő környezet nagy segítséget ad, ugyanis már az adott project létrehozásakor megkérdezi, hogy szeretnénk-e Dockerben futtatni. Ezután a Docker Compose hozzáadása is hasonlóan egyszerű. Ilyen külső alkalmazás volt még a Docker Desktop mely egységesen megjeleníti az éppen működő konténereket és ezeket külön-külön lehet kezelni az alkalmazáson keresztül.

A programunkban 4 különböző konténer fut egyszerre. Egy maga a játéknak, egy a „UserIdentity”-nek elnevezett konténer, ami mindent magába foglal mindent ami nem a játék (váró, barátok, authentikáció). Van egy az adatbázisnak és végül egy az API átjárónak.

```
useridentity.api:
  container_name: UserIdentity
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - ASPNETCORE_URLS=http://0.0.0.0:80
  ports:
    - "15200:80"
  build:
    context: .
    dockerfile: Services/UserIdentity/UserIdentity.API/Dockerfile
```

3. ábra: Docker compose konténer

Egy konténernek be lehet állítani, hogy milyen porton fusson. A képen látható „15200:80” azt jelenti, hogy ami a virtuális linux környezetben, ami a 80-as porton fut azt a külső környezetben az 15200-as porton fogja futtatni. Emellett mivel minden konténerhez tartozik egy Dockerfile, a program indítása során tudnia kell melyik Dockerfilet használja, így annak az elérési útvonalára is szükség van.

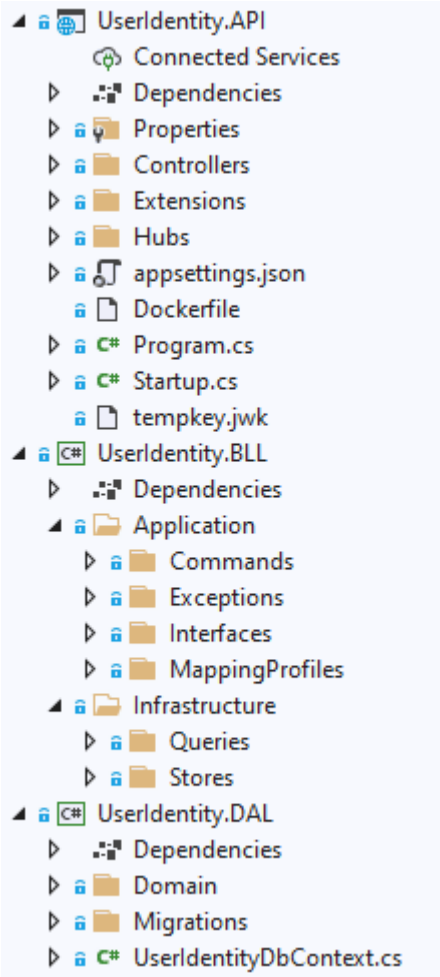
Logikusan következik, hogy bizonyos konténereknek támaszkodnia kell más konténerekre. Ugyanis az általunk létrehozott „Bang” és „UserIdentity” konténerek mindegyike adatbázist használ. Ahhoz, hogy ezt megtehessék mindkettőnek meg kell adni, hogy ők függnek az „sqlserver”-nek elnevezett konténertől. (depends_on)

Azonban nem csak saját úgynevezett „docker imageket” hozhatunk létre, hanem már előre elkészítetteket is felhasználhatunk. Az adatbázis például egy Microsoft által készített Linux Ubuntu-n futó SQL server. Itt be kell állítani saját jelszavunkat és a portot a konténer eléréséhez, hogy használni tudjuk.

3.1.2 Többrétegű architektúra, mappastruktúra

Egy általunk létrehozott mikroszolgáltatás alapvetően a többrétegű struktúrát alkalmaz, azon belül is a háromrétegű felépítés. Ennek lényege, hogy mindegyik réteg más funkciót lásson el, ezáltal a fejlesztők munkája is egyszerűbbé válik, ugyanis így az alkalmazás könnyebben továbbfejleszthető és karbantarthatóbb. A rétegek csak az alattuk lévő réteggel kommunikálnak.

A legalsó szintje az adatelérési réteg (data access layer), felette helyezkedik el az üzleti logikai réteg (business logic layer) és minden felett helyezkedne el a prezentációs réteg, azonban a felhasználó felület itt külön van teljesen szedve, így a legfelső réteg az API réteg melynek szerepe a beérkező hívásokat továbbítani az alatta elhelyezkedő üzleti logikai réteg felé.



4. ábra: Mappastruktúra

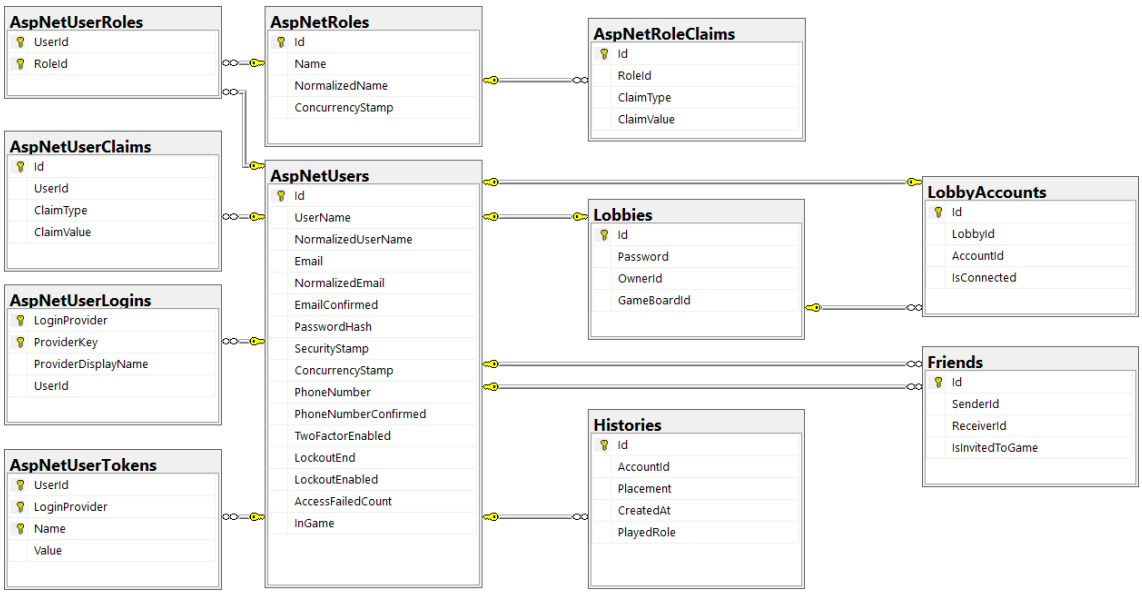
Az adatbázis elérési réteg szerepe, hogy összefogja az adatbázishoz köthető osztályokat, műveleteket. A Domain mappában szerepelnek azok az entitások, melyeket felhasználva létrehoztuk az adatbázist Code-First módszert alkalmazva. A Migrations mappa tartalmazza azokat a szükséges osztályokat melyet az Entity Framwork magának generált a migráció során, az alapján, amit a kontextben definiáltunk.

A felette elhelyezkedő rétegben, az üzleti logikai rétegben foglal helyet minden fontosabb logikai művelet, amely adatbázishívást igényel. Emellett az Application mappán belül szerepelnek azok az egyéb kisegítő osztályok és függvények melyeket felhasználunk egy adatbázis hívás során. Ilyen például az általunk definiált saját kivételek, vagy a

MappingProfiles-ban szereplő AutoMapperhez (3.1.5.4) szükséges osztályok. A CQRS (3.1.5.1) mintának köszönhetően két részre vannak osztva a hívásokra reagáló függvények. Így egyik a műveletek melyek módosítást végeznek az adatbázison a Commands mappában szerepelnek. A műveletek, amelyek csak visszaadják az adatbázis állapotát pedig a Queries mappában. A repository minta miatt létezik egy Stores mappa, ahol a tényleges adatbázis hívások történnek. Ezek interfész osztályai az Interfaces mappában szerepelnek.

Végül a legfelső réteg az API réteg. Ide érkeznek be a külső hívások, alap esetben közvetlenül a frontendtől, azonban a mi megoldásunkban a kettő közé esik egy API átjáró, amely a megfelelő mikroszolgáltatásnak küldi a hívást. A Controllers mappában helyezkednek el a kontrollerek. Ide érkeznek a hívások az átjáróból és továbbítják az üzleti logikai réteg felé. A Hubs mappában szerepelnek azok a szükséges SignalR osztályok, amikkel jelezhetünk a frontendek, ha backenden valami változás történt. Végül azért, hogy átlátható legyen a Startup osztály ezért az abban lévő konfigurációs beállításokat (függőség injektálás, globális kivételkezelés, authentikáció és authorizáció) külön szedtem osztályokra és ezek bekerültek az Extensions mappába.

3.1.3 Adatbázis



5. ábra: A menü adatbázisának felépítése

A projekthez két darab adatbázis táblát használtunk. Ezzel is hangsúlyozva azt, hogy a társammal minél inkább próbáltuk elhatárolni egymás munkáját azonban elkerülhetetlen, hogy a két tábla között kapcsolat jöjjön létre. Elsősorban a menürendszer

adatbázisáról fogok beszélni (5.ábra), ugyanis a játék adatbázisában csak a tervezés szintjén vettem részt és a képen (6. ábra) látható adatbázisfelépítés már a végleges verzió.

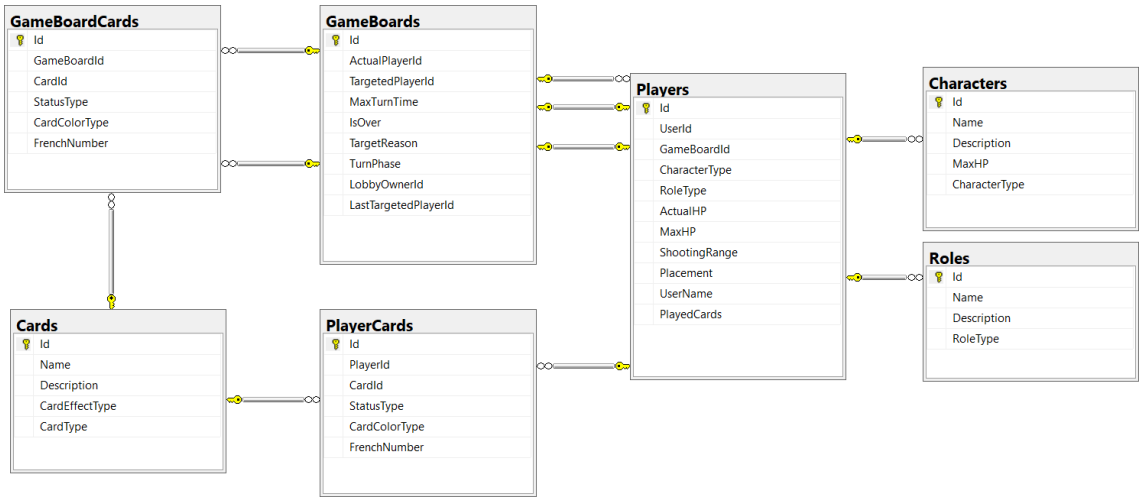
Az adatbázis tényleges létrehozása Code-First módszerrel készült, erre a későbbiekben fogok részletesebben kitérni (3.1.4.1). Röviden annyi, hogy a kódban lévő osztályokat képeztük le adatbázismodellé.

A képen látszódik a sok „AspNet”-el kezdődő tábla, ezek amiatt generálódtak, mert a saját „DbContext”-ünk nem a sima „DbContext” osztályból származik le, hanem az „IdentityDbContext”-ből. Erre a felhasználók kezelése miatt van szükség azonban erről is részletesebben ki fogok térni a későbbiekben (3.1.4.2).

A „Lobbies” tábla tárolja, hogy mi az adott váró jelszava, a felhasználót aki létrehozta a várót és hozzá tartozó „GameBoardId”-t. Ez a játék adatbázisának egy külső kulcsa, ami nulla értéket vesz fel, ha nincs elindítva a játék. Amint viszont elindul, a hozzá tartozó GameBoard kulcsát megkapja. A felhasználó és a váró közé kell egy kapcsolótábla, amely tárolja az adott váróban tartózkodó játékosokat, ennek a neve „LobbyAccounts”.

A „Friends” nevezetű táblának két külső kulcsa van a felhasználókra. A tábla lényegében úgy működik, hogy ha érkezik egy barát bejelölés akkor az bekerül egy küldő-fogadó rekordként a táblába, viszont csak akkor lesz elfogadva a barátság, hogyha ez a kapcsolat fordítva is létezik. Itt van eltárolva még a váróba való meghívás értéke is, ami igazat vesz fel, ha az adott játék meghívott minket és törlődik automatikusan, ha a váróban lévő játékosok elindítják a játékot.

A „Histories” tárolja, hogy melyik felhasználó milyen karakterrel hányadik helyezést ért el. Ennek az időpontja is el van mentve.



6. ábra: A játék adatbázisának felépítése

Mint említettem a játék adatbázisának csak a tervezés szintjén vettem részt, a társam volt, aki ezt később javítgatta, hozzárakott/elvett. Alapvetően úgy épül fel, hogy létezik a „GameBoard” ami igazából a váró táblájának játékbéli megfelelője. Itt kiegészül minden olyan eseménnyel, ami előfordulhat a játékban (IsOver, TurnPhase).

Vannak még a játékosok. Ez a felhasználó megfelelője a játékban, ahol kiegészül egyéb tulajdonságokkal, mint például az aktuális életereje, játszott karakterének típusa stb.

Létezik a „Cards” tábla, amely egy adott kártyát reprezentál. Ennek van leírása, típusa stb. Mivel a játékosoknak és a játéktér asztalán is léteznek kártyák, ezért ezt a feladatot a két darab kapcsolótábla látja el. Érdekesség lehet, a kapcsolótáblákban elhelyezett „CardColorType” és „FrenchNumber”. Minden kártyának van egy franciakártyás jelölése. Azonban az adatbázisban a kártyákat csak egyszer szerettük volna eltárolni, így ezeket a jelöléseket véletlenszerűen kapja meg a kapcsolótábla.

Mint ahogy a menü logikájában szerepel a váró táblájában egy külső kulcs a „GameBoard”-ra, úgy itt is elhelyezkedik egy külső kulcs a „Lobby” táblára. Ennek a neve LobbyOwnerId, ami tulajdonképpen az adott váró tulajdonosának a kulcsa. A társam úgy vélte, hogy számára egyszerűbb, ha ezt az információt tárolja el a saját táblájában, mert a játék működtetése során többször is felhasználja. Ezt meg is teheti ugyanis a váró tulajdonosának kulcsa is egyedivé teszi az adott várót így ezzel nem lenne gond.

3.1.4 Adatelérési réteg

3.1.4.1 Entity Framework, Code-First

Célunk, hogy a C# kódban létrehozott objektum orientált osztályokat, melyek többségében csak tulajdonságokat (property) tartalmaznak átalakítsuk adatbázis táblákká. Erre ad támogatást az Entity Framework, mely pontosan ezt a feladatot segíti elő.

A Code-First módszer arra épül, hogy az adott objektumból létrehozzuk az adatbázismodellt. Ennek a fordított művelete, ahol az adatbázistáblákból generálunk osztályokat, a Reverse Engineered Code-First.

A generáláskor azonban észrevehetjük, hogy létrejött egy olyan tábla is, amelyet nem definiáltunk C# osztályként. Az EFMigrationsHistory tábla lényegében eltárolja, hogy utoljára milyen módosításokat hoztunk létre. Ezt úgy teszi meg, hogy lementi annak a migrációs osztálynak a nevét melyet, mi generáltunk le az utolsó alkalommal amikor létrehoztuk/változtattunk az adatbázison. Az kialakított osztályok a Migrations mappában találhatóak. Ezekben az osztályokban definiálja azokat az utasításokat, amikkel fel és le tudjuk frissíteni (Up és Down függvények) az aktuális adatbázisunkat.

3.1.4.2 DBContext

```
public class UserIdentityDbContext : IdentityDbContext<Account>
{
    0 references
    public UserIdentityDbContext(DbContextOptions<UserIdentityDbContext> options) : base(options){}

    1 reference
    protected override void OnModelCreating(ModelBuilder modelBuilder)...

    8 references
    public DbSet<Friend> Friends { get; set; }
    2 references
    public DbSet<History> Histories { get; set; }
    11 references
    public DbSet<Lobby> Lobbies { get; set; }
    12 references
    public DbSet<LobbyAccount> LobbyAccounts { get; set; }
}
```

7. ábra: DbContext kinézete

A DbContext egy olyan osztály mely a későbbiekben ahhoz ad segítséget, hogy az adatbázist el tudjuk érni LinQ műveletek használatával. Tehát miután létrehoztuk az adatbázissémát, a „UserIdentityDbContext” lesz segítségünkre abban, hogy kommunikáljunk az adatbázissal. A DbSet típusú tagváltozók mindegyike egy táblát jelképez, melynek generikusan adtuk át az objektumot, amiből képezni akarjuk a táblát.

A felüldefiniált „OnModelCreating”-be kerülnek azok az információk, ahol az adatbázisra vonatkozó megkötéseinket be lehet állítani. Ha például szeretnénk, hogy az adatbázis egyes elemei kötelezően megadottak legyenek, vagy a megadott szöveg maximális hosszát szeretnénk korlátozni, akkor azt is meg lehet tenni.

Látszódik, hogy a konstruktorba paraméterként kapott értéket feljuttatjuk az ősosztályba. Erre azért van szükség, mert ahhoz, hogy elérjük az adatbázist meg kell adnunk egy úgynevezett „connection string”-et. Ezt kiszerveztünk egy másik rétegnek, az API rétegnek, ugyanis itt vannak összegyűjtve azok a fontosabb beállítások, amelyek meghatározzák az alkalmazást.

3.1.5 Üzleti logikai réteg

Az előző rétegre felett helyezkedik el az üzleti logikai réteg, amely felhasználja az alatta elhelyezkedő adatelérési réteget.

3.1.5.1 Command Query Responsibility Segregation minta

A minta lényege, hogy a kontrollertől beérkező hívások szét legyenek választva az alapján, hogy az a művelete módosít-e az adatbázison vagy sem.

A „Query”-k lesznek a GET-es HTTP kérések, itt tulajdonképpen csak az adatbázis aktuális állapotának lekérdezése zajlik valamilyen logika alapján, így ez nem módosít azon.

Ezek el vannak különítve minden egyéb HTTP kéréstől, esetünkben a POST, PUT és DELETE voltak azok, amik szóba jöttek. Ők az úgynevezett „Command”-ok amik már ténylegesen módosítás szinten nyúlnak az adatbázishoz. Ugyanis hozzáadnak, változtatnak rajta vagy éppen törlik az adott logika szerinti rekordokat.

3.1.5.2 Repository minta

A repository minta azt a célt szolgálja, hogy egy plusz réteget adjon a beérkező hívás és az adatbázis elérés közé. A programunkban alapvetően úgy határoztuk meg, hogy minden, amihez szükség van a DbContext-en keresztüli adatbázis hívásra, azok belekerülnek ebbe az alsó rétegbe. Ezeket nálunk Store-oknak hívják. Itt történnek azok az adatbázis hívások, amelyeket a középső réteg, az úgynevezett „Handler”-ek gyakrabban használnak fel.

A QueryHandler és CommandHandler-ekbe tehát ezeket a Store-okat függőség injektálás segítségével eljuttatjuk és így az a réteg nem kommunikál közvetlenül az adatbázissal, hanem minden egyéb logikát tartalmaz.

Őszintén ennek esetünkben sokszor nem volt nagy haszna, azonban a programot szerettük volna minden esetleges későbbi változtatásnak megfeleltetni. Így, ha ez komplikáltnak is hangzik és feleslegesen bonyolítottnak, legalább fel van készítve a program az esetleges későbbi módosításkora.

3.1.5.3 Mediator minta

Lényege, hogy a rétegeket méginkább külön lehessen választani egymástól. Erre szolgál a MediatR [10] könyvtár. Az API rétegben az alkalmazás kódjának egy objektumot adunk át, amely meghívja az üzleti logikai rétegben a hozzá megfelelő függvényt.

Tehát ebben a rétegben úgynevezett Handlerek hajtják végre a logikai kéréseket, melyek tudják, hogy pontosan milyen függvényeket kell lekezelni és ezeknek a függvényeknek milyen értéket kell visszaadniuk. Ugyanis a Handlerek leszármaznak az IRequestHandler generikus osztályból melynek meg van adva, hogy melyik utasításra vagy lekérdezésre (CQRS minta miatt különválasztva) milyen értéket kell visszaadnia.

3.1.5.4 Automapper

Az AutoMapper [11] könyvtárra azért van szükségünk, ugyanis az, hogy mit szeretnénk visszaadni a kliensnek, vagy mi az, amit a klienstől kapunk nem mindig egyezik meg azzal az entitással, amit az adatbázisban tárolunk. Tehát több felhasználási célre több osztályt használhatunk. A programunkban ezt külön is választottuk és ViewModelnek neveztük el azokat az osztályokat, amelyet a frontendnek küldünk vissza és DataTransferObject-nek (DTO) azokat melyeket a frontend küld a backendnek.

Tehát a feladata az Automappernek, hogy a két entitást (a ViewModel/DTO és az adatbázis entitást) valamilyen megadott módon össze lehessen egyeztetni egymással. Ezeket az úgynevezett Profile-ok konstruktorában tesszük meg. Itt a CreateMap segítségével beállíthatjuk, hogy melyik entitásra melyik entitás legyen mappelve. Ha a két entitásban található tagváltozó nevek megegyeznek akkor nincs szükség egyéb teendőre, mint csak létrehozni, a CreateMapet a két entitásra. Azonban, ha különböznek és bizonyos tagváltozókat szeretnénk összeakpcsolni és megfelelteni egymásnak akkor a

ForMember metódussal ezt megtehetjük. Itt beállítjuk a cél osztály tagváltózóját és a forrás osztály tagváltozóját.

```
2 references
public class FriendProfile : Profile
{
    0 references
    public FriendProfile()
    {
        CreateMap<Friend, FriendViewModel>()
            .ForMember(dest => dest.Id, opt => opt.MapFrom(src => src.SenderId))
            .ForMember(dest => dest.Name, opt => opt.MapFrom(src => src.Sender.UserName))
            .ForMember(dest => dest.InvitedFrom, opt => opt.MapFrom(src => src.IsInvitedToGame));
    }
}
```

8. ábra: AutoMapper Profile létrehozása

Ezután már csak a kódban kell a mappert függőség injektálás segítségével létrehozni, majd meghívni rajta a Map függvényt. Ennek átadjuk a típust, amelyre leképezni szeretnénk, ez a FriendViewModel lista, majd paraméterként az adatbázis entitását.

```
return _mapper.Map<IEnumerable<FriendViewModel>>(acceptedFriends);
```

3.1.5.5 Globális kivételleképzés

A Globális kivételleképzés lényege, hogy azokban az esetekben, ahol kivételt szeretnénk lekezelni, ezt egy specifikus kivétellel tesszük meg, melyet a későbbiekben egységesen tudunk lekezelni és ezeket átalakítani HTTP válaszokká. Ezt egy külső csomaggal a Hellang.Middleware.ProblemDetails-el tesszük meg. A kivételek HTTP válaszokká átalakítását azt az API rétegben intézzük el.

Az üzleti logikai rétegben így a saját kivételek létrehozásán túl csak annyi feladatunk van, hogy a megadott helyeken el kell dobni az kivételt.

```
throw new InvalidActionException("Lobby is full!");
```

3.1.5.6 Fontosabb logikai megoldások

Eddig többségében a felhasznált mintákról, a struktúráról és a technológiákról beszéltem. Ez a rész viszont érdekesebbnek gondolt logikai megoldásokat fog tartalmazni, hogy hogyan oldottam meg egyes részeket.

Az egyik ilyen a „LobbyStore”-ban elhelyezkedő játékos eltávolítása egy váróból. Ez azért érdekesebb, mert több dologra is figyelni kell egy eltávolítás során.

```
public async Task DeleteLobbyAccountAsync(long lobbyId, string accountId, CancellationToken cancellationToken)
{
    var actualLobby = await _dbContext.Lobbies.Where(l => l.Id == lobbyId)
        .FirstOrDefaultAsync(cancellationToken);
    var kickableAcc = await _dbContext.LobbyAccounts.Where(la => la.AccountId == accountId)
        .FirstOrDefaultAsync(cancellationToken);

    if (actualLobby == null || kickableAcc == null)
    {
        throw new EntityNotFoundException("Lobby or player not found!");
    }

    var playersOfLobby = (await GetLobbyAccountsAsync(lobbyId, cancellationToken)).ToList();

    if (playersOfLobby.Count() == 1)
    {
        await DeleteLobbyAccountsAsync(kickableAcc.LobbyId, cancellationToken);
        await DeleteLobbyAsync(kickableAcc.LobbyId, cancellationToken);
    }
    else
    {
        _dbContext.LobbyAccounts.Remove(kickableAcc);
        await _dbContext.SaveChangesAsync(cancellationToken);
        playersOfLobby.Remove(kickableAcc);

        if (accountId == actualLobby.OwnerId)
        {
            actualLobby.OwnerId = playersOfLobby.FirstOrDefault().AccountId;
            await UpdateLobbyAsync(actualLobby, cancellationToken);
        }
    }
}
```

9. ábra: Játékos eltávolítása egy váróból

A függvény megkapja a váró kulcsát és az eltávolítandó felhasználó kulcsát is. Mint említettem a Store-ok azok az osztályok, amik a Repository minta miatt plusz rétegként bekerültek, így ez a függvény egy Handlerből volt hívva. A lobby kulcsát a frontend biztosítja, azonban a játékos kulcsát külön kérdezte le a Handler egy másik Store felhasználásával (AccountStore).

Az első két függvényhívással megkapjuk, a tényleges váró és felhasználó entitást egy-egy adatbázis hívás segítségével. Ha ezek bármelyike nem található akkor kivételt dobunk, amit az API rétegben átalakítunk egy 404 Not Found HTTP hibává. Ezután lekérdezzük a váróban szereplő felhasználók listáját, ugyanis ennek az ellenőrzésével fogunk megtudni olyan információkat, amikre szükségünk van a tényleges eltávolítás előtt. Ugyanis, ha mi voltunk a váróban szereplő utolsó játékosok, akkor nem csak a felhasználót kell eltávolítani onnan, hanem magát a várót is törölni kell az adatbázisból. Ha viszont több játékos is benne van a váróban, akkor nincs szükség a váró eltávolítására. Azonban itt is előfordulhat egy olyan eshetőség, hogy a váró tulajdonosa az eltávolítandó felhasználó. Ilyenkor kell a szobának egy tulajdonos. Ezesetben a listában a legelsőt kiválasztja.

Itt példát adtam egy Storeban történő függvényről, amely adatbázis hívásokkal volt tele. A következő példa egy olyan függvény lesz, ami a Handler osztályban helyezkedik el. Itt már csak felhasználja a hozzá tartozó Store megadott függvényét.

```
public async Task<IEnumerable<FriendViewModel>> Handle(GetFriendsQuery request, CancellationToken cancellationToken)
{
    var ownId = _accountStore.GetActualAccountId();

    var domain = await _friendStore.GetFriendsAsync(ownId, cancellationToken);

    List<Friend> acceptedFriends = new List<Friend>();

    foreach (var acceptedFriend in domain)
    {
        if (domain.Contains(acceptedFriend))
        {
            acceptedFriends.Add(acceptedFriend);
            if (acceptedFriends.Contains(acceptedFriend))
            {
                acceptedFriends.RemoveAll(f =>
                    (f.ReceiverId == acceptedFriend.SenderId || f.ReceiverId == acceptedFriend.ReceiverId) &&
                    f.SenderId == ownId);
            }
        }
    }

    return _mapper.Map<IEnumerable<FriendViewModel>>(acceptedFriends);
}
```

10. ábra: Barátlista lekérdezése

Az elején az felhasználóhoz tartozó AccountStore-ból lekérdezzük az aktuális játékos kulcsát. Ezt a JWT token miatt tudja, de erre a 3.1.6.5 pontban fogok részletesebben kitérni. A kulcs alapján lekérdezhetőek a barátok. Azonban egy barátság akkor teljesül csak ha az adatbázisban a küldő és a fogadó mindkét irányban meg van adva. (vagyis, ha két rekordként benne van, hogy az A bejelölte a B-t és a B is bejelölte az A-t)

Ezután végigmegyünk a listán, amit megkaptunk és összegyűjtjük azokat a rekordokat, ahol az aktuális játékosunk a fogadó oldalon van, ugyanis az AutoMapper úgy lett beállítva, hogy a küldő felhasználóneve jelenjen meg a ViewModelben. A Barátok entitása implementálja az IEquatable osztály Equals függvényét mely arra jó, hogy innentől kezdve egy Friend objektumon is meghívható a képen is látható Contains függvény, amely visszaadja, hogy tartalmazza-e az általunk definiált egyenlőség szerinti átalakítást. Ez akkor ad vissza igaz értéket, ha a fogadó kulcsa ugyanaz, mint a küldő kulcsa és ez fordítva is igaz. Erre azért volt szükség mert így meg lehet mondani, hogy meg van-e az ide-oda lévő kapcsolat és nem csak egyirányú. Azonban ez kétszer rakná bele az értéket mert mindkét alkalommal igazat fog dobni, nekünk viszont csak arra van szükségünk, ahol a fogadó fél azok mi vagyunk. Emiatt eltávolítjuk azt, ahol mi vagyunk a küldő fél.

Majd a végén a kapott listát leképezzük a szükséges ViewModellé úgy, hogy a Mapnek megadjuk a forrás és cél entitást. Majd visszaadjuk azt.

Az utolsó megemlített logikai megoldás egy felhasználó sztátusz lekérdező függvény. Ennek keretében kitérek az üzleti logikai rétegben megvalósított felhasználókezelésre is.

```
public async Task<StatusViewModel> Handle(GetActualAccountStatusQuery request, CancellationToken cancellationToken)
{
    var status = new StatusViewModel();

    var accountId = _accountStore.GetActualAccountId();
    var actualLobby = await _lobbyStore.GetActualLobbyAsync(accountId, cancellationToken);

    if(actualLobby == null)
    {
        status.LobbyId = null;
        status.GameBoardId = null;
    }
    else if (actualLobby.GameBoardId == 0)
    {
        status.LobbyId = actualLobby.Id;
        status.GameBoardId = null;
    }
    else
    {
        status.LobbyId = actualLobby.Id;
        status.GameBoardId = actualLobby.GameBoardId;
    }

    return status;
}
```

11. ábra: Felhasználó aktuális státusza

A frontendnek szükséges tudnia, hogy az aktuális felhasználót éppen melyik felületre kell helyeznie. Ehhez készült egy státusz lekérdező függvény melynek célja visszaadni, hogy a felhasználónak van-e aktuális várója vagy folyamatban lévő játéka, ahova vissza kell tudnia csatlakozni.

Ha a lekérdezett váró üres, akkor egyértelműen nincs se várója se hozzá tartozó játékmenet, ugyanis a játék csak úgy létezhet, ha van hozzá tartozó váró. Ha várója létezik viszont a benne található „GameBoardId” értéke nulla akkor nincs játéka, tehát nullra állítjuk a játéktábla kulcsát. Minden más esetben létezik a játékmenet is, ilyenkor mindkét kulcsot visszaadjuk.

Az elején látható a nagyon sokat használt AccountStore-os függvény a GetActualAccountId. Ez a függvény a JWT Token visszafejtése alapján vissza tudja nekünk adni, hogy a bejelentkezett felhasználónak mi a kulcsa.

```
_httpContext.User.Claims
      .FirstOrDefault(x => x.Type == JwtRegisteredClaimNames.Sub)
      ?.Value;
```

3.1.6 API réteg

3.1.6.1 Startup, middlewarek

Az alkalmazás elindítását követően a Program osztályban szereplő Main metódus lesz az első, ami lefut. Ez meghívja a szintént ebben az osztályban szereplő CreateHostBuilder függvényt, amely a Startup osztályunkat felhasználva beállítja a hosztolási környezetet.

A Startup osztály az, amivel leírjuk az alkalmazás szerkezetét. Ez tartalmaz minden általunk megadott konfigurációt. Két függvény szerepel alapértelmezetten benne.

A ConfigureService amelyben a fontosabb beállításokat lehet implementálni. A programunkban funkciók szerint külön osztályokba rendeztem a beállításokat, hogy átláthatóbb legyen. Ezeket elneveztem Extensions-nek. Például a ServiceExtensionbe kerültek be azok a MediatR és Storeokhoz kapcsolódó függőség injektálások melyeket muszáj hozzáadni a programhoz. Egy másik az IdentityExtension ahol az IdentityServer4 szükséges beállításait szedtem egy osztályba.

A másik a Configure függvény melyben a middlewareket állítunk be. Az úgynevezett middleware csővezetékek feladata, hogy egy adott beérkező kérést az alkalmazás helyesen tudjon lekezelni. Egymás után helyezkednek el ezért fontos, hogy milyen sorrendben vannak a függvényben hozzáadva. Elsőnek a kivételkezeléshez szükséges „ProblemDetails” csomag függvénye van meghívva, ez lekezeli a későbbi middlewarek által esetlegesen dobott hibákat.

3.1.6.2 Kontrollerek

A kontrollerek feladata, hogy a beérkező http hívásokat a megfelelő helyre továbbítsák az üzleti logikai réteg számára, majd az ott kialakított választ visszaadják.

Egy http hívás lehet GET, POST, PUT és DELETE (ezek, amelyek előfordulnak a programban, azonban ezeken kívül van még sok). Ezek lekérdeznek, hozzáadnak, változtatnak és törölnek. A hívás útvonalát is jelölni kell a kontrollernek úgynevezett attribútum formájában, mely a függvény felett jelenik meg szögletes zárójelben. Az útvonal úgy épül fel, hogy „Hostip:Port/KontrollerNév/AnnotációSzöveg.

Ezen felül adatot is fel lehet juttatni egy kérésbe. Ennek több módja is van. Az egyik, ha magában az URL-ben küldik fel. Ilyenkor a kontrollerben csak a függvényparaméterbe bele kell rakni és magába az attribútumba is. Lehetséges csak

függvényparaméterként belerakni. Ilyenkor ezek úgynevezett lekérdezésként (query) jelennek meg az URL-ben. Például: URL?gameBoardId=2&lobbyId=1. Lehetséges még a kérés testében feljuttatni adatot, ilyenkor jelölni kell a paraméter előtt, hogy [FromBody], ugyanis ilyenkor a feljutatott JSON formátumot automatikusan átalakítja az általunk kívánt objektummá.

Mint említettem korábban a kontrollerek a MediatR használatával kommunikálnak a közvetlen alatta lévő réteggel (BLL). Így minden kontroller számára függőség injektálás (dependency injection, DI) segítségével kell eljuttatni a mediatr-t. A Send függvényének át kell a létrehozott Query (lekérdezés) vagy Command (utasítás) objektumot.

3.1.6.3 WebSocket technológia

Alapvetően az általam elkészített feladatnál 3 helyen kellett használni a SignalR technológiát, melynek feladata a frontend értesítése, ha frissülnie kell. Egyik alapvető ilyen rész a chat funkció működése miatt kell, ugyanis, ha egy váróban/játékban egy játékos üzenni szeretne akkor annak élőben kell tudnia frissülni a többi játékos számára is.

```
public async Task SendMessageToLobby(string message)
{
    var actId = _accountStore.GetActualAccountId();
    var actName = await _accountStore.GetActualAccountName();
    var lobby = await _lobbyStore.GetActualLobbyAsync(actId, new CancellationToken());

    if (lobby != null)
    {
        var messageInstance = new Message
        {
            UserName = actName,
            Text = message,
        };

        Lobbies[lobby.Password].Messages.Add(messageInstance);
        await Clients.Group(lobby.Password).SetMessage(messageInstance);
    }
}
```

12. ábra: Üzenet küldése egy váróban

A hubokban felsorolt függvények azok, amit a frontend hív meg. A SendMessageToLobby is egy ilyen függvény. Paraméterként megkapja a beadott üzenetet. Az első 3 sorban lekérdezzük a fontosabb adatokat, mint például a játékos név,

kulcs meg az aktuális váró kulcsát. Ezek után létrehozza az üzenet objektumot a megadott adatok alapján. Minden hubban el van tárolva a rá vonatkozó adatok. (Például egy chat hubban el kell tárolni a felhasználókat, az üzeneteket) Ez az adat egy Dictionaryban van eltárolva és minden bejegyzés kulcsa az adott váró jelszava. Tehát a képen látható utolsó két sorban, a váró jelszava alapján hozzáadja az üzenetet, majd az összes váróban szereplő felhasználónak beállítja az üzenetet. (SetMessage).

Ezen kívül még a várónak van egy hubja, amelyben a legfontosabb funkció, hogy a ki és becsatlakozó felhasználók alapján frissüljön a frontenden a lista. A barátok hubja is hasonló, ha egy barátot hozzáadunk, vagy eltávolítunk annak mindkét oldalon élőben kell frissülnie.

Ahhoz, hogy egy adott hub tudja, hogy melyik játékosról van szó az egész osztály tetején megkapta az [Authorize] attribútumot. Ehhez szüksége van az adott játékos úgynevezett access_token-jére melyről a 3.1.6.5 pontban fogok részletesebben beszélni. Ezt a tokent az URL query részében kapja meg. Ehhez egy külön függvényt kellett létrehozni, hogy engedélyezze az token fogadását az queryből.

Legutoljára be kell állítani a Startup mindkét függvényében, a konfigurációs beállításokat, hogy hibátlanul működjön. A Configure függvényben az adott hubra vonatkozóan elérési útvonalat is biztosítani kell a számára.

3.1.6.4 Globális kivételkezelés

Az üzleti logikai rétegnél már meg lett említve azonban itt is bemutatom, ugyanis ez a réteg szolgál arra, hogy az általunk elkészített kivételeket http hibákká tudjuk alakítani. Korábban említettem, hogy funkciók szerint szét lett bontva a Startupban konfigurált dolgok. A kivételleképzés is egy ilyen külön osztályt kapott.

A 3 különböző kivételnek be van állítva, hogy a pontos részleteit a hibának ne adja vissza, így a frontenden kezelhető üzeneteket tudunk megjeleníteni. Ezen kívül csak a http hibát (például 404) kell beállítani és a hibához tartozó üzenetet, melyet a kivételünk üzenetével teszünk egyenlővé.

Szokás szerint ezt is bele kell rakni a Startupban a middleware-ek közé. Az alapértelmezett hibakezelőt lecserélve a saját hibakezelőnket állítjuk be a legelső middlewarenek, amely felfogja a többi által dobott esetleges hibákat.

3.1.6.5 Authentikáció, token

A program alapvető követelménye, hogy a felhasználókat tudjon kezelni. Ehhez az IdentityServer4 NuGet csomagot használtam. Az Identity Server egy OpenID Connect és OAuth 2.0 keretrendszer a ASP.NET Core számára. Ezen rész ismertetése során több fogalmat is tisztázni fogok az elején, majd bemutatom, hogy a szoftverben ez hogyan lett implementálva.

A JWT (JSON Web Token) egy általánosan használt token típus. A struktúrája 3 részből áll, melyek egymástól a tokenben pontokkal vannak elválasztva. Egy header (fejléc) melynek feladata, hogy például megadja, hogy milyen algoritmust használt ahhoz, hogy létrehozza az aláírást ("alg": "HS256"). Ezen felül a token típusa is ebben a részben van pontosítva ("typ": "JWT"). A következő szintje a payload (tartalmi rész) melyben minden fontosabb információ megtalálható, például akár a bejelentkezett felhasználó neve. Végül az utolsó rész az aláírás (Signature), melynek feladata, hogy a validálja a tokent biztonságosan. A fejléc és a tartalmi rész Base64-es kódolásával, majd a két rész összekapcsolása után generálódik ez a része a tokennek.

Több fajta authentikáció folyamat létezik, a szoftverünk azonban a token alapú authentikációt használja. Ennek lényege, hogy a kliensen való bejelentkezést követően a szerveroldal generál egy tokent, amit visszaad, majd ezt a kliens esetünkben a helyi tárhelyben (Local Storage) eltárolja. Mivel nem örökéletű egy token így a frontend oldalon megadott időközönként újat kell generáltatni, hogy bejelentkezve tudjon maradni az adott felhasználó.

Az általunk használt authorizációs protokoll az OAuth 2.0, melynek lényege, hogy az adott felhasználó számára személyre szabott elérést biztosít a HTTP hívások számára. Alapvetően két tokent generál le a protokoll. Az egyik az acces tokent aminek a lényege, hogy a szerver be tudja pontosan azonosítani, az adott felhasználót, ezáltal egyedi jogosultságot adva amelyet a szerveroldal pontosan meghatároz. A másik a refresh token. Mint említettem egy token ideje meg van határozva. Erre való a refresh token, hogy a bejelentkezést ne kelljen megszakítani, helyette egy új access tokent igényel a backendtől. Az új access tokennel új refresh tokent is kapunk, ugyanis többször nem lehetne ezeket használni. A refresh tokent ugyanúgy a kliens tárolja esetünkben a local storageban.

Alapvetően az egész folyamat a regisztrációval kezdődik. Ilyenkor kerül bele az adatbázisba a felhasználónak minden szükséges adata. (generált kulcs, felhasználónév, hashelt jelszó stb.) Ezután a bejelentkezéssel tudja magát a felhasználó hitelesíteni. Ilyenkor az IdentityServer segítségével legenerálódik mindkét token. Mivel az access tokennel tudja magát hitelesíteni a felhasználó ezért azt minden http híváshoz küldenie kell a frontendnek. A kontrollerek felett meghatározott Authorize attribútum felel azért, hogy a tokenek validálva legyenek.

A Startupban elsősorban be kell állítani a Cross-origin resource sharing (CORS) konfigurációkat. Ez egy olyan beállítás, amely lehetővé teszi más erőforrások is hozzáférjenek a szolgáltatáshoz. Engedélyeztetni lehet több elérhetőséget is, akik HTTP hívást tudnak kezdeményezni a szerver felé. Emellett a JWT és az IdentityServerre vonatkozó beállításokat is fel kell konfigurálni.

3.1.6.6 Swagger

Egy ASP.NET-es automatikusan generált projekt létrehozása után belekerül kisegítő eszköznek a Swagger. Ez egy olyan felület, amelyen megtekinthetjük az általunk felállított kontrollerek végpontjait. ezeket ki is próbálhatjuk tesztelés céljából. A fejlesztési folyamat elején ezt a felületet is bekonfiguráltam, hogy az bejelentkezés is működjön. [12]

3.2 Backend és Frontend kapcsolata

Mivel a két oldal teljesen le van választva egymástól, ami azt jelenti, hogy a kliens oldal nem foglalkozik a backend számára kijelölt feladatokkal (például adatbázis tárolása) és fordítva. Ezért meg kell oldani a két oldal közti kommunikációt, ez az úgynevezett REST (Representational State Transfer) kapcsolat, mely egy szoftverarchitektúra típus internet alapú rendszerek számára. Ez megkötéseket tartalmaz. Ha egy felépítés megvalósítja ezt, akkor ő REST(ful) API használatával kommunikál.

Az adatokat HTTP-n keresztül adja tovább. Ez több formátumban is megtörténhet, például PHP, HTML vagy az esetünkben is használt JSON. Ez a kommunikációs forma a frontendtől halad a backend oldalig és nem fordítva. Azonban számunkra az is szükséges volt, erre használtuk a WebSocket technológiát.

Az esetek többségében a kliensoldali API hívások egyből beesnek a backenden definiált kontrollerekbe, azonban, mint az az architektúra ábránkon is látszódik, mi felépítettünk a kettő közé egy plusz réteget.

Ez az úgynevezett API gateway (átjáró). Feladata, hogy egy köztes csatornát hozzon létre a kliens és a szerver között, hogy azok ne közvetlen kapcsolatban álljanak egymással.

Erre a célra az Ocelot [13] nevezetű csomagot használtuk. Ennek lényege, hogy egy vagy több JSON formátumú fájlban határozom a végpontokat, melyet a külvilág számára nyilvánosságra akarok hozni. A Startup fájlban beállítottuk a konfigurációs adatokat, mind a felhasználandó JSON fájl(ok)ra mind az Ocelotra.

Egy végpont meghatározásnál több dolgot is be kell állítani. Az URL-t, amit nyilvánosan el lehet majd érni. Ennek a HTTP típusát. Ezután azt is meg kell adni, hogy ez pontosan melyik hosztra és portra legyen továbbítva ugyanis az alkalmazásunk több minkroszoláltatást használ. Azonban itt hosztként a Docker konténer definiált nevét szükséges megadni (useridentity.api). Ezután már csak a továbbítandó URL szükséges, amely a tényleges kontroller elérési útvonala.

```
{
  "DownstreamPathTemplate": "/Account/registration",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "useridentity.api",
      "Port": 80
    }
  ],
  "UpstreamPathTemplate": "/api/identity/registration",
  "UpstreamHttpMethod": [ "Post" ]
},
```

12. ábra: Példa egy végpont meghatározására

3.3 Frontend

Az Angular a Google által készített webes alkalmazás keretrendszer. Az ingyenes és nyílt forráskódú keretrendszer TypsScript alapon működik. A TypeScript egy Microsoft által fejlesztett programozási nyelv. Különlegessége az, hogy mivel JavaScript alapokon működik, ezért átkonvertálható a kódja JavaScript kóddá. Míg a TypeScript egy típusos objektum orientált nyelv, addig a JavaScript egy szkriptnyelv.

Ebben a blokkban is azt az elvet követem, hogy az architektúra szerint lentről felfele haladjak, ám ez a frontenden már nem fog olyan széles spektrumon mozogni, mint a szerver oldalon. Emiatt itt egy adott rész bemutatásánál funkció szerint is szét fogom bontani a megoldásomat és minden általam létrehozott fontosabb egységet bemutatok.

3.3.1 Mappastruktúra

A mappák struktúrája alapvetően logikusan szét vannak bontva. A szervízek bekerültek a services mappába ezen belül funkció szerint szétválasztva, ugyanis a társammal külön kellett szeparálni a feladatokat. Tehát lett egy mappa az az authorizációnak, a chatnek, a játéknak és a menünek.

A komponensek is külön kerültek egy pages mappába itt szintén hasonlóan funkció szerint szétválasztva (hasonlóan authorizáció, chat, játék, menü).

Ezen felül az interceptorok, guardok és pipeok is külön mappákba vannak szétszedve.

3.3.2 Szervízek

A szervízek fogják betölteni azt a funkciót, hogy a HTTP hívásokat végrehajtsák, ugyanis a komponenseknek nem szabad közvetlenül adatot lekérdezni a szerveroldalról, ott inkább az adatok megjelenítésére kell fókuszálni.

3.3.2.1 Authorizáció és tokenek

Ez a két szerviz külön van bontva azonban egyben fogok róluk beszélni, ugyanis valamilyen szinten összetartoznak.

Az authorizációs szervízbe az alapvető bejelentkezés és regisztrációs HTTP hívások kerültek bele. Itt a beadott DTO-ból létrehozza a számunka kívánt adatot olyan formában, amit tovább lehet adni a hívásnak. Ugyanis az adatokat a hívás testében juttatjuk fel, így egy URLSearchParams osztály-t kellett beállítani. Ezen kívül olyanok kerültek be, mint hogy lekérdezhető legyen az aktuális játékos kulcsa és státusza (a státusza, hogy jelen pillanatban melyik oldalon kellene tartózkodnia attól függően, hogy játékban van-e, váróban stb.).

A token szervízbe kerültek bele azok függvények, amik intézik a helyi tárhely (local storage) állapotának lekérdezését és változtatását. A local storageban három dolgot

tárolunk. Az access tokent, a refresh tokent és a felhasználónevet. Ezek írására és olvasására létezik ez a szervíz.

3.3.2.2 Barát és Váró szervízek

Mivel nem szeretném felsorolni minden végponthívást ezért itt inkább bemutatom, hogy hogyan épül fel egy HTTP hívás. Mint már többször is említettem, hogy a GET, POST, PUT és DELETE műveleteket használjuk a programban. A következő egy példa a GET lekérdezésre:

```
getAcceptedFriends(): Observable<Friend[]>{
  return this.client.get<Friend[]>(`${environment.baseUrl}/api/friends`);
}
```

A konstruktorba injektált HttpClienten kell a get függvényt meghívni. Meg kell adni neki, hogy egy Friend típust fogunk visszakapni. Itt nem kell külön a JSON sorosítással foglalkozni, ugyanis, ha a C# kód ViewModeljének tagváltozói megegyeznek a frontenden definiált interface-el akkor magától elintézi a sorosítást. (annyi különbséggel, hogy az első betű kicsi a frontend interfaceben, míg a C# kódban nagy) Ezután csak paraméterbe kell belerakni, a kívánt url-t. Az environment egy külön osztály, amibe kiszerveztük, az API átjárónak és a másik két mikroszolgáltatásnak az elérhetési útvonalát, hogy egységesen lehessen kezelni őket.

```
public joinLobby(password: string): Observable<Object> {
  return this.client.post(
    `${environment.baseUrl}/api/lobby/connect/${password}`,
    undefined,
    { responseType: 'text' }
  );
}
```

Azonban paraméternek nem csak az elérhetési útvonalat lehet megadni. A váróba való csatlakozás esetén a fejléc (header) és a test (body) részt is ki kell töltenünk. A kódban látható módon a body rész undefined, vagyis nem adjuk meg. A fejlécben viszont beállítjuk, hogy milyen fajta válaszra számítunk. Ugyanis belépéskor vissza fogjuk kapni, a váróhoz tartozó jelszót egy string formában és ezt külön kell beállítani.

3.3.3 Komponsek

Az alapértelmezetten generált Angular komponensek négy fájlt tartalmaznak, ezen fileok különböző feladatokat látnak el.

Tartalmaz egy HTML (HyperText Markup Language) fájlt, ami tulajdonképpen egy olyan nyelv, amit weboldalak készítéséhez használnak. Emellett tartalmaz egy CSS/SCSS (Cascading Style Sheets) fájlt, ami a HTML nyelvet egészíti ki. Ez egy stílusleíró nyelv, ami a megjelenését adja a weboldalunknak.

Végül két TypeScript fájl is helyet foglal egy komponensen belül. Ebből számunkra csak az egyik lesz érdekes. Ez fogja meghatározni az adott weboldal működését és logikáját.

3.3.3.1 Bejelentkező/Regisztrációs felület

A felületek bemutatásakor inkább a logikai részére fogok kitérni. A kinézet (HTML és CSS) a háttérbe fognak szorulni, ugyanis erre kevesebb időt szántam.

Ezen két felület egy formos oldalnak számítanak. Ugyanis a beadott bemenetek egy loginForm/registrationForm változóba kerülnek bele, amik FormGroup típusúak.

Minden felületen implementálva van az OnInit és az OnDestroy, ezek lényege, hogy az adott felület betöltésekor és megszűnésekor a függvények lefutnak és így specifikálni tudjuk, hogy milyen esemény menjen végbe ezen műveletek hatására.

A tokenek kezeléséről csak felületesen lesz szó, ugyanis ezeket bizonyos szervízek végzik. Az oldal betöltésekor a program üríti a helyi tárhelyet (local storage) egy szervízen keresztül. Erre azért van szükség, hogyha valaki visszanavigál erre az oldalra, akkor mindenféleképpen újra azonosítani kelljen magát hiába volt már korábban bejelentkezve.

A bejelentkező felületen a bejelentkezés gomb megnyomását követően lekérdezi a formból az adatokat, majd ezt az authorizációs szervíznek továbbítja. A regisztrációs felületen ugyanez a folyamat történik. A bejelentkezéskor a szerviz által visszakapott adatokat eltároljuk. Ugyanis ilyenkor a backend visszaküldi nekünk az access tokent és egy refresh tokent. Ezen felül, ami még lényeges, hogy egy experis_in nevű változót is átad amely azt tartalmazza, hogy mennyi idő alatt fog lejárni a tokenünk. Tehát a két tokent eltároljuk a hely tárhelyen.

Egy szervízben definiált hívásra a subscribe segítségével fel lehet iratkozni. Itt több eset fordulhat elő azonban a programunk csak kettő használ. Az egyik, ha sikeresen visszatért a hívás hibamentesen a másik amikor hibát dobott. Ezt a subscribeon belül egy „response” (válasz) és egy „error” (hiba) résszel választottuk el.

```
this.authService.login(loginDto).subscribe(
  response => {
    this.snackbar.open("Login is successful!");
    this.setLocalStorage(loginDto, response as LoginResponse);
    this.router.navigate(['menu']);
  },
  error => {
    this.snackbar.open("Wrong username or password!");
    this.tokenService.deleteLocalStorage();
  }
);
```

Ha a hívás sikeresen lefutott akkor az esetek többségében snackbar segítségével (3.3.6) kiírtuk a kívánt végeredményt (pl.: „Sikeres bejelentkezés”). Majd ezután a Router beépített osztály segítségével továbbirányítottuk a főmenübe. Ha nem sikerült, akkor a szerveroldal által már egyszerűsített hibaértéket jelenítettük meg az esetek többségében azonban előfordul, hogy saját hibaüzenetet íratunk ki.

3.3.3.2 Barátlista komponens

A barátok megjelenítése nem csak a háttérlogikája miatt érdekes. Ugyanis itt a megjelenítés részéhez is tartoznak újdonságok.Ez a beágyazott komponens alapvetően két helyen tud megjelenni. Az alapján, hogy hol van megjelenítve máshogy is kell viselkednie. Ugyanis, ha a menüben tartózkodik a játékos, akkor nem hívhatja meg a barátait játékra ellentétben, ha a váróban van.

Mivel más felületekbe van beágyazva, így a menü és a váró HTML fájljában kap helyet a barátlista. Megjelenítéskor úgynevezett „@Input()” segítségével tudjuk meghatározni, hogy a lista hol helyezkedik el, ugyanis a menü és a váró is küld egy bool értéket, hogy éppen ez a váró vagy sem. Azonban ugye másnak is kell megjelennie ettől függően. Ezt a HTML kódba beépíthető Angular specifikus „*ngIf” intézi el, amely tulajdonképpen az alapján jeleníti meg az adott részt, hogy milyen értéket kap.

Mint látható a képen ez egy lista, szóval alapvető követelmény, hogy a lista görgethető legyen és ne nyúljon túl a keretein. Erre is egy Angularos eszközt használunk. Az <ng-scrollbar> a nevéből is adódóan egy görgethető felületet jelent. Ezen csak annyit állítottam, hogy a görgető doboz színe menjen a felület általános stílusához. Mivel több

13. ábra: Barát lista

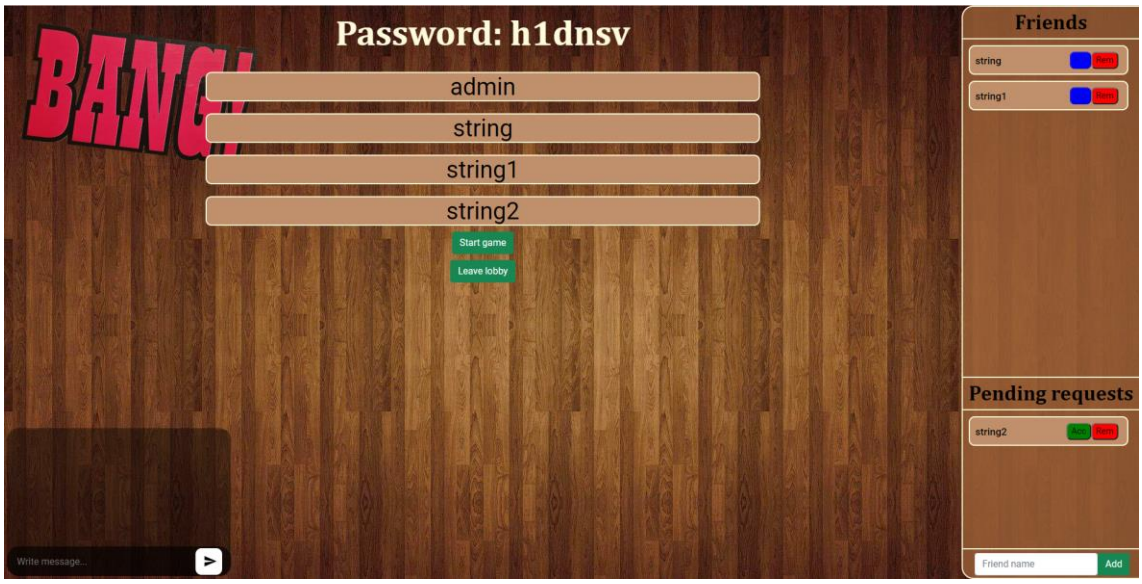
embert kell tudnia megjeleníteni így a szintén Angular specifikus eszközt használom, az „*ngFor” mely hasonló funkciót lát el, mint a C#-os foreach.

Mint látható a képen ez egy lista, szóval alapvető követelmény, hogy a lista görgethető legyen és ne nyúljon túl a keretein. Erre is egy Angularos eszközt használunk. Az <ng-scrollbar> a nevéből is adódóan egy görgethető felületet jelent. Ezen csak annyit állítottam, hogy a görgető doboz színe menjen a felület általános stílusához. Mivel több embert kell tudnia megjeleníteni így a szintén Angular specifikus eszközt használom, az „*ngFor” mely hasonló funkciót lát el, mint a C#-os foreach.

Ránézésre, még egy fontosabb probléma jön szembe. Ugyanis, ha a nevek hossza túlnyúlna azon a területen, amelyet számára kijelöltünk, akkor az belerondítana az összképbe. Erre megoldást egy CSS beállítás ad, mely ezt meggátolja azzal, hogyha túlnyúlna a szó akkor három darab pontot rak a végére.

3.3.3.3 Váró és Menü felület

A menü az a felület, ahova egyből navigál a felhasználó a bejelentkezést követően. Itt több dolgot is tud csinálni. Létre tud hozni egy új saját várót, ilyenkor egyből bekerül abba a váróba és az alábbi felület fogja fogadni, csak egy felhasználóval a listában. Ezen felül a képen látható jelszó használatával be tud a menüből csatlakozni másik várókba is. Természetesen, ha nem a megfelelő jelszót használja akkor hibaüzenettel a menüben marad. Meg tudja még tekinteni a korábbi játékainak eredményét. Ezen felül ki tud jelentkezni és törölni is tudja a felhasználóját.



14. ábra: Váró felülete

A váró felületén láthatjuk a legtöbb dolgot. Ugyanis itt nem csak maga a váró, de a chat komponens és a barátlista is megjelenik. Létrehozásakor a backend generál egy egyedi jelszót a várónak. Ezt meg is jelenítjük rajta. Alatta láthatjuk a felhasználók listáját.

Végül legalul két gomb formájában helyet kap az indítás és az elhagyás. A kettő közül az indítás az érdekesebb, ugyanis erre külön szabályok vonatkoznak. Maga a gomb csak akkor jelenik meg számodra, ha a várót a te tulajdonodban áll. Minden más esetben ez a gomb a többi felhasználónak nem létezik. Ha a váró tulajdonosa elhagyja a várót akkor ez a jog átszáll másra. Ezen felül a gomb megnyomhatatlan mindaddig amíg nem jön be a váróba elegendő játékos. Ugyanis a játék specifikációja alapján minimum 4 és maximum 7 ember tudja játszani. Ez esetben elég a minimumra szűrni, ugyanis a váró alapból se enged be több embert mint 7. Azt is szűri, hogyha egyszer már rányomtunk akkor ne engedélyezze, hogy mégegyszer megtegyük, ne hívja ugyanazt a függvény két alkalommal is.

3.3.4 Pipe

A Pipe osztályok feladata, hogy szövegeket, dátumokat vagy akármilyen adatot átalakítsunk az általunk kívánt formára. Az alkalmazásunkban például azt a feladatot látja el, hogy az általunk kívánt Enum értéket olvasható szöveges formában adja vissza. Ugyanis, ha egy enumot próbálunk megjeleníteni a HTML-ből akkor egy számot fogunk látni.

```
private static lookup = ["Outlaw", "Renegade", "Sheriff", "Vice"];
transform(value: RoleType): string {
  return RoleTypePipe.lookup[value];
}
```

Mint látható, string formában visszaadja az általunk kívánt értéket. Használata egyszerű, a HTML-ben ahol meg szeretnék jeleníteni az enumot amögé beírjuk egy vonal mögé a pipe nevét (| roleType).

3.3.5 Guard

A guardok feladata, hogy a „őrködjenek” afelett, hogy az adott útvonal melyet a felhasználó éppen megnyit arra jogosult e vagy sem. Azonban számunkra ez olyan feladatot lát el, hogy az adott felhasználó mindig a helyes oldalon legyen. Ugyanis, ha a játékos már korábban létrehozott egy várót, de úgy dönt, hogy visszanavigál a bejelentkező felületre, akkor bejelentkezés után alaphelyzetben a menüben lenne. Ez azért okozna problémát mert létre tudna hozni egy új várót, amit alap esetben nem lenne szabad.

Ehhez hasonló problémákra nyújt megoldást számunkra a guard. Megadjuk minden szükséges komponensnek, hogy vegye figyelembe az általunk létrehozott AuthGuard-ot. A Guard implementálja a szükséges CanActivate osztályt. majd annak a canActive függvényét felüldefiniáljuk. Itt tuljadonképen lekérdezzük, hogy az aktuális felhasználónak melyik oldalon is kéne tartózkodnia.

Elsőként megnézzük, hogy érvényes-e még az általa használt token. Ha nem akkor a bejelentkező felületre navigáljuk. Ezután egy végpont hívással megtudjuk, hogy hol kéne lennie. Ez a végpont visszaadja, a várónak és a játéknak a kulcsát. Ha ezek üresek (null értéket vesznek fel) akkor a játékost a menübe navigálja. Ezen logikán haladva, rakja a váróba és a játékba is a játékost. Ez meggátolja, például azt is, hogyha egy váróból vissza gombbal vagy URL átírással szeretne a felhasználó helyet változtatni, ugyanis automatikusa visszakerül a megfelelő helyére.

3.3.6 WebSocket

A frontenden használt WebSocket technológiát a SignalR-t leginkább a beépített chat komponens bemutatása segítésével fogom tudni a legjobban átadni. Ezt a társammal fejlesztettük le úgy, hogy én implementáltam a háttérlogika részét ő pedig a kinézetét fejlesztette.

Ahhoz, hogy létrejöhessen egy kapcsolat a szerveroldal és a kliens között elsőként fel kell építeni ezt a kapcsolatot. Ehhez a signalR.HubConnectiont-t kell definiálni. A szerver oldalon már beállított útvonalat használva query értékként át kell adni az access token-t, ezáltal tudni fogja a szerver, hogy ki használja.

A kapcsolat felállítása után azokat a függvényeket, amelyeket a backenden interfaceben adtun meg, meg kell felelteni egy frontendes függvénnyé.

```
this.connection?.on("SetMessage", message => this.setMessage(message));
```

Az első paraméter a pontos neve annak a függvénynek, ami backend oldalon definiálva lett. Majd ehhez társítunk második paraméterként egy frontend oldalon definiált függvényt, jelen esetben egy üzenet csatolását.

Azonban nem csak arra van lehetőség, hogy a backend oldal visszahívjon a frontendre, ez fordítva is működik, erre szolgál az connection invoke függvénye. A chat komponensnél például, az indításkor be kell csatlakozni a szobába. Paraméterezése hasonló a korábban bemutatott fügvényéhez. Itt is meg kell adni stringbe a függvény nevét, majd egyéb paraméterekben, ha a kiválasztott metódus tartalmaz paramétert.

A komponens megszűnésekor ezekről le is kell iratkozni, majd a végén teljesen megszüntetni a kapcsolatot.

3.3.7 Snackbar

A snackbar egy úgynevezett angular material. Ez tulajdonképpen egy testreszabható alert box. A számára létrehozott serviceben egy függvényt definiálunk. Ennek a függvények paraméterként továbbítjuk a megjeleníteni kívánt üzenetet. Majd a konstruktorban beinjektált MatSnackBar osztály open függvényét felhasználva továbbítjuk az üzenetet neki, beállítjuk, hogy mennyi ideig legyen megnyitva az üzenet és végül állítunk neki egy általunk definiált CSS osztályt.

```
open(message: string) {
  this.snackBar.open(message, 'Close', {
    duration: 3000,
    panelClass: ['snackbar']
  });
}
```

3.3.8 Interceptor

Az alkalmazás felhasználókezelése miatt szükség van arra, hogy egy végpont meghívásakor az adott felhasználó beazonosítsa magát. Erre szolgál a továbbított token.

Azonban ezt a tokent minden híváshoz beállítani fejlécként feleslegesen macerás művelet lenne. Erre szolgál megoldásként az interceptor.

```
intercept(request: HttpRequest<unknown>, next: HttpHandler)
  : Observable<HttpEvent<unknown>>
{
  const authToken = this.token.getAccessToken();
  const authReq = request.clone({
    setHeaders: { Authorization: 'Bearer ' + authToken }
  });

  return next.handle(authReq);
}
```

Ennek feladata, hogy minden általunk kiküldött híváshoz csatolja a fejlécbe a kívánt Bearer tokent.

4 Összegzés, értékelés

4.1 Mit tanultam

Azon túl, hogy a rengeteg új technológiát melyet felhasználtunk a program készítése során megtanultam használni és megértettem a háttérműködésüket, ezen felül még az egyéb programok használatában is nagyobb rutint szereztem melyek szükségesek voltak a fejlesztés során.

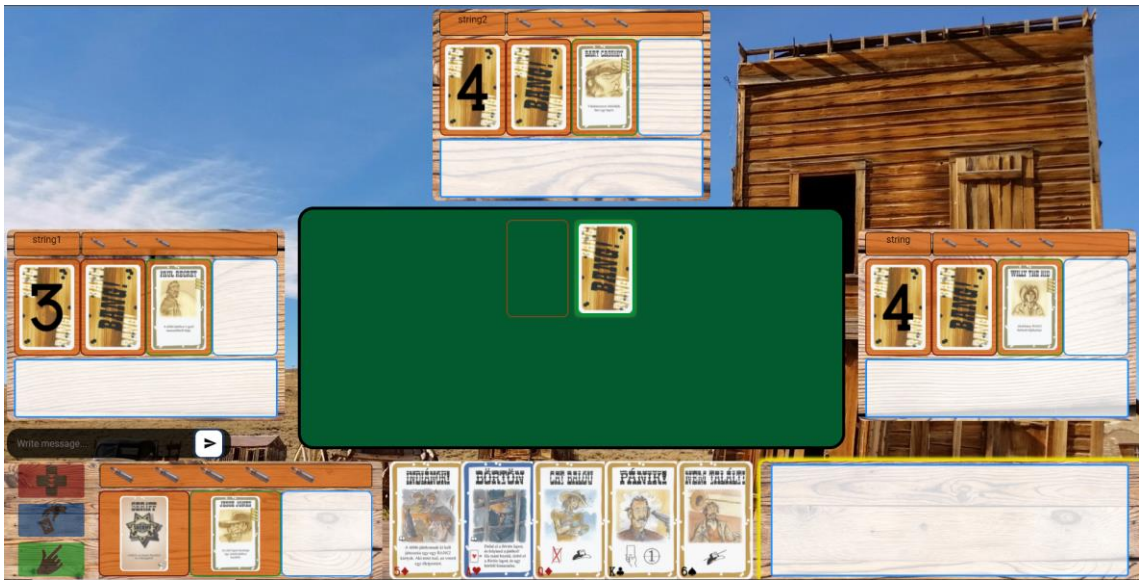
Ilyenek például a Postman [14]. Ez egy alkalmazás mellyel HTTP hívásokat küldhetünk kiszolgálók felé. Alap szinten használtam már, azonban itt sokkal több funkcióját megismertem. A Github működését is részletesebben megértettem, ugyanis azt felhasználva dolgoztunk társammal egy közös repositoryban. Segítségemre volt még a Docker Desktop nevezetű program, amely a Docker konténerek egységes kezelését teszi lehetővé. A témához nem szorosan kapcsolódva, de a Photoshop használta is nagy segítségemre volt a képek elkészítése során, ugyanis azokat körülvágva átlátszó háttérrel kellett elkészíteni.

Azonban nem csak szakmai szemmel fejlődtem, de a közös munka révén abba is belátást nyertem, hogy milyen egy nagyobb szintű projekten közösen dolgozni.

4.2 Értékelés

Összességében elégedett vagyok az elvégzett munka mennyiségével és minőségével. A feladatkiírásban szereplő funkciókat bőven sikerült túlteljesíteni is, bár ez közel se jelenti azt, hogy a program hibátlan és tökéletes lenne.

Ugyan a játék implementálása nem az én érdemem, azonban mivel tervezés és a képek elkészítése szintjén részt vettem benne így gondoltam bemutatom a végső állapotát. Az alsó felület a saját táblánk, a többi a maradék játékosé. Mivel kör eleje van ezért a középen található kártya zölden világít. Mindig az kap zöld keretet, ami egy lehetséges interakció.



16. ábra: Végső játék kinézete

Tesztek és mérések nem készültek a programhoz, így azokat nem tudom bemutatni, azonban továbbfejlesztési lehetőségként felsoroltam.

A technológiák többségét eddig csak felületesen vagy egyáltalán nem ismertem, ugyanis az Angularral előtte minimálisan se foglalkoztam, így azt nulláról kellett megtanulnom használni. A backend oldali technológiák egy részével már foglalkoztam és használtam őket korábban is, azonban voltak, amiket csak felületesen. Így a szakdolgozat elkészítése nagy mértékben hozzájárult a szakmai fejlődésünkhöz.

A csapatmunka is gördülékenyen zajlott. Mint ahogy azt korábban említettem megpróbáltuk minél jobban elszeparálni a feladatokat így a fejlesztés alatt elég volt a minimális szintű kommunikáció is. Az elején a tervezés részénél kellett többet egyeztetni és megbeszélni. A fejlesztés alatt leginkább az elkészítendő kártyák képei miatt volt interakció, azonban előfordult, hogy egymástól kértünk segítséget valami probléma megoldása során, vagy csak egyeztettük, hogy a strukturális felépítés ne legyen túlságos különböző a két mikroszolgáltatásnál.

Mivel a fejlesztés vége felé kapcsoltuk össze a játékot a menü részével, ezért itt is szükséges volt a kommunikáció, hogy ki mit intézzen és melyikünk milyen végpontokra, milyen formában képzelte el az összekapcsolást.

4.3 Továbbfejlesztési lehetőségek

Ahogy említettem, nagy fába vágtuk a fejszénket, de szerintem nagyrészt megvalósítottuk, amit előre elterveztünk. Azonban a program közel se tökéletes, rengeteg

finomítani való lenne benne, vagy esetleg plusz funkciókkal tudna bővülni. A következő részben azt fogom felsorolni, hogy szerintem mi az, amitől használhatóbb lenne a játék és esetleg mivel lehetne továbbfejleszteni.

Angularral sajnos a szakdolgozat alatt foglalkoztam először, így egy tökéletesen struktúrált és szép kinézetű alkalmazáshoz nincs elég ismeretem azonban ami tőlem telt azt beleraktam. A HTML és a CSS nyelvek területén is lenne hova fejlődnöm. Tehát maga a program kinézete, kezelőfelülete lehetne egységesebb és szebb. Photoshopban készítettem is hozzá látványterveket, amiket rétegekre is bontottam, azonban erre sajnos nem maradt elegendő idő, hogy megfelelően belekerüljenek. Inkább a funkcionális működést részesítettem előnybe.

A program egy online játszható játék és az elején azt is terveztük, hogy majd kitelepítjük az alkalmazást Azure-ba vagy valami egyéb külső helyre, hogy ki lehessen próbálni élesben több játékos által magát a játékot, ám ez sajnos elmaradt szintén az idő szűke miatt. Tehát továbbfejlesztésnek mindenféleképpen beleraknám, hogy fusson valami külső szerveren, hogy használható legyen mások által.

A programban megvalósult az eredményjelző is mind frontend mind backend részről. Ez megmutatja, az utolsó tíz játékod játszott karakterét és a helyezést, amit azzal értél el. Ám nem lett végül összekötve a társam oldalával, így a végén nem adja hozzá a listába az adott játékosoknak az elemet, pedig mindkét oldalról megvalósult az ehhez szükséges környezet.

A váró rendszerét lehetne esetleg még bonyolítani olyan elemekkel, hogy játékos kirúgása (backend oldalon elkészült csak figyelmetlenség miatt kimaradt frontend oldalon), nyílt és zárt várórendszer, melynek lényege, hogy az ismerősök nyílt váróba becsatlakozhatnak meghívás nélkül, azonban a zártba csak meghívóval. A váróban és a játékban olyan beszélgetés implementálás, hogy kiválasztasz egy játékost és lehetőséged van csak vele kommunikálni.

Lehetne akár End-to-End teszteket, vagy Unit teszteket írni a játékhoz, hogy backend és frontend oldalról is kiszűrhetőek legyenek az apróbb hibák.

Authentikáció részről még akár bele kerülhetne egy kétlépcsős azonosítás is pluszba, és a mostani állapotán is lehetne finomítani.

Mint látható alapvetően rengeteg ötlettel álltunk neki a feladatnak, így tudtuk, hogy nem fogunk minden elképzelt opciót belerakni, ám kiválasztottuk a

legfontosabbakat és azok kerültek bele. Szóval ezek a fontosabb változtatások és finomítások, amiket beleraknék, ha lenne rá elegendő idő, de meg vagyok elégedve azzal amit elértünk, a feladatkiírást bőven sikerült teljesíteni és még pluszba is kerültek bele funkciók.

Irodalomjegyzék

[1] Wikipedia, „List of most expensive video games to develop” [Online]. Available: https://en.wikipedia.org/wiki/List_of_most_expensive_video_games_to_develop

[2] Wikipedia „Adobe Flash” [Online]. Available: https://hu.wikipedia.org/wiki/Adobe_Flash

[3] Wikipedia „Docker (software)” [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software)

[4] Wikipédia „SignalR” [Online]. Available: https://hu.wikipedia.org/wiki/SignalR

[5] Wikipédia „.NET keretrendszer” [Online]. Available: https://hu.wikipedia.org/wiki/.NET_keretrendszer

[6] Wikipédia „JSON Web Token” [Online]. Available: https://en.wikipedia.org/wiki/JSON_Web_Token

[7] Wikipédia „MongoDB” [Online]. Available: https://en.wikipedia.org/wiki/MongoDB

[8] ElasticSearch [Online]. Available: https://www.elastic.co/

[9] Unity [Online]. Available: https://unity.com/

[10] MediatR [Online]. Available: https://github.com/jbogard/MediatR

[11] AutoMapper [Online]. Available: https://automapper.org/

[12] Swagger [Online]. Available: https://swagger.io/

[13] Ocelot [Online]. Available: https://ocelot.readthedocs.io/en/latest/introduction/gettingstarted.html

[14] Postman [Online]. Available: https://www.postman.com/product/what-is-postman/