

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

[Összefoglaló 1](#_Toc89562791)

[Abstract 2](#_Toc89562792)

[1 Bevezetés 3](#_Toc89562793)

[1.1 Általános bevezető 3](#_Toc89562794)

[1.2 Internetes játékok, társasjátékok 3](#_Toc89562795)

[1.3 Közös munka 4](#_Toc89562796)

[1.4 Saját feladatrészek 4](#_Toc89562797)

[1.5 A dolgozat szerkezete 5](#_Toc89562798)

[2 Tervezés, architektúra 6](#_Toc89562799)

[2.1 Specifikáció 6](#_Toc89562800)

[2.1.1 A menü rendszer 6](#_Toc89562801)

[2.1.2 A játék rendszer 7](#_Toc89562802)

[2.2 A rendszer architektúrájának felépítése 8](#_Toc89562803)

[2.2.1 Adatbázis 8](#_Toc89562804)

[2.2.2 Architektúra 8](#_Toc89562805)

[2.2.3 Használati esetek 9](#_Toc89562806)

[3 Technológiák 11](#_Toc89562807)

[3.1 .NET, ASP.NET 11](#_Toc89562808)

[3.2 Docker, docker compose 12](#_Toc89562809)

[3.3 IdentityServer4 14](#_Toc89562810)

[3.4 SignalR 14](#_Toc89562811)

[3.5 SQL Server 15](#_Toc89562812)

[4 Program részletes bemutatása 18](#_Toc89562813)

[4.1 Backend 18](#_Toc89562814)

[4.1.1 Mikroszolgáltatások 18](#_Toc89562815)

[4.1.2 Többrétegű architektúra, mappastruktúra 19](#_Toc89562816)

[4.1.3 Adatbázis 20](#_Toc89562817)

[4.1.4 Adatelérési réteg 22](#_Toc89562818)

[4.1.5 Üzleti logikai réteg 23](#_Toc89562819)

[4.1.6 Prezentációs réteg 28](#_Toc89562820)

[4.1.7 API átjáró 28](#_Toc89562821)

[4.2 Frontend 28](#_Toc89562822)

[4.2.1 Komponsek 29](#_Toc89562823)

[4.2.2 Szervízek 29](#_Toc89562824)

[4.2.3 Pipe 29](#_Toc89562825)

[4.2.4 Guard 29](#_Toc89562826)

[4.2.5 Interceptor 29](#_Toc89562827)

[4.2.6 Snackbar 29](#_Toc89562828)

[4.2.7 WebSocket 29](#_Toc89562829)

[5 Összegzés, értékelés 30](#_Toc89562830)

[5.1 Értékelés 30](#_Toc89562831)

[5.2 Továbbfejlesztési lehetőségek 30](#_Toc89562832)

[Irodalomjegyzék 32](#_Toc89562833)

[Függelék 33](#_Toc89562834)

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. 04.

...…………………………………………….

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 erre fel voltunk 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.

# Bevezetés

## 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.

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 megkerülhetetlen egy átlagember számára is. Egy komolyabb játék elkészítése felérhet akár egy mozifilm költségeivel, például a 2020-ban készül „Cyberpunk 2077” nevű játék fejlesztési és marketing költsége meghaladta a 300 millió dollárt is [1]. Azonban nem csak a konzolos/számítógépes játékoknak van hatalmas népszerűsége, hanem az online böngészőn keresztül játszható játékoknak is.

## 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 társasjátékok. 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.

## 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 csináljuk.

Azonban a korábbi munkánkkal ellentétben, ahol sok mindent készítettünk közösen együtt, 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, ahol az alapvető felépítést, specifikációt és kinézeti elemeket beszéltük meg még közösen kellett dolgoznunk.

Kérdéses volt, hogy hogyan szeretnénk szétosztani a feladatköröket. Az első ötlet az volt, hogy szétosztjuk backend-frontend vonalon azonban ezt elvetettük. Végül azt a megoldást választottuk, hogy ugyan mindketten fogunk backend és frontend feladatokat is készíteni, de inkább funkció szerint osztjuk el 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 (3.1.1) alatt részletezem, hogy mi volt a konkrét feladat, úgyhogy itt csak felületesen térek ki rá felsorolás szerűen.

## Saját feladatrészek

Az elején közös munka részeként foglalkoztam a program alapvető tervezésén, ideértve az adatbázis kezdetleges megtervezését, a használati esetek elkészítését és a felülethez használt skiccek felrajzolását és 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, mikroszolgáltatásoknak. A bejelentkezés és regisztráció implementálása volt a legfontosabb ugyanis ettől függött utána minden. Ezután jöhettek az egyéb feladataim, mint például a barát rendszer, a váró rendszer és egyéb ezekhez kapcsolódó feladatok. (pl. felhasználóhoz kötött eredmények mutatása, chat rendszer kialakítása)

## A dolgozat szerkezete

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

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

A harmadik részt egy szárazabb technológiai bemutatónak gondoltam, ahol felsorolom, hogy milyen technológiákat használtam a program elkészítése során és ezek mindegyikéről részletesebben írok.

Majd ezt követi az önálló munka bemutatása. Ez egy hosszabb blokk lesz, ahol bemutatom a program szerkezetét a backendtől a frontendig, tehát a felépítése ennek a résznek nem funkciók szerinti szétosztással fog elkészülni, hanem az architektúrában lentről felfele haladva. Kitérek minden számomra érdekesebbnek vagy bonyolultabbnak talált megoldásokat.

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.

# Tervezés, architektúra

## Specifikáció

A program mivel szétosztható két részre ezért külön szedem őket, azonban a második részéről nem fogok nagyon pontos specifikációt adni, ugyanis abban csak a tervezés szintjén vettem részt, nem én implementáltam. Ugyanis 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.)

### 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 bemenetek 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, 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. A jobb oldalon pedig egy barátlista található.

A barátlista két felületen jelenik meg, a menüben és a lobbyban. Annyi a különbség a kettő között, hogyha a felhasználó egy varóban van akkor tud küldeni meghívást egy barátjának, hogy ő is csatlakozzon be. 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. 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 egy 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 birtokosa, 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. A felületen 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.

### A játék rendszer

A játék elindítását követően megjelenik minden játékosnak maga a játéktér. A játéktér több fő elemet is tartalmaz. A saját táblánkat, a középen elhelyezkedő asztalt, ahonnan a lapokat kell húzni és visszarakni, és a többi játékos táblája.

A játék egy körökre osztott kártyajáték. Minden játékos véletlenszerűen megkap egy szerepet, 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. A renegát mindenkin kívül áll, neki csak annyi a feladata, hogy egyedül maradjon.

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” ugyanis ezekkel tudjuk a többi játékos életét csökkenteni, vagy épp a saját életünket megmenteni.

Egy körben annyi kártyát játszhatsz ki amennyit szeretnél, csak arra kell figyelni, hogy maximum annyi kártya maradhat a kezedben amennyi az életerőd. Ha esetleg nem sikerült ennyi kártyát kirakni, akkor a kör vége előtt kénytelen vagy őket eldobni.

## A rendszer architektúrájának felépítése

### Adatbázis

Ebben a részben nem a projekt felépített adatbázisáról fogok írni, ugyanis arról a 4.1.3 pontban fogok beszélni, 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 vagy a Elasticsearch.

A NoSQL adatbázisok alapvető különbsége az SQL-esekkel szemben, hogy nem táblákban tárolják az adatot és a különbség van az adatreprezentációban és sémában is. A NoSQL adatbázisok sokkal flexibilisebben alakíthatóak, így egy olyan alkalmazásnál amelyik folyamatosan fejlődik, gyakori a változás, célszerűbb ezt a módszer alkalmazni. Ugyanis nem kell minden alkalommal migrálni, amely meglehetősen nehéz feladat lenne. A többségük nem támogatja az ACID (atomiság, konzisztencia, izoláció, tartósság) működést (a MongoDB támogatja) az SQL adatbázisokkal szemben.

A játékunkban nem volt szükségünk olyan adatbázisra, amely a későbbiekben sok átlakításon esik át. Az ACID működés is alapvető követelmény volt, amely az általunk választott SQL Serverben érvényesülnek.

### 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 kommunikál egymással.

A képen szöveg, computer, számítógép, sötét látható

Automatikusan generált leírás

1. ábra: Architektúra felépítése

A képen látható, hogy a frontend API (application programming interface) hívásokon keresztül ráhív 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. Ezután az adatbázis szinttől 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.

### Használati esetek

Az úgynevezett „use case diagram” egy grafikus ábrázolása annak, hogy a felhasználónak milyen interakciói lehetnek a rendszerrel. Különböző használati eseteket mutat be akár több fajta szereplőn keresztül.

A képen szöveg, vezérlőelem látható

Automatikusan generált leírás

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.

# Technológiák

## .NET, ASP.NET

A Microsoft által készített .NET keretrendszer (a .NET Framework) gyors alkalmazásfejlesztést (RAD), platformfüggetlenséget és hálózati átlátszóságot támogató szoftverfejlesztői platform. A keretrendszert a korábbi platform, a COM leváltására szánták.

Eredetileg a .NET kifejezés nemcsak fejlesztői környezetet jelentett, hanem fejlesztőeszközök, szoftverek, sőt hardvereszközök összességét is. Az évek során a kép kitisztult, így mostanra a .NET alatt a keretrendszert értjük.

A .NET Framework eszköztára a szoftverfejlesztés szinte minden aspektusát (kliens-, illetve szerveroldali megoldások, adatbázisok kezelése, játékfejlesztés stb.) lefedi.

A kilencvenes évek közepén a Sun kiadta a Java platform első verzióját. Az új környezet hamarosan igen nagy népszerűségre tett szert platformfüggetlensége, hatalmas, jól kezelhető osztálykönyvtára, illetve könnyű megtanulhatósága révén. A Microsoft és más cégek eszközei eleinte nem tudták hatékonyan felvenni a versenyt a Sun termékével, később azonban a fejlesztések beértek. A Microsoft először a Java saját implementációjával próbálkozott, de ezzel soha nem aratott különösebb sikert.

Néhány későbbi .NET technológia eredetileg ezen fejlesztések során jött létre. Amikor 1998-ban a Microsoft elhatározta, hogy nem használja tovább a Sun Microsystems Java technológiáját, a már létező Microsoft J++ (Java++) terméket a .NET projekt alapjaként használták. A .NET CLR kódja egyesek szerint a Colusa Software OmniVM termékéből származik, amit a Microsoft 1996. március 12-én vásárolt meg.

A tényleges fejlesztés a kilencvenes évek végén Next Generation Windows Services munkacím alatt Bill Gates személyes felügyelete mellett indult el. 2000-ben a Microsoft, az Intel és a Hewlett-Packard már a C# nyelv szabványosításán dolgozott, ezt az ECMA 2001-ben el is fogadta.

2000 végére elkészült az első béta változat, 2002. január 5-én pedig a .NET Framework 1.0 verziója végre a nagyközönség kezei közé került.

A következő mérföldkövet 2005. november 7. jelentette, ekkor adták ki a 2.0-s verzió végleges változatát, amely hatalmas változásokat hozott. Tulajdonképpen ez a verzió hozta meg az igazi népszerűséget a .NET számára.

Egy nap híján egy évvel később a 3.0 is napvilágot látott, rá egy évre pedig a 3.5 verzió született meg. 2008 augusztusában a Microsoft kiadta a 3.5 verzióhoz tartozó első szervizcsomagot. 2010 év elején jelent meg a 4.0 verzió, 2012-ben pedig a 4.5 verzió béta változata.

Az ASP.NET egy nyílt forráskódú,[2] szerveroldali webalkalmazási keretrendszer, amelyet webfejlesztésre terveztek dinamikus weboldalak előállítására. A Microsoft fejlesztette ki, hogy a programozók dinamikus weboldalak, alkalmazások és szolgáltatások létrehozását tegyék lehetővé.

Először 2002 januárjában jelent meg a .NET keretrendszer 1.0-s verziójával, és a Microsoft Active Server Pages (ASP) technológiájának utódja. Az ASP.NET a Common Language Runtime-ra (CLR) épül, így a programozók bármely támogatott .NET-nyelven írhatnak ASP.NET-kódot. Az ASP.NET SOAP bővítési keretrendszer lehetővé teszi az ASP.NET komponensek számára a SOAP üzenetek feldolgozását.

Az ASP.NET utódja az ASP.NET Core. Ez az ASP.NET moduláris webes keretrendszerként történő újraimplementálása, más keretrendszerekkel, például az Entity Framework-rel együtt. Az új keretrendszer az új nyílt forráskódú .NET Compiler Platformot (kódneve "Roslyn") használja, és platformokon átívelő. Az ASP.NET MVC, az ASP.NET Web API és az ASP.NET Web Pages (a kizárólag Razor oldalakat használó platform) egy egységes MVC 6-ban egyesült[3].

## Docker, docker compose

A Docker képes egy alkalmazást és függőségeit egy virtuális konténerbe csomagolni, amely bármilyen Linux, Windows vagy macOS számítógépen futtatható. Ez lehetővé teszi, hogy az alkalmazás különböző helyeken fusson, például helyben, nyilvános vagy privát felhőben. Linuxon futtatva a Docker a Linux kernel erőforrás-elszigetelési funkcióit (például cgroups és kernel névterek) és egy egyesítésre képes fájlrendszert használ, hogy a konténerek egyetlen Linux-példányon belül fussanak, elkerülve a virtuális gépek indításának és karbantartásának többletterheit. A macOS-en a Docker egy Linux virtuális gépet használ a konténerek futtatására.

Mivel a Docker konténerek könnyűek, egyetlen szerver vagy virtuális gép egyszerre több konténert is futtathat. Egy 2018-as elemzés szerint a Docker tipikus felhasználási esete nyolc konténer futtatását jelenti állomásonként, és az elemzett szervezetek negyede 18 vagy annál több konténert futtat állomásonként.

A Linux kernel névterek támogatása többnyire elszigeteli az alkalmazásnak az operációs környezetre vonatkozó nézetét, beleértve a folyamatfákat, a hálózatot, a felhasználói azonosítókat és a csatlakoztatott fájlrendszereket, míg a kernel cgroupjai a memória és a CPU erőforrás-korlátozását biztosítják. A 0.9-es verzió óta a Docker saját komponenst (az úgynevezett "libcontainer") tartalmaz a Linux kernel által közvetlenül biztosított virtualizációs eszközök használatához, amellett, hogy a libvirt, az LXC és a systemd-nspawn révén absztrahált virtualizációs interfészeket is használ.

A Docker egy magas szintű API-t valósít meg, hogy könnyű konténereket biztosítson, amelyek folyamatokat futtatnak elszigetelten. A Docker-konténerek szabványos folyamatok, így lehetséges a kernel funkcióit használni a végrehajtásuk megfigyelésére beleértve például az olyan eszközök használatát, mint a strace a rendszerhívások megfigyelésére és beavatkozására.

A Docker Compose egy eszköz több konténerből álló Docker-alkalmazások definiálására és futtatására.[29] YAML-fájlok segítségével konfigurálja az alkalmazás szolgáltatásait, és egyetlen paranccsal végzi el az összes konténer létrehozását és indítási folyamatát. A docker-compose CLI segédprogram lehetővé teszi a felhasználók számára, hogy egyszerre több konténeren futtassanak parancsokat, például képeket készítsenek, konténereket skálázzanak, leállított konténereket futtassanak, és így tovább.[30] A képmanipulációval kapcsolatos parancsok vagy a felhasználó interaktív opciói nem relevánsak a Docker Compose-ban, mivel azok egy konténerhez szólnak.[31] A docker-compose.yml fájl az alkalmazás szolgáltatásainak meghatározására szolgál, és különböző konfigurációs opciókat tartalmaz. Például a build opció olyan konfigurációs opciókat határoz meg, mint a Dockerfile elérési útja, a command opció lehetővé teszi az alapértelmezett Docker-parancsok felülírását, és így tovább. 2013. december 21-én jelent meg a Docker Compose első nyilvános béta verziója (0.0.1 verzió). 2014. október 16-án vált elérhetővé az első gyártásra kész verzió (1.0).[32] A Docker Compose első nyilvános béta verziója (0.0.1 verzió) 2014. október 16-án jelent meg.

## IdentityServer4

Az IdentityServer egy ingyenes, nyílt forráskódú OpenID Connect és OAuth 2.0 keretrendszer az ASP.NET Core számára. A Dominick Baier és Brock Allen által alapított és karbantartott IdentityServer4 magában foglalja az összes olyan protokoll implementációt és bővíthetőségi pontot, amelyek szükségesek a token-alapú hitelesítés, az egyszeri bejelentkezés és az API-hozzáférés-szabályozás integrálásához az alkalmazásokba. Az IdentityServer4 az OpenID Foundation által hivatalosan tanúsított, így specifikációkonform és interoperábilis. A .NET Foundation része, és az ő magatartási kódexük szerint működik. Az Apache 2 (az OSI által jóváhagyott licenc) alatt áll.

## SignalR

A SignalR egy függvénykönvytár ASP.NET fejlesztőknek, mellyel valós idejű kommunikációs funkcionalitás építhető az alkalmazásba. A valós idejű kommunikációs réteggel lehetőség nyílik szerver oldali tartalom célzott vagy üzenetszórásos továbbítására. A függvénykönyvtár a környezetnek megfelelő leghatékonyabb kommunikációs módot választja.[3][4]

A SignalR igazi ereje a WebSocketben rejlik, amivel TCP felett kétirányú kommunikációs csatorna építhető ki. Amennyiben megfelelő a továbbító réteg a függvénykönyvtár automatikusan WebSocket-en keresztül kommunikál, egyéb esetben alternatív kommunikációs rétegen keresztül továbbít üzeneteket.

A SignalR magas szintű interfésszel ú.n. API-val rendelkezik. Szerveroldalon felülbírálhatóak és kialakíthatóak a környezetnek megfelelő események, kapcsolat menedzsment. Az interfész rendelkezik üzenetszórásos képességgel, akár csoportkezeléses megoldást alkalmazva. Kliens oldalon a SignalR rendelkezik .NET-es és JavaScript alapú függvénykönyvtárral is. Így akár JavaScript oldalról hívható natív C# kód.

Nagyon fontos tény, hogy a jelenlegi álláspont szerint nincs semmiféle garancia az üzenetek célba érését tekintve.

## SQL Server

A Microsoft SQL Server a Microsoft által kifejlesztett relációs adatbázis-kezelő rendszer. Adatbázis-kiszolgálóként olyan szoftvertermék, amelynek elsődleges funkciója az adatok tárolása és lekérdezése más szoftveralkalmazások által kért adatoknak megfelelően - amelyek futhatnak ugyanazon a számítógépen vagy egy másik számítógépen egy hálózaton keresztül (beleértve az internetet is). A Microsoft a Microsoft SQL Server legalább egy tucat különböző kiadását forgalmazza, amelyek különböző célközönségnek szólnak, és a kis egygépes alkalmazásoktól a sok egyidejű felhasználóval rendelkező, nagyméretű internetes alkalmazásokig terjednek.

Az adattárolás egy adatbázis, amely tipizált oszlopokkal rendelkező táblák gyűjteménye. Az SQL Server különböző adattípusokat támogat, többek között olyan primitív típusokat, mint az Integer, Float, Decimal, Char (beleértve a karakterláncokat), Varchar (változó hosszúságú karakterláncok), bináris (strukturálatlan adathalmazok számára), Text (szöveges adatok számára). A lebegőszámok egész számokká kerekítése az argumentumoktól függően a szimmetrikus aritmetikai kerekítést vagy a szimmetrikus lekerekítést (fix) használja: SELECT Round(2.5, 0) 3 értéket ad.

A Microsoft SQL Server lehetővé teszi a felhasználó által definiált összetett típusok (UDT-k) definiálását és használatát is. A kiszolgálói statisztikákat virtuális táblázatok és nézetek formájában (úgynevezett dinamikus kezelési nézetek vagy DMV-k) is elérhetővé teszi. A táblákon kívül egy adatbázis más objektumokat is tartalmazhat, beleértve a nézeteket, tárolt eljárásokat, indexeket és korlátokat, valamint a tranzakciós naplót. Egy SQL Server adatbázis legfeljebb 231 objektumot tartalmazhat, és több operációs rendszer szintű fájlra terjedhet ki, a maximális fájlméret 260 bájt (1 exabájt) lehet.[9] Az adatbázisban lévő adatokat .mdf kiterjesztésű elsődleges adatfájlokban tárolják. A .ndf kiterjesztéssel azonosított másodlagos adatfájlok arra szolgálnak, hogy egyetlen adatbázis adatai több fájlban, és opcionálisan több fájlrendszerben is eloszolhassanak. A naplófájlokat .ldf kiterjesztéssel azonosítják[9].

Az adatbázishoz rendelt tárhelyet sorszámozott, 8 KB méretű oldalakra osztják. Az oldal az SQL Server műveleteinek alapvető I/O egysége. Egy oldalt egy 96 bájtos fejléccel jelölnek, amely az oldalra vonatkozó metaadatokat tárolja, beleértve az oldal számát, az oldal típusát, az oldalon lévő szabad helyet és az azt birtokló objektum azonosítóját. A lap típusa határozza meg a lapon található adatokat. Ezek az adatok a következők: az adatbázisban tárolt adatok, egy index, egy allokációs térkép, amely információkat tartalmaz arról, hogy az oldalakat hogyan osztják ki a táblákhoz és indexekhez; és egy változás-térkép, amely információkat tartalmaz a legutóbbi mentés vagy naplózás óta más oldalakon végrehajtott változásokról, vagy nagy adattípusokat, például képet vagy szöveget tartalmaz. Míg az oldal az I/O művelet alapegysége, a tárhelyet valójában egy 8 oldalból álló kiterjedésként kezelik. Egy adatbázis-objektum vagy egy kiterjedés mind a 8 oldalát felöleli ("egységes kiterjedés"), vagy legfeljebb 7 további objektummal osztozik egy kiterjedésen ("vegyes kiterjedés"). Egy adatbázis-tábla sora nem terjedhet ki egynél több oldalra, ezért 8 KB méretre korlátozódik. Ha azonban az adatok meghaladják a 8 KB-ot, és a sor varchar vagy varbinary adatokat tartalmaz, akkor az ezekben az oszlopokban lévő adatok egy új oldalra (vagy esetleg oldalak sorozatára, úgynevezett allokációs egységre) kerülnek át, és az adatokra mutató mutatóval helyettesíthetők[23].

Egy táblázat fizikai tárolásához a sorait partíciók sorozatára osztják (1-től n-ig számozva). A partíció mérete a felhasználó által meghatározott; alapértelmezés szerint minden sor egyetlen partícióban van. Egy táblát több partícióra osztanak fel, hogy az adatbázist egy számítógépes fürtre osszák szét. Az egyes partíciók sorai B-fa vagy halom szerkezetben tárolódnak. Ha a táblához tartozik egy klaszterezett index, amely lehetővé teszi a sorok gyors visszakeresését, akkor a sorok az indexértékek szerinti sorrendben kerülnek tárolásra, és az indexet egy B-fa biztosítja. Az adatok a levelek levélcsomópontjában vannak, a többi csomópont pedig a levéladatok indexértékeit tárolja, amelyek az adott csomópontokból elérhetőek. Ha az index nem fürtözött, a sorok nem az indexkulcsok szerint vannak rendezve. Az indexelt nézetnek ugyanaz a tárolási struktúrája, mint az indexelt táblának. A fürtözött index nélküli táblázatot egy rendezetlen halomszerkezetben tárolja. A táblázat azonban rendelkezhet nem klaszterezett indexekkel, hogy lehetővé tegye a sorok gyors visszakeresését. Bizonyos helyzetekben a halomszerkezet teljesítménybeli előnyökkel jár a fürtözött szerkezettel szemben. Mind a halmok, mind a B-fák több kiosztási egységet is felölelhetnek[24].

# Program részletes bemutatása

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

## Backend

### Mikroszolgáltatások

A Docker és docker compose általánosabb technikai jellegű leírása a 3.2 -es pontban található. Ebben a részben inkább arról lesz szó, hogy a programban hogyan használtuk fel, mit miért csináltunk.

Alapvetően az egész program konténerizált, az adatbázistól kezdve az API átjárón keresztül (frontendet leszámítva). A konténerizált alkalmazások előnye, hogy külső rendszereken is könnyen futtatható és ehhez nem kell minden a szoftver által igényelt függőséget kitelepíteni, hanem a konténer megoldja. Olyan, mint egy virtuális gép, azonban sokkal kisebb az erőforrásigénye.

A képen szöveg látható

Automatikusan generált leírás

3. ábra: Docker compose konténer

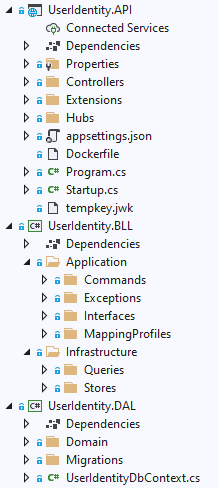
A programunkban 4 különböző konténer fut egyszerre. Egy maga a játék, egy a „UserIdentity”-nek elnevezett konténer, ami mindent magába foglal ami nem a játék (váró, barátok, authentikáció). Van egy az adatbázisnak és egy az API átjárónak.

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 buildelése során tudnia kell melyik Dockerfilet használja. Ezért annak az elérési útvonalára is szükség van.

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

### 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.

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.

4. ábra: Mappastruktúra

A felette elhelyezkedő rétegben, az üzleti logikai rétegben foglal helyet minden fontosabb logikai művelet, amely adatbázishívást igényel. Emelett 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 (4.1.5.4) szükséges osztályok. A CQRS (4.1.5.1) mintának köszönhetően két részre vannak osztva a hívások, így egyik a Commands mappában szerepel, másik pedig a Queries mappában. A repository minta miatt létezik egy Stores mappa, ahol a tényleges adatbázis hívások történnek. Ezek interface osztályai az Interfaces mappában szerepelnek.

Végül a legfelső réteg a megjelenítési 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. Ezek továbbítják a hívást 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.

### 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 (1.ábra), ugyanyis a játék adatbázisában csak a tervezés szintjén vettem részt és a képen (2. ábra) látható adatbázisfelépítés már a végleges verzió. Ebben a blokkban nem fogok kitérni részletesen az adatbázis kódbéli működtetésére, inkább az alapvető tervezői döntésekről ejtek szót.

Az adatbázis tényleges létrehozása Code-First módszerrel készült, erre a későbbiekben fogok részletesebben kitérni (4.1.4.2). 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 (4.1.4.3).

A „Lobbies” tábla tárolja, hogy mi az adott varó 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 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.

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

A képen szöveg, égbolt, képernyőkép látható

Automatikusan generált leírás

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. A játékosoknak is vannak kártyájuk, és az asztalon is van kártya, ezért ezt a feladatot két darab kapcsolótábla látja el.

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.

### Adatelérési réteg

#### Entity Framework, Code-First

Célunk, hogy a C# kódban létrehozott objektum orientált osztályokat, melyek többségében csak tagváltozót (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 a létrehozott 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 lényege, hogy 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. Ezek az 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.

#### DBContext

A képen szöveg látható

Automatikusan generált leírás

7. ábra: DbContext kinézete

A DbContext egy olyan osztály mely a későbbiekben ahhoz nyújt segtséget, hogy az adatbázist el tudjuk érni LinQ műveletek használatával. Tehát miután létrehoztuk az adatbázissémát, a UserIdentityDbContext lesz segítségünkre abban, hogy kommunikáljunk az adatbázissal. A tagváltozók mindegyike egy táblát jelképez.

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, melyet kiszerveztünk egy másik rétegnek, az API rétegnek.

### Üzleti logikai réteg

#### Command Query Responsibility Segregation minta

A minta lényege, hogy a kontrollertől beérkező hívások szét legyenek választva az alapján, hogy az egy lekérdezés vagy sem. A „query”-k lesznek a GET http kérések. Ezek el vannak különítve minden egyéb http kéréstől, esetünkben a POST, PUT és DELETE voltak azok amik szóba jöttek.

#### 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 gyakrabban felhasznál.

A Query és CommandHandler-ekbe tehát ezeket a Store-okat függőség injektálás segítségével eljutottatjuk és így az a réteg nem kommunikál közvetlenül az adatbázissal, hanem minden egyéb logikát tartalmaz. 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, hogy bonyolultabb logikát kapjon.

#### Mediator minta

Lényege, hogy a rétegeket méginkább külön lehessen választani egymástól. Erre szolgál a MediatR 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.

#### Automapper

Az Automapper 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. A programunkban ezt külön is választottuk és ViewModelnek neveztük el azokat az osztályokat, amelyet a frontendnek küldünk vissza és DataTransferObject-nek 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, azonban, ha nem egyeznek és azt szeretnénk megfelelteni akkor a ForMember metódussal ezt megtehetjük.

Ezután már csak a kódban kell a mappert injektálni majd meghívni rajta a Map függvényt és átadni a forrás és a cél entitást.

A képen szöveg látható

Automatikusan generált leírás

8. ábra: AutoMapper Profile létrehozása

#### 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. Az exceptionok http válaszokká átalakítását azt az API rétegben intézzük el.

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

#### 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 LobbyStoreban elhelyezkedő játékos eltávolítása egy váróból. Ez azért érdekesebb, mert több dologra is figyelni kell egy eltávolítás során.

A képen szöveg látható

Automatikusan generált leírás

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 felhasználával (AccountStore).

Az első két függvényhívással megkapjuk, a tényleges váró és felhasználó entitást egy-egy adatbázis hívás segítségével. Ha ezek bármelyik 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 el kell. 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.

A képen szöveg látható

Automatikusan generált leírás

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 4.1.6.3 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 Equals függvényét mely arra jó, hogy innentől kezdve egy Friend objektumon is meghívható a képen is látható Contains. Ez akkor ad vissza igaz értéket, ha a fogadó a küldő és fordítva. 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 átmappeljük a szükséges ViewModellé úgy, hogy a Mapnek megadjuk a forrás és cél entitást. Majd visszaadjuk azt.

### API réteg

#### 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.

#### Kontrollerek

#### WebSocket technológia

Csak kép a kódról a startupban és beszélni róla.

#### Globális kivételkezelés

Elmondani mire jó, majd megmutatni pár sajátot.

#### Authentikáció, token

Az authentikáció flowjáról, tokenekről, konfigurációjáról.

#### Swagger

### API átjáró

Beszélni az ocelotról.

## Frontend

Beszélni az angularról, miért jó, miért ezt választottam hasonló.

### Komponsek

### Szervízek

### Pipe

### Guard

### Interceptor

### Snackbar

### WebSocket

# Összegzés, értékelés

## Értékelés

Összességében elégedett vagyok az elvégzett munka mennyiségével és minőségével. 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 részét ismertem már és használtam korábban is azonban voltak, amiket csak felületesen ismertem. Így a szakdolgozat elkészítése nagy mértékben hozzájárult a szakmai fejlődésemhez.

A csapatmunka is gördülékenyen zajlott. Mint ahogy azt korábban említettem megpróbáltuk minél jobban elszeparálni a feladatokat így a fejlesztés alatt elég volt minimális szintű kommunikáció. Az elején a tervezést részénél kellett többet egyeztetni, megbeszélni. A fejlesztés alatt leginkább az elkészítendő kártyák képei miatt volt interakció. A fejlesztés vége felé kapcsoltuk össze a játék és a menü részét, ezért itt is szükséges volt a kommunikáció.

## 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 eléggé hiányos és részleges tudásom van, de a HTML és a CSS részén is lenne hova fejlődnöm. Tehát maga a program kinézete, kezelőfelülete lehetne egységesebb és szebb. Photoshopban készültek is hozzá látványtervek, hogy mi az, amit szeretnék megvalósítani ám erre sajnos nem maradt elég idő, így a végső változatba se kerültek bele.

A program egy online játszható játéknak készült és az elején azt is terveztük, hogy majd kitelepítjük Azure-ba vagy valami 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. Ilyen kis apróság még a váróból való játékos kirúgás. Ez elkészült backend oldalon azonban a frontenden sajnos figyelmetlenség miatt ez kimaradt a funkciók közül.

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

Összességében ezek a fontosabb változtatások és finomítások amiket beleraknék, 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>

Függelék