ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK	Hallgató neve:	**Vojtkó Márton**
2021	Hallgató törzskönyvi száma:	**T/006437/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Vojtkó Márton**
Törzskönyvi száma: T/006437/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Személyre szabható többjátékos kártyajáték készítése webes platformra
Creating a personalizable multiplayer web cardgame

Intézményi konzulens: Czinder Vendel Bence
Külső konzulens:

Beadási határidő: 2022. május 15.

A záróvizsga tárgyai: Számítógép architektúrák
Szoftvertervezés és –fejlesztés specializáció

A feladat

Tervezzen és valósítson meg egy olyan webes kártyajátékot, amely alkalmas valós idejű online többjátékos mérkőzések lebonyolítására, a felhasználói adatok és eredmények tárolására, illetve statisztikák készítésére. Tekintse át a hasonló fejlesztéseket a játék témáját és mechanikáját illetően, folytasson kutatást a szóba jöhető technológiák, módszerek és szoftverkomponensek területén.
A rendszer tartalmazzon adatbázis, szerver és kliens komponenseket, legyen átlagos böngészőből futtatható és rendelkezzen átlátható menüszerkezettel. A megvalósítás során tegye lehetővé a kártyák testreszabhatóságát és fordítson kiemelt figyelmet a játékmenet gördülékenységére.
A szoftver tartalmazzon gépi játékost is, amely a saját intelligenciája alapján képes az emberi játékosok elleni küzdelemre.

A dolgozatnak tartalmaznia kell:

- a megoldandó feladat megfogalmazását,
- irodalomkutatást a hasonló rendszerek és felhasználható módszerek, algoritmusok témakörében,
- a rendszer tervét a szoftverkomponensek kifejtésével együtt,
- a megvalósítás menetének dokumentációját,
- tesztelési tervet és tesztelést,
- értékelést és konklúziót,
- valamint az elkészült rendszer továbbfejlesztési lehetőségeit.

Ph.

ÓBUDAI EGYETEM * Neumann János Informatikai Kar 1. Budapest

.....
Dr. Vámossy Zoltán
intézetigazgató

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

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens

HALLGATÓI NYILATKOZAT

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

Budapest, 2022.05.10.

.....
hallgató aláírása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Vojtkó Márton..... [redacted] nappali.....

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Személyre szabható többjátékos kártyajáték készítése webes platformra.....

Szakdolgozat / Diplomamunka[2] címe angolul:

Creating a personalizable multiplayer web cardgame.....

Intézményi konzulens: Külső konzulens:

Czinder Vendel Bence..... .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2021.09.15.	Téma egyeztetése, irodalomkutatás megtervezése	
2.	2021.10.13.	Irodalomkutatás áttekintése	
3.	2021.11.10.	Rendszerterv egyeztetése, fő elemek megbeszélése	
4.	2021.12.01.	Rendszerterv áttekintése	

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

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

Budapest, 2021.12.09.

.....
intézményi konzulens

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

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Vojtkó Márton..... [redacted] ..... nappali.....

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka[1] címe magyarul:

Személyre szabható többjátékos kártyajáték készítése webes platformra.....

Szakdolgozat / Diplomamunka[2] címe angolul:

Creating a personalizable multiplayer web cardgame.....

Intézményi konzulens: Külső konzulens:

Czinder Vendel Bence..... .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.02.16.	Rendszerterv egyeztetése	
2.	2022.03.16.	Megvalósítás áttekintése, prototípus konzultáció	
3.	2022.04.20.	Megvalósítás áttekintése	
4.	2022.05.04.	Konklúzió és eredmények egyeztetése, továbbfejlesztési lehetőségek megbeszélése	

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

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

Budapest, 2022.05.09.

.....
intézményi konzulens

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

TARTALOM

BEVEZETÉS

Az ember létezése óta szereti jól érezni magát, ez vitathatatlan. Ahhoz, hogy ez teljesüljön, manapság számtalan lehetősége van mindenkinek: különböző könyvek, filmek, sportok, szociális tevékenységek, játékok, és még számtalan tennivaló közül választhat az ember, ha szeretné élvezni a szabadidejét. Vizsgáljuk meg a játékokat egy kicsit közelebbről.

Azzal sem mondok nagyon nagy újdonságot, hogy játékok nagyon régóta léteznek az emberiség történelmében; rengeteg, különböző fajta példány áll rendelkezésre, legyen az valamilyen táblás társasjáték, kreatív szerepjáték, vagy egy adott társaság barátságára és tudására építő kérdéskör játék. Amit fontos megjegyezni, hogy ezekben a játékokban mind van egy közös pont: mindegyikhez több ember kell, legalább kettő. A videójátékok megjelenésével azonban ez megváltozott.

Ugyanis a videójátékokkal együtt megjelentek az egy embert igénylő játékok (továbbiakban egyszemélyes játékok). Ezeknek egy fontos célja szokott lenni, hogy, akárcsak egy könyv vagy egy film, egy történetet meséljen el, azzal a különbséggel, hogy a játékos maga irányítja a főhőst, esetleg fontos döntéseket hoz meg a nevében, ezzel egy karakterré válva a történetben, illetve azt viselkedésével alakítva.

Érdekesség, hogy manapság egyre több egyszemélyes játékban megtalálhatóak az úgynevezett minijátékok, melyek általában valamilyen egyszerű felépítésű, valós, fizikai játékokon alapulnak, mint például a kockapóker. Így volt ez a CD Projekt Red (továbbiakban CDPR) által készített Witcher 3: Wild Hunt (továbbiakban csak Witcher) nevű videójátékkal is, melyben először jelent meg a Gwent nevű minijáték.

Gwent

A Gwent a Witcher világában működő, egy viszonylag egyszerű szabályrendszerű kártyajáték. A játékos ahogy halad előre a történetben, úgy gyűjthet össze egyre jobb és ritkább kártyákat. Ezeket a kártyákat játékbeli ellenfelek legyőzésével, illetve különböző kereskedőktől való vásárlással tudja megszerezni; tehát tisztán épít arra, hogy az egyszemélyes játék történetével párhuzamosan lehessen folyamatosan játszani.

A minijáték hatalmas sikert ért el a játékosok körében, főként egyszerű, könnyen megérthető szabályrendszerének köszönhetően. Ennek következtében fejlesztette le és adta ki a CDPR különálló játékként a Gwent: The Witcher Card Game nevű többjátékos játékot.

Gwent: The Witcher Card Game

A különálló többjátékos játék csak kezdetén hasonlított az alapjául szolgáló Gwentre. Az idő előrehaladtával egy, az elődjénél jóval komplexebb és kompetitívebb játék fejlődött ki, mely azóta is folyamatos fejlesztés alatt áll. Ebből származnak előnyök és hátrányok.

Az előnyök közé tartozik, hogy, a rengeteg kártyának, és összetett szabályrendszernek köszönhetően, egy viszonylag sokáig, változatosan játszható játék alakult ki, mely rendelkezik megfelelő rangrendszerrel, ezzel is támogatva a verseny szerű mérkőzéseket.

Hátránya, ami ebből kialakult, hogy elvesztette egyszerűségét, mely tulajdonsága miatt nagyon sokan szerették ezt a játékot; illetve ami még fontosabb az én szempontomból, hogy ezt a kártyajátékot, ha előveszi az ember egy baráti társaságban, akkor nem lehet azt 5-10 perc alatt bemutatni, majd játszani ismerőseinkkel, mivel a játék arra épül, hogy a játékosok 10-20, vagy akár több száz óra alatt gyűjtik össze a kártyáikat.

Ezek ismeretében mutatom be a projektem tervét, a megvalósítandó programot.

A feladat: Gwent Custom / Gwent Friends

Projektem céljaként egy olyan játék-keretrendszer fejlesztését tűztem ki, mely a fentebb említett, már létező két implementáció alapjait ötvözi, illetve azt kiegészíti.

Az eredeti Wicher Gwentből megtartanám a rá jellemző egyszerű szabályrendszert, illetve a viszonylag kevés kártyatípust, míg a később megjelent többjátékos Gwentből pedig magát a többjátékos jellegét hoznám át; ezzel kapva egy olyan játékot, melyet lehet baráti társaságokban használni, minimális betanulási idővel.

Ahhoz, hogy ez ebben a formában működni tudjon, szükséges az, hogy a játékosok tudjanak gyakorlatilag helyben kártyákat gyártani, épp amiatt, mert nincs előzetes X órájuk arra, hogy ezeket gyűjtögessék.

Ezért egészíteném ki ezt a megoldást a saját ötletemmel: ahelyett, hogy a játékosok gyűjtik a kártyáikat gyakorlatilag egy randomgenerátor által, saját maguk tudják őket létrehozni egy megadott, fair szabályrendszer alapján.

Ez egyrészt azért előnyös, mert így mindenki személyre tudja szabni a kártyáit, mind játékmechanikai szempontból (milyen erős, milyen képességei vannak, stb.), mind perszonális szempontból (ki/mi van a kártyán, mi a neve, mi a leírása); ezzel jócskán megnövelve a játékélményt.

Másrészt ezzel elérhető az, hogy ténylegesen rövid idő alatt lehessen mind megismerni a játékot, majd egy adag saját kártyát legyártani, szintén nem sok időt pazarolva rá, a tényleges játék megkezdése előtt.

1. TECHNOLÓGIAI ÁTTEKINTÉS

A projekt technológiai felépítését tekintve két fő komponenst tudunk megkülönböztetni: egy kliensoldalt (frontend), illetve egy szerveroldalt (backend).

1.1. Kliensoldal

A kliens oldali komponensnek két főbb feladatköre van: a játékfelület megjelenítése, illetve a játékostól érkező bemenetek (kattintás, írás, stb.) kezelése.

Fontos követelmény bármilyen játéknál, hogy a megjelenítés és a vezérlés megfelelő sebességgel történjen, kényelmesen lehessen kezelni a játékot. Ennek egyik fontos mérőszáma a képkocka/másodperc (Frames Per Second, FPS). Játékonként és emberenként változó, hogy mennyi az a legkevesebb FPS, amennyivel a játékos kellemesnek érzi a játékot, nem érzi azt, hogy diavetítést néz; illetve az általa megadott inputok akkor fejtik ki hatásukat a játékban, amikor azt ő szeretné, elvárná. A manapság elterjedtebb FPS értékek: 30, 60, 120, 144, azonban a dolgozat írásakor már megjelentek 240, 360 Hz-es monitorok, ami azt mutatja, hogy ezek az elvárt értékek egyre jobban tolódnak felfelé.

Fontos megjegyezni azonban, hogy jelent projekt célja egy kártyajáték implementálása, amelyben nem sok interakció történik másodpercenként, ellentétben például egy lövöldözős játékkal. Ennek függvényében kijelenthető, hogy a 60 FPS (amely jól illeszkedik a széleskörben elterjedt 60 Hz-s monitorokhoz) elérése bőven elég esetünkben, illetve rosszabb körülmények közt elfogadható a 30 FPS érték is.

A sebességet befolyásoló tényező továbbá a kliensoldalon jelen levő hardver erőssége. Mivel ez gyakorlatilag ismeretlen, és minden játékosnál más, ezért itt azt a megkötést tudjuk tenni, hogy a kliensoldali szoftvernek jól kell skálázódnia, a felhasználó által biztosított hardvert maximálisan ki kell tudnia használni.

További kritérium a kliensoldali komponenssel szemben, hogy képes legyen az úgynevezett platformfüggetlen működésre, azaz futtató környezettől függetlenül képesek legyenek a játékosok együtt játszani, legyen a hardver akár egy Windows-t futtató PC, vagy egy Apple Mac.

1.1.1. HTML, CSS, JavaScript

A HTML, CSS, JavaScript hármas napjaink egyik legalapvetőbb és legmeghatározóbb technológiája webes kliens alkalmazások területén. A HTML leíró nyelvvel olyan dokumentumokat lehet létrehozni, amelyeket a böngészők tudnak megjeleníteni. Ezek a dokumentumok tartalmazzák az egyes weboldalak alapvető tartalmát, felépítését. A HTML által leírt oldalak formázását segíti a CSS, melynek segítségével érhető el az, hogy a megjelenített weboldal ne csak egyszerű formázást tartalmazzon (alap betűtípus, egyszerű felsorolások, táblázatok, stb.), hanem az emberi szemnek is kellemesebb, szép megjelenítés jöjjön létre. A harmadik elem pedig a JavaScript nyelv. Ennek segítségével

lehet a kliensoldali, böngészőben futó programokat írni, melyek főként a felhasználói élmény további növelésére szolgálnak (pl. már lekérdezett adatbázisban találatok szűkítése, vagy keresésnél automata kiegészítés).

Mára már ez a hármas hatalmassá nőtte ki magát, ennek következtében nagyon sokszor nem is szokás önmagában HTML, CSS illetve Javascript nyelven létrehozni a webdokumentumokat, hanem az ezekből létrejött, különböző keretrendszereket használni, melyek rengeteg előre elkészített stílust, funkciót tartalmaznak. Ilyen keretrendszer pl. a rendkívül elterjedt Bootstrap, vagy a Foundation, melyek a CSS használatát könnyítik meg számtalan előre elkészített sablonnal. JavaScript oldalról legismertebb keretrendszerek pl. az Angular, a React, vagy a Vue.js.

1.1.2. Blazor

A Blazor a Microsoft által fejlesztett webkliensoldali keretrendszer, mely egy új megközelítést alkalmaz a böngészős megjelenítés területén. A széleskörben elterjedt JavaScript helyett C# kódot futtat kliensoldali scriptként. Ennek működése az úgynevezett WebAssembly technológia megjelenésének köszönhető, mely 2017 óta van jelen, és azóta a legtöbb létező böngésző támogatja [1].

A Blazornak jelenleg két típusát lehet megkülönböztetni: szerveroldali illetve kliensoldali.

A szerveroldali megközelítés esetén egy ASP.NET Core szerver intéz mindenféle interakciót, legyen szó üzleti logikáról, vagy akár csak a megjelenítés frissítéséről. A szerver a klienssel SignalR-en keresztül kommunikál. Enne következménye, hogy a kliensoldali követelmények rendkívül kevesek, mind számításigény, mind tárhely tekintetében: csak a kívánt parancsokat továbbítja a kliens a szerver felé, illetve annak válaszát jeleníti meg; a letöltendő tartalom kevés. Ebből azonban adódik az a hátrány is, hogy szerver nélküli Blazor szoftver ebben a felállásban nem készíthető, illetve a szerver-kliens közti kommunikációs hiba esetén szintén megszakad a működés.

A kliens oldali megközelítés esetén a kapcsolat létrejöttekor a futtatandó program a kliens eszközre települ, ezzel elérve akár szervernélküli működést. További előnye ennek a módszernek, hogy a kliensoldali erőforrások így teljes mértékben kihasználásra tudnak kerülni. Hátránya azonban, hogy több tárhelyet igényel (mivel az applikációt le kell tölteni), illetve a program funkciói szűkülhetnek a szerveroldali megközelítéshez képest, a böngésző szabja meg így a határt.

Megközelítési módtól függetlenül a kliensoldali megjelenítést elsősorban a böngésző végzi, azonban létezik kiegészítő könyvtár (Electron), mellyel különálló számítógépes program készíthető a forráskódból. Folyamatban van továbbá ennek a telefonos applikáció testvére (Mobile Blazor Bindings), azonban ez még fejlesztés alatt áll. [2] [3]

1.1.3. Unity

A Unity egy széles körben elterjedt, kifejezetten játékkészítésre kifejlesztett keretrendszer, játékmotor. Mára egy rendkívül kiforrott rendszer, rendkívül sok beépített funkcióval önmagában is, de számtalan harmadik féltől származó könyvtár letölthető hozzá; a felhasználóbázis is jelentős mértékű.

Működését tekintve szolgáltat grafikus szerkesztőfelületet, melyet főként a játék grafikus elemeinek tervezéséhez, létrehozásához lehet használni, valamint a főként üzleti logikai scripteket, programokat C# nyelven lehet írni. Megjegyzendő azonban, hogy maga az elkészített játék C++-ra fordul le, mielőtt futtatható állomány képződik belőle.

A motor egyik fontos tulajdonsága, hogy rendkívül platformfüggetlen, jelenleg több mint 25 rendszerre lehet vele játékot írni, ugyanabból a forráskódból [4]. További tulajdonsága még, hogy lehetőségünk van két-, illetve háromdimenziós játékok létrehozására is, melyet a projekt elkezdése előtt lehet kiválasztani.

A Unity mai fontosságát a játékiparban az is mutatja, hogy manapság akár a legnagyobb játékfejlesztők is használják, többek közt a CDPR is Unity-ben írta a Gwent többjátékos verzióját.

1.2. Szerveroldal

A szerveroldali feladatokat szintén két főbb alcsoportra bonthatjuk: fontos a felhasználók és adataik nyilvántartása, valamint a meccsek tényleges lebonyolítása.

A felhasználók adatainak nyilvántartása alatt értjük mind a játékos személyes adatainak (felhasználónév, e-mail cím, jelszó), mind a játékhoz szükséges adatok (az egyes játékosok által létrehozott egyedi kártyák, illetve ezen kártyákból összerakott paklik) tárolását, az adatok kezelését.

Ezen adatok közül kiemelendőek a kártyák, ugyanis esetükben az egyszerű szöveges adatok mellett (név, leírás, képesség, stb.) szükséges még a kártyán szereplő képet is eltárolni. Ez felveti a kérdést, hogy a képeket milyen formátumban, illetve milyen platformon tároljuk el.

Ezekből következik, hogy szükségeltetik a szerveroldalon egy adatbázis, mely ezeket az adatokat tudja tárolni, illetve az elérésüket megfelelő sebességgel biztosítani.

A meccsek tényleges bonyolítása pedig a szerveroldal másik lényeges feladatköre. Itt lényeges kritérium, hogy a szerver egy játékos kliensoldali kérésére biztosítson egy „várószobát”, melybe a játékos által kívánt másik játékos(ok) tudnak becsatlakozni; ez történhet például egy egyedi szobaazonosítóval.

A csatlakozás megtörténte után szükségeltetik a játékosok közötti kapcsolat fenntartása, a felhasználók interakcióinak egymáshoz való eljuttatása (kártya lerakása, passzolás, stb.). Egy mérkőzés vége után pedig a szoba megszüntetése a feladat.

1.2.1. Node.js

A Node.js egy nyílt forráskódú, JavaScripten alapuló szerveroldali keretrendszer. Az első verzió 2009-ben jelent meg, és napjainkig egy folyamatosan fejlődő, karbantartott rendszer. A rendszer forráskódja elérhető GitHubon [5], itt lehet követni a projekt állapotát.

Tulajdonságai közé tartozik, hogy natívan csak JavaScript kód futtatását támogatja, azonban nem zárkózik el az olyan kódoktól sem, melyek csak végső soron fordulnak le JavaScript kódra, ennek következtében lehetséges pl. a Microsoft által implementált TypeScript nyelven írni a weboldalon futó kódot.

Fontos tulajdonsága még, amelynek közlésére a fejlesztőcsapat is nagyobb hangsúlyt fektet [6], a szálmentes, aszinkron működés. A többi keretrendszerrel ellentétben itt nem indítunk operációs rendszer szintű szálakat, folyamatokat minden egyes, a szerverhez érkező kapcsolathoz, hanem egy főszál intézi az összes kliens kiszolgálását. A rendszerfejlesztők állítása szerint ezzel egy olyan működés érhető el, mely rendkívül kevés erőforrást igényel, de mégis nagy hatékonysággal, jó skálázhatósággal bír. Amit még szintén kiemelnek, hogy nem található a rendszerben *lock*, ennek következtében nem is következhet be nem kívánatos holtpont.

1.2.2. Django

A Django egy szintén nyílt forráskódú keretrendszer, ami szintén a GitHubon érhető el [7], a keretrendszer Python nyelven alapul. Alapvetően a Model-Template-View architektúrális mintát követi, mely a böngészős megjelenítésre specializálódott.

Amire a fejlesztőcsapat nagyobb hangsúlyt fektet, az a kód újrafelhasználás, kevés kódolás (mely a sok előre elkészített könyvtárnak köszönhető), laza függőségek, és a „ne ismételd magad” elv támogatása.

1.2.3. ASP.NET, ASP.NET Core

Az ASP.NET egy 2002-ben megjelent, a Microsoft által fejlesztett szerveroldali keretrendszer, mely a .NET keretrendszert egészíti ki. A rendszer segítségével különféle webes projektek hozhatók létre, lehet az akár egy MVC mintára illeszkedő monolitikus alkalmazás, mely egyszerre felel a szerveroldali üzleti logikáért és a kliensoldali megjelenítésért is, vagy egy API végpont, mely JSON válaszokat szolgáltat, melyeket szinte bármilyen kliensoldali alkalmazással fel lehet dolgozni. További előnye, hogy a rendkívül kiforrott C# nyelv köré összpontosul, mellyel mély komplexitású feladatok is megoldhatóak. Megjegyzendő továbbá, hogy a hasonló szerveroldali keretrendszerekhez képest kiemelkedő teljesítményt nyújt [8]. Hátránya azonban, hogy .NET keretrendszer lévén csak Microsoft Windows operációs rendszereket támogat.

A .NET Core (és ezzel együtt az ASP.NET Core) a .NET utódja, mely az előbb említett platformfüggésre ad megoldást. A .NET Core már egy nyílt forráskódú keretrendszer [9],

mely egyúttal platformfüggetlen is, ezzel elérve, Windows mellett Linuxon, vagy akár MacOS-en is futtatható ugyanabból a forráskódból származó program [10].

1.3. Szerver-kliens valós idejű kommunikáció

A szerver és kliensek közti kommunikáció lényeges pontja a projektnek, ugyanis a játszó felek közti interakciókra épül az egész játék. A kommunikációs technológia kiválasztásakor fontos szempont, hogy mennyi kérés-válasz párra számíthatunk a felhasználás során.

A felhasználók beléptetése kevés kommunikációt igényel, gyakorlatilag egy lépés. Kijelenthető az is, hogy az egy meccs (azaz két játékos mérkőzése) alatti kérés-válasz párok száma szintén nem sok, percenként maximum 10-15 kérés, ez is inkább gyorsabb játékmenet esetén jellemző.

Fontos megjegyzendő azonban, hogy amennyiben sok meccs játszódik időben párhuzamosan (azaz a szervernek sokkal több különböző játékost kell kiszolgálnia, ráadásul várhatóan mindenkinek egyedi adatokat szolgáltatnia, az egyedi kártyák miatt), úgy a kérések száma az egyidejű játékosok számával aránylik, mely könnyen sok lehet. Ebből következik, hogy olyan kommunikációs technológiát, hálózati protokollt kell használni, mely képes jelentős mennyiségű kérést továbbítani, illetve a szervernek ezeket a kéréseket tudnia kell megfelelő sebességgel kiszolgálni.

1.3.1. WebSocket

A WebSocket egy hálózati kommunikációs protokoll, melyet 2011-ben standardizáltak a 6455 számú RFC-ben [11]. Lehetőséget biztosít HTTP protokoll feletti, valós idejű, teljes-duplex kommunikációra, az az mind a küldő, mind a fogadó felek képesek egyidejűleg üzenni egymásnak. Portok szempontjából a standard HTTP 80, illetve 443-at használja, mely előnyt jelenthet abból a szempontból, hogy nem szükséges egyedi portok használata. További előnye a korábbi hasonló technológiákkal szemben (mint pl. a HTTP Polling [12]), hogy jóval kisebb kommunikációs overhead-del jár.

Implementációk tekintetében manapság a legtöbb böngésző támogatja. Megtalálható továbbá Microsoft ASP.NET Core-ban is, mint különálló technológia, illetve a szintén Microsoftos SignalR is erre a technológiára épít többek között [13]. Mindezek mellett a Unity motorhoz is található hozzá harmadik féltől származó implementáció [14].

1.3.2. gRPC

A gRPC egy Google fejlesztésű, RPC (távoli metódus hívás) keretrendszer. Hasonlóan a többi eszközhöz, ennek is egyik funkciója a kétirányú, akár teljes-duplex kommunikáció kliens és szerver közt. Főként mobil illetve böngésző eszközök szerverekkel való kommunikációját teszi lehetővé, HTTP/2 felett.

A gRPC egyik nagy előnyre, hogy sok programozási nyelvet támogat, köztük a C#-ot is, de megemlíthető még pl. a C++ vagy a Java is. Ennek következményeként akár különböző nyelven megírt komponensek közti kommunikáció jöhet létre, mely sok lehetőséget biztosít a komponensek megválasztása szempontjából [15].

A gRPC, sok társához hasonlóan, szintén nyílt forráskódú, mely elérhető GitHub-on [16].

1.3.3. SignalR

A SignalR egy, a Microsoft által fejlesztett, valós-idejű kommunikációra alkalmas eszköz. Szerveroldalon a szintén Microsoft által implementált ASP.NET (Core) platformon használható, míg kliensekből különböző programozási nyelveken írt programokból (pl. JavaScript, Java, de ugyanúgy .NET is) hívható.

A SignalR egy magas absztrakciós szintű hálózati kommunikációs eszköz, amely az alábbi technológiákat használja:

1. WebSocket,
2. szerver által küldött események,
3. HTTP long-polling.

A technológiák prioritás szerint vannak sorrendbe rendezve, azaz ahol csak lehetséges, a rendszer megpróbál WebSocketet használni, ha ezt a körülmények (szerver és/vagy kliens) nem teszik lehetővé, úgy próbálja a további technológiákat alkalmazni.

A SignalR használatával elérhető, hogy hálózati kommunikációt magas szinten, viszonylag kevés kóddal lehet megírni, azonban mégis több, az adott szerver-kliens képességeihez legjobban alkalmazkodó hálózati technológiát használjunk, mindezt teljesen automatikusan. Ez egy rendkívül nagy előnye ennek a rendszernek [17] [18].

További információ, hogy a SignalR is nyílt forráskódú, elérhető GitHubon [19].

1.4. Adatbázisok

A játékosok adatainak eltárolásához (felhasználói adatok, kártyák, meccsek, stb.) szükséges lesz egy adatbázis használata. Erre a célra a legjobban elterjedt adatbázis típus a relációs adatbázis. Ezek jellemzően az adatokat azok típusai szerint táblákban tárolják, melyekben rekordok találhatóak, ezek reprezentálják a tényleges adatokat (pl. a „Játékosok” táblában az első rekord lehet „Minta Béla, 22 éves”, és a második rekord lehet „Minta Petra, 19 éves”, stb.). További jellemzőjük a relációs adatbázisoknak, hogy SQL nyelv segítségével lehet velük kommunikálni, az adatokat manipulálni. SQL-lel lehet pl. lekérdezni egy tábla adatait (pl. milyen kártyák vannak, kik a játékosok), vagy beszúrni egy új rekordot (pl. regisztráció esetén egy új játékos felvétele a játékosok táblába, stb.

Manapság rengeteg SQL adatbázis rendszer létezik, a jelenlegi legelterjedtebbek közül az első három:

- MySQL
- PostgreSQL
- Microsoft SQL

A MySQL egy nyílt forráskódú [20] rendszer, melyet az Oracle tart karban. A PostgreSQL szintén egy nyílt forráskódú rendszer [21], míg a Microsoft SQL a Microsoft által létrehozott adatbáziskezelő [22].

A három rendszer között e feladat szintjén számottevő, lényeges különbség nincs, kiválasztásukat inkább a környező komponensek határozzák meg.

1.5. Választott technológiák

Amellett, hogy egy adott technológia mennyire felel meg a fentebb kifejtett követelményeknek, még az alábbi tulajdonságait fogom figyelembe venni:

- objektum-orientált fejlesztés támogatása,
- elérhető külső könyvtárak,
- .NET, C# nyelven való programozás.

Az objektum-orientált programozás hatalmas előnye a jól olvasható, modularizálható, újra felhasználható kód. A külső könyvtárak nagyban megkönnyíthetik a nem játékspecifikus, gyakran használt műveletek elvégzését (pl. JSON konvertálás, hálózati kommunikáció, stb.). A C# nyelven való programozás pedig az egyetemi tanulmányaimba illeszkedik bele, ebben a nyelvben rendelkezem a legtöbb programozási tapasztalattal.

A követelmények, illetve a hasonló implementációk áttekintése után az alábbi rendszereket kívánom felhasználni a projektmunka megvalósításához:

1.5.1. Kliensoldal: Blazor

A kliensoldali rendszer kiválasztása során az elsődleges szempontok a megfelelő teljesítmény és irányíthatóság volt.

Első választásom a Unity motorra esett, azonban ezt az ötletet végül elvetettem. Bár a motor számtalan lehetőséget kínál játékfejlesztésre, valamint a nagyobb játékok egy része szintén ebben a rendszerben van megírva, úgy érzem, hogy a feladat nem igényel ilyen komplexitású (és mellesleg hardverigényű) motort az elkészítéshez.

Ebből kifolyólag a megjelenítés céljából egy böngészős kliens mellett tettem le a voksomat. Ennek implementálásához választhattam volna a HTML, CSS, JS kombinációt, mely széles körben elterjedt, és a korábbi példák is azt mutatják, hogy a feladat elvégzéséhez teljesen elegendő eszközök. Mivel azonban Blazor is ezt a funkcionalitást nyújtja, azzal a különbséggel, hogy JS helyett az általam preferált C# nyelven lehet a scripteket megírni, ezért végső soron emellett a keretrendszer mellett

döntöttem. A Blazor által igényelt WebAssembly technológiát manapság szinte minden böngésző támogatja, így amiatt sem kell aggódni, hogy viszonylag új technológia ellenére kevés lenne a hozzá fellelhető futtatókörnyezetek száma.

1.5.2. Szerveroldal: ASP.NET Core

A kliensoldali Blazor kiválasztása után szinte magától értetődő volt az ASP.NET Core használata szerveroldali rendszerként. Bár a Blazor valamennyi szerverrel együtt tud működni, a Microsoft által támogatott és ajánlott ASP mégis a legoptimálisabb választás. Az ASP biztosítja továbbá az Entity Framework-öt, mely a webalkalmazás által igényelt adatbázis műveletek elvégzését könnyíti meg, annak tényleges implementációjától függetlenül. Megemlítendő még, hogy ASP.NET Core-ban is C# nyelven lehet programozni, így ennek választásával szerveroldalon is használható a nyelv.

1.5.3. Kommunikáció: SignalR

A Blazor és ASP.NET Core kombináció gyakorlatilag hozza magával a SignalR-t, amennyiben szükségünk van valós idejű kommunikációra a szoftverünkben. A SignalR előnye, hogy a korábbi komponensekkel megegyezően a Microsoft fejleszti és támogatja, ennek köszönhetően tökéletesen illeszkedik az eddigi választott rendszerekhez.

1.5.4. Adatbázis: Microsoft SQL

Adatbázisok közül, a teljes Microsoft lefedettség végett, a Microsoft SQL Server-re esett a választásom. A rendszer gyártója végett remekül alkalmazható a többi rendszerrel, az ASP.NET Core is alapértelmezetten ezt a rendszert használja; az Entity Framework-höz szintén található olyan hivatalos Microsoft könyvtár, melynek segítségével könnyedén tudunk kapcsolatot létesíteni az adatbázis és az ASP szerver közt az EF-t használva.

1.5.5. Választott rendszerek összegzése

A fent kiválasztott technológiák segítségével elérhető az, hogy az egész projektet, kliens és szerveroldalt egyaránt egységesen C# nyelven lehet megírni. Ez kódtisztaság szempontjából mindenképp kedvező, mivel így lehet talán a legolvashatóbb, legfenntarthatóbb kódot létrehozni. Tagadhatatlan az a tény is, hogy ebben a nyelvben vagyok a legjártasabb, valamint az egyetemi tanulmányaimba is beilleszkedik a C# nyelv.

További előny, hogy így minden termék Microsoft által támogatott. Ez eddigi személyes tapasztalataim szerint inkább pozitívum, hiszen így a jövőben érkező potenciális frissítésekre való átállás minimális problémát jelent (legyen az akár csak egy használt könyvtár, vagy az egész .NET keretrendszer). A talált hasonló rendszerek is azt mutatják, hogy egyszerűbb úgy tervezni, implementálni és fenntartani az alkalmazásunk, hogy csak egy cég eszközeit használjuk, több, kicsi, esetleg kevésbé megbízható harmadik féltől származó eszközzel szemben.

2. HASONLÓ RENDSZEREK

A projekt alapötletéül szolgáló hivatalos Gwent játékok mellett még az alábbi hasonló implementációkat tekintettem át.

2.1. Egyéb Gwent implementációk

Publikált, aktívan működésben lévő Gwent játékot nem találtam, így nyílt-forráskódú projektek után kezdtem el keresni, ezt a GitHub-on tettem. A talált projektek különböző nyelveken íródtak, a leggyakoribbak a JavaScript, Python, illetve C# voltak; azonban ezek közül sajnos azonban egyiket sem tudtam működésre bírni; ez legtöbbször annak volt betudható, hogy a projektek átlagosan 3-5 éve készültek, és azóta nem voltak frissítve, karbantartva, és a használt harmadik féltől származó könyvtárak elévültek az évek alatt.

2.2. Egyéb kártyajátékok

Következő lépésként a különböző, elterjedt kártyajátékokat vizsgáltam meg. Ezek, a teljesség igénye nélkül, az alábbiak voltak:

- a Blizzard által fejlesztett Hearthstone [23],
- a MegaCrit által fejlesztett Slay the Spire,
- illetve a Richard Garfield által fejlesztett Magic: The Gathering [24].

A felsorolt játékok bár mindegyikében a játékos maga tudja összeállítani a meccsekhez használt pakliját, azonban (a játékok versengő szerűségéből fakadóan) egyikben sincs lehetőség a teljesen saját kártyák előállítására.

Szintén megjegyzendő, hogy a felsorolt három játékból kettő (Hearthstone, Magic: The Gathering) is a Unity motort használja a kliensoldali szoftver implementálásához.

2.3. Egyéb egyedi kártyajátékok

Utolsó lépésként az legelterjedtebb játékoktól valamilyen szinten eltávolodva olyan már létező projektek után kutattam, melyek az egyedi játékmenetet helyezik középpontba, ezzel elérve, hogy a játékosok személyre tudják szabni a játékaikat.

Az első érdekes találat a „PlayingCards” névre keresztelt böngészős táblajáték szimulátor [25]. Ezen termék keretében egy rendkívül színvonalas implementációval ismerkedhettem meg. A játékosoknak előre elkészített, fizikai formában is létező játékok közül van lehetőségük választani, legyen az sakk, dáma, vagy csak egy üres virtuális asztal egy általános, 52 lapos franciakártya paklival. A játékosoknak ebben a rendszerben nem szükséges regisztrálniuk, de megvan rá a lehetőségek. A játékok lebonyolítása a kiválasztás után egy virtuális játékszoba létrehozásával kezdődik, melybe a további játékosok egy azonosítókóddal tudnak belépni. Megemlítendő, hogy a játékosok közti valós-idejű kommunikáció is rendkívül jól működött tesztelésem során; még arra is van

lehetőségük a játékosoknak, hogy egymás „kezét” (ami jelen esetben a kurzor pozícióját jelenti) valós-időben lássák.

A következő megemlítendő szoftver a „Card Game Simulator”, vagy röviden csak CGS [26]. Ez a megoldás az előzővel ellentétben nem böngészős klienst szolgáltat a játékosok számára, hanem egy Unity-ben írt programot; valamint itt alapvetően csak az általános 52 lapos franciakártyákkal játszhatunk. Ami miatt azonban mégis kiemelendő ez a projekt, az az, hogy itt a fejlesztő lehetővé tette az egyedi kártyák illetve játékok létrehozását, mindezt JSON formátumú dokumentumok segítségével. Míg ez egy nagyon jó kezdeményezés a személyre szabhatóság tekintetében, sajnos azzal a negatív mellékhatással is jár, hogy az informatikához kevésbé értők, laikusok számára problémát jelenthet a megfelelő JSON dokumentumok létrehozása illetve publikálása, mely egyenesen elveheti a felhasználó kedvét a használattól.

A harmadik és egyben utolsó találat amit megemlítenék a Dulst [27]. Ez egy böngészőből kezelhető, gyakorlatilag teljeskörű kártyajáték keretrendszer. A felhasználók regisztrálást követően létre tudják hozni saját kártyáikat, paklijaikat, de akár a teljes saját szabályrendszerű kártyajátékaikat; a személyre szabhatóságra rendkívül sok lehetőséget nyújt. A rendszer hátrányaként azt lehetne megemlíteni, hogy a felhasználók elé tárt felület nem a legátláthatóbb, használhatósága nem a legkényelmesebb. Továbbá érződik a rendszeren, hogy inkább azoknak nyújt fejlesztési lehetőséget, akik széleskörben szeretnék kártyajáték ötleteiket elterjeszteni, sem mint egy baráti társaságnak. Az fenti állítást támasztja alá az a tény is, hogy az oldal kínál a publikált játékokhoz nagy teherbírású szerverszolgáltatást is, természetesen felárért.

3. SPECIFIKÁCIÓK

3.1. Kliensoldal

A kliensoldali szoftvernek négy, jól megkülönböztethető komponenssel kell rendelkeznie: egy felhasználókezelővel, egy kártya- és egy paklikezelővel, illetve egy meccset lebonyolító felülettel.

A felhasználókezelő komponens feladata egy olyan oldal nyújtása, melyen a még nem regisztrált játékosok létre tudják hozni a profilukat (név, játékosnév, email cím, stb.), illetve a már regisztrált felhasználók be tudnak jelentkezni. Mivel a szakdolgozatban erős a hangsúly azon, hogy minden játékosnak lehetnek egyedi kártyái, ezért személyes felhasználói fiók nélkül nem lehet majd használni az alkalmazást.

A kártyakezelő komponens segítségével tudják a játékosok létrehozni az egyes kártyáikat. Ezen a felületen tudják feltölteni a kártyának a képét, annak nevét és leírását megadni, illetve a kártya különböző tulajdonságait (erő, speciális képességek, stb.) beállítani. Szintén ennek a komponens használatával lehet a meglevő kártyákat módosítani, illetve törölni.

A paklikezelő komponens felel az egyes paklik összeállításáért. A felhasználó itt tudja az általa már korábban létrehozott kártyákról eldönteni, hogy melyik kerüljön be egy adott pakliba, valamint a játékhoz szükséges további beállításokat (pakli képesség, vezér képesség) konfigurálni. Egy játékosnak több paklija is lehet, valamint lehetősége van őket szerkeszteni és törölni.

Az egyes mérkőzések lebonyolítását végzi az erre hivatott komponens. Itt az egyes játékosoknak lehetőségük van várószobák létrehozására, ahova játékostársaikat tudják meghívni, legyen az akár személyes hívás, vagy a korábban, az egyik hasonló rendszerben látott szobaazonosítós hívás. A játékosok csatlakozása után ezen a felületen tudják a játékosok a meccset irányítani, a játékban elvégezhető műveleteket (kártya lerakás, húzás, passzolás, stb.) elvégezni.

3.2. Szerveroldal

A szerveroldali szoftvernek gyakorlatilag a kliensoldalnál meghatározott feladatokat kell tudnia kiszolgálnia. Ez az adatbázissal való kommunikációt jelenti (felhasználók, kártyák, paklik CRUD műveletei), valamint az egyes játékosok közti kommunikáció megteremtését a meccsek során.

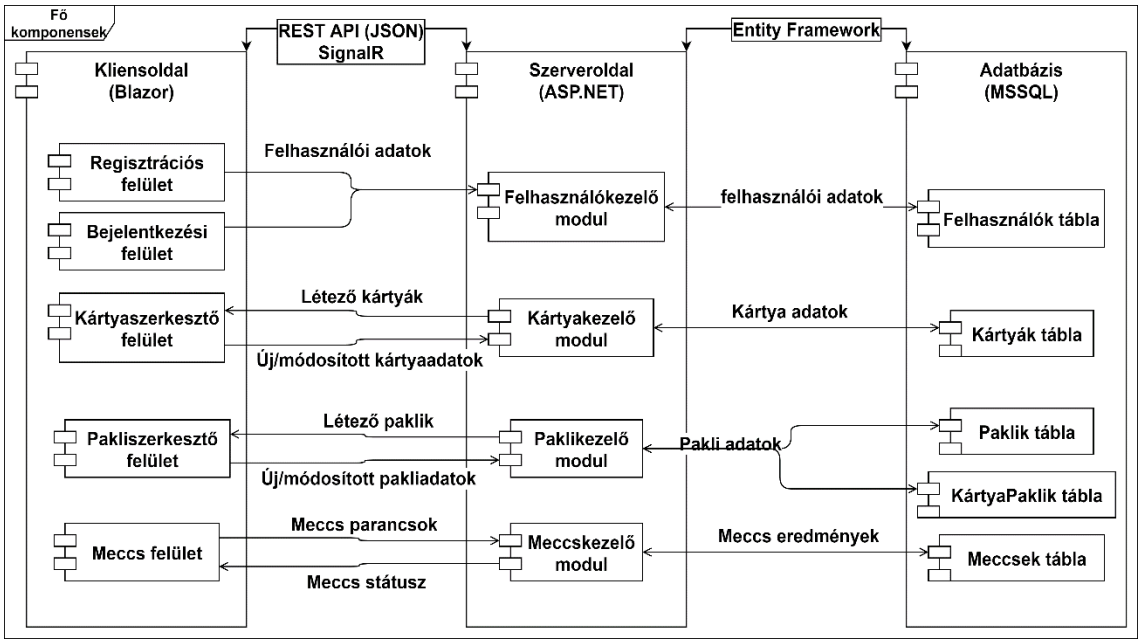
3.3. Kliens-szerver kommunikáció

Kijelenthető, hogy a feladatban gyakorlatilag mindennek valós időben kell történnie, legyen szó regisztrációról, játékosok közti üzenetváltásról, vagy a meccsek tényleges lebonyolításról. A szervernek tudnia kell fogadni az egyik kliens kéréseit (pl. kártya letétele), és erről a megfelelő másik klienst azonnal tudnia kell értesíteni (előző pl. A

játékos kártyát rakott le). Ebből következik, hogy valós idejű kommunikációs technológia használata elengedhetetlen.

4. RENDSZERTERV

4.1. Fő komponensek



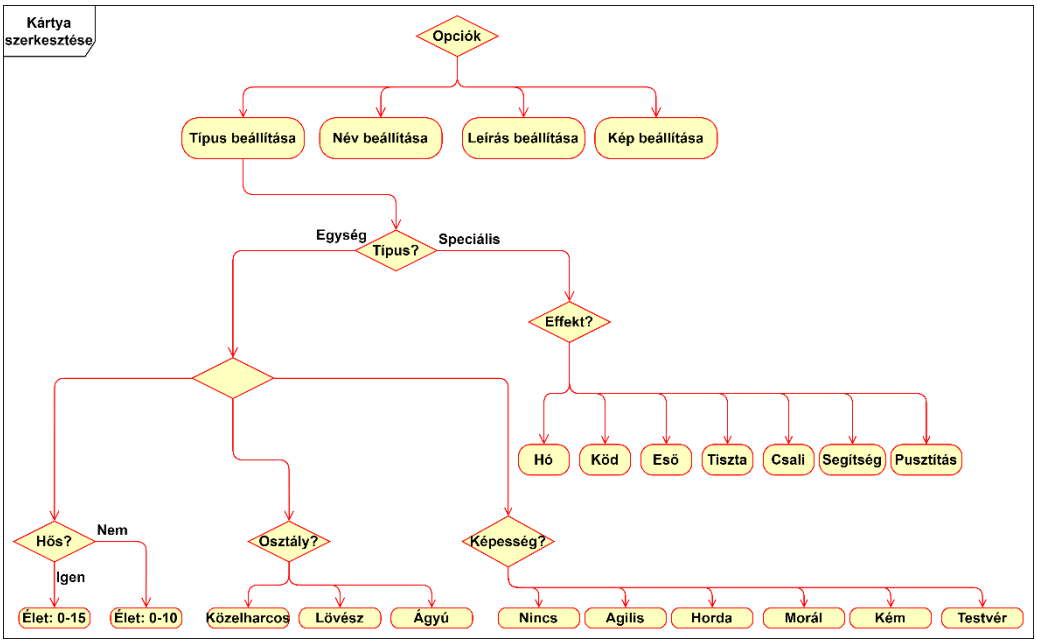
4.1. ábra: Magas szintű kompononens diagram

A specifikációknak megfelelően a program három fő modulra bontható, melyet a 4.1. ábra mutat be: kliensoldalra, szerveroldalra, és egy adatbázisra. A kliens és szerver közti kapcsolat két módon valósul meg: elsődleges kommunikációs forma a REST API, melyen keresztül JSON fájlok formájában történik az adatcsere. Ahol pedig szükséges a valós-idejű kommunikáció, illetve több felhasználó kezelése egyidejűleg, ott pedig a már említett SignalR-t fogom használni. A szerveroldal és az adatbázis közötti kapcsolatot pedig az Entity Framework teremti meg.

A felhasználókezelés, kártya, illetve pakliszerkesztés folyamatát túlzottan nem szeretném részletezni, mivel a mindennapokban használt CRUD[1] műveletek, a játék megfelelő elemeinek felhasználásával való implementációja. Ezekről a műveletekről általánosságban elmondható, hogy a szerveroldal csak, mint egy közvetítő fog részt venni a felhasználó és az adatbázis között. A felhasználó a megfelelő adatok megadásával tud majd fiókot létrehozni és abba bejelentkezni, kártyákat, illetve paklikat létrehozni. Ezen adatok ellenőrzés után kerülnek a szerveroldalra, mely szintén ellenőrzi őket, és beírja az adatbázisba az értékeket. Hasonló módon járunk el az adatok listázásánál, frissítésénél, törlésénél.

[1] Create, Read, Update, Delete azaz Létrehozás, Listázás, Frissítés, Törlés műveletek

4.2. Kártyák képességei, kártyaszerkesztési opciók



4.2. ábra: Kártyaszerkesztő menütérkép

Az egyes kártyák szerkesztési lehetőségeit a 4.2. ábra mutatja be: a felhasználók tudják a kártya nevét, annak leírását szerkeszteni, illetve a kártyához egyedi képet feltölteni. Ezen opciók a kártya működését semmilyen módon sem befolyásolják, csupán a személyre szabhatóság szabadságát hivatottak növelni.

A kártyának továbbá szükséges megadni a fő típusát, ami lehet egység kártya, vagy speciális kártya.

Amennyiben speciális kártyát hoz létre a játékos, úgy az alábbi effektek közül kell egyet választania:

- hó, mely az első sorban levő kártyák erejét 1-re állítja,
- köd, mely a második sorban levő kártyák erejét 1-re állítja,
- eső, mely a harmadik sorban levő kártyák erejét 1-re állítja,
- tiszta idő, mely az előbbi három effekt közül az összeset feloldja,
- csali kártya, mellyel a játékos egy korábban lehelyezett kártyáját tudja visszavenni,
- segítség kártya, mellyel egy tetszőlegesen választott sor kártyáinak az értékét lehet megduplázni,
- pusztítás kártya, mely a legerősebb lehelyezett kártyá(ka)t helyezi a temetőbe.

Amennyiben egység kártya létrehozása mellett dönt a felhasználó, az alábbi beállítási lehetőségek adottak:

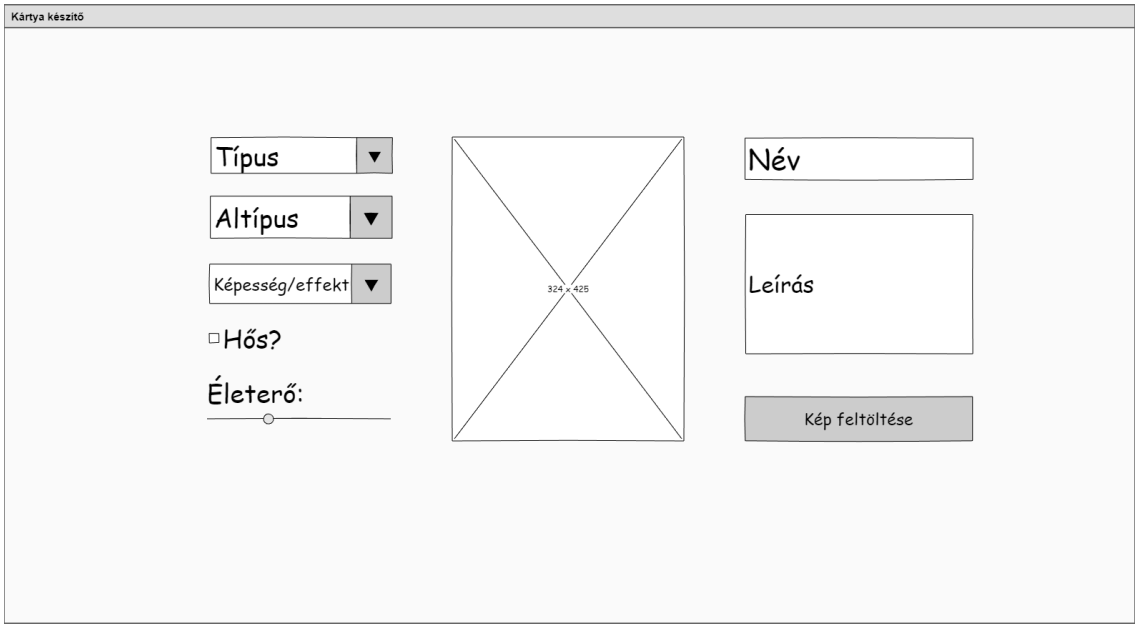
Szükséges eldönteni, hogy hős-e az adott kártya. Amennyiben igen, úgy az életereje 0 és 15 között állítható, amennyiben nem, úgy 0 és 10 között.

Szükséges kiválasztani az egység osztályát, melyek azt határozzák meg, hogy a kártyát melyik sorba lehet lehelyezni. A közelharcos kártyákat az elsőbe, a lövészeket másodikba, az ágyú kártyákat pedig a leghátsó, harmadik sorba lehet kijátszani.

Továbbá be lehet állítani az egységeknek is egy képességet. Ezek lehetnek az alábbiak:

- agilis, így bármelyik sorba lehelyezhető,
- horda, így az összes vele azonos kártyát egyidejűleg helyezi le a játékos,
- morál, mely a lehelyezett sorban levő összes többi kártyának eggyel növeli az erejét,
- kém, melyet az ellenfél oldalára lehet lehelyezni, cserébe a játékos húzhat másik két kártyát,
- testvér, melyből az egyforma kártyák megduplázzák egymás erejét.

A kártyaszerkesztő tervezett felületét a 4.3. ábra mutatja be:



4.3. ábra: Kártyaszerkesztő tervrajz

4.3. Paklik képességei, pakliszerkesztési opciók

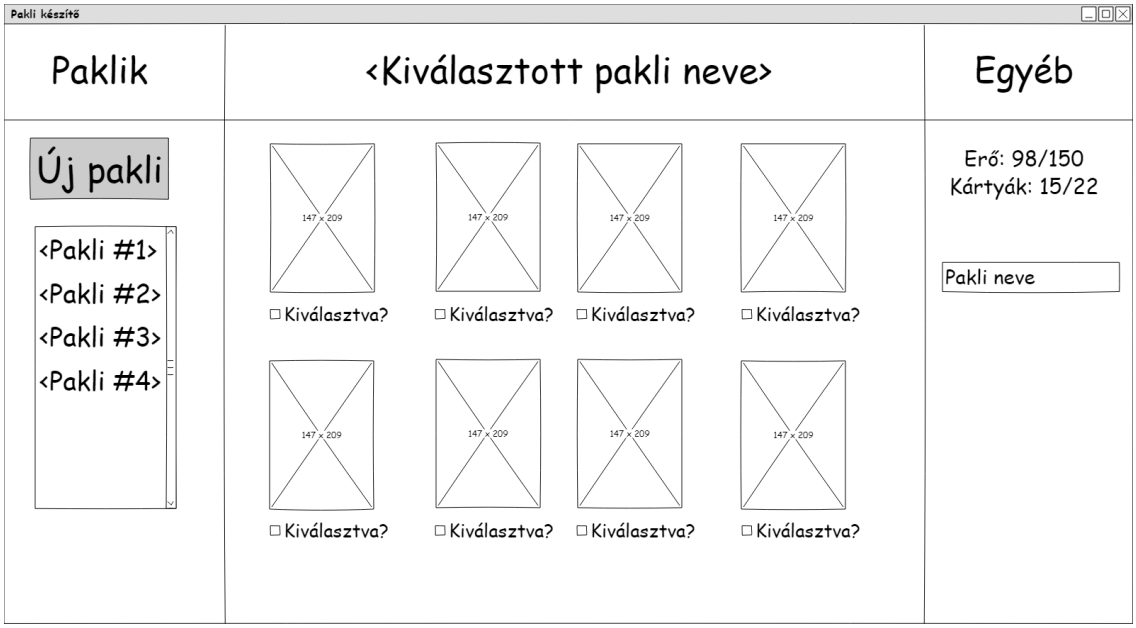
Az egyes paklik szerkesztése során az felhasználó meg tudja adni a pakli nevét, illetve be tudja állítani annak képét. Hasonlóan magukhoz a kártyákhoz, ezek csak esztétikai beállítások, a játékmenetet nem befolyásolják.

Amivel a felhasználó értelemszerűen tudja befolyásolni paklija működését, az a benne található kártyák kiválasztása.

Ahhoz, hogy egy pakli érvényes legyen, és lehessen vele játszani, az alábbi követelményeknek kell teljesülnie:

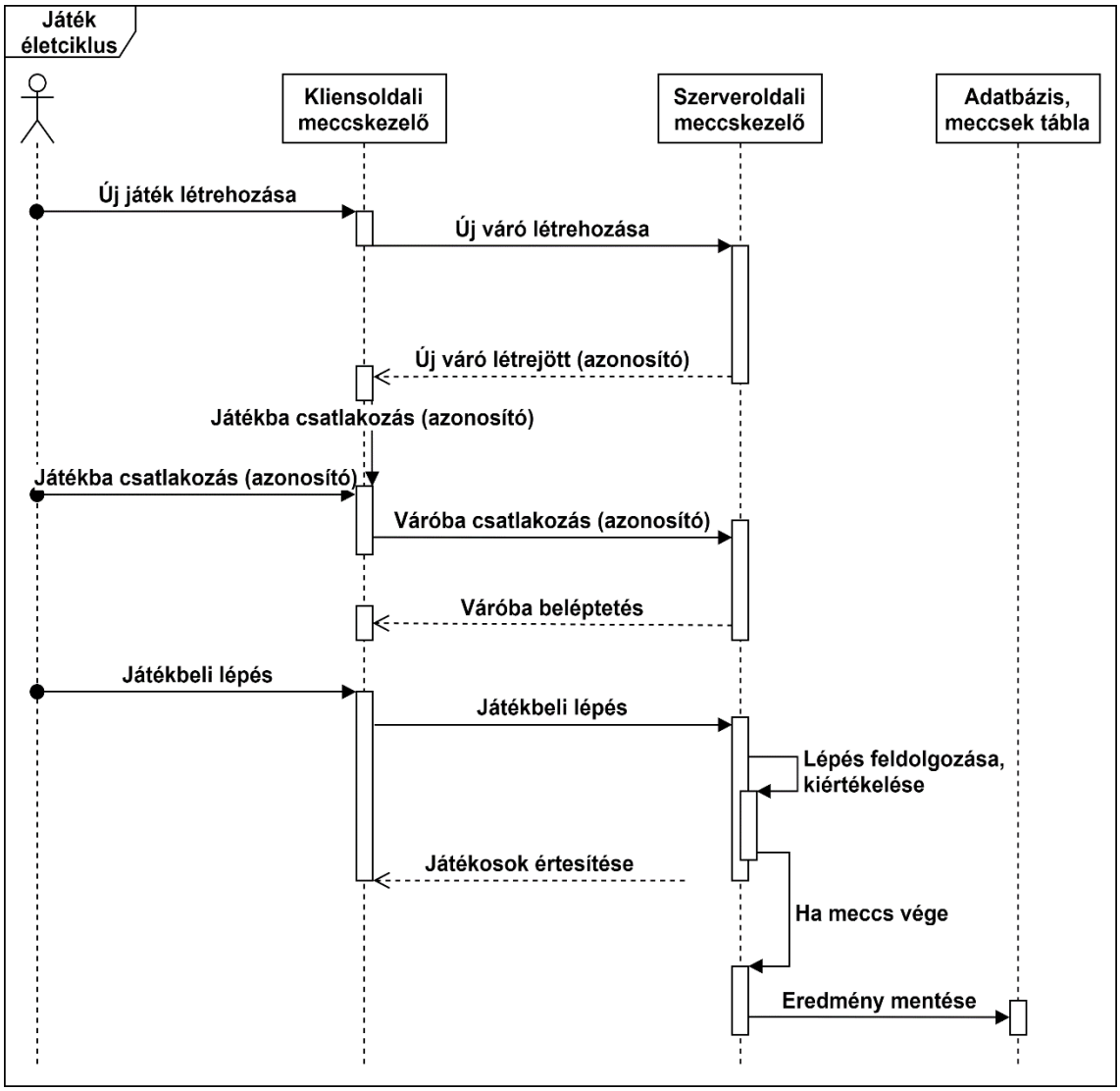
- a pakliban minimum 22 egységkártyának lennie kell,
- a pakliban található kártyák összegzett ereje nem haladhatja meg a 150-et,
- a kártyában maximum 10 speciális kártya lehet.

A pakliszerkesztő tervezett felületét a 4.4. ábra mutatja be



4.4. ábra: Pakliszerkesztő drótváz

4.4. Játék folyamata



4.5. ábra: Alapvető játékinterakciók

A játékosfelülettel való főbb interakciókat a 4.5. ábra szemlélteti. A játékosnak lehetősége van új várót[1] létrehozni, mely gyakorlatilag egy kérés a szerver felé. Ezt a kérést a szerver feldolgozza, és a szükséges műveleteket végrehajtja (egy új váró létrehozása, benne a játékosok létrehozása stb.), majd a hívó klienst értesíti a váró létrejöttéről, valamint küldi a várónak az azonosítóját, mellyel csatlakozni lehet.

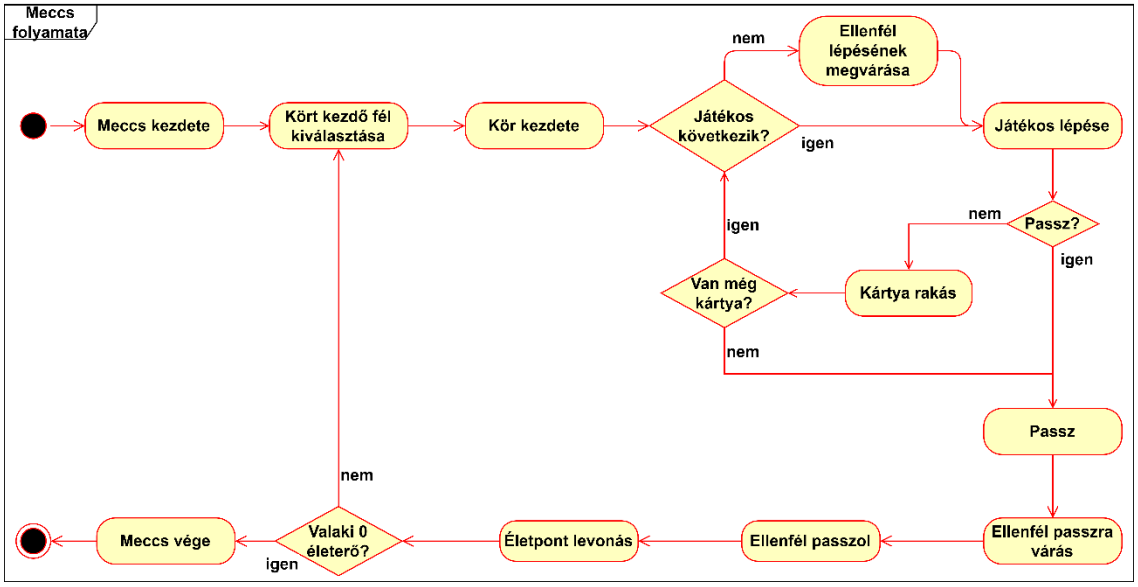
A játékosnak lehetősége van már létező várókba csatlakozni. Ehhez szükséges a megfelelő szoba azonosítójának elküldése a szerver felé, melybe a felhasználó csatlakozni akar. Ezt a kérést megkapva a szerver azonosítja a felhasználót, majd a megfelelő helyre belépteti a felhasználót, ezután pedig értesíti őt a művelet befejeztéről.

[1] várószoba, lobby

Amit érdemes kiemelni, hogy a váróba való beléptetés mindig ugyanazzal az eljárással történik, attól függetlenül, hogy a felhasználó egy új szobát hoz létre, vagy már egy meglévőbe kíván csatlakozni. A különbség csupán abban rejlik, hogy új szoba létrehozása esetén a kliens automatikusan, a felhasználó számára transzparensen, meghívja a csatlakozás eljárást a szervertől kapott azonosítóval.

A játékba való csatlakozás után a felhasználók játékbeli lépései követik egymást, melyet a szerver dolgoz fel és értékel ki, majd a megfelelő értesítéseket küldi ki a játékosoknak. Amennyiben a játék véget ér, úgy a szerveroldal az eredményt az adatbázis megfelelő táblájába menti el.

A tényleges játékmenetet a 4.6. folyamatábra szemlélteti:



4.6. ábra: Játékmenet folyamatábra

1. A kört kezdő fél kiválasztása: ez az első kör esetén a meccset (várót) létrehozó játékos, a további körökben mindig az azt megelőző kör nyertese kezd.
2. Ezt követi a kör kezdete; ha a játékos következik, akkor ugrás a 3. pontra, amennyiben azonban nem, abban az esetben meg kell várnia az ellenfél lépését.
3. Következik a játékos lépése.
4. A játékos eldönti, hogy passzol-e abban a körben. Ha passzol, akkor ugrás a 6. pontra, amennyiben nem, úgy
5. A játékos lehelyezi a kiválasztott kártyáját. Amennyiben ezután nem marad több a kezében, automatikusan passzol, tehát ugrás a 6. pontra, ellenkező esetben visszaugrás a 2. pontra.

6. A játékos passzol, várakozik az ellenfél passzára. Az ellenfél passz után a megfelelő játékos veszít egy életpontot.
7. Amennyiben valamelyik játékos 0 életponttal rendelkezik úgy ugrás a 8. pontra, ellenkező esetben következik a következő kör, visszaugrás az 1. pontra.
8. A meccs vége, értékelése: a 0 életerős játékos vesztett, akinek maradt életereje, nyert.

A játékosfelület tervezetét a 4.7. ábra mutatja:

4.7. Játékosfelület tervrajz

4.4.1. Egyjátékos mód tulajdonságai

Ebben a játékban (és az összes hasonló felépítésű pakli-építő-játékban[1] egyaránt) meg lehet határozni három tényezőt, melyek befolyásolják, hogy mi lesz egy meccs kimenetele; ezek az alábbiak:

Első tényező a játékos által összeállított pakli tartalma. Az ilyen játékokban, mint amilyen ez is, természetesen nagy hangsúlyt kap az, hogy ki milyen kártyákkal szeretne meccsbe szállni. A játékosnak saját magának kell rájönnie, kitapasztalnia azt, hogy mely kártyákat, milyen helyzetben tud jól, hatékonyan felhasználni, és ez alapján kell létrehoznia egy paklit. Szintén megjegyzendő az, hogy az ilyen játékokban (és természetesen ebben is),

[1] deck-building-game

valamilyen formátumban megjelennek korlátozások, amiket betartva lehet csak érvényesnek tekinthető paklit létrehozni, ez a 4.3. pontban van kifejtve. Erre a játékbeli egyensúly fenntartása végett van szükség, elvégre nem szeretném azt engedni, hogy olyan paklit lehessen létrehozni, ami minden esetben megverhetetlen.

Második tényező maga a játékos által követett logika, ami alapján a létrehozott kártyáit játssza ki. Ezt az előző ponthoz hasonlóan, a játékosnak kell kitapasztalnia, hogy az egyes kártyákat, esetleg több kártya kombinációját milyen helyzetben, hogyan, milyen sorrendben érdemes letenni, vagy akár adott esetben passzolni.

A harmadik tényező pedig maga a véletlen. Lehet a játékosnak akármilyen jó paklija, és hozzá tartozó terve, ha a szerencse épp nem áll az ő oldalán. Ez főként a kártyahúzások során érezhető: a 10 kezdőkártya véletlenszerűen van kiosztva a játékosoknak, és a további húzások (legyen ez a meccs eleji 2 újra-húzás, vagy egy kém kártya kijátszása) során is véletlen kap a játékos további kártyákat. Itt jön képbe az első tényező, miszerint valamilyen balansz jelen van a paklik létrehozása során, ezért muszáj olyan kártyáknak is jelen lennie, amik gyengébbek, kevésbé jól használhatóak. Ha a játékos épp ezeket húzza ki mind, akkor elég nehéz dolga lesz, amennyiben pedig épp a legjobbakat kapja meg, akkor könnyű győzelem várhat rá.

A fenti hármas jól leírja, hogy mikre kell figyelni a játék során, ezáltal az számítógép játékos (bot) implementálásakor is. Bár elsőre nem tűnnek kifejezetten bonyolult dolgoknak, azonban ha belegondolunk, hogy csak egy kártyát hány féleképpen lehet elkészíteni, illetve egy pakliban hány kártyának kell szerepelni ahhoz, hogy érvényes legyen, könnyen beláthatjuk, hogy a lehetőségek száma igen nagy, és ekkor a logikáról, ami alapján ezeket kijátsszuk, még nem is beszéltünk.

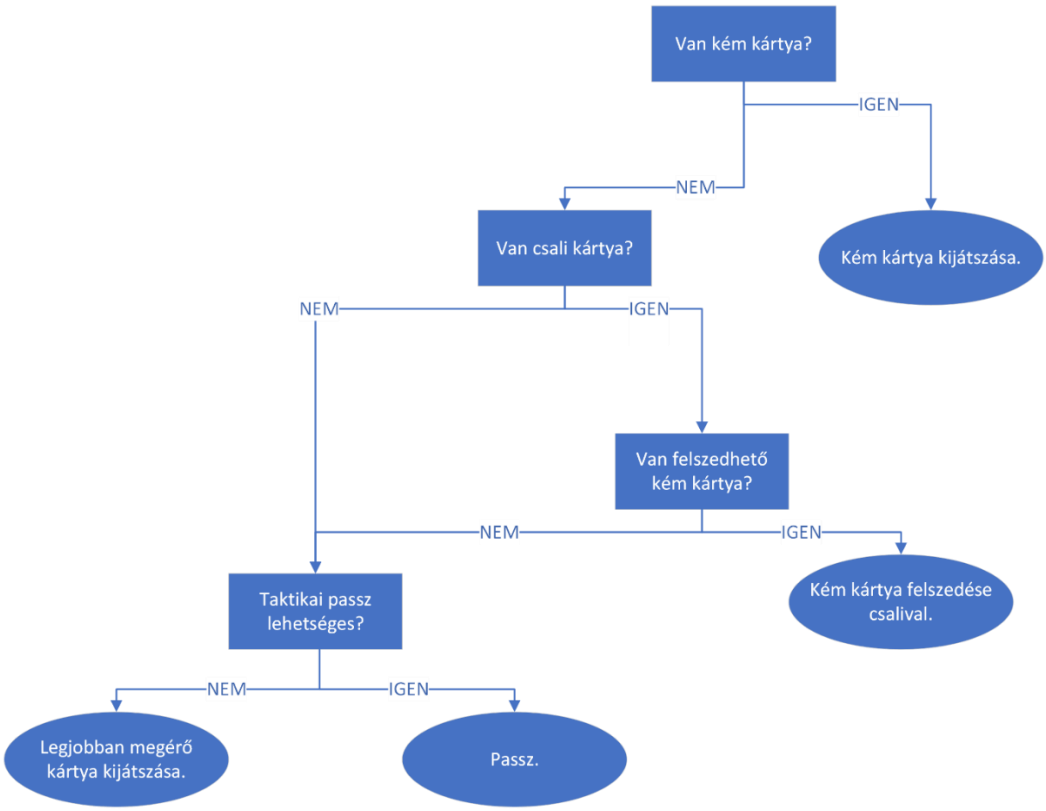
4.4.2. Egyjátékos bot logikája

Mindezeket figyelembe véve, az általam tervezett bot egy előre elkészített paklit használ, és az ellenfélhez és a véletlenhez valamelyest adaptálódva hoz meg játékbeli döntéseket. Mind az összeállított pakli, mind a játékmenet logikája a saját tapasztalatomból ered, mivel a játékom alapját képező eredeti Gwent játékkal viszonylag sokat játszottam, így sikerült kialakítani egy kombinációt, amely viszonylag széles körben használható hatékonyan (azonban természetesen ez sem verhetetlen). Az így született bot játékos mechanikája a következő, a három szempontot vizsgálva:

A pakli összeállításakor törekszem a minimális kártyaszám használatára, mivel így nagyobb valószínűséggel húzza ki a bot a jobb, erősebb kártyákat. Szintén használok kém kártyákat, a fentebbi ok miatt, ezzel is elősegítve az erős kártyák kihúzását. Továbbá a pakli tartalmaz csali kártyákat is, melyek hasznosak lehetnek, ha az ellenfél is használ kémeket, mivel azokat a csalival fel tudja a bot szedni, és ismételten kijátszani őket az ellenfél ellen. Jelentős hangsúlyt kapnak a segítség kártyák, melyekkel egy adott sor erejét lehet megduplázni, ezeket jellemzően az utolsó körben használja a bot. Maguk az

egységek közt viszonylag erősebb testvér kártyák lesznek, melyek szintén tudják egymás erejét szorozni, illetve hős kártyák, melyeket lehelyezés után már semmilyen effekt nem érint, legyen az a bot számára előnyös vagy hátrányos.

A bot által végrehajtott logikát a 4.8. diagram szemlélteti:



4.8. ábra: Egyjátékos bot logika

Kém kártyát minden esetben megéri kijátszani, mert bár így az ellenfél számára adunk pontokat (már amennyiben a kémünk nem 0 pontos), a bot két másik kártyával gazdagodik. Amennyiben a bot fel tud szedni csalival egy ellenfél által lehelyezett kémet, azt is érdemes mindig megtenni, ezzel elősegítve azt, hogy az a következő körben ismételten ki tudja játszani azt.

Amennyiben fentebbieket (már) nem lehet végrehajtani, a bot eldönti, hogy stratégiailag megéri-e passzolnia; ehhez az alábbi kritériumok szükségesek:

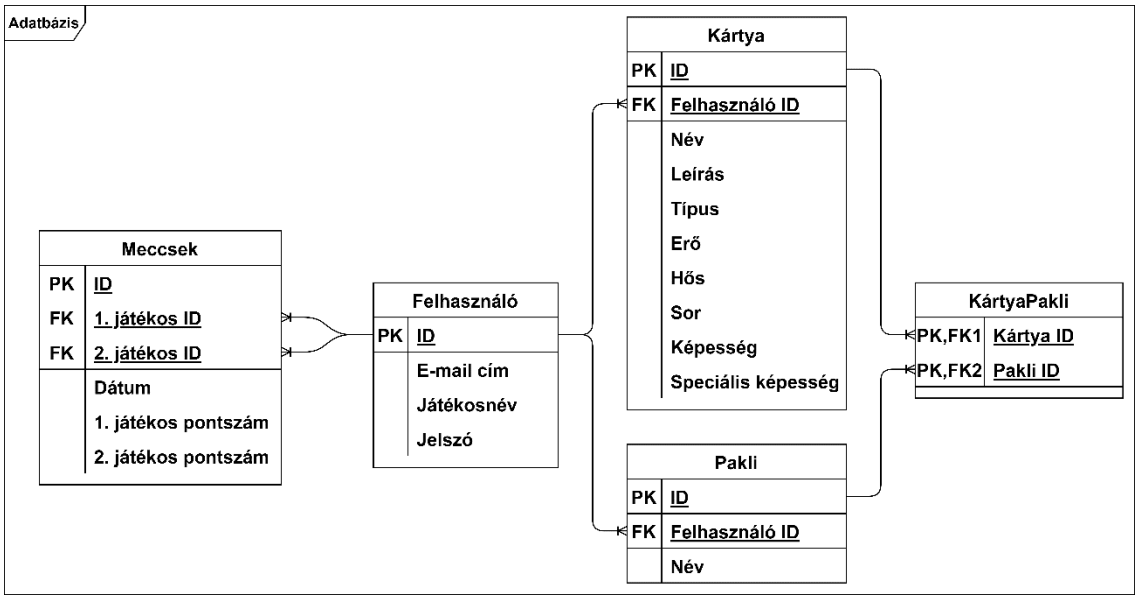
- legalább 2-vel több kártyája legyen, mint az ellenfelének;
- kevesebb pontja legyen, mint az ellenfelének;
- „bukható” legyen az adott kör, vagyis az állás 2:2 vagy 2:1 a bot javára;
- az ellenfél még nem passzolt, tehát lehet, hogy tovább küzd.

Amennyiben teljesülnek a feltételek, és a bot passzol, úgy a következő, második vagy utolsó kört több kártyával kezdi, mint ellenfele, ezzel nagy valószínűséggel biztosítva a győzelmét.

Amennyiben nem engedheti meg magának a bot a passzolást, úgy kiválasztja a kezéből a legjobban megérő kártyát, és azt játssza ki.

A leírásból is talán látszik, de a bot játékstílusára jellemző, hogy elsősorban minél több kártya kihúzására törekszik, illetve a saját maga pontjainak növelésére, semmint az ellenség gyengítésére. Ez jól használható sok esetben, azt tapasztaltam, hogy az esetek nagy részében elég pontra lehet szert tenni az ellenfél legyőzésére. Továbbá észre lehet venni, hogy a bot érzékeny a különböző negatív effektekre; amennyiben például az ellenfél sok időjárás kártyával, pusztítás kártyával rendelkezik, netalán ezek kompenzálására még hős kártyákat is használ, az igencsak nehéz helyzetbe tudja hozni a botot.

4.5. Adatbázis



4.9. ábra: Adatbázis terv

Az adatbázis felépítését a korábbiakban kifejtett kritériumok szerint terveztem meg, a táblák diagramja a 4.9. ábrán látható.

Felhasználók tábla tartalmazza az azonosításhoz szükséges adatokat, azaz a felhasználó egyedi azonosítóját; e-mail címét, amellyel regisztrált, ez szintén egy egyedi érték; az általa használni kívánt játékosnevét; valamint a belépéshez szükséges jelszavát.

A kártyák tábla segítségével tárolom el az elkészült kártyákat, vagyis a kártya egyedi azonosítóját; a kártyát létrehozó felhasználó azonosítóját, ő a kártya tulajdonosa; továbbá a kártyaspecifikus adatokat, melyeket a 4.2. pontban fejtettem ki.

A pakli tábla felel az elkészült paklik tárolásáért. Elmentem az adott pakli egyedi azonosítóját; a kártyákhoz hasonlóan eltárolom a létrehozó játékos azonosítóját; valamint a pakli nevét.

Mivel egy pakliban több kártya van, és egy kártya is benne lehet több pakliban, egy klasszikus több-a-többhöz[1] kapcsolat áll fent. Ennek kezelésére az egyik legelterjedtebb és leghasználhatóbb megoldást választottam, egy kapcsolótáblát vezettem be, *KártyaPakli* néven. Ez tartalmazza az adott kártya azonosítóját, illetve a pakli azonosítóját, amelybe a kártyát behelyezte a felhasználó. Mivel mind a kártyák, illetve mind a paklik egyedileg vannak azonosítva, illetve azok is tárolják a tulajdonos azonosítóját, így további azonosítókra nincs szükség.

[1] Many-to-Many

Az utolsó tábla pedig a lejátszott meccsek eredményeinek tárolásáért felel, ez az alábbi adatokat tartalmazza: a meccset játszó játékosok azonosítóját, az egyes felek által elért pontszámot, valamint a meccs dátumát.

5. IMPLEMENTÁCIÓ

A program fejlesztését a megfelelő projekt létrehozásával kezdtem, ehhez a Microsoft Visual Studio 2019-es verzióját használtam fel. A Blazor Webassembly projekt kiválasztása után lehetőségünk van kiválasztani, hogy ASP.NET Core szerveroldalt szeretnénk, vagy esetleg mi magunk valamilyen más kiszolgálót használni. Mivel nekem pontosan erre volt szükségem, így éltem ezzel a lehetőséggel, ahogyan a beépített *Individual Accounts*[1] autentikációval is. Ez egy előre implementált felhasználókezelő komponens, mely rengeteg, a mindennapokban használt funkcióval rendelkezik, és könnyen személyre lehet szabni (pl. játékosnév hozzáadása). A felhasznált keretrendszer pedig a .NET 5. verziója. Amit erről érdemes megemlíteni, hogy a Microsoft az 5-ös verzióval egyesítette a korábbi platformfüggetlen .NET Core, és a Windows specifikus .NET Framework rendszereket, ezzel egy egységes, platformfüggetlen rendszert kapva.

5.1. A projekt felépítése

A fentebbi folyamat következtében létrejött projekt három fő részre tagolódik:

Kliensoldal: a böngészőben futó Blazor kódja, mely tartalmazza az elérhető oldalak leírását, a szerverrel való kommunikációhoz szükséges API hívásokat.

Szerveroldal: a tényleges kiszolgáló ASP.NET kódja. Itt találhatóak a kontrollerek, szolgáltatások, hubok, modell osztályok.

Megosztott erőforrások: azon kódok érhetőek el itt, melyeket mind a kliens, mind a szerveroldal egyaránt használ. Itt érezhető igazán, hogy jó döntés volt a Blazor + ASP.NET kombináció, mivel rengeteg kódduplikáció elkerülhető így. Jó példa erre a DTO[2] osztályok: a szerver által használt modell osztályokat nem szeretnénk a kliens felé publikálni, ezért használunk DTO osztályokat, melyekbe tényleg csak azok az adatok kerülnek, amik feltétlen szükségesek a két oldal közti kommunikációhoz. Amennyiben eltérő nyelv lenne a kliens és szerveroldal, úgy ezeket az osztályokat kétszer kéne megírni; így azonban elég egyszer, és a megosztott helyről mindketten elérik.

5.2. Általános eszközök

A projekt írása közben felhasználtam néhány általános eszközt, melyek konkrét komponenshez, feladathoz nem köthetőek, a program egészében megtalálhatóak:

ILogger – naplózó szolgáltatás, mellyel az egyes műveletek fázisait, annak sikerességét/sikertelenségét iratom ki a szerver naplóba. Általánosságban is hasznos a visszakövethetőség miatt, azonban hibakereséshez is remek eszköz, fejlesztés során is.

[1] Egyéni felhasználói fiókok
[2] Data Transfer Object

AutoMapper – a projekt során több helyen is szükségem van volt hasonló osztályok közti konverzióra (jellemzően adatbázis modellosztályok és DTOk között), erre használtam egy külső könyvtárat, a .NET-es környezetben legelterjedtebb AutoMapper-t.

Konfigurációs állomány – a program indulásakor ebből a fájlból (konvenció alapján az *appsettings.json*) olvas be a szoftver bizonyos beállításokat; én az MSSQL adatbázishoz tartozó elérést állítottam be (*ConnectionString* érték), valamint létrehoztam egy egyedi változót is, mely a kártyák képeit tartalmazó mappa elérési útvonalát adja meg.

5.3. Entity Framework, CodeFirst

A korábbiakban specifikált adatbázis táblák létrehozása volt az első tényleges feladat. Itt az úgynevezett *Code First*[1] megközelítést választottam, mely azt jelenti, hogy nem a tényleges adatbázisban hozom létre a szükséges táblákat illetve mezőket, hanem a programkódban írom meg az azoknak megfelelő osztályokat (táblák) illetve tulajdonságokat (oszlopok). Ezek az osztályok lesznek később az Entity Framework segítségével valós táblákká, oszlopokká, illetve rekordokká konvertálva.

Az Entity Framework egy ORM[2] rendszer, mely egy absztrakciós réteget alkot a program és az adatbázis között. Ezzel elérhető az, hogy a programkód függetlenedjen az adatbázistól, számára lényegtelen lesz az, hogy milyen a tényleges adatbázis implementáció. Amit ez számomra a gyakorlatban jelent például, hogy nem SQL lekérdezéseket írok az MSSQL szervernek, hanem a C#-ban elterjedt LINQ lekérdezéseket, melyeket majd az EF fordít le az adatbázis számára SQL nyelvre.

5.3.1. A felhasználó osztály

A felhasználó osztályt külön emelném ki, mivel az előzőektől eltérően ez nem egy teljesen önálló osztály. Ez a korábban említett, Microsoft által beépített autentikációs rendszer miatt van. A létrejött felhasználó osztály a már létező *IdentityUser* leszármazottja, attól megörökölve az azonosításhoz szükséges adattagokat (pl. email, felhasználónév, jelszó); ezt egészítettem ki a számomra szükséges adatokkal (pl. játékosnév).

5.3.2. Adatbázis kontextus, adatbázis migrációk

A létrehozott modellosztályokat, hogy ténylegesen bekerüljenek majd az adatbázisba, egy úgynevezett adatbázis kontextus osztályba kell elhelyezni, mint publikus tulajdonság. Ez szintén egy származtatott osztály, mégpedig az *ApiAuthorizationDbContext<IdentityUser>* az őse. Mint az a nevéből is látszódik, ez az ősosztály elfogad egy *IdentityUser* típust, vagy annak egy leszármazottját. Pontosan ide

[1] Kód először
[2] Object-Relational Mapping

illeszkedik az előző pontban kiemelt felhasználó osztály, mely így már rendelkezik a kiegészítésekkel. A többi modell osztályait, mint publikus tulajdonság vettem fel.

Ahhoz, hogy a létrejött konfiguráció valóban létrejöjjön adatbázis táblák formájában, úgynevezett adatbázis migrációra van szükségünk. Ez egy egyszerű parancs, mely egy kicsit hasonlít verziókezelő szoftverekhez: megadjuk az adott migráció nevét, a rendszer detektálja a változásokat, majd azokat végrehajtja a tényleges adatbázison. Az egyes migrációk között tudunk visszalépni (*rollback*), amennyiben esetleg hibát követnénk el.

5.4. Szolgáltatások és kontrollerek

A programírás közben törekedtem a kontroller-szolgáltatás-adatkontextus[1] réteghármas használatára. A kontrollerek a szerveroldali végpontokhoz érkező webes kéréseket szolgálják ki; erre egy példa lehet a *KártyaKontrollerhez* érkező *GET* kérés, mellyel lekérdezi a felhasználó az összes kártyáját, míg egy *POST* kéréssel egy új kártyát ad hozzá a meglévőkhöz.

Ahhoz hogy a kontroller ezeket a feladatokat el tudják végezni, a megfelelő szolgáltatásokat kell használniuk; folytatva az előző példát, a *KártyaKontroller* segítségül a *KártyaSzolgáltatást* használja. A szolgáltatás végzi a kérések tényleges kiszolgálását, legyen az az adatbázisból való lekérdezés, vagy a meccsek lebonyolítása. Feladattól függően a szolgáltatás még igénybe veheti az *AdatKontextust*, mely az előző fejezetben tárgyalt Entity Framework által biztosított felület az adatbázis elérésére.

Amit ezzel a komponens hármassal elérek, az az, hogy az egyes részegységek függetlenek egymástól, és tényleg csak azokat a feladatokat végzik el maguk, amelyeket nekik szükséges. Ahhoz, hogy ez működhessen, szükséges az alábbi technológiát felhasználni:

5.5. Függőség injektálás

A függőség injektálás[2] egy olyan szoftvertechnika, melynek segítségével egy objektumot (függés) tudunk átadni, átpasszolni egy másik objektumnak (függő). Például az alkalmazásunk szerveroldala szeretné jelenteni minden kártyalerakás után a meccs állapotát, ehhez pedig szüksége van (függ) egy olyan objektumtól, szolgáltatástól, ami ki tud írni egy üzenetet, ez legyen pl. „Az állás 1:0”. Egy másik, konkrétabb példa a fentebbi: az egyes kontrollereknek szükségük van a megfelelő szolgáltatásokra.

Amennyiben DI-t használunk, úgy a függő objektum nem mondja meg konkrétan, hogy melyik komponenst szeretne használni, csak azt, hogy annak az objektumnak mit kell tudnia; ezt interfészekkel lehet megoldani. Azt, hogy a függő objektum milyen tényleges függést kap, egy külön konfigurációban tudjuk megadni. A példánkban a szervernek

[1] Controller, service, context
[2] Dependency Injection

szüksége van egy kiíró metódussal rendelkező komponensre, ez lehet akár csak egy konzol, amire írunk, de lehet akár egy adatbázisba való írás is.

Amit a DI használatával elérünk, az az, hogy az egymástól függő objektumok csak lazán függenek, és amennyiben egy függést cserélni kell (pl. konzolról adatbázisba írásra), nem kell a fő üzleti logikai réteget átírni, csak az ezt leíró konfigurációt módosítani, ezzel megfelelve az OOP és a SOLID elveknek.

Az automatikus injektálásra létezik többféle implementáció, könyvtár is, azonban az ASP-ben elérhető egy beépített is, mely tökéletesen megfelel számomra. Csupán a szükséges konfigurációt kell elvégezni, azaz meg kell adni az injektálható függéseket.

5.6. Kártyaszerkesztő

A fejlesztést a kártyákat kezelő komponens létrehozásával kezdtem, mivel ez az egész projektnek az alapja. Ez valójában egy CRUD szolgáltatás, azonban szeretném részletesebben is kifejteni, és ezen keresztül bemutatni a felhasznált eszközöket.

A szerveroldalon ehhez létrehoztam a kártyákat leíró modell osztályt, mely tartalmaz minden olyan tulajdonságot, amivel egy kártya azonosításához szükséges. Ezen tulajdonságok az alábbiak:

- Id – *string* típusú, teljesen egyedi azonosító, melyet a rendszer generál a kártya létrehozásakor
- Name – *string* típusú, a kártya neve
- Description - *string* típusú, a kártya leírása
- OwnerId – *string* típusú, a tulajdonos felhasználó azonosítója
- Type – *CardType enum* típusú, melynek két értéke lehet: egységkártya vagy speciális kártya
- Power – *int* típusú, a kártya erejét írja le
- IsHero – *bool* típusú, a kártya hős-e
- Row – *UnitRow enum* típusú, mely leírja, hogy melyik sorbéli a kártya: közelharci, lövész, vagy ágyú
- Ability – *UnitAbility enum* típusú, mely leírja, hogy az adott egységkártyának milyen képessége van
- Special – *CardSpeciali enum* típusú, mely leírja, hogy az adott speciális kártyának milyen effektje van
- PicturePath – *string* típusú, a kártyához tartozó, a felhasználó által feltölthető kép fájljának a neve
- CopyCount – *int* típusú. Egyes képességeknél (horda, testvér) ugyanabból a kártyából több is szükséges, hogy a képesség működőképes legyen. Ez az érték tárolja el, hogy adott kártyából hány másolatot szeretne a felhasználó.
- GroupId – *string* típusú, a fentebbi másolat funkcióhoz tartozó változó. Minden önálló kártya a saját maga csoportjába is tartozik, azonban az egyes kártyából

létrejött másolatok egy csoportba tartoznak, így könnyen lehet az összetartozókat azonosítani.
- IsCopy – *bool* típusú, a másolatfunkcióhoz tartozó utolsó változó. Mivel a másolatokat nem lehet direkt módon szerkeszteni, így a csoporton belüli eredeti kártyáról eltárolom eredeti mivoltát. Ennek a szerkesztése esetén lesznek a másolatok frissítve.

A modell osztály létrehozása során felhasználtam a nyelv által biztosított annotációkat ahol szükségesek, hasznosak voltak. Ilyen annotáció például a *Key*, mellyel az entitás azonosítóját lehet megjelölni, ami ebben az esetben az *Id*. Továbbá használtam még a *StringLength*-et, mellyel egy szöveges mező maximum hosszát lehet megadni, illetve a *Range*-t is, mellyel a számoknak lehet megadni egy érvényes intervallumot; utóbbit az a kártya életerejénél használtam (0-15), illetve a másolatok számánál (0-2). Ezeket az attribútumokat az adatbázis felhasználja, és nem fogja engedni a nem megengedett értékek eltárolását, ezzel is egy biztonságot szolgáltatva.

A modell létrehozása után az azokat kezelő szolgáltatást írtam meg, mely biztosítja az egyes kártyák létrehozását, módosítását, törlését, illetve az összes kártya lekérdezését.

Minden szolgáltatásbéli metódus ún. *tuple* visszatérési értékkel rendelkezik. Ez a funkció C# 7.0 óta érhető el, és csak annyit tesz lehetővé, hogy egy metódus ne csak egy, hanem több visszatérési értékkel rendelkezhet. Ezt arra használtam fel, hogy a szerveroldal ne csak a művelet eredményét adja vissza (pl. kártyák listája lekérdezésnél, módosított kártya adatai módosításnál), hanem minden eljárás eredménye mellé szolgáltasson egy **szerveroldali kódot**, melyeket én határozok meg; egy **HTTP kódot**, melyeket a webes világban elterjedt szokások határoznak meg; illetve egy szöveges üzenetet, mely a felhasználó számára jól megfogalmazott, értelmezhető. Ezen adatok továbbítására létrehoztam a *BackendResult* nevű osztályt, mely pontosan ezen adatok tárolására szolgál. Ebből az osztályból létre is hoztam néhány statikus példányt is, melyeket többször előforduló eseteknél használok fel (pl. sikeres művelet esetén 0-s kód, http 200 kód, „OK” üzenet).

Ez a funkció sikeres műveletvégzés esetén annyira nem érdekes, azonban amikor hibára fut a rendszer valamilyen okból kifolyólag, rendkívül hasznos lehet. Itt térnék át a másik dologra, amit minden egyes szolgáltatásbéli metódus elvégez, mielőtt a ténylegesen kért műveletet végrehajtaná: a bejövő paraméterek ellenőrzése, validálása. Minden paraméter átesik pl. *null* ellenőrzésen, annak érdekében, hogy a későbbi műveletvégrehajtás során ne fusson hibára a program. Ezután következnek az egy fokkal bonyolultabb ellenőrzések, pl. a megadott felhasználó/kártya azonosító valóban létezik-e az adatbázisban. Amennyiben ez is sikeres, további ellenőrzésképpen lefut még egy tulajdonos azonosítás is, ez leginkább módosításnál/törlésnél lényeges, mivel nem szeretném, hogy A játékos illetéktelenül módosítsa B játékos kártyáit.

Látható, hogy optimális esetben, megfelelő szolgáltatás hívásokkal ezek a hibák nem következhetnek be. Azonban esetleges programozási hibák vétésekor, vagy akár egy támadás esetén ezekkel az ellenőrzésekkel elkerülhetőek a hibák.

Itt szeretnék még szót említeni egy szintén fontos és elterjedt technológiáról, amit alkalmaztam a szolgáltatás során, ami pedig nem más mint az adatbázis tranzakciók. Vannak esetek, amikor egy művelet végrehajtásához több adatbázisbeli módosítás is szükséges, erre lehet jó példa egy olyan kártya módosítása, melyből léteznek másolatok is, melyek szintén önálló rekordok. Értelemszerűen az az elvárt működés, hogy amennyiben módosítunk egy ilyen kártyát, úgy a másolatai is megváltozzanak, azonban ha a folyamat során bárhol hiba lép fel, úgy mindegyik entitás őrizze meg az eredeti állapotát. Erre nyújt megoldást a tranzakció, mellyel elérhető, hogy több adatbázisbéli művelet is egy lépésben fusson le, ezáltal vagy minden módosul hibamentesen, vagy hiba esetén semmisem.

A szolgáltatás létrehozása után a végső szerveroldali lépés a kontroller létrehozása volt. A kontroller osztály feladatköre a különböző beérkező HTTP kérések kiszolgálása a megfelelő szolgáltatásbéli metódus meghívásával. A kontroller elején elhelyeztem az *Authorize* attribútumot, ezzel elérve, hogy a kontroller bármely metódusát csak akkor lehet elérni, ha a kérésben szerepelnek érvényes felhasználói adatok. Így minden metódus elején elvégezhető a hívó fél azonosítása, erre a beépített *UserManagert* használtam fel. Az így beazonosított felhasználó azonosítójával hívom tovább a megfelelő szolgáltatásmetódust, majd annak a korábban kifejtett eredményét küldöm vissza a hívó félnek JSON-ként.

Ezzel a szerveroldali fejlesztést megoldottam, így következhetett a kliensoldal implementálása.

Kliensoldalon a munkám megkönnyítése végett először létrehoztam egy generikus *ApiServicet*, mely a beépített *HttpClient* köré épül. Ez gyakorlatilag egy wrapper osztály, ami a szükséges HTTP GET, POST, PUT illetve DELETE hívásokat végzi el, és adja vissza eredményül a korábbiakban kifejtett *BackendResultot*, illetve a T paraméterül átadott elvárt eredményt. Ez a kis segédosztály jó arra, hogy ne kelljen minden egyes oldal esetén megírni ugyanazt az ismétlődő kódot, mely a szerver által nyújtott JSON válaszok deszerializálását végzi.

A konkrét kliensoldali *CardService* osztály így már egy viszonylag kevés kódot tartalmazó szolgáltatás, ami tényleg csak azokat a változókat foglalja magában, ami kifejezetten a kártyák kezeléséhez szükséges, mint például a kártya API végpontjának az elérési útvonala, illetve az azzal való kommunikációhoz szükséges DTO-k típusa; ezek segítségével hívja a fentebbi generikus *ApiServicet*.

Végső lépésként magát a megjelenő oldalt hoztam létre, ez már a tényleges *.razor* kiterjesztésű Blazor fájl.

Blazorhoz is rendelkezésre áll számos CSS keretrendszer, melyek rengeteg, a mindennapi életben gyakran használt, előre elkészített komponenst tartalmaznak (pl. fő elrendezés, valamilyen grid rendszer, táblázatok, gombok, stb.), melyek a fejlesztést teszik könnyebbé, átláthatóbbá. Azonban ezek használatát végül elvetettem, ennek két oka is volt: egyrészt a projekt nem fog sok oldalból állni, és azok is viszonylag eltérű felépítésűek, így nem sok lett volna az újra felhasználható komponens; másrészt pedig mivel egy kártyajátékról van szó, melynek egyedi nézetet szerettem volna létrehozni, így nem is feltétlen lett volna célravezető mások által előre elkészített komponenseket felhasználni.

Így a nézetet, az ahhoz tartozó HTML és CSS részeket teljes mértékben egyedileg készítettem el. JavaScript használatára természetesen nem volt szükség, mert a Blazor által nyújtott C# kóddal meg lehetett oldani minden olyan dinamikus oldalbéli változtatást, amire szükség volt.

A kártyaszerkesztő felület (és a többi oldal) a fő elrendezését a CSS által nyújtott *Flexboxokkal* oldottam meg. Ez egy viszonylag újabb CSS tulajdonság, 2018-ban jelent meg, és rendkívül megkönnyíti a tartalmak elrendezését a korábbi megoldásokhoz képest. Kifejezetten segíti a reszponzív oldalak létrehozását, melyre az elmúlt időkben egyre nagyobb a szükség az egyre jobban elterjedő okostelefonok, tabletek miatt, melyek képaránya jellemzően nagyon eltérő egy laptop vagy számítógép monitorétól, ezáltal más féle megjelenítést igényel. [28]

Mint említettem C# kóddal teljes mértékben meg lehet oldani az egyes oldalak manipulációját, erre néhány példa a kártyaszerkesztő felület esetén:

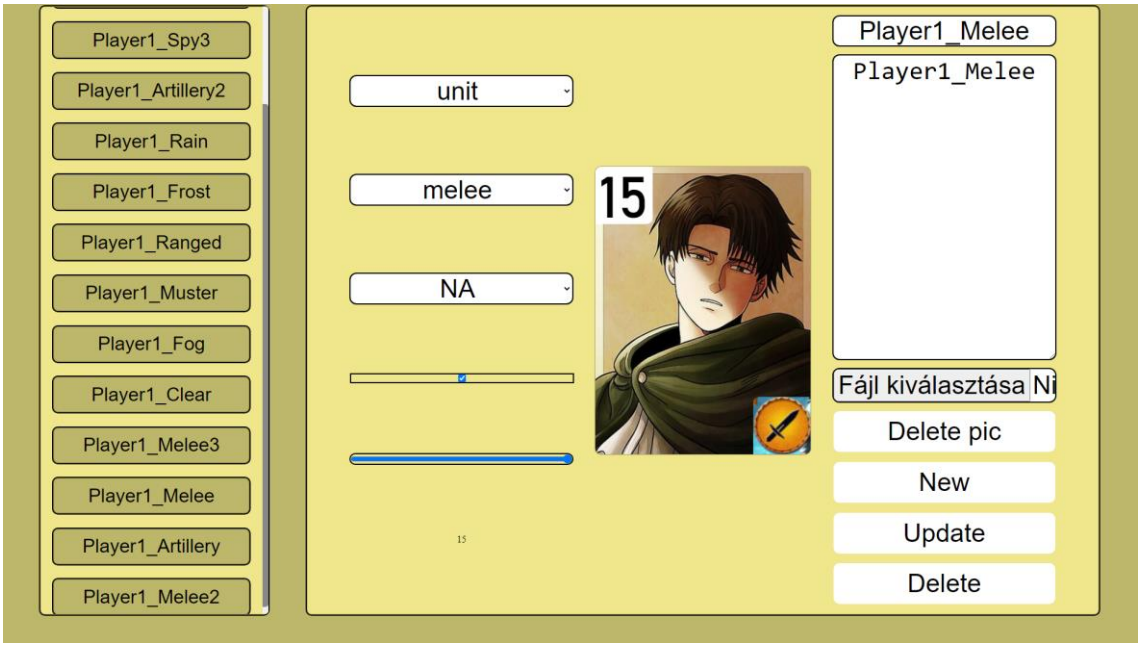
Magában a HTML tagek között elhelyezhetőek már jól ismert *if* elágazások, melyet például arra használtam fel, hogy a kártya típusától függően jelenítsem meg a beállítható paramétereket.

Szintén felhasználható pl. a *foreach* ciklus, melynek segítségével pl. a <*select*> HTML tagek <*option*> értékének adtam meg egy *enum* összes opcióját, illetve ezzel listáztam ki a már létező kártyákat is.

Gomboknál használtam az *@onclick* eseményt, így hívtam meg a fentebbi kártyaszolgáltatás metódusait (pl. kártya mentése).

Illetve az *input* mezőknél felhasználtam a *@bind* opciót, mellyel egy bemeneti mező értékét lehet adatkötni a modellosztály egy paraméteréhez.

Ezek segítségével készült el az 5.1. képen látható kártyaszerkesztő felület, mely a terveknek megfelelően néz ki:



5.1. ábra: Kártyaszerkesztő felület

(A „Fájl kiválasztása” gomb a böngésző / operációs rendszer nyelvén jelenik meg, ezért tér el a többi angol szövegtől.)

5.7. Pakliszerkesztő

Mivel a pakliszerkesztő mechanizmus gyakorlatilag megegyezik a kártyaszerkesztőnél látottéval, így csak a különbségeket szeretném kiemelni.

Paklik szerkesztéséhez két modell osztályt használtam fel, ezek az alábbiak:

- pakli modell
  - Id – *string* egyedi azonosító
  - Name – *string* pakli neve
  - OwnerId – *string* tulajdonos azonosítója
- kártya-pakli modell
  - CardId – *string* kártya azonosítója
  - DeckId – *string* pakli azonosítója

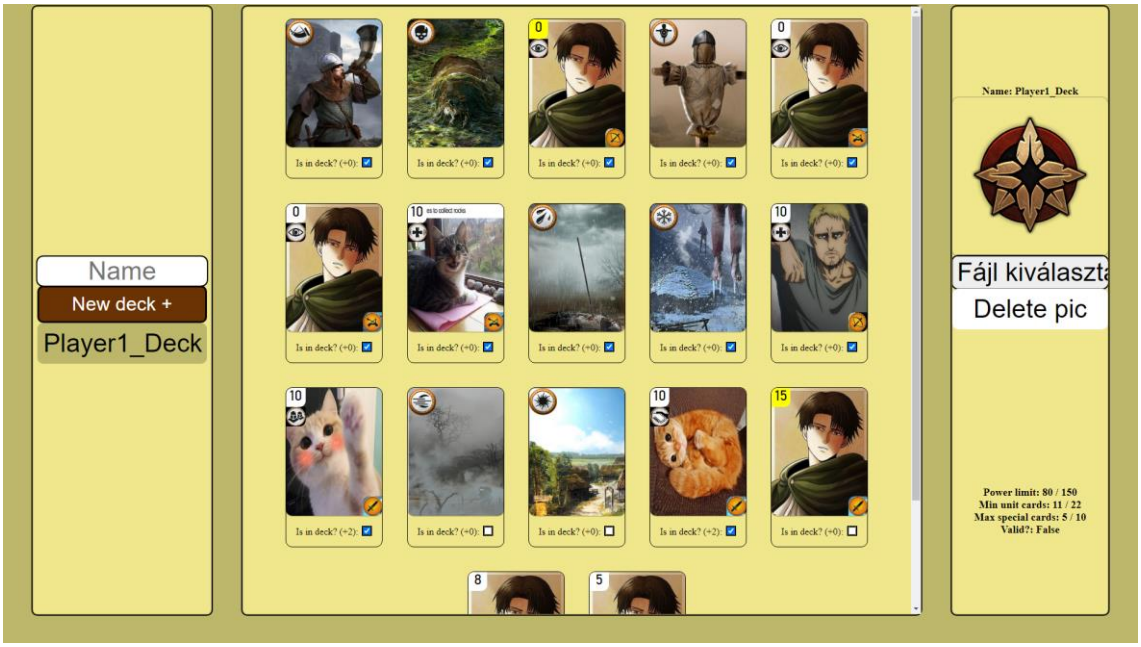
A paklikezelő szolgáltatás szintén végrehajtja a pakli létrehozás, módosítás, törlés és listázás funkciókat. Amivel még kibővül, az a paklihoz való hozzáadás illetve pakliból való eltávolítás funkciója. Ezek a többihez hasonlóan végrehajtják a bemenő paraméterek alapvető ellenőrzéseit, illetve azok sikeressége után ellenőrzöm, hogy hozzáadás esetén a pakliban megtalálható-e már az adott kártya, és csak akkor adom hozzá, ha még nem; eltávolítás során pedig értelemszerűen ennek a fordítottját vizsgálom, azaz, hogy az eltávolítandó kártya valóban megtalálható-e a pakliban.

Hozzáadás illetve eltávolítás esetén még megemlítendő, hogy azok a kártyák, melyekből léteznek másolatok, mindig csoportostul kerülnek kezelésre. A felhasználó feladata hogy

ezt figyelembe vegye, és az ő stratégiai döntése, hogy úgy hozza létre a kártyákat és paklikat, hogy azok érvényesek, de jól használhatóak legyenek.

A kontroller szintén hasonló a kártyaszerkesztőéhez, azonban rendelkezik a további két metódussal, melyek a szolgáltatásbeli hozzáadás/eltávolítást hívják meg, így ezek külön API végponton érhetőek el.

A megjelenítésnél sincs lényeges eltérés, szolgáltatásnál felhasználtam a korábbi generikus API-t, illetve elkészítettem az egyedi felületet, melyen el lehet készíteni a paklikat, ezt az 5.2. ábra mutatja:



5.2. ábra: Pakliszerkesző felület

5.8. Meccs

Ami minden meccsbéli interakcióról elmondható, hogy teljes mértékben kérés-válasz alapú kommunikáción alapul, ehhez használom fel a *SignalR*-t, melynek működését az 5.8.3. pontban fejtem ki.

Egy meccs egy váró létrehozásával kezdődik, ennek folyamata az alábbi: a felhasználó küld egy *kérést* a szerver felé, hogy szeretne egy újat készíteni. Ezt feldolgozza a szerveroldal, végrehajtja a megfelelő feladatokat, majd küld egy *választ* a hívó félnek, ami tartalmazza a létrejött szoba kódját is. Ez alapján a válasz alapján tudja a kliens, hogy frissítenie kell a megjelenítő felületet.

Egy váróba való csatlakozás, ahogy korábban a rendszertervnél ismertettem, egy eljárással valósul meg, mely egy szobakódot vár. Ezt a kódot kapja meg a szoba létrehozója automatikusan, illetve ezt a kódot kell közölnie a másik felhasználóval, akivel játszani szeretne. Ezt követően a felhasználó küld egy *kérést*, benne a szoba

azonosítójával, melybe csatlakozni szeretne. A szerveroldal ezt megkapva azonosítja a felhasználót, belépteti a tényleges váróba, majd küld egy *választ* az összes érintett kliensnek, hogy egy felhasználó csatlakozott.

A *SignalR* segítségével ezek a kérés-válasz párok gyorsan és egyszerűen implementálhatók. Amit még érdemes megemlíteni, hogy szintén könnyen megvalósítható az, hogy egy kérés során a két játékos eltérő választ kapjon. Ezt szintén egy, a projekt során írt példával tudom szemléltetni: amennyiben az „A” játékos le szeretne helyezni egy kártyát, ő erről egy kérés formájában értesíti a szervert, mely feldolgozza a kérést. Ezután „A”-t arról értesíti, hogy a kártya letétele sikeres, így „A” úgy frissíti a megjelenítést, hogy a saját térfelén lehelyezte a megfelelő kártyát. Innen következik, hogy a másik, „B” játékos egy kicsit eltérő értesítést kap: a másik játékos lehelyezett egy kártyát, így neki ennek megfelelően kell eljárnia, tehát az ellenfél térfelén kell megjelenítenie ugyanazt a kártyát.

5.8.1. Meccs modell

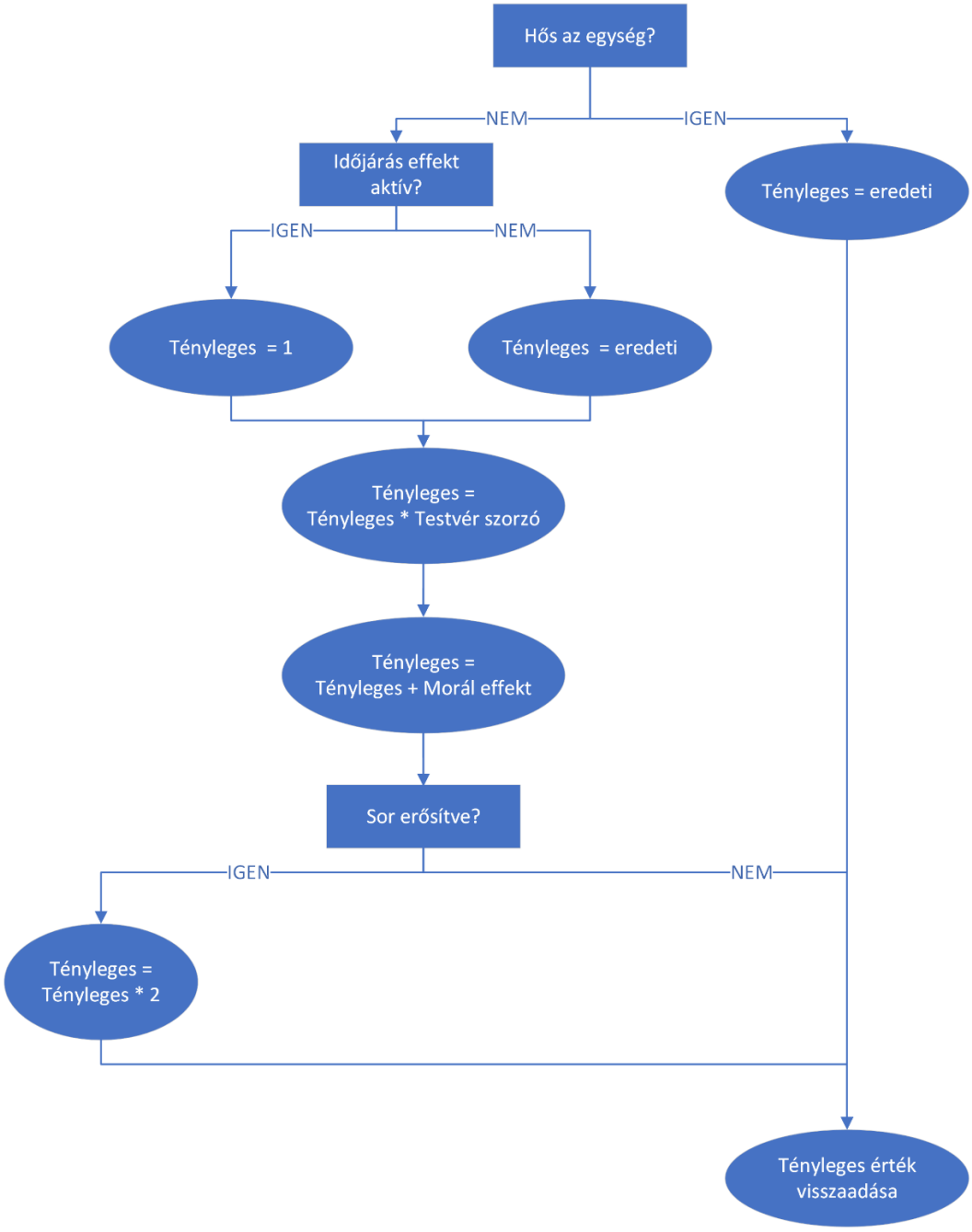
A meccsrendszer implementálását szintén egy modellosztály létrehozásával kezdtem, melyet a szerveroldalon fogok használni a tényleges logika megvalósításához. A korábbi osztályokkal ellentétben ennek az osztálynak az értékeit nem fogom adatbázisban tárolni, mivel egy-egy példány életciklusa csak az adott meccs végéig tart. Az osztály felépítése a következő:

- Id, *string* azonosító
- Players, *Players[]* tömb, mely a két játékost tárolja
- RoundCounter, *int*, a meccs köreit számolja
- LastStarter, *Player* típusú referencia, mely mindig az utolsó kört megkezdő játékosra mutat
- FrostActive, FogActive, RainActive, *bool* logikai változók, mely az megfelelő sorok időjárás állapotát jelzik;

A felhasznált *Player* osztály pedig az alábbiakat tartalmazza:

- NickName, *string* a játékosnév
- Deck, Hand, PlacedCards, Graveyard, *List<CardGameModel>* típusú gyűjtemények, melyek rendre a játékos még paklijában található, kezében levő, lehelyezett és temetőben levő kártyáit tárolják
- IsJoined, IsReady, IsRedrawDone, IsInMatch, HasLeftMatch, *bool* logikai változók, a játékos aktuális meccsbéli állapotát írják le
- IsMyTurn, HasPassed, *bool* logikai változók, a játékos körbéli állapotát írják le
- Health, *int* a játékos életereje;
- RedrawnCounter, *int* a meccs eleji újrahúzások számlálója
- MeleeBoosted, RangedBoosted, ArtilleryBoosted *bool* logikai változók, melyek a megfelelő sorok erősítési állapotát írják le

Az említett *CardGameModel* pedig az adatbázisban használt kártya modell osztályra hasonlít, azonban azt kibővíti néhány elemmel, mely a játéklogika megvalósítását segíti. Az egyes effektek, melyek a kártyákat érinthetik (pl. időjárás effekt, életerő duplázás, stb.) kártyánként vannak eltárolva, ezáltal könnyen is flexibilisen lehet állítani, hogy mely kártyáknak milyen állapotokat engedélyezünk. A kibővített modell egyik fontos tulajdonsága az *ActualPower*, mely a kártya tényleges, lehelyezett életerejét mutatja, értékének kiszámítását az 5.3. ábra mutatja:



5.3. ábra: Egység kártya tényleges erejének számítása

Amennyiben a kártya hős, úgy mindentől függetlenül az alapereje a tényleges is. Amennyiben nem az, úgy ki kell számítani. Ehhez el kell dönteni egy kiindulóértéket, mely 1, ha a kártyán aktív a rossz időjárás effekt, ellenkező esetben pedig a kártya alapereje. Ezt szorzom meg a Testvér képesség általi szorzóval, valamint adom hozzá a Morál képesség értékét. Ha a sor erősítve van, úgy ezt még 2-vel megszorzom, és az így kapott szám lesz a tényleges érték.

5.8.2. Meccs szolgáltatás, container

A felsorolt osztályok manipulálásával hoztam létre a meccs szolgáltatást, mely a meccsek kezeléséért felel. Ehhez azonban szükségem volt egy tárolóra, ahol az aktuálisan folyó mérkőzéseket tárolhatom el. Erre a célra hoztam létre a *MatchContainert*, melyet mint egy *Singleton* szolgáltatást konfiguráltam be az ASP-ben, és egyetlen funkciója egy lista tárolása, amiben a meccsek vannak.

Minden meccs szolgáltatásbéli metódus az alábbi sémát követi: először jön a tárolóban levő meccsek elérése, a megfelelő meccs kiválasztása: az azonosítás történhet a meccs kódja által is, azonban sok esetben a megadott felhasználó alapján veszem ki a meccset, mivel egyidejűleg egy felhasználó csak egy játékban lehet. Ezt követik a CRUD szolgáltatásoknál látottakhoz hasonló paraméter validálások, majd a játéklogikai ellenőrzések (pl. tényleg a hívó játékos következik-e, tényleg a kezében van-e a lerakni kívánt kártya). Ezek sikeressége esetén következik az adott játékbeli logikai művelet végrehajtása, majd a utolsó lépésként a megváltozott meccsmodell visszaadása két féleképpen: egy modell a hívó játékos számára, egy modell az ellenfél számára.

Az utolsó lépés teszi lehetővé az 5.8. pontban már említett értesítéseket mind a két fél számára. Abból a szempontból is előnyös minden eseménynél a teljes meccs modell visszaadása, mert így a megjelenítési oldalon sokkal egyszerűbb feladatom van: a kapott modell megjelenítése minden esetben. A kezdeti megvalósításomban minden egyes esemény külön visszatérési értékkel rendelkezett (pl. lerakott kártyánál csak a kártya maga, passzolásnál logikai érték, mely a sikert jelezte, stb.), melynél valamivel kevesebb adat került feleslegesen küldésre, azonban jóval megnehezítette és lassította a fejlesztést. A teljes modell leküldése esetén esetenként felesleges adatokat is küldök (nem változó elemek), azonban sokkal tisztább, átláthatóbb, kezelhetőbb kódot kapok.

A létrejött meccs szolgáltatás az alábbi funkciókkal rendelkezik:

- új játék létrehozásánál egy új meccsmodell példányosítása, és a tárolóban való elhelyezése.
- játékba való csatlakozás esetén a meccs azonosítása a megadott kód alapján. Amennyiben a játékosok közül senki sem csatlakozott, úgy az első játékos elhelyezése, ha egy valaki már jelen van, úgy a második csatlakoztatása.
- Pakli kiválasztása lépésnél a pakliszolgáltatás segítségével a megfelelő kártyák beállítása a játékosnak a meccsen belül.

- „Meccsre készen” gomb megnyomása esetén az *IsReady* logikai változó igazra állítása; ha mindkét játékos készen áll, a meccs automatikus indítása: kezdeti változók értékadása, véletlenszerű kártyaosztás a játékosoknak.
- A játékosok meccs eleji újrahúzásának lebonyolítása. Amennyiben az adott játékos mind a kétszer újrahúzott kártyát/nem kíván többet húzni, úgy az *IsRedrawnDone* logikai változó igazra állítása. Ha mindkét játékosnál megtörtént a művelet, a tényleges csata kezdete.
- Kártya lehelyezés esetén a kártya játékos kezéből való eltávolítása, a kijátszott kártyák közé való hozzáadása. Amennyiben a kártya egységkártya, a korábban aktivált effektek érvényesítése a kártyán. Amennyiben rendelkezik speciális effekttel, annak a többi kártyára gyakorolt hatásának kifejtése.
- Passzolás esetén az adott játékos *HasPassed* logikai változó igazra állítása. Amennyiben mindketten passzoltak, a kör kiértékelése, a lehelyezett kártyák temetőben való elhelyezése.

5.8.3. Meccs hub, SignalR

A SignalR használatához szükséges volt létrehozni egy osztályt, mely a beépített *Hub* osztály leszármazottja. A hubokat a kontrollerekhez lehetne hasonlítani, azzal a különbséggel, hogy valós időben működnek, illetve meg tudnak szólítani úgy klienseket, hogy azok előtte nem feltétlen fordultak hozzá kéréssel; utóbbira példa egy kártya lehelyezése: míg a játékos egyértelműen megszólította a hubot, hogy le szeretne rakni egy kártyát, addig az ellenfele nem csinált semmit, csak várt. A hub felelős azért, hogy a várakozó ellenfelet értesítse afelől, hogy új kártya került a táblára, és frissíteni szükséges a megjelenítést.

Az általam létrehozott *MatchHub* metódusainak létrehozásakor törekedtem arra, hogy csak ténylegesen azokat a feladatokat hajtsa végre, amelyeket neki szükséges; hoztam létre az 5.8.2. pontban leírt meccs szolgáltatást, melyet a *MatchHub* is használ. Így voltaképpen az összes hub-beli eljárás a következőképpen épül fel: először a hívó fél azonosítását hajtom végre, majd a meccs szolgáltatás megfelelő metódusát meghívom az azonosított felhasználó paraméterként való átadásával. Amennyiben a szolgáltatás valamilyen hibát jelez, a felhasználó értesítése; sikeres lefutás esetén pedig a visszakapott érték, értékek továbbítása a hívó játékos és ellenfelének egyaránt.

5.8.4. Meccs kliensoldal

A kliensoldalon levő megvalósítás a kinézet kialakításán kívül még az alábbiakban tér el a többi oldalétől:

Itt nem volt szükséges a generikus api szolgáltatás használata, mivel nem direkt API hívásokat használok; hanem a szerveroldalon kialakított SignalR hubhoz kellett csatlakozni. Ezt a célt szolgálja a kliensoldali *HubConnection* példány, illetve a hozzá tartozó, azt létrehozó *HubConnectionBuilder* osztály. Egy kapcsolat létrehozásához

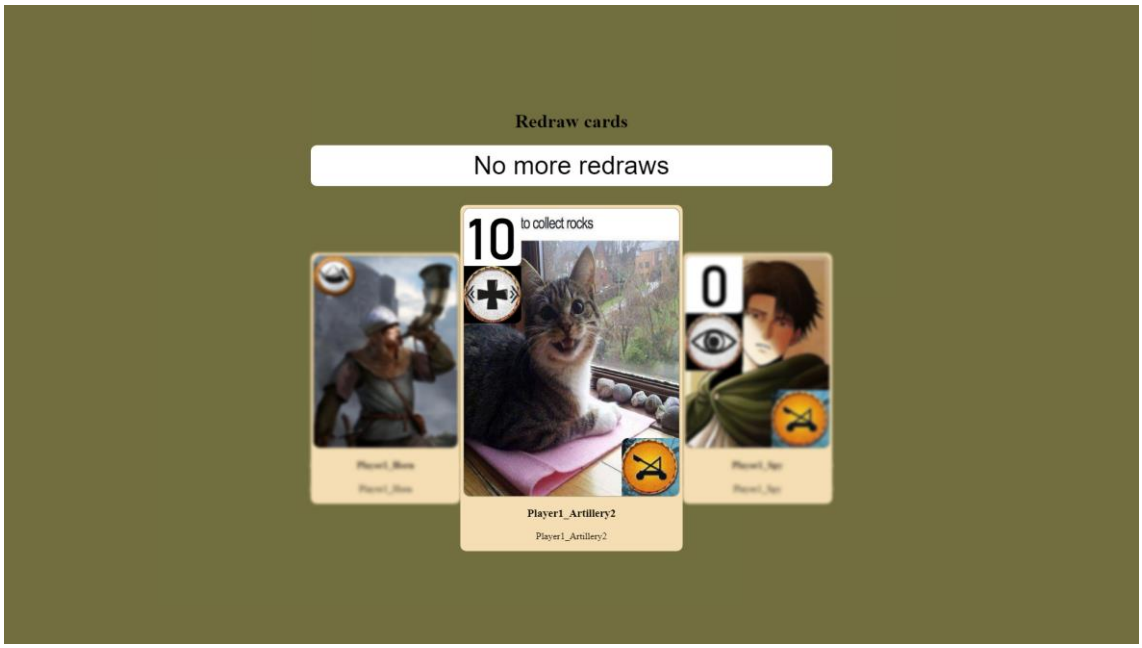
szükséges megadtam a végpontot, amin a Hub elérhető, az autentikációhoz szükséges tokent, valamint bekonfiguráltam, hogy az elvárt hub által küldött üzenetekre milyen kliensoldali kód fusson le. Ez utóbbira a legjobb példa a „Meccs frissítés” üzenet kezelése, melynek során csak szimplán a kapott modellt beállítom a régi helyett, és máris látható a meccs legfrissebb állapota. A kapcsolat indítása után már él is a hub, tud a kliens üzeneteket fogadni. Ezután már csak a másik, küldő irányú metódusok megírása volt a feladatom. Egy hub üzenet küldéséhez egy *string* típusú metódusazonosító küldése kötelező, mellette pedig kívánt típusú paraméterek passzolhatóak át. Ily módon hoztam létre a játékos által végrehajtható műveleteket meghívó kódokat. Mivel a *string* azonosítók szükségesek a hívásokhoz, illetve egy azonosítót esetleg több helyen is felhasználok (erre jó példa a „Kártya lerakás” azonosító, melyet a különböző típusú kártyánként máshogy használok), ezért ezeket kiszerveztem egy statikus *Globals* névre hallgató osztályra, a későbbi elírások elkerülése, esetleges módosítások megkönnyítése végett.

A megvalósult játékosfelületet az 5.4. és 5.5. képernyőképek prezentálják:

5.4. ábra: Meccs felület

Amiben még eltér a meccs oldal a többitől, hogy itt jónak láttam az úgynevezett sablonok használatát. Mint azt a meccs képernyőképen (5.4. ábra) is látható, egyszerre több kártya van a mezőn, és a kártyák is sorokba vannak rendezve. Ebből adódott, hogy szerettem volna minél jobban elkerülni a kódismétlést, ezért létrehoztam két paraméterezhető Blazor komponenst, melyek sablonként viselkednek: egy kártya és egy sor komponenst. A kártyakomponens leírja, hogy egy kártyát hogyan kell megjeleníteni (kép, képesség piktogram, életerő és annak színe értéktől függően, stb.), míg a sor komponens ezt a kártya komponenst felhasználva egy sor megjelenítését írja le. Azt, hogy mely konkrét

példányokat kell kirajzolni, paraméterekkel adom meg. Ezekből következik, hogy a fő meccs oldalon már csak a sorkomponenseket hívtam meg közvetlenül, ezzel egy letisztultabb kódot kapva.



5.5. ábra: Meccs újrahúzás felület

5.8.5. Egyjátékos mód

Az egyjátékos mód megvalósítása során két főbb problémakört kellett megoldanom: egyik a technikai megvalósítás kérdése, a másik pedig a játéklogika implementálása. Utóbbiról részletesen a 4.4.1. illetve 4.4.2. fejezetben írtam.

Technikai oldalról kritérium, hogy a gép általi játékos (bot) magától tudjon lépéseket végrehajtani, bármilyen esetben: akkor is, amikor a valós játékos egy lépésére kell reagálnia (pl. a játékos után a bot következik, és neki kell kártyát leraknia), és akkor is, amikor a botnak kell kezdeményeznie egy lépést (pl. ő kezdi a kört, mert ő nyerte az előzőt). Fontos kikötés volt még magammal szemben, hogy semmiképp sem szerettem volna már létező játékmechanizmusokat, logikát újraírni, ezzel kódismétlést okozva, hanem valamilyen formában felhasználni a már többjátékos módhoz létező kódot.

Technikai oldalról ezeket a kritériumokat sikerült is megvalósítani. A tényleges játékhoz tartozó szolgáltatásból, illetve a hozzá tartozó SignalR Hub-ból készítettem leszármazottakat, és egészítettem ki a szükséges lépésekkel. A hub leszármazottja az ős metódusait azzal egészíti ki, hogy a valós játékos lépése után automatikusan meghívja a megfelelő játékmechanikai funkciót a bot, amely megtörténte után visszakerül az irányítás a valós játékoshoz. A bot „gondolkodásának” szimulálásához egy másfél másodperces késleltetést használtam fel, hogy ne azonnal történjenek a reakciók. Így elértem azt, hogy a játékmenet teljesen ugyanúgy zajlik, mint a többjátékos mód esetén,

azzal a különbséggel, hogy a második játékost a rendszer vezérli. Itt jegyezném meg, hogy ehhez az is szükséges volt, hogy a bot játékos egy ugyanolyan entitás, mint bármelyik másik játékos, csak az ő lépéseit a gép hívja meg, a játékos számára transzparensen.

A meccs szolgáltatás leszármazottjában került implementálásra a bot által használt logika, amely a korábban kifejtett tervek alapján lett elkészítve. Amit itt még technikai oldalról megemlítenék, az a bot által használt módszer, mellyel a számára legjobban megérő kártyát határozza meg, mely az alábbi módon történik: ahhoz, hogy a bot számszerűen tudja, hogy melyik kártyával jut a legelőrébb, ahhoz ki kell tudnia számolni azt, hogy a kártya letétele esetén mennyi ponttal rendelkezne. Azonban egy kártya letétele több másik kártya értékét is módosíthatja, illetve a lehelyezett kártya aktuális pontja is függ az aktuálisan kijátszottaktól, ezért a háttérben elmentem a meccs jelenlegi állapotát, a bot kipróbálja az összes lehetőséget, és elmenti, hogy melyikkel mennyi pontra tudott szert tenni; ezután visszaállítom az eredeti állapotot a mentésből, a bot pedig kijátssza a legjobban megérő kártyát, és folytatódik a kör.

5.9. Eredmények, statisztika

Az eredményeket körönként tárolom el, így jól mutatva hogy egy meccs hogyan alakult.

Egy eredmény az alábbi mezőkből áll:

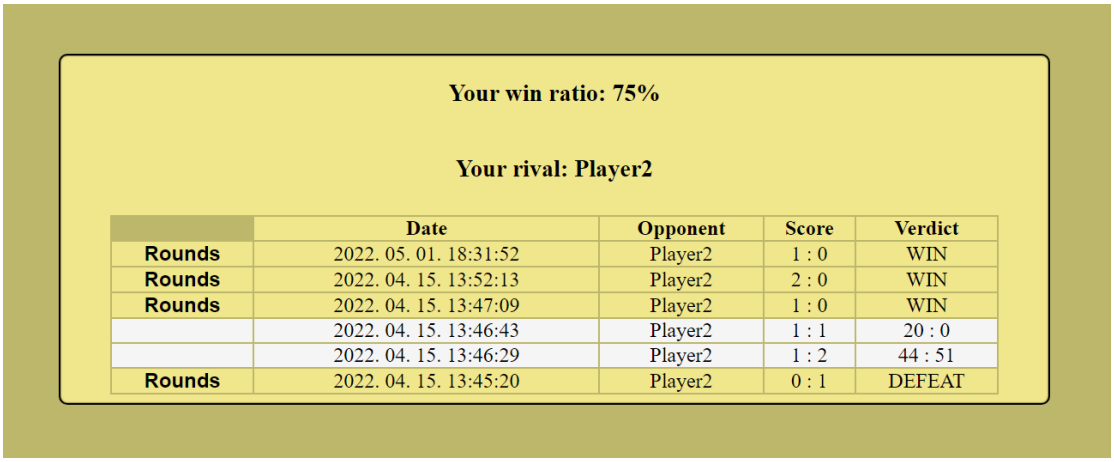
- Id, *string* azonosító
- MatchId, *string* a meccs azonosítója
- MatchOver, *bool* azt mutatja, hogy az adott meccs utolsó köre volt-e
- Date, *DateTime* a rögzítés dátuma
- Player1 és Player2, *MatchResultPlayerInfo* beágyazott osztály típusúak, mely az alábbiakat tartalmazza:
    - PlayerId, *string* játékos azonosító
    - Health, *int* játékos életereje
    - Points, *int* játékos pontjai

A *MatchResultPlayerInfo* osztályt elláttam az *Owned* attribútummal, ami azt eredményezi, hogy míg a kódban külön osztályként van jelen egy kör rekordjában, azonban az adatbázisban az adatok nem fognak külön táblába kerülni, hanem a többi adattal együtt, egy külön oszlopban lesznek eltárolva.

Az eredmények mentése a felhasználó számára transzparensen történik. A meccs szolgáltatás feliratkozik a meccs által kiváltott kör végét jelző eseményre, és meghívja a *MatchHistory* szolgáltatást, mely a szokásos ellenőrzések után végrehajtja az adatbázisba való írást. A szolgáltatás további három metódussal rendelkezik: egy a felhasználó korábbi eredményeinek a visszaadását hajtja végre, egy a felhasználó legnagyobb riválisait adja vissza, egy pedig a nyerési rátát számítja, melyet az alábbi képlettel számol ki:

$$NyerésiRáta = \frac{2 \times győzelmek\ száma + döntetlenek\ száma}{2 \times összes\ meccs\ száma}$$

Az ehhez tartozó kontrolleren keresztül ez utóbbi három funkciót lehet elérni, és ezt jelenítem meg táblázat formájában, melyben a meccs többi körét lenyitva lehet megtekinteni; ezt mutatja az 5.6. ábra:



5.6. ábra: Eredmények felület

6. TESZTELÉS

6.1. Automatikus

Automatikus tesztelés során elsődlegesen a különböző szolgáltatásokat szerettem volna tesztelni, mivel nagyrészt ezek valósítják meg a tényleges logikát, legyen szó a játékmenetről, vagy egyszerű adatbázisműveletekről.

Ezen tesztek közül elsődlegesen az egységtesztekre[1] fektettem nagy hangsúlyt, melyek jellemzője, hogy egy adott osztály (szolgáltatás) egyetlen metódusának egyetlen elvárt működését teszteli, legyen az egy „műveletvégrehajtás sikeres” üzenet visszaadása, vagy egy kivétel dobása. Egységtesztelni gyakorlatilag bármit lehet; ami fontos ezen tesztek során, az a függőségek feloldása, hogy ténylegesen csak az adott osztály helyességét értékeljük, az általa használt szolgáltatások ne befolyásolják az eredményt.

Ilyen függőség feloldás pl. a CRUD szolgáltatások esetén a használt adatbáziskontextus helyettesítése, mivel nem szeretném a valós MSSQL adatbázist minden egyes tesztfuttatásnál feltölteni tesztadatokkal. A másik fontos ok amiért ezt meg kell oldani, az a tesztek megismételhetősége. Azt szeretném elérni, hogy minden egyes egységteszt egymástól teljesen függetlenül futtatható legyen, ehhez szükséges az, hogy mindegyik teszt egy egységes állapotból tudjon elindulni.

Az adatbáziskontextus helyettesítésére először egy korábban tanult, és egyébként is széleskörben elterjedt technikát akartam alkalmazni, a *mock*olást. Mockolás esetén egy lebutított, hamis példányt adunk át a szolgáltatásnak, mely csak a helyben bekonfigurált műveleteket tudja végrehajtani, értékeket visszaadni. Jól látható, hogy ez optimális lett volna számomra, hiszen bekonfiguráltam volna, hogy a mock adjon vissza pl. egy darab teszt felhasználót, három darab kártyát és két paklit, mindezt úgy, mintha egy rendes adatbázis lenne; a szolgáltatást elvégre nem érdekli, hogy honnan jönnek az adatok, csak legyenek meg. Azonban sajnos mockoláshoz szükséges a helyettesítendő objektumnak paramétermentes konstruktorral rendelkeznie, ami jelenleg nem áll rendelkezésemre, mivel a használt adatbáziskontextus ősosztályának nincs ilyen konstruktora.

Némi kutatás után találtam egy megfelelő megoldást a problémámra, ez pedig a memóriabéli ideiglenes adatbázis[2], abból is egy SQLite verzió. Az ilyen típusú adatbázisokról azt kell tudni, hogy a tesztek indulásakor elindulnak és léteznek a memóriában, azok végeztekor pedig törlődnek onnan; semmilyen valós fájl nem jön létre a háttértáron. Ennek az is az egyik előnye, hogy rendkívül gyorsak, nem kell operációs rendszer szolgáltatásokat indítgatni, ez kifejezetten elősegíti a gyors tesztfuttatásokat. Még megemlíteném, hogy azért esett a választásom az SQLite verzióra az alapvetőbb Microsoft-os beépített helyett, mivel utóbbi nem támogatja az adatbázis tranzakciók

[1] Unit test
[2] In-memory database

használatát, ami számomra komoly probléma, mivel fontos részét képezik a műveleteimnek.

Itt szeretném kiemelni, hogy ez egy mintapélda arra, hogy jó döntés volt az Entity Framework használata, mivel a fentebb részletezett adatbázisra való átállás rendkívül egyszerű volt, csak a megfelelő NuGet csomagot kellett telepítenem, és egy pársoros konfigurációt elvégeznem a tesztprojekt indításakor, ezzel előállítva a megfelelő állapotot. Ezen kívül a tesztek futtatásához még szükséges volt a tesztadatok előállítása, melyeket minden egyes teszt futtatása előtt elhelyeztem[1] az adatbázisban, ezzel elérve a tesztesetenkénti azonos kiinduló állapotot.

A CRUD szolgáltatások tesztelése után hasonló módon a meccs szolgáltatást teszteltem. Itt hasonló módon jártam el, nem ütköztem problémába; annyi a különbség, hogy a tesztesetek megírásakor értelemszerűen itt nem adatok létrehozását, módosítását imitáltam, hanem egy vagy két játékos lépéseit, programkód formátumban.

A megírt tesztek egyrészt segítségemre voltak abban, hogy megbizonyosodjak a program aktuális állapotának helyes működéséről, másrészt pedig nagyon hasznosak voltak olyan esetekben, mikor már létező kódon hajtottam végre módosítást, ezzel akár minimálisan is megváltoztatva annak működését. Ebben az esetben ugyanis bár a tesztek „pirosak lettek” – hibát jeleztek, ezek nem tényleges hibák voltak, hanem visszajelzések felém, hogy a korábbi működés megváltozott. Ezt fel tudtam használni arra például, hogy egy szerveroldali változtatásnak mindenképp implementáljam a kliensoldali változatát is, semmiképp se felejtsem el azt.

6.2. Manuális

Az automata tesztelés mellett természetesen végeztem manuálisakat is, melyek főként a böngészőből történtek; ebből kifolyólag ezek hívhatóak teljes rendszertesztnek, mivel a tényleges felhasználói felületen keresztül ellenőriztem, hogy az elvárt funkciók működnek-e.

Természetesen itt is lehetett volna valamilyen front-end tesztelő keretrendszert felhasználni (pl. Selenium), azonban mivel kevés oldal van, egyszerűbb és gyorsabb volt kézzel végig nézni a rendszert különböző böngészőkből. A program minden elterjedt böngészőben (Google Chrome, Microsoft Edge, Mozilla Firefox, Apple Safari) helyesen működött.

Mint azt az 1.1.2. fejezetben is említettem, a WebAssembly technológiát a legtöbb böngésző ma már támogatja, ezt támasztja alá a sikeres tesztelés is. Szintén megbizonyosodtam arról is, hogy a viszonylag újabb CSS flexbox is működik böngészőtől függetlenül.

[1] seed

7. TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

A játék további funkciójaként elsősorban a kártyarepertoár kibővítésére törekednék. Az eredeti játékból kimaradt néhány funkció, azok implementálása a többiéhez hasonlóan könnyen megvalósítható (további speciális kártyák, vezér képességek). Valamint megfelelő kreativitás esetén kitalálhatóak teljesen új képességek is, csak át kell gondolni, hogy miféleképpen lehet a játék logikájába balanszosan beilleszteni.

További lehetőség lenne egy technológiai fejlesztés, az állapot kezelése[1] futó meccs esetén. A jelenlegi verzióban, amennyiben meccs közben lefrissíti a játékos a böngésző oldalát, úgy az teljesen elveszíti a tartalmát, és a meccs állapotát. Ennek kiküszöbölésére több megoldást is el tudok képzelni, mind szerver, mind kliens oldalon, a SignalR segítségével. Erre egy példa lehet egy időtúllépés[2] bevezetése: amennyiben a SignalR elveszti a kapcsolatot az egyik játékossal, csak egy bizonyos idő elteltével kezelni kilépettnek, és értesítené a másik játékost a meccs végéről. A limiten belül a játékos vissza tudna csatlakozni a meccsbe és folytatni azt.

Az egyjátékos módot is lehetne különböző formátumokban továbbfejleszteni. Legegyszerűbben további pakli-logika párosok elkészítését lehetne implementálni a jelenlegi megoldáshoz hasonlóan, melyek közül a bot induláskor tudna választani véletlenszerűen. Bár ennek a komplexitása és reagálási képessége azonos szinten lenne a jelenlegi megoldáséval, valamilyen szintén azért növelni a bot változatosságát. Eggyel komolyabb szint lehetne valamilyen súlyozás bevezetése. A bot jelenlegi lépéseit ki lehetne egészíteni oly módon, hogy minden egyes műveletet kiértékel valamilyen képlet segítségével, hogy az mennyivel helyezte őt előnyösebb helyzetbe. Ezeket az értékeket minden meccs minden lépésénél eltárolja, és ezeket a jövőbeli meccseknél felhasználja.

A legkomolyan egyjátékos továbbfejlesztéshez szükséges lenne az alkalmazás éles környezetbe való kitelepítése, és viszonylag nagyobb játékosszám elérése. Ebben az esetben a valós játékosok minden egyes lépését lehetne menteni, és elég adat összegyűlése esetén el lehetne gondolkodni valamilyen mélytanulásos[3] rendszer bevezetésén, mely már nem csak egy előre elkészített paklit használna, hanem a paklit is ő maga tudná létrehozni, a korábbi tapasztalatok alapján; elvégre a létrehozható kártyák száma véges, csak a létrehozott opciókból lehet választani.

[1] state management
[2] timeout
[3] deep learning

8. KONKLÚZIÓ

A szakdolgozat során létrejött szoftver teljesíti az elvárt követelményeket, valamint az általam szükségesnek tartott funkciókat. A program jól elkülönült komponensekre lett osztva, melyek közt laza csatolás jött létre, így könnyen módosítható igény esetén. A létrejött felület lényegre törő, a menürendszerben csak a szükséges menüpontok jelennek meg, navigációra a böngésző által nyújtott funkciók jól használhatóak. A játékmenet közben az adatok gyorsan, valós időben jelennek meg. A felhasználókezelést a beépített felhasználókezelő megfelelően ellátja, a további adatok kezelését a megírt szolgáltatások az elvártaknak megfelelően végzik.

A megvalósítás során sok új ismeretre tettem szert. Megismertem az ASP.NET Core által nyújtott lehetőségeket, különös hangsúlyt fektetve a valós idejű hálózati kommunikációra a SignalR segítségével. Kliensoldal terén volt lehetőségem a Blazorban való elmélyülésben, mely az elkövetkező években rendkívül hasznos és releváns lehet a webfejlesztés világában. Az egyjátékos mód implementálása jó lehetőséget biztosított arra, hogy megtudjam milyen lehet egy valós életbeli folyamatot (a játékmenetet) programként modellezni, rávilágított az ezekkel felmerülő kérdésekre és komplexitásra, illetve felvetett lehetőségeket melyekkel még kifinomultabban is meg lehet oldani ezt a problémát.

A tesztelési fázis során megtanultam a korábbiakban tanult módszereket egy valós projekten is alkalmazni, valamint egyes esetekben rávilágítottak szoftverfejlesztési hibákra, melyekből szintén sokat fejlődtem (ez jellemzően túl komplex osztályok kiszűrését jelentette).

ÖSSZEGZÉS

A szakdolgozatom célja egy olyan webes játék létrehozása volt, melyben a játékosoknak lehetőségük van egyedi kártyákat, és azokból paklikat létrehozni, majd azokat egymás elleni mérkőzésekre felhasználni. A kártyák személyre szabhatóságát itt a teljesen egyedi név, leírás és képfeltöltés jelenti, illetve a kártyák képességeinek beállíthatóságát, melyet a játékos előre elkészített opciókból, játékfunkciókból válogathat össze.

A dolgozat első részeiben a projekt alapjairól írtam. Utánanéztem milyen technológiák jöhetnek szóba a megvalósítás érdekében, valamint hasonló rendszerek után kutattam tapasztalatgyűjtés szempontjából. Zárásképpen az általam választott technológiákról írtam.

A következő fejezetekben leírtam a szoftverrel szembeni elvárásokat, valamint megterveztem a fő létrehozandó komponenseket, azok működését. Ismertettem a kártyák, paklik képességeit, a játékmenet fő mechanikáját. Szintén leírtam az egyjátékos mód követelményeit.

A dokumentum második fele az implementációról szól. A létrejött program szerveroldali része ASP.NET Core alapú. A kártyák, paklik, eredmények kezelésére REST API-t hoztam létre megfelelő háttérszolgáltatásokkal, a játékosok közti meccsek lebonyolítására pedig a SignalR-t használtam fel, mint valós idejű kommunikációs eszközt. A kliensoldali webes megjelenítésért a Blazor felel, mely egy előremutató technológia, és a jövőben remélhetőleg egyre relevánsabb lesz. A létrejött program egyedi, letisztult kezelőfelülettel rendelkezik, valamint nagy hangsúlyt fektettem a különböző ellenőrzésekre, a program helyes működése érdekében.

A dolgozat zárásaként a tesztelési eredményekről, valamint az általam elképzelt továbbfejlesztési lehetőségekről ejtettem néhány szót.

SUMMARY

The goal of my thesis was creating a web based multiplayer card game, in which the players have the ability to create customized cards, which they can use to create decks in order to play with them against each other. The customization includes giving the cards unique names and descriptions, and uploading images. The players can choose from preset abilities, which will determine how the card can be used in a match.

In the first part of my thesis I wrote about the basis of the project. I looked up technologies which might be useful for the implementations, and I've searched for similar software in order to get some experience about them. As a conclusion I wrote about the selected technologies.

In the following chapters I wrote about the software requirements, and created plans about the main components which are to be implemented. I described the functions of the cards and decks, the main mechanics of the game. I also specified the singleplayer mode's plan.

The second half of the document contains the process of the development. The implemented software's backend is based in ASP.NET Core. The CRUD methods of the cards, decks and match results can be accessed through REST APIs, while the real-time matches between players use SignalR. The front end browser framework is Blazor, which is a promising technology, and it's possibly becoming more and more relevant in the future. The software that was created has an easily usable, unique interface, and I've put great emphasis on thorough validations of user inputs, in order to enforce the correct operation of the program.

As a closing I wrote briefly about the testing and the possible improvements of the project.

IRODALOMJEGYZÉK

[1] [Online]. Elérhető: https://webassembly.org/. [Hozzáférés dátuma: 12 11 2020].

[2] [Online]. Elérhető: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor. [Hozzáférés dátuma: 12 11 2020].

[3] [Online]. Elérhető: https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0. [Hozzáférés dátuma: 12 11 2020].

[4] [Online]. Elérhető: https://unity.com/. [Hozzáférés dátuma: 12 11 2020].

[5] [Online]. Elérhető: https://github.com/nodejs. [Hozzáférés dátuma: 28 10 2020].

[6] [Online]. Elérhető: https://nodejs.org/en/about/. [Hozzáférés dátuma: 28 10 2020].

[7] [Online]. Elérhető: https://github.com/django. [Hozzáférés dátuma: 28 10 2020].

[8] [Online]. Elérhető: https://dotnet.microsoft.com/apps/aspnet. [Hozzáférés dátuma: 12 11 2020].

[9] [Online]. Elérhető: https://github.com/dotnet/aspnetcore. [Hozzáférés dátuma: 12 11 2020].

[10] [Online]. Elérhető: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core. [Hozzáférés dátuma: 12 11 2020].

[11] [Online]. Elérhető: https://tools.ietf.org/html/rfc6455. [Hozzáférés dátuma: 26 10 2020].

[12] [Online]. Elérhető: https://www.telerik.com/blogs/real-time-communication-techniques. [Hozzáférés dátuma: 26 10 2020].

[13] [Online]. Elérhető: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/websockets?view=aspnetcore-3.1. [Hozzáférés dátuma: 26 10 2020].

[14] [Online]. Elérhető: https://github.com/sta/websocket-sharp. [Hozzáférés dátuma: 26 10 2020].

[15] [Online]. Elérhető: https://grpc.io/about/. [Hozzáférés dátuma: 12 11 2020].

[16] [Online]. Elérhető: https://github.com/grpc/grpc. [Hozzáférés dátuma: 12 11 2020].

[17] [Online]. Elérhető: https://dotnet.microsoft.com/apps/aspnet/signalr. [Hozzáférés dátuma: 12 11 2020].

[18] J. M. Aguilar, SignalR Programming in Microsoft ASP.NET, Microsoft Press, 2014.

[19] [Online]. Elérhető: https://github.com/dotnet/aspnetcore/tree/master/src/SignalR. [Hozzáférés dátuma: 12 11 2020].

[20] [Online]. Elérhető: https://github.com/mysql/mysql-server. [Hozzáférés dátuma: 24 11 2020].

[21] [Online]. Elérhető: https://git.postgresql.org/gitweb/?p=postgresql.git. [Hozzáférés dátuma: 24 11 2020].

[22] [Online]. Elérhető: https://www.microsoft.com/en-us/sql-server/sql-server-2019. [Hozzáférés dátuma: 24 11 2020].

[23] [Online]. Elérhető: https://playhearthstone.com/en-us. [Hozzáférés dátuma: 27 11 2020].

[24] [Online]. Elérhető: https://magic.wizards.com/. [Hozzáférés dátuma: 27 11 2020].

[25] [Online]. Elérhető: https://playingcards.io/. [Hozzáférés dátuma: 27 11 2020].

[26] [Online]. Elérhető: https://www.cardgamesimulator.com/. [Hozzáférés dátuma: 27 11 2020].

[27] [Online]. Elérhető: https://dulst.com/. [Hozzáférés dátuma: 27 11 2020].

ÁBRAJEGYZÉK