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

Markovics Gergely

KÁRTYAJÁTÉK MEGVALÓSÍTÁSA MIKROSZOLGÁLTATÁS ALAPOKON

KONZULENS

Dr. Kővári Bence András

BUDAPEST, 2023

Tartalom

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2023. 12. 12.

...…….

Markovics Gergely

Összefoglaló

A kártyajátékok régóta nyújtanak szórakoztatást és kikapcsolódást világszerte. Legyen az családi vagy baráti környezetben, könnyed közösségi élményt biztosítanak minden korosztály számára. Ezen játékok közül a Sushi Go egy komplex, változatos és egyedi játékélményt ad az összegyűlt társaságnak.

A kártyajátékok egyik hátránya, hogy fizikai jelenlétet igényelnek a játékosoktól. Legyen akármilyen élvezetes a játék, a játékosoknak össze kell gyűlniük a használatához, ami a mai világban gyakran nehezen megoldható. Ennek a problémának az áthidalására szolgál a játék szoftveres megvalósítása, amivel a játékosok globálisan, akárhonnan tudják élvezni a játék adta lehetőségeket.

Az alkalmazás implementációja webes, böngészőben játszható formában készül el, aminek a segítségével a játékosok akárhonnan, platformtól függetlenül csatlakozhatnak be a játékba, élvezhetik annak szolgáltatásait. Azért, hogy igazi közösségi felhasználói élményt nyújtson az alkalmazás, a fejlesztés során implementálva lettek további játékon túlmutató funkciók is, mint barátok kezelése, vásárlás, váróterem kezelés vagy szöveges beszélgetés nyújtása.

A szoftver implementációjában kiemelt szerepet kapott, hogy a szerveroldali komponens mikroszolgáltatás alapokon valósuljon meg, annak lehetőségeit aktívan és hatékonyan kihasználva.

A feladat tehát egy ismert kártyajátékot implementáló szoftver megvalósítása webes klienssel és mikroszolgáltatásokra épülő szerverrel, figyelmet fordítva a felhasználói igényekre és élményre.

Abstract

Card games have long provided entertainment and relaxation around the world. Whether with family or friends, they provide a light social experience for all ages. Of these games, Sushi Go provides a complex, varied and unique gaming experience for the assembled group of people.

One of the drawbacks of card games is that they require the physical presence of the players. No matter how enjoyable the game is, the players must come together to use it, which is often difficult in today's world. This problem is overcome by the software implementation of the game, which allows players to enjoy the game’s features from anywhere globally.

The implementation of the application will be in a web and browser-based format, allowing players to connect to the game and enjoy its services from anywhere, regardless of the platform. In order to provide a true social user experience, additional features beyond the game, such as managing friends, shopping, lobby management or chatting, were implemented during the development.

In the implementation of the software, it was a priority to implement the server-side component on a microservice basis, actively and efficiently utilising its potential.

The task is therefore to create a software implementing a well-known card game with a web client and a server based on microservices, paying attention to the needs and experience of the users.

1 Bevezetés

1.1 Böngészős- és társasjátékok

Társasjátékok évszázadok óta nyújtanak szórakoztatást és kikapcsolódást családoknak és baráti társaságoknak. Ezek az interaktív, változatos játékok nem csak szórakoztatást nyújtanak, de különböző készségeket is fejlesztenek. Ilyen például a kommunikáció, vagy stratégiai és logikai gondolkodás.

Fizikai társasjátékokkal ellentétben a böngészős játékok csak az internet korával jelentek meg, és azóta rohamosan nő a népszerűségük. A folyamatosan frissülő és fejlődő felület sok figyelmet és érdeklődést vonz maga után, legyen az több- vagy egyszemélyes játék. Ezen játékokhoz a hozzáférés általában teljesen vagy részlegesen ingyenes, így szélesebb közösséget tudnak kialakítani.

Böngészős játékok további előnyei, hogy könnyű hozzáférést biztosítanak, mind a használatuk és időigényük rugalmas, emellett könnyed közösségi interakcióra is adnak lehetőséget. Kényelmes a használatuk platformtól függetlenül, legyen az komplett asztali számítógép, laptop, vagy kisebb mobil vagy tablet készülék. Ezek lehetőséget biztosítanak arra, hogy igényektől függően bárhol és bármikor élvezhessük a játékot.

A dolgozat témájában a megvalósítás során a kettőnek a közös pozitív tulajdonságaira összpontosítottam. Tehát egy könnyen elérhető és közösségi élményt nyújtó, de emellett változatos játék megvalósítása.

1.2 Motiváció

Ez a diplomaterv a Sushi Go Party! nevű változatos kártyajátékot hivatott megvalósítani webes felületen. Az interneten keresztül való használatával nagyobb célközönség felé tud terjeszkedni, több ember ismerheti meg a játékot.

A szoftver elkészítésének egyik alap motivációja, hogy egy teljeskörű szoftvert akartam létrehozni, ami egyszerre tartalmaz szerveroldali logikát, és kliensoldali dizájn megvalósítást is. Ehhez a feladathoz egy kártyajáték implementációját gondoltam megfelelőnek, mivel azontúl, hogy könnyen megfogalmazható milyen végterméket szeretnék megvalósítani, sokféle kreatív bővítési lehetőséget biztosít magában.

A szoftverhez használt kártyajátéknak a „Sushi Go Party” -t választottam, mivel egyedi és izgalmas játékelemeket tartalmaz, ami a játékosoktól a játékhoz illő egyedi stratégiákat vár el. Emellett nem csak maguk a lapok és használatuk egyedi, de maga a pakli is sokféleképpen változtatható a játékosok idejétől, számától, vagy csak kedvétől függően. Az ilyen részletesség lehetővé teszi a szoftverben további konfigurációs, közösségi és gazdálkodási lehetőségeket is.

1.3 Sushi Go Party!

A „Sushi Go Party!” kártyajáték a „Sushi Go” nevű kártyajáték második verziója, további lapokkal és többféle pakli összerakásának lehetőségével kiegészítve.

A játék szabálya röviden leírva úgy hangzik, hogy minden játékos a játék elején kap a kezébe egy kártyacsomagot különböző japán konyhát idéző ételek kártyáival. Az egyes kártyáknak különböző képességeik vannak, de a legfőbb céljuk, hogy valamilyen módon az asztalra lerakva pontot szerezzen velük a játékos.

A játékban úgy zajlik egy kör, hogy minden játékos kiválaszt egy lapot titokban, és kirakja maga elé az asztalra. Ha mindenki kiválasztott egyet, akkor egyszerre felfordítják. Itt az egyes lapoknak más-más képességük van, amiktől függően vagy simán pontot érve lent maradnak a játékos előtt, vagy további akciót hajtanak végre. Ha ezek lezajlottak, minden játékos továbbadja a kezében lévő lapokat a mellette ülőnek, és kezdődik a következő kör. Ez a mechanika adja a játéknak az egyik stratégiai egyediségét.

A játékosok célja, hogy a kihasznált lapokkal pontokat gyűjtsenek, és a játék végén a legtöbbet szerezzék meg belőlük. Például a nigiri kártyák annyi pontot érnek, amennyi szerepel rajtuk, további logika nélkül. Viszont a tempura esetén minden második lerakott után lehet pontokat szerezni. Vannak kártyák, amiknek az akcióit kártyák kirakása után lehet kijátszani. Lehet az az újonnan kirakott vagy asztalon lévő lap.

Egy körmenetnek akkor van vége, ha minden lap elfogyott a játékosok kezéből. A menet végén minden lerakott kártya visszakerül a pakliba a pontja beszámítása után, leszámítva a desszert típusú lapokat. Egy játék 3 ilyen menetből áll, és az utolsó végén a legtöbb összesített ponttal rendelkező játékos nyer. A desszert lapok az utolsó menet végén kerülnek beszámításra.

A játékot lehet játszani akár csak ketten is, amire van külön kifejezett erre kitalált pakli, és akár nyolcan is, amire ugyanúgy van sok játékosra kifejlesztett kártya

kombináció. Az aranyos rajzokkal ellátott lapokon egyértelműen megtalálhatjuk melyik lapnak mi a képessége, így fiatalok és öregek is könnyedén betanulhatják a játékot könnyed családi vagy baráti program keretében is.

1.4 Felhasználói élmény

A feladat keretében a hangsúly nemcsak a kártyajáték implementációjára összpontosult, hanem a felhasználói élmény és a vele járó segédfunkciók megvalósítására is. A szoftverben ezáltal fontos szerepet játszik, hogy közösségi elemeket, funkciókat is szolgáltasson a felhasználói számára.

A játékban tehát a felhasználókezelési funkciók és annak megvalósítása is kiemelt szerepet kapott. Az egyes felhasználóknak regisztrálniuk kell, hogy igénybe vegyék a játék adta fő és segédfunkciókat. Ilyen segédfunkció például, hogy egymást barátnak tudják jelölni, amivel könnyedén megtalálják egymást a rendszerben.

Ezenkívül, hogy kihasználjuk azt a helyzetet, hogy többfajta paklilehetőség van, egyfajta belső bolt is üzemel a honlapon. Ebben a boltban a játékból szerzett pontokkal tudnak a játékosok beváltani egyes paklikat, amikkel játszani szeretnének. Ezzel is növelve a játékélményt, játékhoz való motivációt.

A játékhoz értelemszerűen szükség van egy váróteremrendszerre is, ahol a játékosok a játék létrehozásánál megtalálhatják egymást. Itt játszik szerepet a megvásárolt paklik jelentősége is. Az ilyen termekben szabadon tudnak szövegesen csevegni egymással a játékosok, míg a játék indítására és további játékosokra várakoznak.

További felhasználói élményt segítő funkciók is megtalálhatóak a szoftverben, mint a honlapon beállítható többnyelvűség, vagy sötét és világos stílus közötti választási lehetőség.

1.5 Megvalósítás

A projekt részeként megvalósult egy szerveroldali és egy kliensoldali alkalmazás. A szerveroldali komponens implementálásánál fontos tényező volt, hogy az mikroszolgáltatás technológia alapjain legyen megvalósítva.

A szerveroldali komponens a főbb funkciók mentén mikroszolgáltatásokra lett osztva. Ezáltal elhatárolva egymástól elemeit a képességeik, felelősségeik mentén, közöttük elvágva a kapcsolatot. Ezek a főbb funkciók a felhasználó és barátkezelés, a

boltnak a kezelése, a váróterem és beszélgetésnek a kezelése, és magának a játéknak a kezelése. Ezek a funkciók között laza kapcsolat van, és csak ritkán van igény közöttük való kommunikációra. Ez funkcionálisan be van építve a szerverbe, mint konténerek közötti kommunikáció.

A szerveroldali mikroszolgáltatások használatánál fő szempont volt, hogy a felosztottság a külvilágból transzparens legyen, tehát a szerver egyetlen kapun keresztül kommunikáljon bárkivel. Ebbe beletartozik a WebSocket kommunikáció is, és a statikus adatok, mint például a felhasználók profilképeinek, vagy más képeknek, illusztrációknak a linkjei is.

Szerveroldalon a perzisztens relációs adatbázis mellett egy temporális gyorsítótárazásra használt Redis alapú cache adatbázis is üzemeltetve van, ami sokban növeli a szerver teljesítményét, gyorsaságát. Ezzel tovább növelve a felhasználói élményt.

Emellett a szerver a szokványos REST alapú kommunikáción felül lehetőséget biztosít WebSocket technológiára épülő kommunikációval való csatlakozást is, amivel élőben értesülhetünk kliensoldalon az egyes eseményekről. Például, ha egy váróteremben egy másik felhasználó üzenetet küldött.

Kliensoldalon kiemelt figyelmet fordítottam az igényes, átlátható dizájnra. A kliens feladata a kommunikáció a szerverrel, mind REST mind WebSocket formában. Az utóbbinál kezelnie kell, hogy melyik oldalakat megtekintve melyik kapcsolatot tartsa fenn, és hogy az egyes kapcsolatok eseményeivel milyen műveleteket hajtson végre.

Kliens szerepe ezenkívül a többnyelvűség és stílus megvalósítása is. A kiválasztott nyelvet és stílust egy-egy Cookie segítségével tárolom, hogy az oldal frissítésekor, vagy újra megnyitásakor is a preferált, korábban kiválasztott mód segítségével jelenjen meg.

Egyelőre két nyelven, magyarul és angolul jelenik meg a honlap, de végtelen bővítési lehetőség van. A stílusokból is kettőt választhatunk, egyet, ami a honlap világos témáját teszi ki, egyet meg ami a sötétet. Viszont itt is könnyedén tudjuk bővíteni további témákkal.

Ezenkívül, hogy a kommunikációhoz szükséges szenzitív információk biztonságos helyen legyenek tárolhatóak, az egyes kulcsok nem a környezeti változók között vannak tárolva, hanem script segítségével töltődnek be oda indításkor.

2 Technológia

2.1 Bevezetés

Az elkészült szoftver számos és változatos technológiai skálát tartalmaz, kezdve az adatbázistól a böngésző stílusáig. Ebben a fejezetben pár főbb technológiát sorolok fel, amik meghatározóak voltak a szoftver fejlesztése során.

Nagy befolyással voltak rám mind a BSc, mind az MSc alatt elkészített korábbi munkáim, amikből ihletet és tudást merítettem. Az elkészült alkalmazás tekinthető technológiai szempontból a korábbi munkáim továbbfejlesztésének, mivel az alattuk megismert technológiák részletesebb megismerése és használata volt a szoftver célja.

2.2 Szerveroldali technológiák

A szoftver fejlesztése során szerveroldalon számos technológia volt felhasználva. Ezeknek elsődleges feladata a mikroszolgáltatások kialakítása és kommunikációja, emellett bennük a háromrétegű architektúra kialakításának megvalósítása volt.

2.2.1 Docker

A Docker [1] egy olyan nyílt forráskódú technológia, ami lehetővé teszi alkalmazások gyors és egyszerű csomagolását és konfigurációját egy környezetfüggetlen környezetben. Egy konténerizációs platform, ami operációs rendszer szinten biztosít virtualizációt.

Segítségével alkalmazások és függőségeik környezettől függetlenül, szabadon futtathatóak, mivel egységesen csomagolva alakítja ki hordozhatóságukat. Az így kialakított konténerek el vannak szeparálva egymástól, ezáltal beállításaik és függőségeik is külön-külön kezelhetőek redundancia nélkül, átláthatóan. Az egyes konténerekre API felületet biztosít, így könnyedén megfigyelhetjük a belső folyamatokat.

A létrejött konténerizált rendszer könnyűsúlyú, jól skálázódik a szerver létrehozásánál. A rendszer kialakításához én a Docker Compose segítségét használtam, ami egy YAML konfigurációs fájl segítségével engedi, hogy megfogalmazzam az egyes konténereket és a hozzájuk tartozó paramétereket, környezeti változókat.

A Docker tehát megkönnyíti a fejlesztők és szakemberek feladatát a fejlesztésben, telepítésben és a komponensek skálázásában is. Hátránya viszont, hogy a fejlesztett kódban redundanciát okozhat, ha az egyes konténerekben hasonló felelősségek vannak. Figyelni kell a kialakítás során, hogy megfelelően válasszuk el a konténerek funkcióinak a határát, esetleg segédkomponensekkel támogatva. Emellett meg kell valósítani a konténerek közötti esetleges kommunikációnak a lehetőségét, ami az elizolált rendszerek közötti logikai kapcsolatot hozza létre.

2.2.2 ASP.NET CORE

Az ASP.NET Core [2] egy open-source, cross-platform keretrendszer, amit a Microsoft fejlesztett ki webes alkalmazások és API-k készítésére. A szoftverhez én a .NET 7-es verziójú változatát használtam, ami dolgozat írása alatt a legújabbnak számított. A keretrendszer lehetővé teszi fejlesztők számára hatékony és skálázható szoftverek fejlesztését. Dockeres technológiákkal is széleskörű támogatás található benne. Egy megbízható és hatékony keretrendszer szerveroldali alkalmazások fejlesztésére, ami webes alkalmazások hátterének könnyen használható.

A keretrendszer segítségével létrehozott alkalmazásokban könnyedén tudunk további segédkönyvtárakat telepíteni NuGet package-ekként, ami ki is volt használva a dolgozatban. Emellett beépített támogatást tartalmaz az egyes szolgáltatások injektálására, amivel rendezett, átlátható struktúrát tudtam létrehozni a fejlesztés során.

2.2.3 Ocelot

Az Ocelot [3] egy olyan segédkönyvtár, aminek elsődleges célja, hogy .NET technológiával futó mikroszolgáltatások fölé egy kívülről transzparens, egységes felületet biztosítson. Ez a közös felületet biztosító komponens az „API Gateway” a szerver implementációjában.

Az Ocelot az üzenetek átirányítása során többféle stratégiát vagy transzformációt alkalmazhat. Emellett átlátható integrációt valósít meg az IdentityServerrel, ami a szerver felhasználókezelő komponense. Ezáltal könnyedén lekezeli a felhasználók authentikációját, ha a helyzet úgy kívánja.

A Gateway-en keresztül nem csak a REST alapú kérések, de a WebSocket technológiára épülő kétirányú kommunikáció is támogatva van, elősegítve a szerverben való implementálását is és a kliensoldali regisztrációját is.

Az Ocelot tehát egy hatékony és rugalmas könyvtár konténerizált rendszerek elrejtésére egy API Gateway mögé, ezzel megkönnyítve a szerver külső használatát, esetleg elrejtve a nem publikálandó belső működést. Ezzel bizonyos szinten biztonságot is nyújtva.

2.2.4 MediatR

MediatR [4] egy C#-hoz készült segédkönyvtár, ami a mediátor minta megvalósítását hivatott segíteni a fejlesztésben. Használata leegyszerűsíti a kérések és parancsok regisztrálását és a megvalósításuk delegálását a megfelelő komponensek felé. Segíti a szoftverben a kód jobb széttagoltságát és a felelősségek megfelelő elválasztását. Laza csatolás biztosításával a funkciók jobb skálázhatóságát is biztosítja.

A könyvtár elég rugalmasan és egyszerűen használható a beépített osztályok segítségével, továbbá a WebSocket események lekezeléséhez is ad további segítséget. Lehetőséget ad a CQRS minta implementálására, tehát a parancsok és lekérdezések szétválasztására. Emellett biztosít az esemény feliratkozás minta használatára is, ami a cache adatbázis naprakészen tartásában és a WebSocket használatához adott segítséget.

2.2.5 SignalR

A SignalR [5] egy open-source segédkönyvtár, ami leegyszerűsíti a valós idejű kommunikáció megvalósítását ASP.NET Core alkalmazásokban. A használt keretrendszer beépített eleme, így könnyen felhasználható vagy kombinálható más komponensekkel, mint például az authentikáció kezelésében.

Az alkalmazás szerves és fontos része volt a kétoldalú kommunikáció megvalósítása, mivel a megvalósított funkcióknál nagy szerepe volt, hogy a felhasználók élőben, rögtön értesüljenek a változásról, ne kelljen manuálisan nekik lekérdezniük.

A SignalR rugalmasan használható. Megfogalmazhatunk számos kapcsolatot, amiknek megszabhatjuk, hogy mi a kommunikációs interfésze a klienssel. Emellett a kapcsolatok dinamikus kezelésével, csoportosításával nyomon tudjuk követni az élő kapcsolatokat. A kialakított csoportok segítségével nem csak egy-egy tudunk üzeneteket váltani, hanem jól definiált csoportosítások mentén broadcast módon is tudunk értesítéseket küldeni a megfelelő felhasználóknak. Erre egy példa, ha egy játékos kirak egy lapot, akkor arról az aktuális játék résztvevői értesüljenek.

Mivel az egyes konténerek el vannak zárva a külvilágtól, ezért a megvalósított WebSocket kommunikáció nem közvetlenül a klienssel zajlik le, hanem az Ocelot feladata elosztani a megfelelő végpontok között ki kivel kommunikál. Szerencsére az Ocelot lehetőséget ad nem csak REST alapú kommunikációra. Az így kialakított átirányítást továbbá authentikáció is védi, így magasabb védelmet biztosítva az alkalmazásnak a felhasználói számára.

2.2.6 Redis

Redis [6] egy gyors és könnyen skálázható kulcs-érték alapú adatbázis, amit a gyorsasága miatt cache-elés megvalósítására használtam a kérések megvalósítása során. A Redis egy in-memory típusú adatbázis, tehát minden adatot a memóriában tárol el. Ez adja a rendkívüli gyorsaságát és hatékonyságát, ami miatt gyakran használják. Emellett egyszerű és rugalmas integrációt biztosít a konténerizált és .NET alapú rendszereknek, amit én is kihasználtam.

Gyakorlatban elsősorban a MediatR lekérdezései és parancsai kezelésénél volt szerepe. Bizonyos lekérdezéseknél be lett állítva, hogy a lekérdezés továbbküldése helyett először ellenőrizze le, hogy a megadott kulcshoz volt-e már elmentett érték a cache adatbázisban. Ha talált ott értéket, akkor a lekérdezés lefuttatása nélkül visszatért. Az ilyen kulcs-érték párok frissítését vagy a MediatR esemény feliratkozás lehetőségével oldottam meg, vagy egyes parancsok visszatérését használtam fel.

2.2.7 RabbitMQ

RabbitMQ [7] rendszer elsődleges feladata a konténerizált rendszerben a konténerek közötti kommunikáció megvalósítása volt. A RabbitMQ egy message queue rendszer, ami a megvalósított mintában a message broker szerepet tölti be. Itt az egyes konténer komponensek felvehetik a kibocsátó és fogadó szerepet, elküldve az üzeneteket a feliratkozó konténerek felé, és fogadva a feliratkozott üzeneteket a küldő konténerek felől.

Mivel az egyes felelősségeket nem lehet teljesen elválasztani egymástól, ezért kellett gondoskodni arról, hogy az egyes konténerek tudjanak egymással kommunikálni amikor szükséges. A RabbitMQ szerepe tehát az, hogy bizonyos események mentén parancsokat továbbítson más-más konténerek felé.

2.2.8 IdentityServer4

Az IdentityServer4 [8] szerepe volt az alkalmazásban a felhasználói műveletek kezelése. Ebbe beletartozik a regisztráció, belépés vagy esetleg törlés vagy módosítási műveletek is. Nagy szerepet töltött be mind a szerver mind a kliens fejlesztésében is.

Maga az IdentityServer4 a felhasználókezelő konténerbe lett beépítve, ahonnan többféle jogosultság, token kezelés vagy authentikációs és authorizációs szerepet lát el. A kliensnek bejelentkezés során átadott tokeneket sokszínűen tudjuk testre szabni, így a szükséges jogosultságokat könnyen át lehet adni a weboldalnak a bejelentkezett felhasználóról.

Az OAuth2 technológiára épülő authentikációt széles körökben használják, és az IdentityServer4 is ad rá lehetőséget, hogy a szerverünk felhasználóit ezzel a technológiával védjük meg. Emellett egyszerű integrációt is biztosít a szerver többi komponensével, mint az Ocelot vagy SignalR könyvtárak.

2.2.9 AutoMapper

Az AutoMapper egy egyszerű segédkönyvtár, ami az egyes domain osztályok átfordításában segít, vagy adatátviteli objektumokra, vagy nézetmodellekre. Könnyen kezelhető, de jól testre szabható a profiljai megfogalmazásával.

Használatával átláthatóbban tudjuk kezelni a bemenetek és kimenetek feldolgozását és kialakítását anélkül, hogy tele szemetelnénk a lekérdezéseket vagy adatosztályokat. A keretrendszerben egyszerű támogatás van biztosítva a használatára.

2.2.10 Hangfire

A Hangfire [9] egy olyan segédkönyvtár, ami lehetővé teszi .NET alapú rendszerekben háttérben futó és ismétlődő események kezelését, létrehozását vagy törlését. Széleskörű konfigurációs lehetőségekkel, perzisztens adatbázissal könnyen megfogalmazhatunk olyan folyamatokat, amiket felhasználói események nélkül, vagy azoknak a hatására adott idővel később akarunk végrehajtani.

Emellett lehetőséget ad arra, hogy az időzített folyamatok során a CQRS mintában megfogalmazott parancsokat hajtsa végre a megadott paramétereket JSON-be konvertálva tárolva.

2.2.11 xUnit

Egy tesztelési eszköz, ami .NET alapú rendszerekben Unit tesztek megfogalmazására ad lehetőséget. Különálló tesztelési projektekben megfogalmazva a Visual Studio segítségével tudjuk kezelni a velük létrehozott teszt függvényeket. Így a Moq könyvtár segítségét is felhasználva átlátható Unit teszteket tudunk létrehozni, amiket a fejlesztőkörnyezetből követhetünk milyen eredményt adnak.

2.3 Kliensoldali technológiák

Kliensoldali technológiák főleg a dizájn kialakítására fókuszálnak. Használatuk az egységes, átlátható felület kialakítását vagy a szerver felé irányított kapcsolat megvalósítását segítik elő.

2.3.1 Angular Material

Az Angular Material [10] az Angular keretrendszer által kifejlesztett UI komponenskönyvtár. Elsődleges célja, hogy egy egységes és esztétikus dizájnt adjon a felületnek, amit professzionális környezetben is gyakorian használnak fel. A könyvtár számos beépített funkciót és komponenst szolgáltat, amelyeknek a segítségével a legtöbb elemet ki lehet alakítani, amire egy honlapon szükség lehet. Ezekkel a komponensekkel gyorsan, egyszerűen létrehozhatjuk a kliensoldali alkalmazásunkat a mély testreszabhatóságuk segítségével. A dokumentációját böngészve könnyen megtaláljuk a nekünk szükséges elemeket az interfészükkel és változatos példákkal együtt. Egy gyorsan fejlődő és változó keretrendszerről van szó, de a verziók között is átláthatóan tudunk barangolni.

A világos és sötét stílus kialakításánál is nagy szerepet játszott az Angular Material, mivel a számos beépített komponens stílusával összhangban volt szükség a beállítására. Szerencsére a könyvtár a stílus szerkesztésére is ad konfigurációs lehetőséget.

2.3.2 SCSS

A CSS kiterjesztése, a stílus kialakításában volt fő szerepe. Az alap CSS funkciókon felül számos további lehetőséget biztosít a stílus fejlesztésében. Kialakíthatunk vele hierarchikus, átlátható stílusfákat, bevezethetünk később

felhasználandó változókat, vagy további kiegészítő függvényeket írhatunk bele. A változókezelésnek nagy haszna volt a világos és sötét stílus kialakításában is.

Segítségével változókba lehet szervezni az Angular Material által adott és beállított színsémát. Így csökkentve az erőforrásigényét az egyes stílus fájloknak, mivel csak ezekre a színekre kell hivatkozniuk.

2.3.3 SignalR

Mivel a WebSocket technológia kétoldalú kommunikáció, ezért meg kell említeni a kliensoldali felhasználását is. Angular felől az elsődleges feladata a kapcsolatfelépítés kérése a szerver felé a már említett authentikáció felhasználásával is.

Kliensoldalon arra kell figyelni a használatakor, hogy milyen esetekben akarjuk inicializálni és leállítani a kapcsolatokat. Milyen oldalak megtekintésekor van szükségünk milyen kapcsolatra. Ezenkívül, a már kialakított kapcsolaton lehetőséget biztosít a szerver felől érkező eseményekre való feliratkozásra, az onnan érkező adatok kezelésére.

2.3.4 NGX-Translate

Az ngx-translate [11] egy erőteljes internalizációs könyvtár Angularra, amivel egyszerűen és jól skálázhatóan tudjuk kezelni a kliensoldali többnyelvűséget. A könyvtár segítségével a statikus, honlapon megjelenítendő szövegeket nem közvetlenül égetjük az alkalmazásba, hanem kiszervezzük őket assets fájlokba, ahol a könyvtárban beállított nyelv segítségével választja ki melyik fájl segítségével állítsa be.

A könyvtárnak több más segédfunkciója van, ami sokban elősegíti segítségével a többnyelvű fejlesztést. Például nem csak statikus szöveget tud többnyelvesíteni, hanem paraméterezett értékeket is tud kiértékelni a megjelenítés előtt. Ezzel például a névsorrendet is az adott nyelv szerint tudjuk megjeleníteni, de egyes mondatoknak a tárgyai is ugyanígy könnyen megadhatóak kívülről.

2.3.5 Dotenv

A dotenv egy hasznos és népszerű, mégis egyszerű könyvtár, aminek a segítségével biztonságosan tudjuk kezelni a kliensoldali szenzitív adatainkat anélkül, hogy azokat kiadnánk a külvilág felé. Ha fejlesztők külön-külön dolgoznak kliens vagy szerveroldali alkalmazáson, akkor gyakran előjön a probléma, hogy kulcsokat és titkokat

kell megosztaniuk egymás között. Ezeket nem akarjuk a Git repository-ban nyilvánosságra hozni, mivel a jövőben felhasználhatják ellenünk. Emellett az assets-ek közé se akarjuk őket helyezni, mert akkor bárki kiolvashatja kívülről.

Ilyenkor jön a dotenv könyvtár szerepe, ami egy olyan konfigurációs fájlból szolgáltatja az értékeket, ami nem része a Git repository-nak, de futás közben kiolvashatóak az értékek belőle.

Jelen szoftverben olyan felhasználási módszert valósítottam meg, ami a Dotenv konfigurációs értékeket a kliens inicializálásakor egy script segítségével kiolvassa, és az Angularhoz használható tárolóba tölti be őket. Így amellett, hogy nincsenek a Git irányítása alatt az értékek, a kliens is könnyedén használni tudja a megszokott módon.

2.3.6 Cypress

A Cypress [12] egy szoftverfejlesztési segédeszköz, aminek a segítségével webes felületek end-to-end tesztelésére kapunk lehetőséget. Használatával könnyen ellenőrizhetjük alkalmazásokról, hogy az oldal az elvárt funkcionalitásnak megfelel-e.

Használata során lehetőségünk van külön-külön egységekre bontani a tesztelést, amiken belül akár elő és utó logikát is megfogalmazhatunk. Ilyen logikákat megadhatunk a teljes egység elejére és végére, vagy akár minden teszt köré kialakítva.

A tesztelés során gyakran szükségünk lehet konstans értékekre, amikre a Cypress a saját fixture rendszerével ad lehetőséget. Ezzel a fixture mappába kiszervezett JSON alapú fájlok értékét aktívan tudjuk használni a tesztek futtatása alatt a Cypress cy.fixture függvénye segítségével.

Cypress lehetőséget ad saját függvények hozzáadására is, amivel könnyedén ki tudunk szervezni olyan segéd implementációkat, amik szükségesek a teszteléshez, de nem aktív részei a tesztelt funkciónak. Ilyen függvény lehet például a szerver felé elküldött előregisztráció vagy belépés, amiket a felület tesztelése nélkül akarunk elküldeni. Ezeket a Cypress interfészébe beregisztrálva szabadon tudjuk meghívni a tesztelési fájlok bármelyik részén.

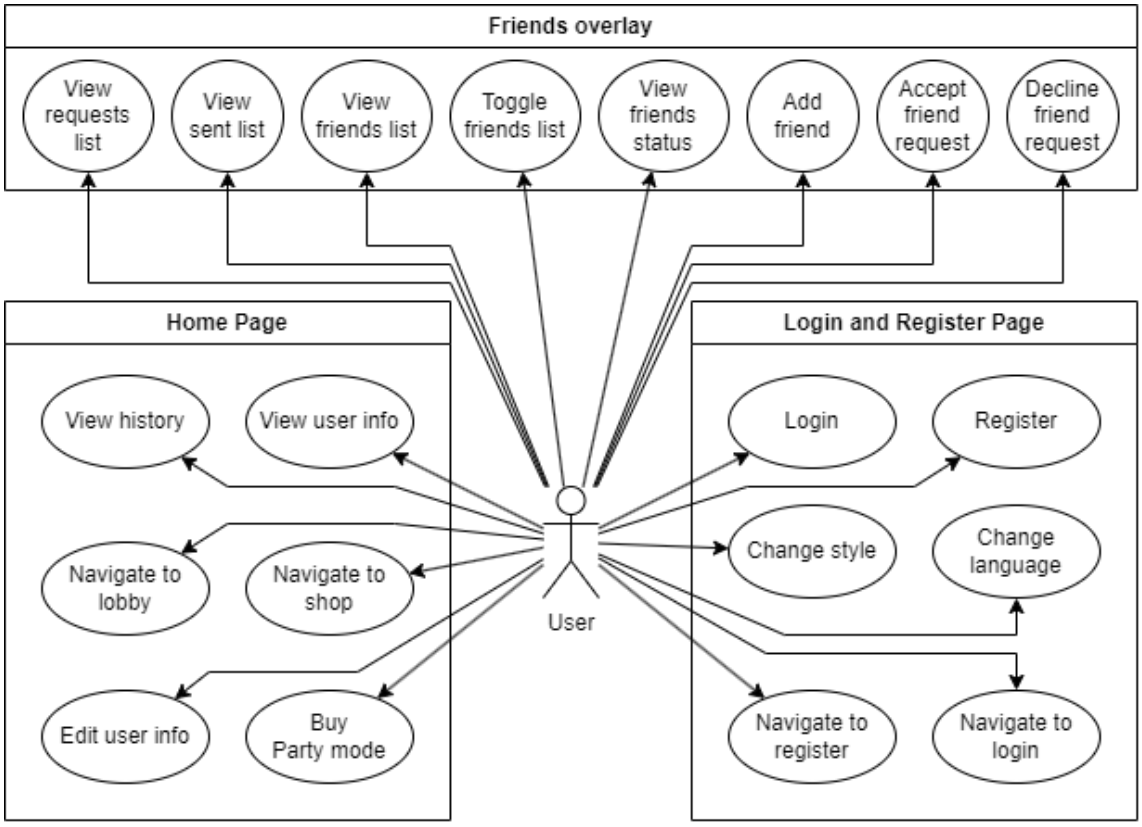
3 Tervezés

3.1 Bevezetés

Az elkészített alkalmazás egy komplex, különálló egységekre bontható szoftver implementációja volt. Ezek a funkciók mentén feldarabolható az alkalmazás specifikációja is.

3.1.1 Felhasználókezelés

A webalkalmazás felhasználókat kezel, hogy személyre szabott játék élményt tudjon nyújtson a játékosoknak. Elsősorban azért van rá szükség, mivel sok olyan funkciót valósít meg az alkalmazás, ami személyre szabott előzményeket és állapotot igényel, így be kell azonosítani ki szeretné használni a weboldalt.



1. ábra Felhasználókezelési használati esetek

Emiatt az oldalra látogatva az első, amit egy megtekintő megtehet, hogy vagy belép, vagy regisztrál. Enélkül nem mehet tovább az oldalon semerre. Egy felhasználótól adatokat gyűjtünk, ami a neve, felhasználóneve, e-mail címe és egy biztonságos jelszava.

Ezek az adatok validálva vannak eltárolva. Ezenkívül a játékosok bejelentkezés után profilképet is állíthatnak maguknak, ami megjelenik a váróteremben és a játékban is.

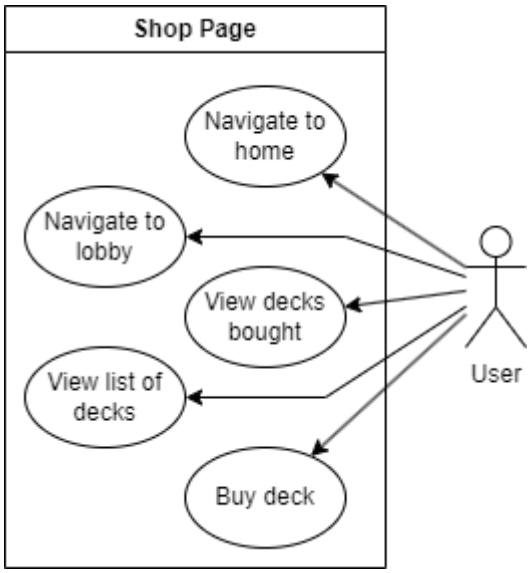
További funkció a felhasználókezelésben a barátok kezelése. A felhasználók bejelölhetnek más felhasználókat barátnak, és azokat a kéréseket a túloldalon elfogadhatják vagy elutasíthatják. Barátok látják egymást egy listán, mellettük egy indikátorral jelezve, hogy éppen aktívak-e a honlapon. Ez elősegíti, hogy a játékosok megtalálják egymást.

A felhasználók ezenkívül tudják módosítani a felhasználójukat, akár a nevüket vagy felhasználónevüket is. Emellett a honlap megtekintése közben változtathatják a stílus vagy nyelv preferenciájukat is, amit egy Cookie-ba elmentve megjegyez.

3.1.2 Bolt rendszer

A webalkalmazás tartalmaz egy beépített boltot, amiben a játékosok a játékok végén szerzett pontokból tudnak vásárolni.

Maga a megvalósított játék úgy épül fel, hogy a játék kezdete előtt ki kell választani egy pakli összeállítást, aminek a kártyáit szeretnénk felhasználni. Itt jön képbe a boltrendszer, hogy ne adjuk oda a játékosoknak a teljes összeállítási listát, hanem az alkalmazás használatával tudjanak tetszőlegesen felnyitni új módokat.



2. ábra Boltrendszer használati esetei

Ezzel további motivációt adunk a játékosoknak, hogy többet használják az alkalmazást. Emellett a nagyobb változatosságon túl döntési lehetőséget is ad, hogy milyen pakli tetszett meg melyik felhasználónak.

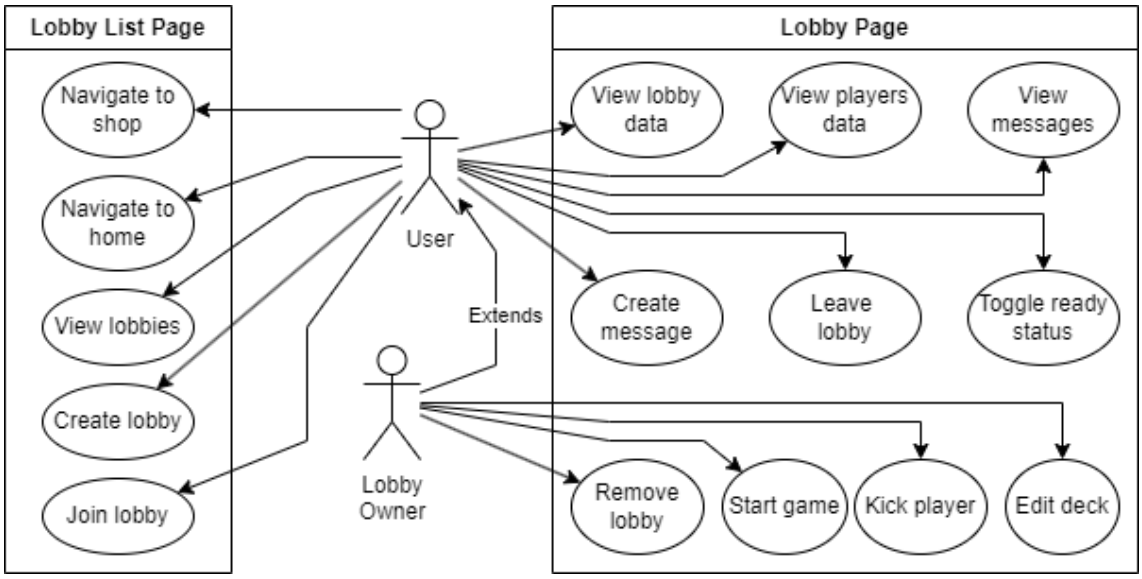
Minden játékos a játék alap Sushi Go verziójának paklijával tud kezdetben játszani, és elég pontot szerezve tudja magának feloldani a boltban vásárlási lehetőséget. Ezzel kicsit elrejtve a plusz ponthasználat szükségességét, de mégis újdonságot adva.

3.1.3 Váróterem rendszer

Az alkalmazás várótermeket, lobby-kat kezel. Ennek a rendszernek a segítségével tudják a felhasználók megtalálni egymást egyes játékokra.

A várótermek komplex funkciókkal lettek ellátva. Először is megtekinthetjük az aktív várótermek listáját, aminek az aktuális állapotát élőben látják változni a honlap megtekintői. Tehát ha valaki elindít egy szobát, akkor azt egy másik felhasználó rögtön látja a lista változásával. Ugyanez fordítva, ha egy szoba megszűnik, akkor arról minden megtekintő rögtön értesül.

Egy szobát névvel és jelszóval tudunk létrehozni, amibe a játékosok a jelszó segítségével tudnak belépni. Ezzel valamelyest levédve a szobát, hogy csak azok léphessenek be, akiknek elárultuk a jelszót.



3. ábra Váróterem használati esetei

A szobába lépve a lehetséges funkciók aszerint változnak, hogy a felhasználó hozta-e létre, vagy csak utólag lépett be. A szobában többféle információval találkozunk, kezdve a szoba nevével és jelszójával, amivel további játékosokat hívhatunk meg.

Látjuk a már belépett játékosokat, a profilképüket és egy indikátort arról, hogy készen vannak-e a játékra. Ehhez minden játékosnál megjelenik egy gomb, amivel ki-be tudják ezt kapcsolni. Ez az indikátor is élőben követhető, frissítés nélkül az oldalon.

A szoba tulajdonosa külön jogokkal rendelkezik. Először is módosítani tudja, hogy milyen paklival akarjuk játszani a játékot. Ez automatikus kiveszi a kész állapot indikátorát minden játékostól. Olyan paklit tud választani, amihez megvette a boltban a paklit. Emellett a többi játékos olyan pakli mellett tud kész gombot nyomni, aminek ő is megvette a pakliját. Tehát csak olyan paklival indulhat a játék, amihez mindenki megvette a hozzá illő paklit.

A másik joga a szoba tulajdonosának maga a játék elindítása. Ezt akkor teheti meg, ha minden játékos kész van, és szerepet játszik az is, hogy az adott paklinak mi az ajánlott alsó és felső határa a játékosszámának. Ha ez a két feltétel nem teljesül, akkor nem engedi elindítani a játékot a honlap.

Egy további funkció a szobában, hogy a belépett játékosok tudnak élőben chatelni egymással, míg a játék indítására várakoznak. Ez időrendben megjelenik a szoba alján, ahol látják ki és mit írt a szobában.

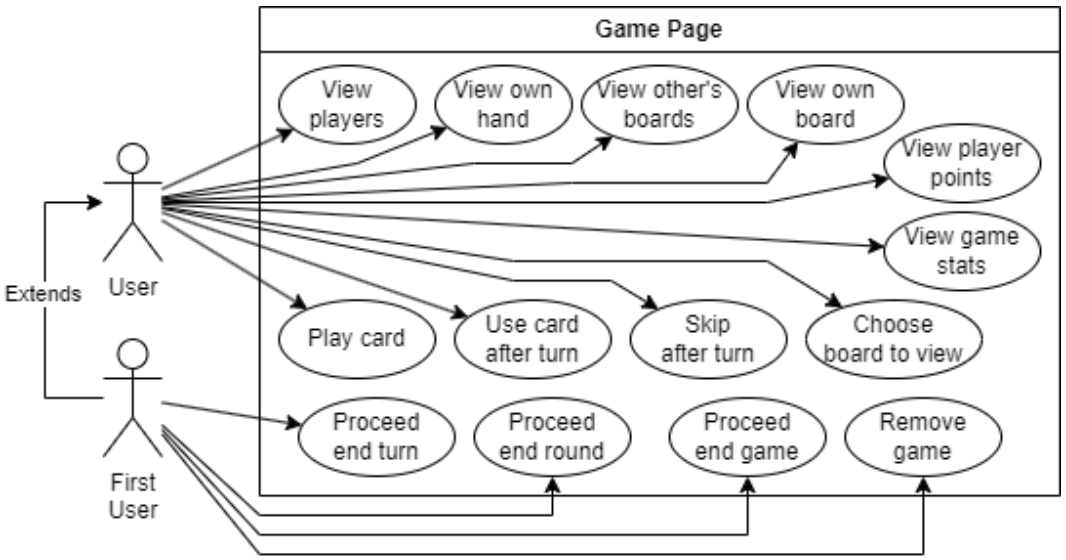
3.1.4 Játék rendszer

Az alkalmazás fő egysége maga a játéknak a megvalósítása. Erre a felületre egyedül a váróteremből elindított játék után lehet kerülni, és a játék befejezéséig vagy törléséig az oldalon bárhova lépve ide irányít át minket.

A játékosok egy átlátható felületet kapnak a játék állásáról. Jobb oldalt egy listán látják a játékosok listáját és sorrendjét, amin az elemekre kattintva a baloldali mezőn megjelenik az adott játékos asztalára kirakott lapok listája. Emellett a képernyő alján látják a saját kezükben található lapokat, amit átválthatnak az asztaluk nézetére, ha onnan akarnak indítani valamilyen akciót.

A játékban a körben első játékos felvesz egy vezető szerepet, neki külön jogai vannak. Minden játékos tudja, hogy kinek van az aktuális köre. Ennek a játékosnak az oldal jelzi, hogy hol és milyen akciókat tud végrehajtani. Így nem tud véletlen rossz lépést

tenni. Egy átlagos játékmenetben egyszerű lapokat, vagy lapokkal végrehajtott akciókat kell végrehajtaniuk. További akciója van az első játékosnak, akinek a feladata elindítani az új kört és menetet, viszont ez is megtörténik a szerver által, ha egy idő után nem indítja el.



4. ábra Játék használati esetei

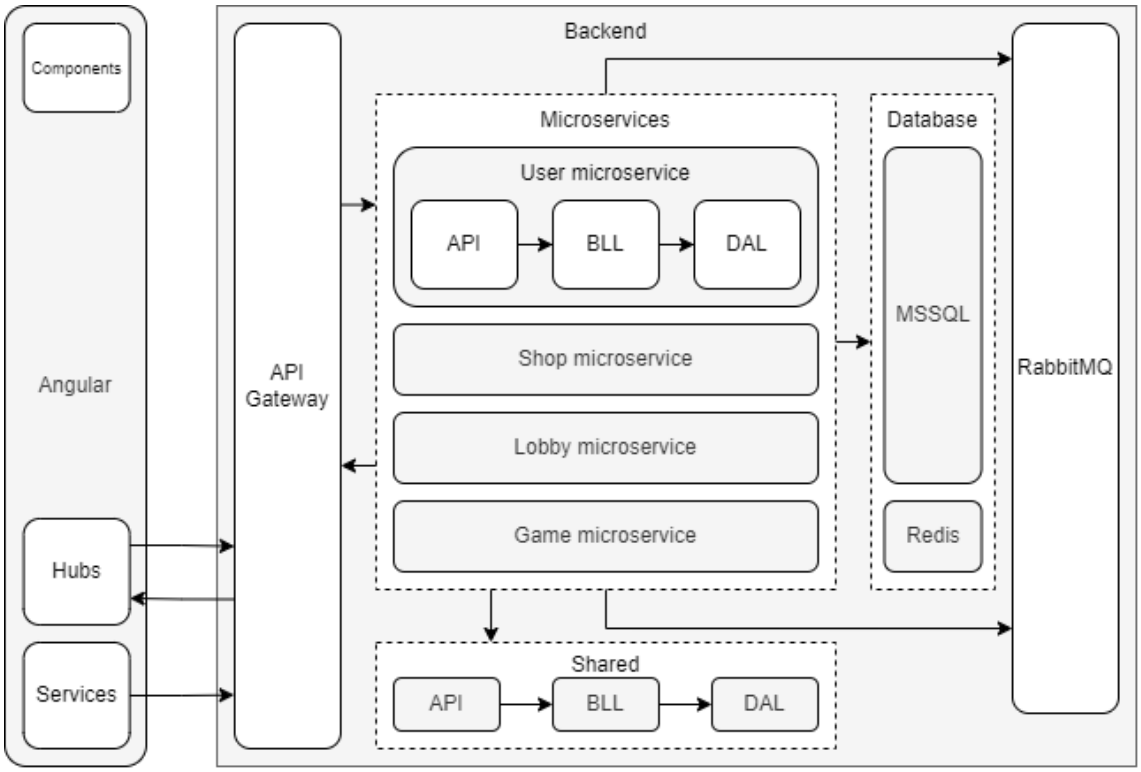
A játékosok látják a játék aktuális állapotát, kezdve a nevétől, aktuális menettől a játékfázis állapotáig. Emellett látják azt is, hogy ki hány pontot halmozott fel eddig a játékban. Amikor a játék véget ér, akkor egy listában megtekinthetik a végső helyezésüket és pontjukat. Ekkor az első játékosnak van joga törölni a játékot, ami mindenkit visszadob a főoldalra.

3.2 Architektúra

3.2.1 Áttekintő

Az elkészült alkalmazás több szempont szerint is egységekre bontható. Ezekben az egységekben többféle architektúra és tervezési minta valósult meg vagy lett felhasználva.

Mivel egy full-stack alkalmazás valósult meg, ezért elsősorban különválaszthatunk egy kliensoldali és egy szerveroldali komponenst. A kliensoldali része foglalkozik közvetlenül a felhasználói interakciókkal és a webes a környezet kezelésével. A szerveroldali komponens pedig az alkalmazás logikájának megvalósításával és az adatok perzisztens tárolásával és transzformációjával.



5. ábra Architektúra

3.2.2 Adatbázis

A szerver adatbázisa alatt két egységet érthetünk. Van a szervernek egy perzisztens, relációs adatbázis alapú tárhelye, és egy ezt segítő gyorsítótárként funkcionáló tároló is.

Az alkalmazás perzisztens tárhelye MSSQL-ben lett megfogalmazva, amit egy külön konténerizált kapcsolaton keresztül érhetünk el. Tehát az egyes konténerek ugyanazzal az adatbázis szerverrel kommunikálnak, viszont külön megfogalmazott adatbázisokat kezelnek. Ezáltal nincs olyan, hogy egymás tudta nélkül egymás keze alá dolgoznak. Az adatbázisokban robosztus, rögzített mezőkkel ellátott adatokat akarunk kezelni, ezért is előnyösebb a relációs adatbázis használata a NoSQL alapúakkal szemben. Emellett a sokfelé elágazó kapcsolatot az adatok között is hatékonyabban tudjuk kezelni relációs alapokon.

Ezeknek az adatbázisoknak a sémája az Entity Framework Code-First technikája segítségével, az adatelérési rétegből lettek megfogalmazva. Itt megadhatóak az egyes modellek, amiket adatbázisban táblákra lehet fordítani. Továbbá megadhatóak táblák közötti kapcsolatok és azoknak fajtáik, változók és mezők közötti konverziók vagy

inicializálási adatok is. Tehát széleskörű támogatást biztosít az Entity Framework használata az adatbázis összeállítására.

A gyorsítótár adatbázisa egy ugyanúgy különálló konténerben futó Redis alapú szerver. Ez a szerver a memóriában tárolja el az adatokat, így kisebb, rövidtávú értékek tárolására van tervezve. Benne az értékek kulcs-érték párokban szerepelnek, ami a jelen alkalmazásban szöveges azonosítókhoz JSON-be fordított objektumok formájában valósult meg. Ebben az adatbázisban tároljuk el a ritkán változó, de gyakran lekérdezett adatokat, hogy meggyorsítsuk a szerver válaszidejét a kliensnek.

3.2.3 Szerveroldal

Az alkalmazás szerveroldala a témából adottan konténerekre van felosztva. Ezek a konténerek funkciók, felelősségek mentén lettek felosztva. Tehát úgy vannak egységekbe zárva, hogy csak a közös logikai elemek tartozzanak össze.

Ezeken a konténereken belül egy-egy háromrétegű architektúrát megvalósító szerveralkalmazás található. Egy API réteg, egy üzleti logikai réteg és egy adatelérési réteg. Ezeken az alkalmazásokon belül további ismert tervezési minták lettek megvalósítva, mint például a CQRS, repository vagy a unitofwork.

Az egyes konténereken belül gyakran találhatóak olyan egységek, amik nagyon hasonlóak, vagy teljesen megegyeznek egymással. Erre lett bevezetve egy „Shared” komponens, ami egy-egy segédkönyvtárt ad az egyes rétegeknek. Erre a könyvtárra a konténerek közösen hivatkozhatnak, így nem kell feleslegesen duplikálni az implementációjukat.

Különösen hasznos a közös könyvtár a konténerek közötti kommunikáció megvalósításakor. A RabbitMQ üzenetküldő implementációjánál ugyanis egy-egy közös modellre kell hivatkozniuk a konténereknek, amikben adatot küldenek vagy fogadnak az esemény során. Ezeket a modelleket a közös segédkönyvtárban megfogalmazva könnyedén felhasználhatjuk.

3.2.4 Kliensoldal

A kliensoldali komponens felépítése egyszerűbben néz ki, mivel egy egységes Angular webalkalmazás készült el. Ezen az alkalmazáson belül lettek feldarabolva az egyes felületek komponensei, és a hozzájuk tartozó szolgáltatások, guard-ok vagy WebSocket hubok.

4 Önálló munka bemutatása

4.1 Bevezetés

A szoftver bemutatásánál azt láttam célszerűnek, ha először általánosságban bemutatom alulról felfele a megvalósított komponenseket, és utána bemutatom funkciókra lebontva az egyes konténerek milyen feladatokat látnak el, hogyan valósulnak meg a webes kliensen.

4.2 Szerveroldali funkciók

4.2.1 Adatelérési réteg

A szoftverben minden konténerben megtalálható egy adatelérési réteg. Ennek a rétegnek a felelőssége az adatbázissal való kapcsolat kiépítése és a modellek megfogalmazása.

4.2.1.1 Repository

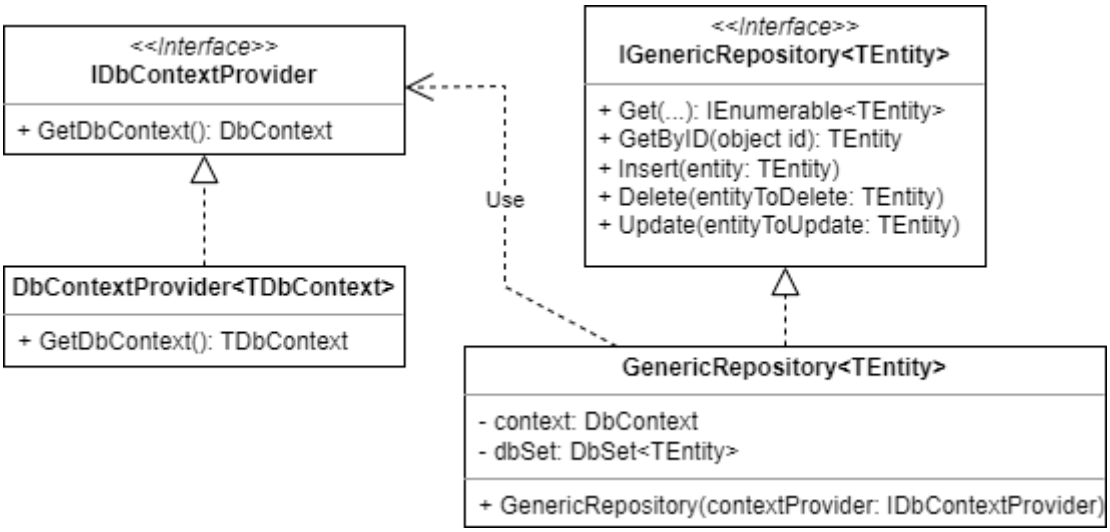
Az adatbázis elérése elsősorban a repository minta segítségével van kialakítva. Ennek a mintának az elsődleges feladata, hogy egy absztrakciós réteget biztosítson az adatbázis fölé az üzleti logikai rétegnek.

A megvalósított repository-k egységes interfészt adnak az adatbázis fölé, amivel egy elszigetelt, lekorlátolt adatelérést biztosítanak. Ezáltal kontrollálni tudjuk, hogy milyen műveleteket biztosítunk a logikai réteg felé. Ezek lefedik a CRUD műveleteket, és elrejtik a mögöttük megtalálható komplexitást, hogy a logikai réteg felől meghívva ne azon legyen a hangsúly.

Ezenkívül mivel egy egységes interfészt bocsát ki, így tesztelés esetén is könnyen mockolhatóak ezen az interfészen végrehajtott műveletek.

4.2.1.2 Repository implementáció

Mivel minden konténerben megegyezik az egységes interfész kialakításának módja, ezért ezt az elemet a közös „shared.dal” segédkönyvtárba emeltem ki. Ez a megoldás viszont magával vonta, hogy az adatbázis kontextusát is generikusan kell megadni benne, mivel az konténerenként változó.



6. ábra Repository struktúra

Ehhez egy provider típusú segéd struktúrát alakítottam ki, ami egy segédfüggvényen keresztül szolgáltatja az adott kontextust. Ezt a segédosztályt az implementáció a függőség injektálással kapja meg, így az injektált objektum típusa szabályozható az egyes konténerekben található tranziens regisztrációk során. Például:

```
services.AddTransient(typeof(IDbContextProvider),
typeof(DbContextProvider<GameDbContext>));
```

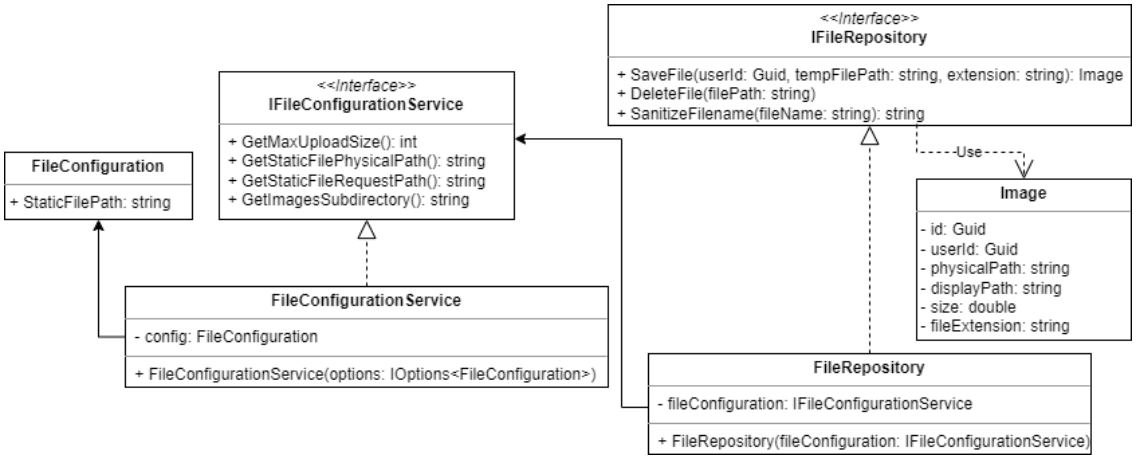
Így az egyes repository komponensek a saját konténerük adatbázisával tudnak dolgozni a közös implementáció során is.

Ezekben a generikus repository megvalósításokban elrejtjük az adott DbSet entitásokon megvalósított műveleteket egy-egy egységes CRUD művelettel. Itt kiemelt szerepet kap a Get függvény, ami részletes paraméterezhetőséget kap. Megadható rajta szűrő funkció, ami a bemenő paraméterként megadott „Expression” objektumot továbbítja az adatbázis lekérdezés szűrése felé. Megadható egy transzformációs objektum, ami LINQ átalakításokat továbbít. Emellett megadható egy szöveges vesszővel ellátott lista, hogy az adatbázis kapcsolatokat milyen mélyen akarjuk szolgáltatni a kimenő eredményben.

4.2.1.3 FileRepository

A konténerek nem csak entitásokat tárolnak perzisztensen, hanem fájlok kezelését is megvalósítják. Ezek a fájlok lehetnek konstans értékek, amiket a szerver szolgáltat a kliens felé, vagy esetleg felhasználók által feltöltött és kezelt fájlok is.

Konstans érték például a megjelenítendő kártyák illusztrációi, felhasználó által kezelt érték meg lehet a profilképük. Mindkettő kezelésére ad lehetőséget a szerver.



7. ábra Fájlkezelő struktúra

Ugyanúgy, mint az entitások kezelésénél, itt is a fő implementáció a „shared.dal” könyvtárban található, mivel elsősorban közös módszerrel van megvalósítva. A fájlok kezeléséhez is tehát van egy kialakított repository minta. Viszont itt nem kell adatbázis kontextussal bajlódnunk, mint az entitások kezelésénél, helyette a konténer szintű fájl konfigurációk kezelését kapja meg kívülről a repository.

Ezeket a konfigurációkat egy szolgáltatás adja át, amiben egy konfigurációs objektum segítségével van megoldva a függvények implementációja. Ez az objektum konténer szinten az appsettings-ből van betöltve a tranziens regisztráció segítségével:

```
services.Configure<FileConfiguration>(configuration.GetSection("FileConfig
uration"));
```

Ezáltal külön tudjuk kezelni az esetleges konténer szintű beállításokat, viszont a jelen implementációban nem kapott nagyobb szerepet.

Ahhoz, hogy használni tudjuk a fájlokat, ezenkívül be kell állítani az egyes konténerekben, hogy szeretnénk statikus fájlokat kezelni. Ennek a beállítására is felhasználható a kialakított fájl konfigurációs szolgáltatás:

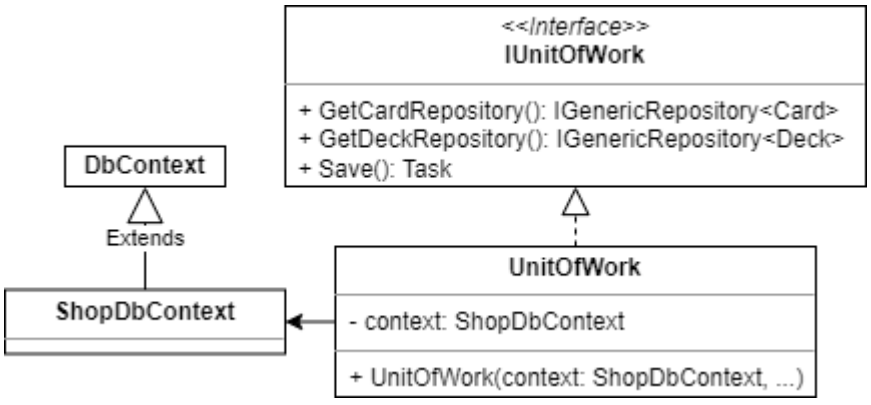
```
var configService =
app.Services.GetRequiredService<IFileConfigurationService>();
app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new
PhysicalFileProvider(configService.GetStaticFilePhysicalPath()),
    RequestPath = $"/{configService.GetStaticFileRequestPath()}"
});
```

Ezekkel a beállításokkal megoldottuk, hogy a szerver fájlokat kezeljen és szolgáltasson a beállított végpontokon keresztül.

4.2.1.4 UnitOfWork

A repository-k kezelését egyszerűsítve bevezetett szoftverfejlesztési minta. A célja, hogy összefogja az egyes repository-kat, ezzel az összes adatbázis műveletet is egyetlen konzisztens interfész alá sűrítve.

Az implementációban a szerepe az egyes generikus repository-k betöltése függőség injektálással, és ennek property-k segítségével a külvilág felé nyújtása. Emellett az adatbázis kontextusát is elrejti az üzleti logikai rétegtől, és egy egységes tranzakciókezelési függvényt nyújt a külvilágnak.



8. ábra Bolt UnitOfWork komponense

Minden konténerben megtalálható ez a komponens, és a szükséges repository-kkal párhuzamosan tartalmaz rájuk egy-egy property-t. Így az üzleti logikai rétegben egyedül ezt a komponenst kell kezelni és használni. Ezáltal könnyen karbantartható és skálázható további entitásokkal az implementáció.

4.2.1.5 Transzformációk

Az adatelérési rétegnél egy további általános kérdés a modellek és azok változóinak relációs adatbázis mezőire tábláira és mezőire való fordítása. Gyakori probléma, hogy egy C# változót vagy nem tudunk közvetlenül SQL adatra fordítani, vagy máshogy szeretnénk megoldani, mint az alapértelmezett módja.

Itt lehetőség van az alap működést felülírni vagy kiegészíteni saját megoldással. A saját megoldás során két konfigurációs értéket kell megadni. Egy Converter osztályt, ami a ValueConverterből leszármazva a nevéből adódóan azt adja meg, hogy mit mire

fordítson oda és vissza. Ezután egy Comparer osztályt, ami a ValueComparerből leszármazva a converterrel hasonlóan azt adja meg, hogy a kapott generikus értékeket milyen módon hasonlítson egymáshoz. Erre azért van szükség, mivel egyes típusokat nem feltétlen az egyszerű equals segítségével akarunk összehasonlítani. Például, ha deep copy-ra van szükségünk.

A szoftverben háromféle transzformáció kiegészítés lett implementálva. Egy, amikor van egy lista a C# modellben, viszont ez egyszerű értékeket tartalmaz, így célszerű egyetlen mezőben eltárolni az adatbázisban. Ilyenkor ezt Newtonsoft segítségével JSON transzformációval szöveges értékké alakítva tárolja el. Ez a lista tartalmazhat akár primitíveket vagy enum értékeket is.

```
public class CollectionJsonValueConverter<T> :
ValueConverter<ICollection<T>, string> where T : notnull
```

Másik kettő transzformáció a Dictionary modell értékek átalakításáért felel. Van, hogy a modellben egy-egy kisebb konfigurációs értéket akarunk eltárolni szabad vagy részlegesen korlátozott kulcsokkal. Ilyenkor, mivel elég kicsi a mérete ezeknek az értékeknek, ezért nem éri meg külön struktúrát kialakítani az adatbázisban hozzájuk. Ha egyetlen mezőbe vannak besűrítve, akkor könnyen kiolvasható további művelet végrehajtása nélkül, ezáltal végeredményében növelve a teljesítményt.

Tehát olyan transzformációs lehetőségek is vannak, ami vagy szöveges vagy enumerációs kulcs segítségével Dictionary változóban kezeli adatokat. Ezeket az adatbázisba való mentés előtt szöveges JSON objektumba mentjük, és lekérdezés során olvassuk ki belőle.

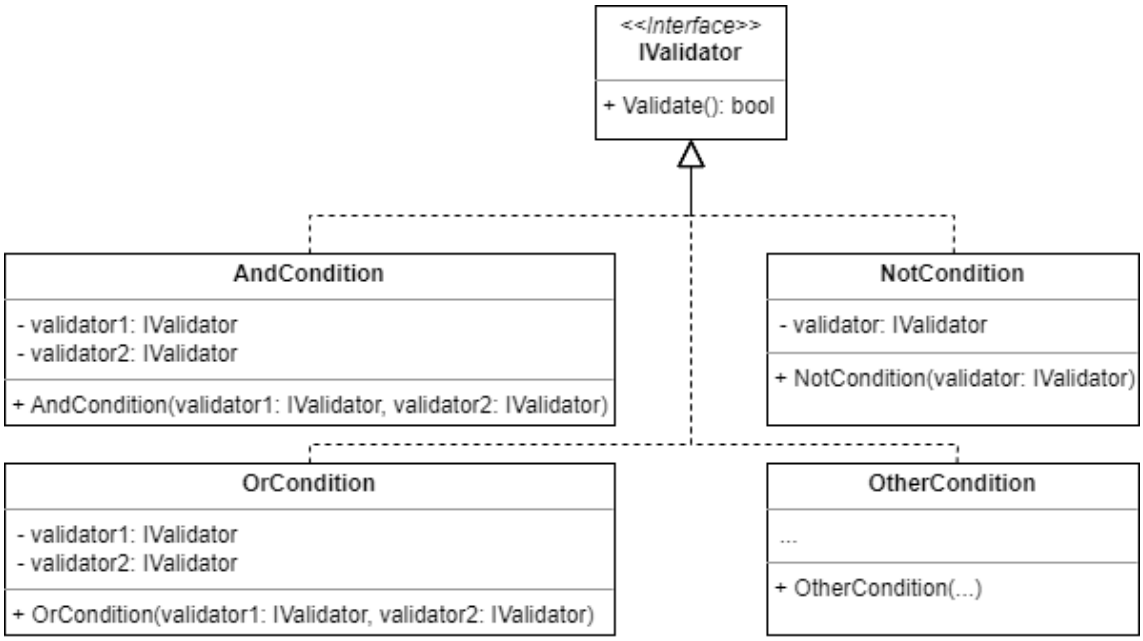
4.2.2 Üzleti logikai réteg

Az üzleti logikai rétegben van megvalósítva az egyes funkcióknak az implementációja és logikai háttere. Itt derül ki, hogy milyen bemenetek hatására milyen adatbázis műveletek kerüljenek végrehajtásra.

Ugyanúgy, mint az adatelérési rétegnél, itt is beszélhetünk pár implementációról, ami általánosságban jelen van minden konténeren belül. Ezek a funkciók a shared.bll segédkönyvtárba lettek kiszervezve, amit minden konténerből meghívva szabadon fel tudnak használni.

4.2.2.1 IValidator

A logikai réteg egyes funkcióiban – főleg, amik módosítást hajtanak végre – be lett vezetve egy validator szoftverfejlesztési mintát követő segéd struktúra, amivel egyszerűen lehet megfogalmazni, hogy az egyes műveletekhez milyen jogokra van szükség.



9. ábra IValidator struktúra

A validációs objektumokat szabadon tudjuk kombinálni az egyes logikai relációkat megvalósító segéd validációk segítségével a saját feltételeink összevonásánál. Az implementáció tehát ilyen IValidator interfészt megvalósító osztályokból képezett objektumok létrehozásával valósul meg.

Ezek az objektumok szabadon kaphatnak paramétert a konstruktoraikban, amiket így fel tudnak használni a validációs függvényük megvalósításában. Tehát az így létrehozott validációs fa struktúrák ősén meghívott validációjából egyszerűen kideríthetjük, hogy a jogot, amit meg akarunk adni a funkció további részéhez az jelen van-e a hívásban. Például egy játékos eltávolításánál a váróteremből az a feltétel, hogy a saját várótermünk legyen, és hogy vagy mi vagyunk a váróterem tulajdonosai, vagy mi magunk akarunk kilépni:

```
_validator = new AndCondition(
    new OwnLobbyValidator(lobby, request.User),
    new OrCondition(
        new LobbyCreatorValidator(lobby, request.User),
        new OwnPlayerValidator(player.UserId, request.User)
```

```
        )
    );
    if (!_validator.Validate())
    {
        throw new ValidationErrorException(nameof(RemovePlayerCommand));
    }
```

4.2.2.2 Pipeline

A MediatR könyvtár széleskörű támogatást nyújt a lekérdezések és parancsok futtatására. Ezekből az egyik a kérések fölé konfigurálható pipeline-ok. Ezek a pipeline-ok middleware jelleggel működnek. Többféle felhasználásuk is lehet. Például kérések előtt vagy után további folyamat megfogalmazása, vagy esetleg hibakezelési folyamat beépítése. Ezáltal szabadon tudjuk kiegészíteni a kérések működését.

Az alkalmazás implementációjában kétféle pipeline lett beépítve. Egy gyorsítótár kezelő és egy naplózó pipeline. Ezeket az egyes konténerek a többi függőség injektálással együtt tudja beépíteni az alkalmazásba az IPipelineBehavior típus alatt, konkrét generikus kérés osztály megadása nélkül is:

```
services.AddTransient(typeof(IPipelineBehavior<,>),
typeof(LoggingBehavior<,>));
```

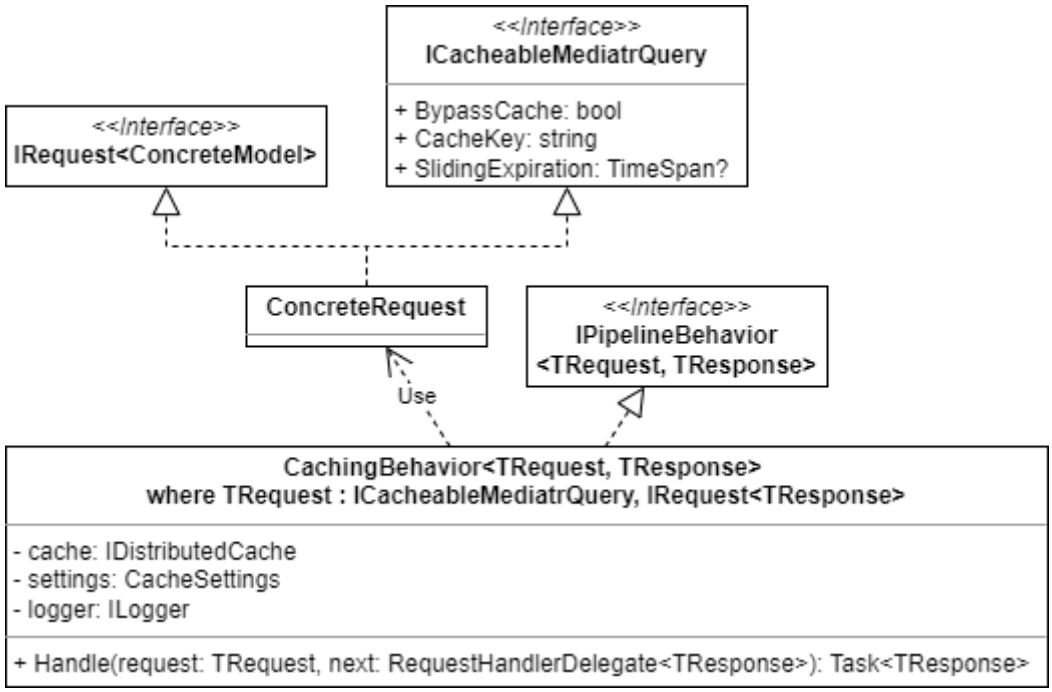
4.2.2.3 LoggingBehavior

Az egyes konténerekben megadható egy naplózó pipeline. Ennek a pipeline-nak egyszerű feladata van. Először naplóznia kell az egyes kérések előtt, hogy mikor milyen kérés indult el, egy generált azonosítóval. Ezután pedig naplóznia kell a kérés végrehajtása után, hogy mennyi időt tett ki az adott kérés lefutása, ugyanezzel a generált azonosítóval címezve.

Ezzel a pipeline-nal tehát fejlesztés közben átláthatóan tudjuk követni, hogy a lefuttatott kérések mikor jöttek létre, és hogy megfelelően végre lettek-e hajtva.

4.2.2.4 CachingBehavior

Egy olyan pipeline a MediatR felett, aminek a gyorsítótár kezelése a feladata a megjelölt kérések felett. Segítségével meg tudjuk mondani, hogy egyes kéréseket gyorsítótárból akarunk betölteni. Kétféle módja van a pipeline-nak. Egyik, ami be is tölti és el is rakja a gyorsítótárba. Másik, ami nem tölt be semmit, csak a kimenetét rakja el. Ez többnyire a parancsok eredményére van felhasználva.



10. ábra Cache pipeline struktúra

Az implementáció úgy kezdődik, hogy azoknak a kéréseknek, amiknél be akarjuk kapcsolni a gyorsítótár pipeline-t, azoknak meg kell valósítaniuk az alap IRequest MediatR interfész mellett egy további interfészt is, ami jelzi a pipeline-nak, hogy ezzel a kéréssel foglalkoznia kell.

Ez az interfész tartalmazza a cache működéséhez szükséges információkat, például, hogy mi a kulcsa a Redis adatbázisban ennek a kérésnek. Itt fontos információ, hogy ez a változó az interfészen van egy property-vel meghatározva, tehát a kérés további változóiból számított érték is lehet. Például egy adott játék lekérdezése a kérésben található felhasználó játék azonosítójából állítja össze a gyorsítótár kulcsát.

A pipeline logikája a Handle függvényében található. A függvényen belül több kisebb egység található. Ezek a gyorsítótár lekérdezése a kulccsal és objektumba fordítása, az adott kérés lefuttatása, és az eredmény lementése a gyorsítótárba ugyanúgy az adott kulccsal.

Első lépés, hogy leellenőrizzük, hogy a BypassCache változó igaz-e. Ha igen, akkor lefuttatjuk a kérést a gyorsítótár kérdezése nélkül, és lementjük a gyorsítótárba is az eredményt. Ha hamis, akkor először megnézzük a gyorsítótárban megtalálható-e a keresett és lefordítható objektum. Ha igen, akkor további lépés nélkül visszaadjuk. Ha nem találtuk meg a gyorsítótárban, akkor ugyanúgy lefuttatjuk a kérést és lementjük a végén az eredményt. Ezáltal egy könnyen kezelhető előellenőrzést kialakítva.

4.2.2.5 Exceptions

A szoftverben saját hibaosztályok vannak implementálva. Kevés jelentőségük van azonkívül, hogy velük átláthatóan kezelhető a saját megfogalmazott hibáink. Ilyen például, mikor az IValidator ellenőrzése elbukik, vagy amikor üres eredménnyel zárul egy adatbázis lekérdezés.

Lekezelésük egységesen van megfogalmazva a ProblemDetails segítségével, validációs hibára 400-as hibával, üres eredményre pedig 404-es hibával válaszolva. Az így kialakított hibarendszer további jövőbeli bővítésre is könnyedén ad lehetőséget. A hiba osztályok száma és bonyolultsága ezáltal könnyedén skálázódik.

4.2.2.6 Extensions

Egy egyszerű de fontos egysége a konténereknek a kiegészítő segédfüggvények. Elsősorban a felhasználó tokenében található adatok kiolvasására lettek bevezetve. A szerepük tehát a bejelentkezett felhasználót jelképező ClaimsPrincipal objektumból a megfelelő Claim értékének kiolvasásának kiszervezése egy-egy átláthatóbb függvény mögé. Ezáltal átláthatóbbá téve a felhasználókezelést az egyes lekérdezések implementációjában.

Például a felhasználó azonosítójának lekérdezése:

```
public static string GetUserIdFromJwt(this ClaimsPrincipal
claimsPrincipal)
{
    return claimsPrincipal.Claims.FirstOrDefault(x => x.Type ==
JwtClaimTypes.Subject)?.Value ?? string.Empty;
}
```

4.2.3 API réteg

Az egyes konténerekben az API rétegben számos funkció található. Az itt megvalósított kiegészítő komponensek nagyban különböznek konténerektől függően, ilyen például a felhasználókezelés vagy Hangfire ütemezés.

Ebbe a részlegbe azok a funkciók kerülnek, amik általánosságban megtalálhatóak az egyes konténerek megvalósítása közben.

4.2.3.1 Authentication

A szoftverben be van építve felhasználókezelés. Ez a funkció egy azonosító tokent ad át a kliensnek, amit az egyes kérések előtt érvényesít a szerver. Ezt az érvényesítést nevezzük authentikációnak, amit minden konténer elvégez.

A JWT alapú authentikáció beállítása az API rétegen valósult meg. Ehhez az IdentityServer modulhoz beállított értékek lettek felhasználva az appsettings.json konfigurációs fájlból kiolvasva.

További biztonsági beállítás a CORS bevezetése. CORS segítségével meghatározhatjuk, hogy a szerver milyen klienssel és milyen módon akar kommunikálni. A módszer alapja, hogy a szervertől eltérő címtől érkező kérések előtt a böngésző egy preflight kérést küld, amivel megkérdezi a szervert, hogy az adott címről, ilyen http fejlécekkel és http metódussal hajlandó-e kommunikálni. Ez egy options http metódusú kérés. Ha a szervernek megfelelő, akkor engedélyezi és elindul a kérés. Ha nem, akkor 403 CORS hibával tér vissza.

A szerveroldali alkalmazásban ezzel be lett állítva, hogy az egyes konténerek vagy egymással, vagy a kliens alkalmazással kommunikálhatnak.

Emellett egy egyszerű segédfilter is bevezetésre került, ami arra szolgál, hogy ha nincs megadva authentikáció az adott kérésen, akkor ránéz a kérés query paraméterei közé is, hogy nem lett-e megadva ott a token értéke. Ez például a WebSocket kapcsolatnál vagy az egyes képek lekérdezésénél hasznos. A filter függvény bekötése az alkalmazás kérései elé elég egyszerűen megoldható az inicializáció konfigurációi között:

```
app.Use(AuthenticationExtension.AuthQueryStringToHeader);
```

4.2.3.2 Swagger

A Swagger fejlesztői felület beállítása is egységesen történt meg. Ezzel a felülettel könnyedén tudjuk tesztelni az egyes konténerek működését azáltal, hogy az API réteg interfészét kivetítjük egy webes felületre.

A Swagger felületet a controllerek függvényeiből automatikusan hozzák létre az egyes konténerek. Emellett szükség van a bejelentkezés megoldására is a felületről, mivel a kérések nagyrésze egy bejelentkezett felhasználót követel. Ezért a Swagger konfigurációjánál biztonsági sémaként be van kötve a felhasználókezelési konténer IdentityServer beléptetési végpontja és a hozzá tartozó azonosító és jelszó.

4.2.3.3 WebSocket

A konténerekben megvalósított kétirányú kapcsolatoknak is azonos logikai alapjaik vannak. Egy kétirányú kapcsolat kiépítéséhez először ki kell alakítani egy központi osztályt, aminek a Hub ősosztályból kell leszármaznia. Ezen az ősosztályon

generikusan megadhatunk egy interfészt, ami konkrét felületet ad arra, hogy a kliens felé milyen eseményeket akarunk továbbítani, milyen paraméterekkel.

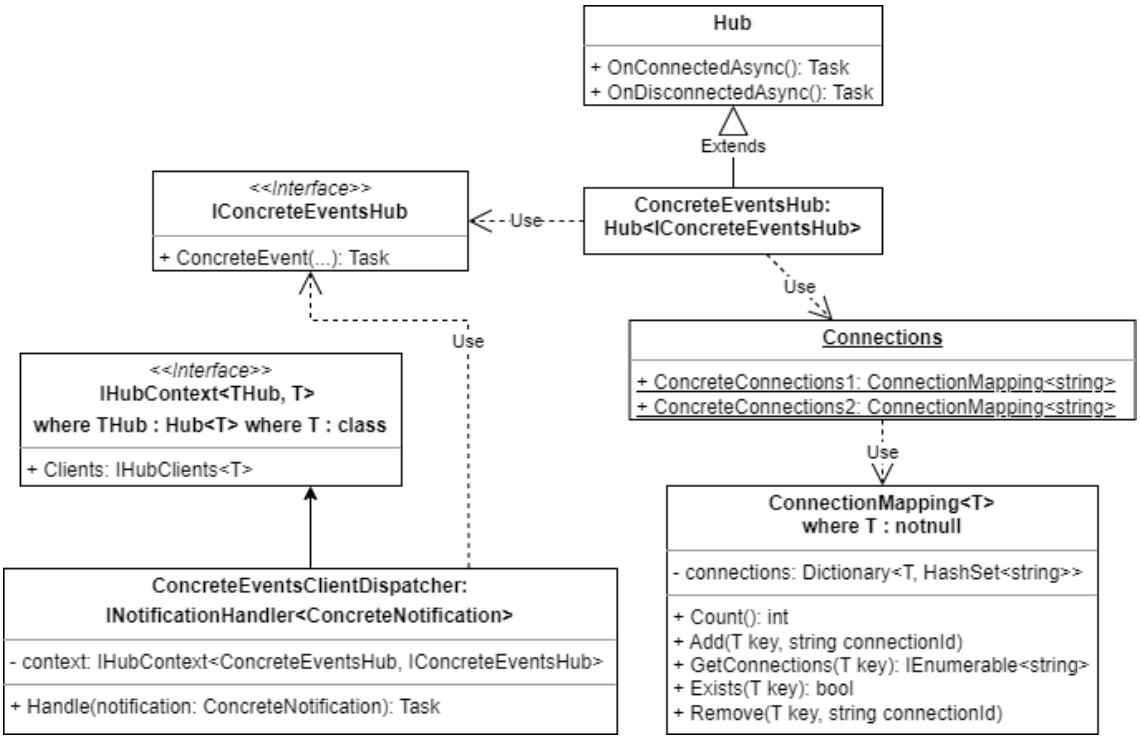
Ebből az ősosztályból kapunk két fontos függvényt. Egyik egy kliens csatlakozásának az eseménye, a másik pedig a lecsatlakozása. Mindkettőre tudunk implementációt kötni, kapcsolatot csoportba rendezni. A csatlakozás eseményén a kontextusból kapunk egy ConnectionId azonosítót, ami az adott kapcsolatot jelképezi. Ez különbözik a felhasználó azonosítójától, így további implementációt igényel a komplex alkalmazása.

Mivel korábban implementáltunk authentikációs filtert, így a Hub kontextusán is el tudjuk érni az adott felhasználónak az adatait. Ezáltal a felcsatlakozás eseményén ki tudjuk olvasni mindkét azonosítót.

A SignalR nem tartja számon a létrehozott kapcsolatokat, úgyhogy komplexebb feladatok támogatására be lett vezetve a ConnectionMapping struktúra. Ez egy egyszerű tárhely, amiben generikusan lehet tárolni kulcshoz értékeket, és azon segédfüggvényeket hívni. Elsősorban a szöveges kapcsolat és felhasználó azonosítók összekötésére van felhasználva. Például a barátok kezelésénél szeretnénk látni, hogy ki van jelenleg online. Ezt ki tudjuk nyerni abból, hogy az aktív kapcsolatok között jelen van-e az adott barátnak a felhasználói azonosítója a segéd tárhelyben kulcsként.

Mivel ez a ConnectionMapping osztály egy változóban kezeli, perzisztens adatbázis nélkül a kapcsolatokat, ezért használata közvetetten egy statikus központi osztályon keresztül történik. Itt az egyes konténerek a hubjaikkal párhuzamos darabszámmal tartalmaznak ConnectionMapping típusú statikus property-ket, amiken hozzáadjuk és töröljük az adott csatlakozó és lecsatlakozó felhasználókat az eseményeik mentén.

Ez a fajta tárhely nagyban hasonlít a Hub beépített csoportkezelőjéhez, viszont ezen ismerjük a tárhely elemeit, és ki tudjuk keresni mi a tartalma. A beépített csoportkezelő így megmaradhat az általános csoport és kapcsolatazonosítók kapcsolatára, amit ez kiegészít, támogat.



11. ábra Szerveroldali hub struktúra

A kapcsolatok és csoportok kiépülésével készenállnak a konténerek az események küldésére. Ezeket megfogalmazhatnánk az egyes hubokon is, viszont letisztultabb és egyszerűbb implementációt kapunk a dispatcher minta felhasználásával.

A hubokon üzenet küldése a kliens felé elsősorban valamilyen esemény hatására szokott történni. Ezért, hogy az alap logikai funkciók implementációja elkülönített maradjon, felhasználjuk a MediatR publish–subscribe mintára épülő funkcióját. Ezzel a mintával INotification osztályból leszármazó MediatR eseményeket tudunk elküldeni, amikre az egyes dispatcher osztályok feliratkozhatnak.

Az ilyen elkapott eseményekből kinyerjük, hogy milyen azonosítójú vagy csoportú kliensek felé szeretnénk elküldeni az esemény adatait. Ezután a generikusan megadott interfész segítségével megadhatjuk, hogy a kiválasztott kliensek csoportja felé milyen függvénnyel és paraméterekkel szeretnénk továbbítani az adatokat. Itt nem kell feltétlen továbbítanunk paramétert, mivel maga az esemény küldése is információt ad a kliensnek.

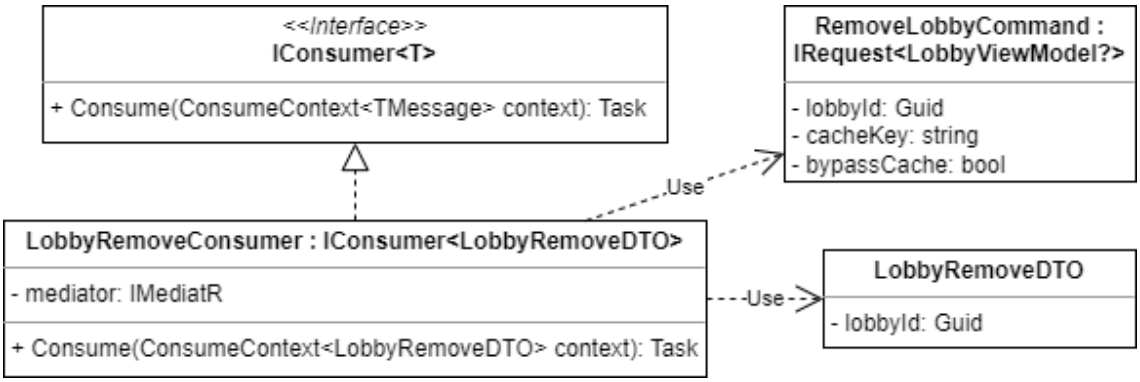
4.2.3.4 RabbitMQ

Ennek a komponensnek az elsődleges és egyetlen feladata események továbbítása konténerek között. Mivel a konténerek határai jól lettek meghúzva, ezért ritkán van

szükség adatok közvetítésére közöttük, viszont fontos szerepet töltenek be a funkciók megvalósításakor.

A jelek továbbításának megvalósítása a MassTransit könyvtár segítségével a producer-consumer mintát követi. Ez egy gyakori minta elosztott rendszerek közötti kommunikáció implementálása során. A mintában a producer küldő fél eseményeket pakol egy üzenetsoron, amiről a fogadó consumer rendszer leveszi őket és feldolgozza a kapott adatokat. Az alkalmazásban a minta csak korlátozottan van felhasználva egy-egy kommunikáció megvalósítására, de komplexebb problémák megoldására is ad lehetőséget.

A RabbitMQ egy különálló konténer, ami elvégzi az üzenetek továbbítását. Ahhoz, hogy küldeni lehessen először is szükség van modellekre, amikben továbbítjuk az információt a konténerek között. Mivel ennek a modellnek meg kell egyeznie mindkét oldalon, ezért célszerűen a „shared.dal” segédkönyvtárban lett megadva, ahol közösen lehet szerkeszteni vagy bővíteni változóit mindkét oldalnak.



12. ábra Consumer példa minta

A küldő oldalról tehát egy ilyen modellt kell létrehozni az események előtt. Ezeket a modelleket az IPublishEndpoint injektált szolgáltatásával kell elküldeni a RabbitMQ konténere felé. Például amikor egy felhasználó csatlakozik egy váróteremhez, akkor azt el kell küldeni a felhasználókezelő konténernek is, hogy eltárolhassuk a felhasználó adatai között is:

```
await _endpoint.Publish(new LobbyJoinedDTO
{
    UserId = request.PlayerDTO.UserId,
    LobbyId = request.PlayerDTO.LobbyId
}, cancellationToken);
```

Ahhoz, hogy küldeni vagy fogadni tudjunk üzeneteket, definiálnunk kell az API rétegben a MassTransit kapcsolatot. MassTransit egy olyan keretrendszer, ami .NET-es alkalmazásokban segít kialakítani aszinkron kommunikációt applikációk és szolgáltatások között, többek között a RabbitMQ segítségével is. Az IPublishEndpoint is ennek a könyvtárnak a része.

Az alkalmazásban a kapcsolat elég alapszintű, mivel kevés funkciója van kihasználva. Viszont sokszínű, komplex kialakítást tesz lehetővé. A kapcsolat létrehozása nagyvonalakban így néz ki:

```
services.AddMassTransit(options =>
{
    options.AddConsumer<LobbyRemoveConsumer>();
    options.SetKebabCaseEndpointNameFormatter();

    options.UsingRabbitMq((context, cfg) =>
    {
        cfg.Host("rabbitmq", "/", h =>
        {
            …
        });
        cfg.ConfigureEndpoints(context);
    });
});
```

A kapcsolatban a hozzáadott consumerek teszik ki a fogadó oldalát az eseményeknek. Ezekből tetszőleges számút definiálhatunk az alkalmazásokban. A regisztráció során a host beállítása tartalmaz további konfigurációkat, mint például a felhasználónév és jelszó megadása a RabbitMQ konténerhez.

4.2.4 API Gateway

A Gateway alapvető feladata a konténerek elfedése, és egy közös interfész definiálása felettük. Tehát meg kell benne határozni, hogy milyen konténerekben, milyen végpontok hova legyenek bekötve. Erre az Ocelot keretrendszer egy ocelot.json nevű konfigurációs fájl létrehozásaként ad lehetőséget, ahol az átkötés mellett tetszőleges további beállítást tudunk hozzáadni a végpontokhoz. Ilyen beállítás például az authentikáció ellenőrzése.

A JSON fájlban konténeren belüli útvonalat és konfigurációt a downstream előtagú értékek jelölik, és a külvilág felé irányított értékek pedig az upstream előtaggal vannak ellátva. Például a „DownstreamHostAndPorts” tartalmazza, hogy pontosan milyen konténer cím felé irányítjuk az adott kérést.

4.2.4.1 SwaggerForOcelot

Egy praktikus segédkönyvtár, aminek a segítségével Swagger felületet tudunk biztosítani az ocelot által kezelt interfésznek. Mivel az interfész mindenféle konténerből tartalmaz végpontokat, ezért lehetőséget ad saját elválasztást adni a konfigurációk megjelenítésének. Ezáltal a Swagger beépített lapozója segítségével tudjuk kiválasztani a megtekintendő konténerek Swagger sémáit.

A könyvtár felhasználásánál nem csak a Swagger felületet tudjuk felosztani, hanem az ocelot.json konfigurációs értékeket is szétszedhetjük külön JSON fájlokba. Ehhez a könyvtár regisztrálásakor meg kell határoznunk, hogy milyen útvonalon található ez a könyvtár. Ebben a projektben a Routes mappába lett szervezve minden ocelot konfigurációs fájl:

```
builder.Configuration
    .AddOcelotWithSwaggerSupport((options) =>
    {
        options.Folder = "Routes";
        options.FileOfSwaggerEndPoints = "ocelot.swagger";
    });
```

A fájlok olyan konvenciót kaptak, hogy „ocelot.{konténer neve}.json”. Emellett található egy ocelot.ws.json fájl, ami a WebSocket útvonalak megfelelő átirányításáért felel, és egy ocelot.swagger.json, ami a Swagger feldarabolásának beállításáért felel.

Ahhoz, hogy fel tudjuk darabolni a kéréseket külön felületekre, ahhoz minden kérésnek tartalmaznia kell egy SwaggerKey konfigurációs értéket, ami az ocelot.swagger.json fájlban hozzápárosít egy belső swagger konfigurációt. Ennek a fájlnak az elemei tartalmazzák a kulcsot, amit a kérésekbe rakunk, és mellette a swagger felület beállításait. Például a felhasználókezelő konténer konfigurációs értéke:

```
{
  "Key": "user",
  "Config": [
    {
      "Name": "User API",
      "Version": "v1",
      "Url": "http://user.api/swagger/v1/swagger.json"
    }
  ]
}
```

4.3 Kliensoldali funkciók

4.3.1 Dotenv

A könyvtár felhasználásánál a cél az volt, hogy verziókezelés nélkül, de a beépített Angular módszerrel lehessen kezelni a környezeti változókat. Az implementáció során egy .env nevű verziókezeletlen fájlban helyezzük el a konfigurációs értékeket, például a szerver címét:

```
baseUrl=http://localhost:5000/api
```

Viszont ezeket az értékeket a környezetfüggő environment konfigurációs fájlokból akarjuk felhasználni. Ehhez ezek az environment fájlok kikerültek a verziókezelésből, és megvalósításra került egy config script, ami a .env értékekkel feltöltve létrehozza ezeket az environment fájlokat. Ez a script a ts-node segítségével futtatható, és a package.jsonbe is beépítésre került az alábbi módon:

```
"config": "ts-node ./scripts/setenv.ts",
"start": "npm run config -- --env=dev && ng serve",
"build": "npm run config -- --env=prod && ng build",
```

Itt átadjuk a konfigurációs scriptnek, hogy fejlesztői vagy éles környezetben akarjuk futtatni, ami szerint a megfelelő environment fájlokba tölti be az értékeket. Ezáltal beépül az általános alkalmazás indítási scriptekbe a környezeti változók betöltése is, ha szükséges a kitöltésük.

4.3.2 Témák

Kétféle Angular Material alapú téma lett megvalósítva. Egy kék alapú világos, és egy rózsaszín alapú sötét téma. A témák beállításai ki lettek szervezve a sushi-theme.scss fájlba, amit egyedül a styles.scss fő stílusfájl tölt be.

Az alkalmazás alapértelmezett témája a világos, amit a sötéttel tudunk felülírni felhasználó által Cookie-val vezérelve. Ez a vezérlés úgy működik, hogy a body tagnek a css osztályát állítjuk light-theme vagy dark-theme értékre. Ezáltal tudjuk a témát css osztályokon belül kezelni és felülírni. Például a fő téma regisztráció:

```
@include mat.all-component-themes($light-theme);
.dark-theme {
  @include mat.all-component-colors($dark-theme);
}
```

Ezzel betöltjük alapértelmezetten a világos téma beállításait. Ettől a sötét téma csak a színbeállításokban tér el, ezért elegendő a sötét osztályon belül csak a színek regisztrációját átállítani.

Ahhoz, hogy a Material színeit hatékonyan lehessen használni nem Angular komponenseken is, segédváltozók lettek bevezetve. A segédváltozók nélkül minden komponensben hivatkozni kéne a Material könyvtárra, ami megterheli a stílusok betöltését. A helyette bevezetett segédváltozók globálisan megtalálhatóak az alkalmazásban, így egyéb komponensek betöltése nélkül lehet rájuk hivatkozni, például:

```
background-color: var(--color-primary);
```

Az egyes változókat egy mixin segítségével tölti be, ami létrehoz egyszerre egy hex alapú és egy rgb alapó színváltozót is:

```
@mixin addColor($color, $value) {
  --color-#{$color}: #{$value};
  --color-#{$color}-rgb: #{red($value)}, #{green($value)}, #{blue($value)};
}
```

Ehhez a mixinhez a változókat egy manuálisan definiált listából olvassa ki, ami tartalmaz Materialból hozott értékeket és saját értékeket is.

```
:root {
  .light-theme {
    @each $color in $colors {
      @include addColor(…);
    }
  }
  .dark-theme {
    @each $color in $colors-dark {
      @include addColor(…);
    }
  }
}
```

4.3.3 Nyelvesítés

A nyelvesítés megvalósításához az ngx-translate könyvtár volt használva. Az alkalmazásban két nyelv található. Ez a magyar és az angol, de tetszőlegesen bővíthető. A megvalósítás részeként az assets mappában az i18n almappában lett kialakítva az en.json és hu.json fájl, ami egymással párhuzamosan, hierarchikusan kialakítva tartalmazza a nyelv specifikus konstans értékeket.

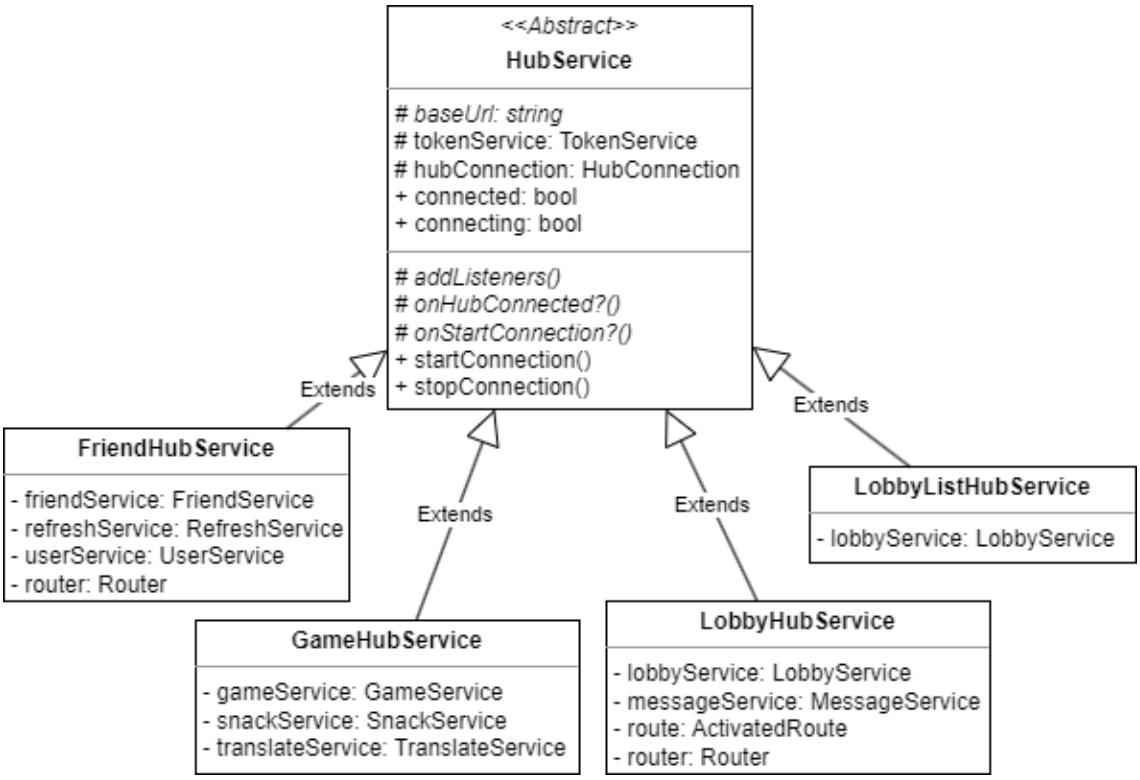
Ezeket az értékeket ki tudjuk olvasni komponens osztályán belül is szolgáltatás segítségével, de gyakoribb és egyszerűbb megoldás az ngx-translate csővezetékének a használata a HTML kódon belül. Például:

```
{{ 'confirm.buttons.yes' | translate }}
```

Az alkalmazás nyelvét vezérlő Cookie alapértelmezett értéke angol. Ezt az oldal megtekintője böngésző specifikusan tudja vezérelni felhasználótól függetlenül. Ezáltal az oldal vagy a böngésző bezárása után is ugyanazzal a beállítással találkozik, ha belépett, ha nem. Az alkalmazás stílusa is megegyező módszerrel állítható be.

4.3.4 WebSocket

A SignalR alapú kommunikáció kiépítése elkészült kliensoldalon is. Az implementáció ki lett szervezve a komponensektől független szolgáltatásokba. Mivel a kapcsolat kiépítésének a módszere megegyezik az egyes hub kapcsolatokban, ezért annak az alap implementációja egy közös absztrakt ősben valósult meg. Minden kapcsolat ebből a közös ősből leszármazva készült el.



A közös ős szolgáltat egy kapcsolat indítási és leállítási függvényt. Emellett lehetőséget ad a leszármazottaknak további implementáció hozzáadására a kapcsolat kiépítésének fázisaiban. A kapcsolat kiépítésének állapotáról is ad két jelző flag változót.

Egyik megvalósítandó függvénye a leszármazottaknak a szerver hub eseményeire való feliratkozások regisztrációja. Itt tudjuk elkapni azokat az eseményeket, amiket a szerverben megadott hub interfészen határoztunk meg. Ezért a feliratkozások az ottani interfészekkel párhuzamos nevekkel és paraméter modellekkel lett kialakítva.

A kapcsolat kiépítése a startConnection függvényben történik meg. Alatta a SignalR HubConnectionBuilder komponensének segítségével sokféle beállítást tudunk megadni. Ilyen a logolás szintjének beállítása, vagy a kapcsolat kiépítése alatt az authentikáció beállítása. Az authentikáció során beállításra került, hogy a Cookie-ban eltárolt access token legyen továbbítva a szerver felé, ami azonosítja a felhasználót.

4.3.5 Interceptor

Az interceptorok feladata, hogy az Angular HttpClient eszköze segítségével elküldött kérések előtt és után további egységes logikát implementáljanak. Az alkalmazáshoz egyetlen interceptor lett hozzáadva, de tetszőlegesen bővíthető továbbiakkal.

Ennek az AuthInterceptor osztálynak az a feladata, hogy a kérések fejlécébe felhelyezze authorization értékként a bearer tokent az alkalmazás access token Cookie-jából. A felhelyezés mellett további feladata, hogy elkapja a hibákat azután, hogy elküldte az egyes kéréseket. Ha az elkapott kérésben authentikációs vagy authorizációs hibával találkozott, akkor letisztítja a tokeneket, mivel azok vagy érvénytelenek vagy lejártak. Emellett egy egyszerű hibaüzenetet is kiír a felhasználónak a kapott hiba alapján.

4.3.6 Guard

Olyan komponens, aminek a segítségével eldönthető, hogy egy útvonal az alkalmazásban megnyitható-e vagy sem. Emellett további logikák is hozzáadhatóak egy oldalra lépéskor, mint például átirányítás. Az alkalmazás sokféle guardot tartalmaz. Ezeket az útvonalak beregisztrálásakor tudjuk nekik megadni, akár többet is.

4.3.6.1 AclGuard

Ennek a guardnak a szerepe, hogy kliensoldali authorizációt hajtson végre az oldalakra lépéskor. Előkészítésnek ide tartozik, hogy az egyes útvonalakon meghatározhatunk kiegészítő konstans adatokat a data blokkon belül. Ebben a data blokkban megadunk minden útvonalnak egy „name” értéket, amit az AclGuard felhasznál, mint kulcs az authorizáció során.

Az authorizáció implementációjában egy konfigurációs ACL modellből olvassa ki, hogy adott kulcsokhoz milyen felhasználói Claimekre van szükség. Ezek a kulcsok tetszőleges szöveges értékek lehetnek, amiket itt a guard a name data értékből olvas ki.

Az így megkapott Claim listából kell, hogy meglegyen legalább egy a felhasználónak, hogy megtekinthesse az adott oldalt.

Emellett a guard további kiegészítő ellenőrzéseket végez el, például, ha van a felhasználónak váróterem vagy játék azonosító megadva, akkor ugorjon át a megfelelő oldalukra, akárhol van éppen.

4.3.6.2 LoginGuard

Egy egyszerű ellenőrző réteg a bejelentkező és regisztrációs oldal felett. Feladata, hogy ha van az oldal megtekintőjének érvényes felhasználója a böngészőben elmentve, akkor dobja tovább az alkalmazásba. Ez abból hasznos, hogy a felhasználóknak ne kelljen belépniük újra, ha egyszer már bejelentkeztek korábban.

4.3.6.3 HubGuard

Feladata, hogy kezelje az egyes hubok elindítását és leállítását a szolgáltatásaikon meghívott startConnection és stopConnection függvények meghívásával.

A vezérlés úgy működik, hogy a data értékek tartalmaznak egy „hub” listát is, amiben szövegesen kulcsok vannak átadva. Ezek a kulcsok alapján a guard eldönti, hogy mik azok a hubok, amiket el kell indítani. Minden más hubot leállít. Ebben benne van, hogy azok a hubok, amik már el vannak indítva nem indulnak újra, hanem üzemelnek folytonosan.

4.3.6.4 GameGuard és LobbyGuard

Egyszerű ellenőrző egységek. Feladatuk, hogy megnézzék van-e a felhasználónak érvényes váróterem vagy játék azonosítója, és az megegyezik-e a megnyitott oldallal. Ha nem megfelelő helyen van a felhasználó, akkor egyszerűen át lesz irányítva a főoldalra.

4.3.7 Directive

Az Angular lehetőséget ad rá, hogy saját magunk is létrehozzunk directive komponenseket. Ezek olyan komponensek, amiket az Angular a HTML kódjának közvetlen irányítására használ Angular kódon keresztül. Ezáltal olyan vezérlő elemeket tudunk hozzáadni az oldalakhoz, amik különálló egységes logikát igényelnek. Használatuk csökkenti a megvalósított HTML oldalak komplexitását.

Az alkalmazásban két directive implementáció került be, de tetszőlegesen bővíthető. Minden directive implementáció esetén meg kell adni egy selector értéket,

amivel az Angular beazonosítja a használt komponenst. Ezen a selectoron keresztül paramétereket is tudunk átadni az implementációnak. Selector megadása például:

```
@Directive({
  selector: '[can]',
})
export class CanDirective { … }
```

4.3.7.1 CanDirective

Feladata és logikája sokban egyezik az AclGuard implementációjával. Szerepe, hogy a felhasználó Claim értékeitől függően jelenítsen meg vagy sem HTML kódrészleteket.

Az implementáció során átadhatjuk paraméterként akár szöveges vagy tömbös értékként az ellenőrizendő Claim értékeket. A logika megvalósítása során fontos szempont volt, hogy egy felhasználónak a Claim listája nem konstans érték az alkalmazás futása során, ezért nem elegendő, ha az alapértelmezett statikus módon hozza létre a megjelölt komponenseket.

A CanDirective tehát dinamikusan tölti vagy üríti ki a megjelölt HTML komponenst a felhasználó tokenének változásával párhuzamosan. Például:

```
<mat-grid-tile [colspan]="12" [rowspan]="1" *can="'Party'">
```

4.3.7.2 RespMatGridTileDirective

Egy olyan segédkomponens, aminek a feladata az Angular Material Grid rendszerében reszponzív beállítás biztosítása. A directive regisztrációja során felülírjuk a Material colspan értéket a saját selectorral, ezáltal a komponens transzparensen használható a Material elemek felett.

A grid elemnek az alapértelmezett logikája, hogy a szülő lista elemen megadjuk, hogy hány oszlopegységből álljon a listánk. Ebbe a listába van annyi oszlopba sűrítve elhelyezve az elem, amennyi értéket megadunk a colspan paramétereként. Ez a logika lett kiegészítve a directive által úgy, hogy további értékek megadásával a HTML kódban felülírhatjuk bizonyos képernyőméreteken ezt az oszlop méretet. Tehát az elem betöltésekor és a képernyő méretének változtatásakor dinamikusan erre a segéd értékre állítja be a méretet.

Ugyanez a logika be lett vezetve a sorméreten is, tehát az elemek magassága is dinamikusan és reszponzívan beállítható az egyes grid elemeknek. Például:

```
<mat-grid-tile [colspan]="8" [rowspan]="4" [md]="12" [smr]="6">
```

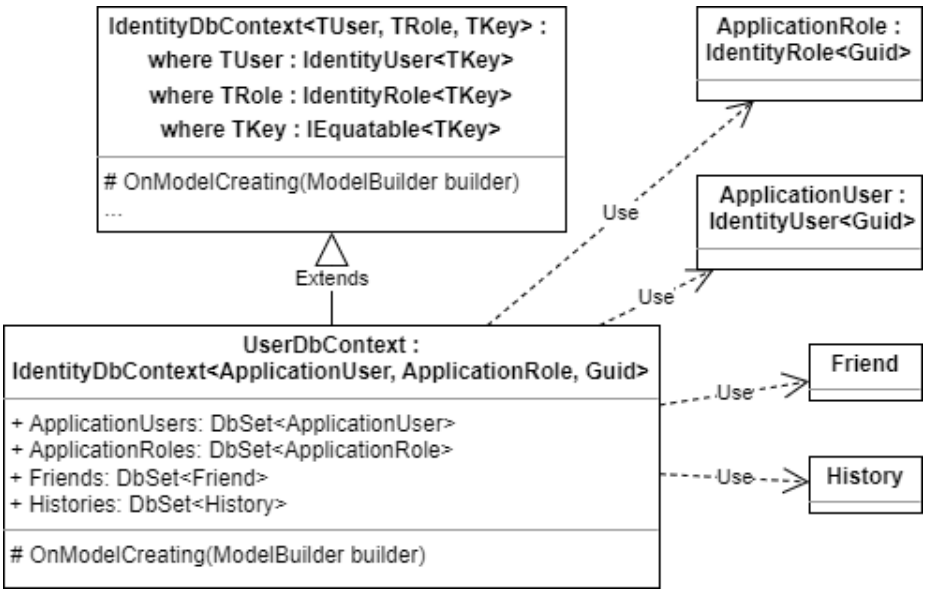
Ebben a példában az elem alapértelmezetten 8 oszlop és 4 sorban van elhelyezve. Ezt Small méreten felülírja 6 sorra, és Medium méreten pedig 12 oszlopra.

4.4 Felhasználókezelés

Az alkalmazásban megvalósított felhasználókezelés és annak a konténere nagyban függött az IdentityServer által szolgáltatott funkcióktól.

4.4.1 Adatbázis és adatelérés

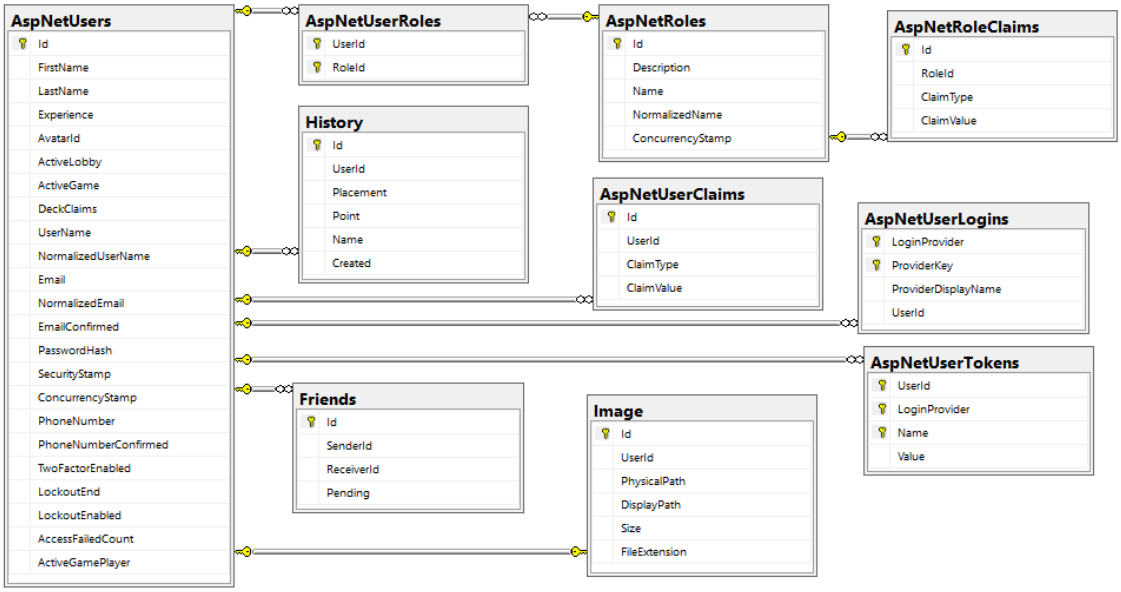
A felhasználókezelő adatbázis kontextusa a többi konténerrel ellentétben nem a EF-ből származó DbContext ősosztályból származik le, hanem ennek az IdentityServeres IdentityDbContext változatával. Ez az ősosztály létrehozza a háttérben a felhasználókhoz szükséges beállításokat, amikhez további konfigurációs lehetőséget biztosít.



13. ábra UserDbContext struktúrája

Az IdentityDbContext alapértelmezetten a beépített IdentityUser és IdentityRole osztályok segítségével hozza létre az egyes felhasználókat és szerepeiket, viszont lehetőséget ad ezeket kiegészíteni. Erre generikusan és az ősosztályból leszármazással ad módot. Minden más felhasználókhoz köthető tábláját a háttérben létrehozza, és lehetőséget ad a logika kialakításánál a felhasználásukra. Ilyen például a kiadott Claimek vagy tokenek perzisztens kezelése.

A felhasználókhoz köthető saját segéd táblák is itt lettek implementálva. Ilyen a barátok kezelése, ahol két felhasználó között alakít ki egy kapcsolótáblát. Ebben a táblában egy-egy azonosítóval jelöli a küldő és fogadó felet, és egy flag bool értékkel tárolja, hogy a kérés el lett-e már fogadva. Egy másik implementáció a játékelőzmények tárolása, ahol időbélyeggel ellátva tartalmazza az egyes felhasználók korábbi játékokban elért eredményeit.



14. ábra Felhasználókezelő adatbázis struktúra

4.4.2 Üzleti logika

Ha a konténer logikája a téma, akkor központi elemnek tekinthetjük mindegyikben a CQRS mintát megvalósító MediatR-ra épülő Handler osztályokat. Ezeknek az osztályoknak a feladata, hogy az API rétegtől kapott kérésekre a DAL rétegen a Repository-k mentén a megfelelő műveleteket hajtsák végre.

A felhasználókezelő konténerben ezért megvalósításra került a barát és felhasználó funkciók mentén egy-egy lekérdezés és parancs kezelő központi egység. Emellett az előzmények kezelésére is létrejött egy központi elem, de csak eseményeinek feliratkozásra.

Külön megemlítendő, hogy a bejelentkezés megvalósítása szerver oldalon nem itt történik meg, mivel az IdentityServer biztosít hozzá az API rétegen egy különálló végpontot. Ez a végpont a „/connect/token” útvonalon található. Itt cserél a felhasználónév és jelszó párosra egy JWT alapú tokent a felhasználónak, amit a kliens kezel le. A token claimekkel kiegészítésére is ad lehetőséget. Emellett nem csak a

bejelentkezésre, hanem a tokenben található adatok frissítésére is felhasználható a végpont a refresh_token segítségével, amit az access_token mellett szolgáltat a bejelentkezés során.

4.4.2.1 FriendCommandHandler

Feladata a barátokhoz köthető parancsok lekezelése. Ebből az egyik a barát hozzáadása. Itt egyetlen függvény kezeli le azt is, ha újonnan adunk hozzá valakit barátnak, és azt is, ha visszaigazoljuk. Új barátkérés esetén létrehozzuk az új rekordot, míg visszajelölés esetén a „Pending” flag értékét állítjuk hamisra, ezzel teljesértékűvé téve a kapcsolatot.

Másik funkció, ami itt van lekezelve az a barát törlése. Mivel a barátkérelem elutasítása és a már létrejött barátság törlése is ugyanazt az eredményt vonzza maga után, ezért egy közös függvényben vannak lekezelve. Ez a barát rekord törlését jelenti.

Harmadik funkció az nem végez változtatást az adatbázisban, csak az offline státuszok értékének frissítésére küld eseményt. Egyébként a másik két funkció is küld eseményeket a kliensnek a sikeres változtatás után.

4.4.2.2 FriendQueryHandler

Ennek a komponensnek az egyetlen célja, hogy az egyes felhasználóknak a barátlistáját szolgáltassa. Ezt úgy teszi meg, hogy azelőtt, hogy odaadná az API rétegnek, különválogatja fajtája szerint a kapcsolatokat. Két felhasználó között háromféle baráti kapcsolat állhat fent, nem számítva, ha nincs semmilyen. Lehetnek teljesértékű barátok, vagy barátkérelmet küldők vagy fogadottak. Eszerint külön-külön listában szolgáltatja tovább őket.

4.4.2.3 UserCommandHandler

Logikai réteg legfontosabb eleme, mivel ennek a feladata a felhasználók létrehozása vagy módosítása. Az Identity alapú belső működéshez a felhasználókhoz és szerepekhez köthető funkciókat megvalósító beépített UserManager<ApplicationUser> szolgáltatás volt felhasználva. Segítségével könnyedén lehet új felhasználókat regisztrálni, módosítani, szerepeiket kezelni vagy esetleg törölni is.

Ezenfelül a saját kiegészítések kezelésére, mint például a felhasználó megvett paklijai, a megszokott Repository alapú implementáció is elegendő volt. Több olyan

parancs is megtalálható itt, ami nem a controllerek felől, hanem más konténerektől eseményeken keresztül érkezett.

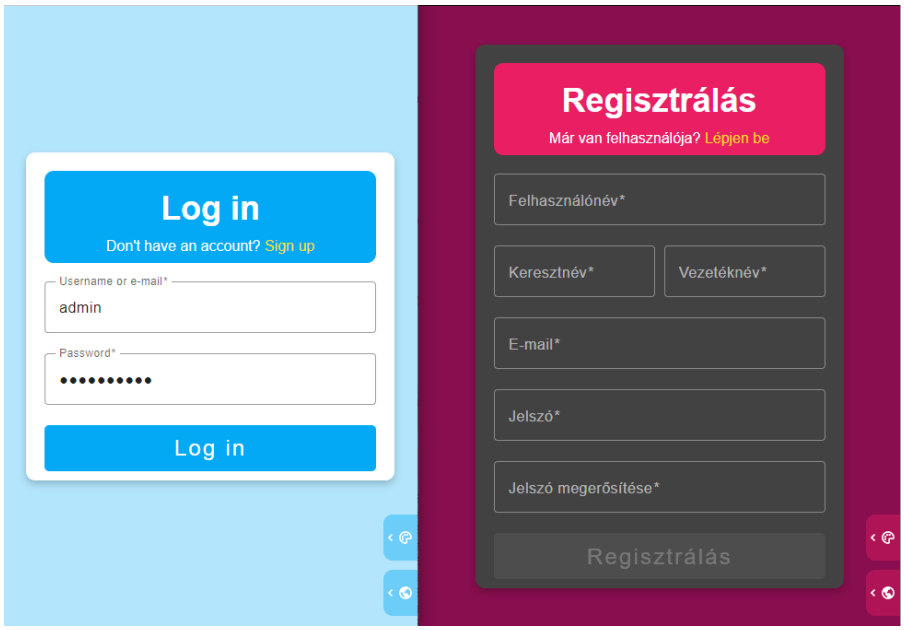
4.4.2.4 UserQueryHandler

Feladata a felhasználónak a közvetlen adatainak a szolgáltatása. Ezek az adatok nagyrészt a bejelentkezés utáni kezdőoldalon kapnak helyet, és emiatt gyakran gyorsítótár segítséget kapnak a felhasználói élmény magasabb szintű szolgáltatásáért.

4.4.3 Felület

A felhasználókezeléshez sorolhatjuk a bejelentkezési, regisztrációs és a kezdőoldal felületeteket. Az oldal megtekintői először itt találják magukat, ahol az Angular Form alapú komponensein keresztül tudják megadni a szükséges adataikat.

A bejelentkezési és a regisztrációs oldal alapjai és dizájnjai megegyeznek és oda-vissza lépési lehetőség van közöttük a fejlécen található linkre kattintva. Mindkettő felületen egy-egy kártya található, ami mutatja milyen értékeket szükséges megadni bennük. Ezekben a kártyákban az egyes értékek mentén kliensoldali validáció is megtalálható. Segítségével a felhasználó rögtön megtudja, ha valamilyen alap hibát vétett a mezők kitöltésében, ezáltal segítséget nyújtva neki. Az elküldési gombok is akkor aktiválódnak, ha minden validáció sikeres. Ilyen validáció például az e-mail cím formátumának ellenőrzése, de olyat például nem tud előre ellenőrizni, hogy létezik-e már felhasználó vele.



15. ábra Bejelentkezési és regisztrációs felület

Bejelentkezés után a felhasználók a kezdőoldalon találják magukat. Ezen a felületen a grid rendszer segítségével reszponzívan vannak az egyes egységek téglalapjai elhelyezve. Ezekben az egységekben találhatóak külön-külön funkcióhoz köthető kártyák vagy egyéb elemek. Van egy fejléc, ami mindig felül található meg, és még két navigációs gomb más oldalakra, amik mindig alul. A bolt navigációjához szüksége van a felhasználónak a Party szerepre, ami nélkül meg sem jelenik a kártyája a grid rendszerben.

Ezek az elemek között található a felhasználó profilja és az előzményei, amik a képernyőmérettől függően egymás alá csúsznak, így kis képernyőn is olvashatóak maradnak a bennük található adatok. A felhasználó profilján képernyőmérettől függően elrendezve található meg az általános adatai, mint a neve, e-mail címe vagy profilképe. Emellett megjelenik neki, hogy a játékok során hány pontot gyűjtött össze, és hogy milyen szerepe van aktiválva. Elegendő ponttal itt lehet rögtön aktiválni is a Party játékmódhoz szükséges szerepet, ami azonnal életbe lép utána a szerver segítségével.

A felhasználóhoz köthető változtatások kezelése a stílus és nyelvkezelővel hasonlóan egy kinyíló fül mögé lett szervezve. Ez azt segítette elő, hogy a kezdőoldal letisztultabb maradhat kisebb és nagyobb képernyőn is, és akár más-más oldalakról megtekintve is rögtön elérhető opciót ad. Ezek a funkciók a kijelentkezés, profilszerkesztés és a felhasználó törlésének indítása.

16. ábra Kezdőoldal felülete

Bejelentkezés után az általános kinyíló elemek mellett lehetősége van a felhasználónak a barátkezelő felület lenyitására is, ami a jobb felső sarokban jelenik meg

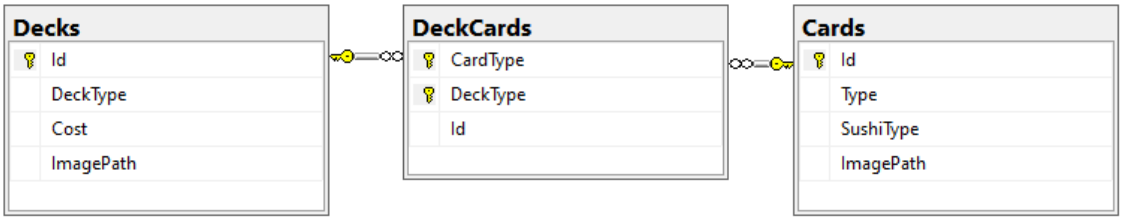
bármelyik oldalon, ha van neki érvényes authentikációs tokene. Ebből a gombból lenyitva tudjuk böngészni a 3 féle típusú barátainkat vagy barátjelöltjeinket. Látjuk, hogy mikor vannak aktívak, törölhetjük vagy elfogadhatjuk őket, vagy hozzáadhatunk új felhasználót.

4.5 Bolt kialakítása

Az alkalmazásban kialakított bolt elég egyszerű alapokra épít, mivel funkciói részben túlmutatnak a konténerének a határain. Elsődleges feladata a boltfelülethez az adatok nyilvántartása, emellett az elvégzett műveletek validálása és a megfelelő irányba továbbítása.

4.5.1 Adatbázis és adatelérés

A konténerek közül a legegyszerűbb adatbázis struktúrával rendelkezik. Egyetlen feladata, hogy a bolt tartalmát szolgáltassa a felületnek, így csak az egyes paklik alap adatai találhatóak meg az adatbázisban.



17. ábra Bolt adatbázis struktúra

Úgy, mint a felhasználókezelőnél a profilképekkel, itt is foglalkozunk fájlok kezelésével. Annyi a különbség, hogy mivel itt csak az egyes paklik illusztrációit tároljuk el, ezért nincs lehetőség módosításra, csak lekérdezésre. Tehát nincs fájlkezelés teljeskörűen bekötve, csak a lekérdezésük engedélyezése az alkalmazásban.

A fájlok bevezetése látszik az adatbázis struktúrából is, mivel a felhasználásuk megkönnyítéséért az egyes rekordokon fel van tüntetve, hogy az illusztrációjuk milyen útvonalon találhatóak.

4.5.2 Üzleti logika

A boltkezelő logikája az adatbázis sémával párhuzamosan elég vékony réteget tesz ki. Tartalmaz egy-egy MediatR kezelő osztályt a lekérdezésekhez és parancsokhoz, amiken keresztül kommunikál az API réteg a konténerben.

4.5.2.1 ShopQueryHandler

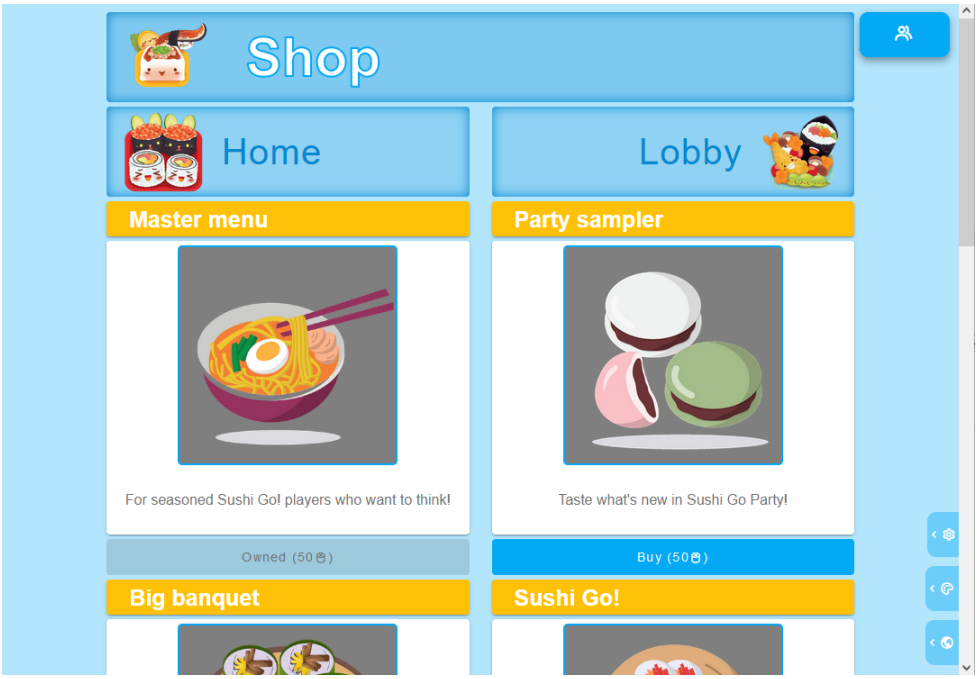
Kétféle lekérdezést kezel. Egyik a paklik teljes listájának lekérdezése, amit a boltfelületen akarunk majd felsorolni. A másik pedig egy specifikus paklinak a lekérdezése az enumerációs típusa alapján. Megvalósított logikájuk egyszerű, így különösebb figyelmet nem igényel.

4.5.2.2 ShopCommandHandler

A konténer alapvetően nem kezel változtatást az adatbázisán, mégis van két funkció, ami ide köthető. Ez a bolt használatához való jog megvásárlása, és a boltban pakliknak a vásárlása.

Mindkét funkció egy jelet küld a felhasználókezelőnek a RabbitMQ segítségével, hogy frissítse a felhasználó adatait és jelezzen az aktív klienseinek. Ezeken a funkciókon még a boltkezelő oldalon validációk találhatóak. Ez egy-egy validáció, hogy az aktuális felhasználónak van-e elegendő pontja a vásárlásra. Ezt a konténer a tokenből kiolvasott Claimek alapján állapítja meg. Ezután a felhasználókezelő oldalon is ráellenőriz a biztonság kedvéért a valódi adatok alapján. A paklivásárlás ezenkívül használ egy további validációt arra, hogy ne próbáljon megvásárolni olyan paklit, ami megvan már a felhasználónak. Ezt is hasonlóan a Claimek segítségével.

4.5.3 Felület



18. ábra Bolt felület

A boltkezeléshez egyetlen felület tartozik. Itt egy kártyás rendszerben látjuk felsorolva a játéknak az egyes paklijait. A paklilistát felülről és alulról közrefogja egy-egy sor a más oldalakra való navigációval, hogy az aljáról se kelljen feltekerni.

A paklik kártyái reszponzívan vannak elosztva, tehát aszerint van az egy sorban megjelenő darabszámuk elosztva, hogy mekkora a képernyő mérete. Ezáltal kicsi és nagy méretben a legkényelmesebb módon böngészhető.

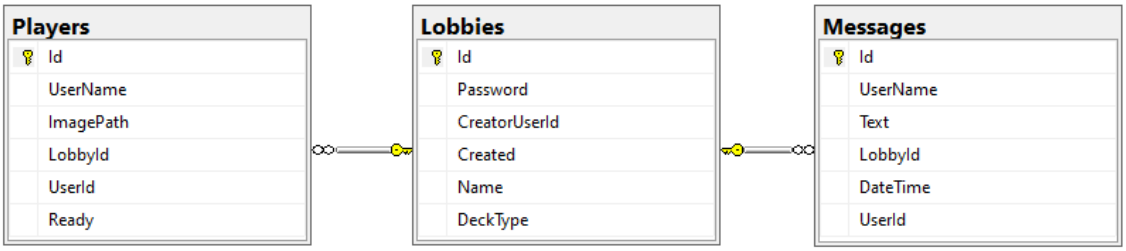
Az egyes kártyák teljesmértékben nyelvesítve és sötét-világos téma szerint stílussal ellátva vannak. Legfelül található a pakli neve, alatta pedig egy egyszerű illusztráció a beazonosítására. A kép alatt található a hivatalos leírása a pakliknak angolul és magyarul is. Végül a legalján található a gomb, amivel megvehető az adott pakli. Ez a gomb kliensoldali validációval van ellátva. Tehát ha egy felhasználó megvette már a paklit, vagy nincsen rá elég pontja, akkor egy magyarázó üzenettel ellátva nem engedi, hogy megvegye. Ehhez a validációhoz a kliens a tokenből könnyedén ki tudja olvasni a szükséges információkat.

Az egyes paklik megvétele után rögtön visszajelzést is ad a megvételről és az új validációval kerülnek kiszámolva a paklik gombjainak szabályai.

4.6 Váróterem megvalósítása

A váróterem megvalósításánál egy komplex problémát kellett megoldani. A konténernek kezelnie kellett a létrehozott várótermek listáját, a hozzájuk való csatlakozást vagy kilépést, és a bennük található beszélgető és más-más funkciókat is. Mindezt a klienssel összefogva kétoldalú kommunikáció segítségével.

4.6.1 Adatbázis és adatelérés



19. ábra Váróterem adatbázis struktúra

A váróterem adatstruktúrája összesen 3 táblából áll, mindegyik az alkalmazás egy-egy komponensét reprezentálva. Vannak a várótermek, amiket kezelünk az

alkalmazásban. Ezeknek van egy neve és egy jelszava, egy tulajdonosa, egy létrehozási dátuma és az aktuális kiválasztott paklinak az azonosítója. Ezek mind olyan adatok, amiket a létrehozás során kap meg a szerver, és később csak a beállított paklinak a módosítására lesz lehetőség.

Az egyes várótermekben kezeljük a csatlakozott felhasználókat, akikből a játék játékosai lesznek. Náluk csak az alap felhasználói adataikat tároljuk, mint a név, azonosító és a profilkép útvonala. Emellett minden játékos kap egy „Ready” flaget, aminek a segítségével tároljuk el, hogy készenállnak-e a játékra.

Az alkalmazásnak egy további funkciója a beszélgetés megvalósítása a termekben, ezért az egyes üzenetek is kapnak egy-egy rekordot az adatbázisban. Ezekben a rekordokban csak alapadatok vannak feljegyezve, minthogy ki, mikor és milyen üzenetet írt az adott teremben.

4.6.2 Üzleti logika

A váróteremkezelő logikában számos és változatos funkció megvalósult. Ezeket fedi le a külön-külön kialakított lekérdezés, parancs és eseménykezelő központi osztályok a várótermek és üzenetek funkciói mentén.

Az itt megvalósított funkciók során meghatározó szempont volt az élő és gyors információcsere a klienssel. Emiatt részletes támogatást kapott a gyorsítótárral és a kétoldalú kommunikációval összekapcsolt eseményekkel is.

4.6.2.1 LobbyQueryHandler

Tartalmaz egy lekérést a várótermek listájára, és egyet egy bizonyos terem lekérdezésére. Mindkettő fölött gyorsítótár támogatás van bekapcsolva, így a felhasználók gyorsan megkapják a kért információt. A lista lekérdezés során a kliensen az aktuális termeket akarjuk megjeleníteni, ezért csak a nevük és azonosítójuk kerül lekérdezésre. Az azonosító alapú lekérdezésben pedig egy részletes státuszt kapunk a teremről, viszont már validáció is védi, amivel csak a saját termünket kérdezhetjük le.

4.6.2.2 LobbyCommandHandler

A várótermeknél megvalósított módosításoknál meghatározó volt, hogy a módosítások során ügyelni kellett arra is, hogy a felhasználókezelés felé is szólni kell a változásról, és a gyorsítótárnak is frissítenie kell magát, ha a terem változott.

Egy terem létrehozása során a felhasználó egyedül a nevét és jelszavát adja meg. Ezt a nevet levalidálja, hogy nincs-e ilyen aktuális terem az adatbázisban. Ezután kiegészíti a dátummal, az alapértelmezett paklival és a saját felhasználójával. Ha minden elkészült, akkor küld egy jelet a MediatR eseménykezelőn keresztül, és egy jelet a RabbitMQ-n keresztül, hogy csatlakozott egy teremhez.

Minden egyes parancs küld egy jelet a MediatR eseménykezelőnek. Emellett a felhasználóhoz köthető parancsok, mint a teremhez csatlakozás vagy kilépés, az küld a RabbitMQ-n keresztül is a létrehozással párhuzamosan. A parancsok emellett részletes validációval vannak ellátva, ami lefedi az előfordulható jogokat a termek kezelésénél, mint például, hogy a saját termünkön dolgozunk, vagy hogy mi vagyunk-e a terem tulajdonosa. A játékos kirúgása például akkor történhet meg, ha a saját termünkből akarjuk kirúgni, és vagy mi vagyunk a terem tulajdonosai vagy magunkat akarjuk kiléptetni [IValidator].

További módosítás a pakli lecserélése, ami automatikusan leszedi a Ready flaget minden játékosról. Egy másik parancs pedig, ami ezt a flaget cserélgetni tudja a saját játékosunknak. Ezáltal nem kerülnek a játékosok olyan paklival a játékba, amivel nem szeretnének játszani. Emellett megemlítendő, hogy a termek törlése kétféleképpen mehet végbe. Vagy elindul egy törlési parancs, például a játék indítása után, vagy a terem tulajdonosa elhagyja a szobát, ezzel kiléptetve mindenki mást is.

4.6.2.3 LobbyEventHandler

Feladata a parancsok által kibocsájtott események szerveroldali logikájának lekezelése. Ez a termek gyorsítótárának kezelését teszi ki. Ugyanezeket az eseményeket a Hubok is elfogják, amiket a megfelelő útvonalakon küldenek a klienseknek.

Ebben az eseménykezelő osztályban tehát két dolog történhet. Vagy egy konkrét terem gyorsítótárát kell frissíteni, vagy a termek listáját. Tehát a két lehetőség egy-egy segédfüggvénybe lett szervezve, és az egyes események hatására a megfelelő van meghívva. A termek listájánál emellett arra is figyel, hogy létrehozási idő szerint csökkenően legyen eltárolva, ezzel is segítve a kliensoldalon a megjelenítést a későbbi lekérdezése után.

4.6.2.4 MessageQueryHandler

Egyetlen feladata, hogy termenként szolgáltassa az elküldött üzeneteket. Ezen a lekérdezésen gyorsítótár is van bevezetve, így kevésbé terheli meg az adatbázist a rekordok hosszú listájának lekérdezése.

4.6.2.5 MessageCommandHandler

Ez is egyetlen funkciót szolgáltat, ami egy elküldött üzenet rögzítése. Ebben a parancsban a felhasználó egyedül a rögzítendő szöveget és a terem azonosítóját szolgáltatja. A kapott azonosítót ezután a parancs levalidálja, és kiegészíti az aktuális idővel és a felhasználó adataival, hogy utána eltárolja. Legvégül elküld egy eseményt a sikeres üzenetről, amit elkap a szerveroldali komponens és a kliensoldalnak továbbító Hub kapcsolat is.

4.6.2.6 MessageEventHandler

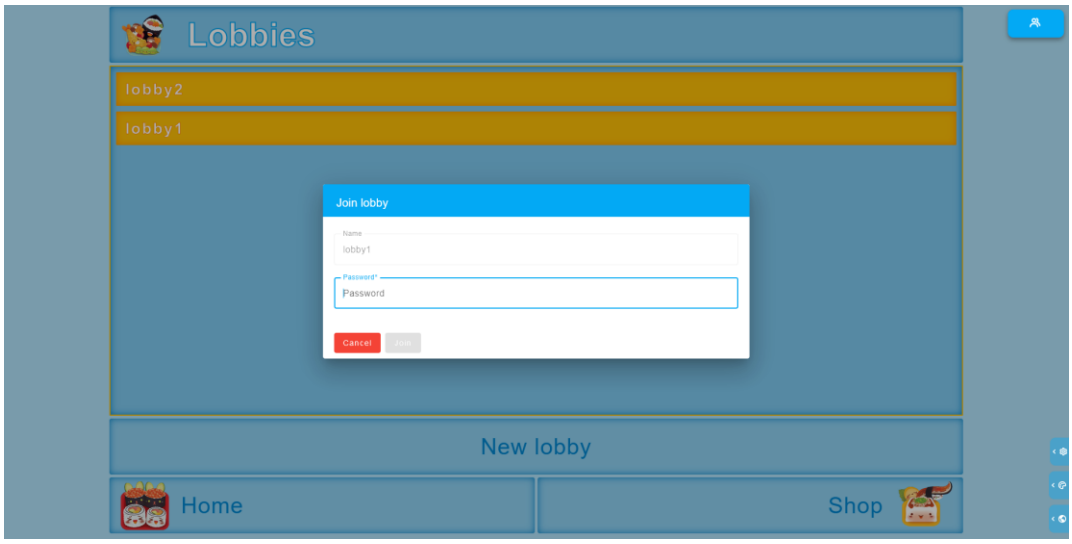
Feladata, hogy elkapja az üzenetek létrejöttének az eseményét. Minden üzenet eseménye után frissíteni akarjuk a gyorsítótárban megtalálható adott terem üzeneteinek a listáját. Hogy ezek a memóriában eltárolt listák ne növekedjenek túl nagyra, egy konfigurációs értékkel meg van határozva, hogy hány elem szerepelhet bennük. Tehát a frissítés során ezzel az értékkel számolva és idő szerint csökkenő sorrendben veszi ki az üzeneteket, amiknek a végére beszúrva az új üzenetet tárolja el újra.

4.6.3 Felület

A várótermek kezeléséhez két felületet lehet kötni. Egyik a termek böngészésére kitalált listafelület, a másik pedig a váróterem tagjainak megjelenített részletes nézet. Mindkét felület aktívan részt vesz a validációban és a kétoldalú kommunikációban is.

A várótermek listanézetén egy letisztult felületet látunk, ahol a navigációk felett egy listadobozt és egy létrehozási gombot találhatunk. Ennek a gombnak a segítségével bárki létrehozhat egy termet a név és jelszó megadásával. Az így létrehozott terembe a létrehozó játékost rögtön be is dobja, és mindenki másnak feltűnteti az újonnan létrejött szobát a listán. A listában az aktív szobák egyedi nevei találhatóak létrehozási sorrendben csökkenően, tehát könnyen megtalálják egymást a játékosok.

Az itt felügyelő hub kapcsolatnak tehát egyedül a szobák létrehozására és törlésére kell, hogy figyeljen. Ezek hatására kiszűri a törlendő termet és betolja a lista elejére a létrejött termet. Mindezt szinkronban a szerver gyorsítótárának szerkesztésével.



20. ábra Váróterem csatlakozás

Az egyes létrejött termekhez a lista elemükre kattintva tudunk csatlakozni. Ennek a dialógusa hasonlóan néz ki, mint a létrehozásének, de egyedül a jelszót kell megadni. Ezt a jelszót pedig a szoba tagjaitól tudjuk elkérni, megakadályozva a kéretlen belépőket.

A termekbe belépve egy komplex, kisebb egységekből álló felületen találják magukat a felhasználók. A felület tetején egy navigációs sáv található. Ezen a sávon találhatóak meg a kiegészítő funkciók, amiket egy adott felhasználó megtehet a teremben. Ez egy belépett felhasználónak a kilépést és a saját Ready flag értékének állítását jelenti. Ezt egészíti ki a terem tulajdonosának két funkcióval.

Egyik, hogy meg tud nyitni egy dialógust, amiben kiválaszthat a saját megvett paklijai közül egyet, hogy azzal induljon a játék. Ezzel automatikusan vissza is veszi mindenkitől a Ready flaget. Emellett a másik funkció magának a játéknak az indítása. Ezen a gombon külön validáció található kliensoldalon is, ami akkor engedi meg a játék indítását, ha minden játékos készenáll, és a paklin megadott határokon belüli játékosszám található a teremben.

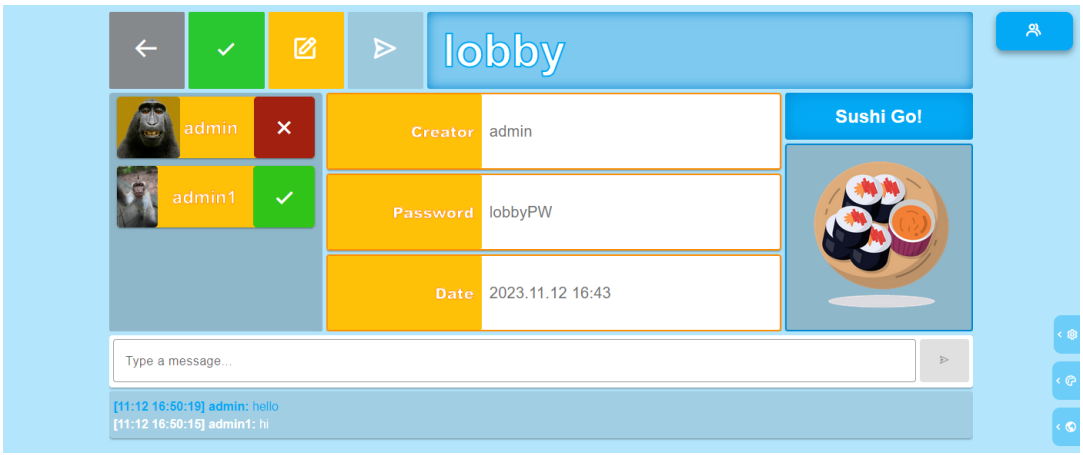
Ezek a gombok mellett található meg a terem neve, ami a képernyő méretétől függően reszponzívan a gombok alá tud csúszni.

Alattuk található a teremnek a játékos listája és az általános adatai, ugyanúgy reszponzívan elhelyezve egymás alatt vagy mellett. A terembe belépett játékosokról élőben követve adja vissza az információkat. Ez azt jelenti, hogy arról is rögtön visszajelzést kapunk, ha valaki belépett vagy elhagyta a termet, és arról is, ha a készenálló értékét billenti át. Ebben a listában az egyes leendő játékosokról látjuk a profilképüket és

a felhasználónevüket a készenálló ikonjuk mellett, amivel be tudjuk azonosítani ki lépett be a termünkbe. Ha esetleg nem férne ki a belépett játékosok listája, akkor egyszerűen görgethető felületté alakul a dobozuk.

A terem általános adatai közé tartozik a kiválasztott pakli is, amit a terem tulajdonosa tud lecserélni. Ennek a cseréjét a többi a játékos is élőben tudja lekövetni, a pakli nevének és az illusztrációjának változásával. A terem neve és jelszava minden belépett felhasználónak elérhető, így további játékosok is könnyedén meghívhatóak a játékba. Persze a játék indítása és a játékosok eltávolítása a tulajdonos kezében van.

A felület alján található egy chat felület. Itt a szoba tagjai tudnak egymással élőben beszélgetni. Tetején egy szöveges mező található, amiből az elküldött üzeneteket a szerver visszaküldi időbélyeggel ellátva a terem minden tagjának. Ezeket az üzeneteket jeleníti meg a mező alatt egy görgethető felületen. Az üzenetek a kényelem kedvéért idő szerint csökkenő sorrendben jelennek meg, és új üzenet hatására automatikusan a konténer tetejére görget az oldal, ha olyan hosszú, hogy görgethetővé alakul.



21. ábra Váróterem felülete

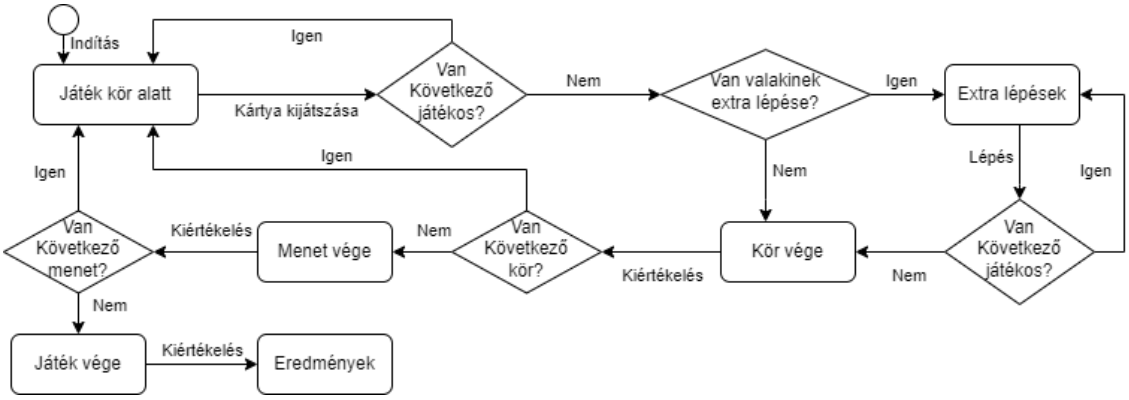
4.7 Játék

Az elkészült alkalmazás legfontosabb és legvastagabb komponense. Feladata a játék logikájának a megvalósítása, ami komplex műveleteket igényelt kliens és szerveroldalon is.

A megvalósítás során a játékszabályt pontosítva a játék fázisokra lett osztva. A játék kezdetén áll egy kör fázisból, amiben a játékosok a lapjaikat tudják kirakni, opcionálisan további interakciókkal kiegészítve. Ez a fázis addig tart, amíg minden játékos sorra nem került és ki nem rakott egy-egy lapot. Ezután a szerver leellenőrzi, hogy

van-e olyan játékos, aki jogosult további extra lépésre a kiválasztott lapja vagy az asztala alapján. Ha vannak ilyen játékosok, akkor teszünk még egy kört csak ezekkel a játékosokkal, ahol ki tudják játszani ezeket a speciális akciókat, ha akarják. Ezután a játék a kör végére érkezik. Minden kör végén a játékosoknak véglegesíti a lerakott lapjait, és körbeadja a kezükben maradt lapokat. Ennek az elindítására az első játékos kap jogot. Az így lejátszott körök végén, mikor nem maradt több lap, akkor a szerver a menet végi fázisba érkezik, ahol kiszámítja a szerzett pontokat és újrakeveri a paklit. Ezt ugyanúgy az első játékos indítja. Legvégül, a 3. menet végén jutunk a játék végi fázisba, ahol az első játékos vezérlésére elvégzi a számításokat, és a véglegesített eredmények fázisba állítja a játékot.

Egy bevezetett segédfunkció, hogy az egyes kiértékeléseket nem kötelező az első játékosnak elvégeznie, mivel egy időlimit után a háttérben elvégzi a szerver is egy feladatban, amiről értesíti a klienseket. Erről a felhasználók egy visszaszámlálót is látnak.



22. ábra Játék menete

4.7.1 Adatbázis és adatelérés

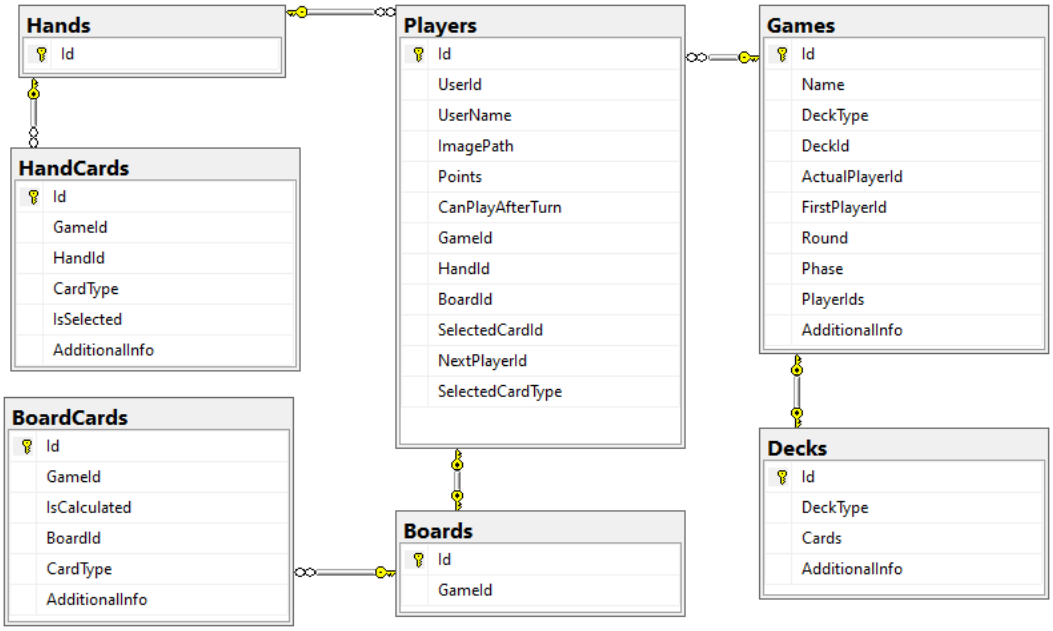
Az egyes konténerek közül a legvastagabb adatstruktúrával rendelkezik a játéknak a kezelése. A struktúra központi egysége a Games tábla, ami az egyes játékokat reprezentálja. Itt vannak eltárolva a játékhoz köthető állapotok és adatok, mint a menet száma, fázis vagy az aktív és kezdő játékos azonosítója is. Több táblán megtalálható az AdditionalInfo mező, amiben JSON-be fordított szabad kulcs-érték formájú kiegészítő információk találhatóak.

Az átláthatóság kedvéért a használt pakli egy külön táblába lett kivezetve, amibe megadott típus szerint töltjük be a lapokat és az esetleg szükséges kiegészítő információkat. Maga a kártyák listája a Cards nevű egyetlen mezőbe van JSON-ként

befordítva. Viszont csak a lapoknak a típusának a listája található meg benne, amivel minimalizálható az adatbázis terhelése. Ezekhez a lapokhoz, ha szükség van altípusra, mint például melyik lap hány pontot ér, akkor azt az AdditionalInfo mezőből tölti be az alaptípussal együtt.

Az egyes játékokhoz csatlakoznak a játékosokat reprezentáló rekordok. Ebben vannak eltárolva a játszó felhasználók adatai és nekik a játék további elemeivel kiépített kapcsolatainak azonosítói. Emellett tartalmaz további, a játékhoz köthető adatokat is, mint az eddig összegyűjtött pont, vagy hogy van-e extra lépése az adott körnek a végén.

A kártyák reprezentálására két tábla van kialakítva. Egyikben található a játékosok kezében található lapok, a másikban pedig az asztalra maguk elé kijátszott lapok. A kettéválasztást az indokolta, hogy más-más kiegészítő információt igényel a használatuk, és az utólagos szűrés is egyszerűbbé, hatékonyabbá válik.



23. ábra Játék adatbázis struktúra

A kártyák és játékosok közötti kapcsolat nem közvetlenül lett kiépítve, hanem egy kéz és asztalt reprezentáló kapcsolótáblán keresztül lettek összekötve. Ennek az az oka, hogy a megvalósított játéknak egyik alapeleme, hogy a körök végén a játékosok körbeadják a kezükben található lapokat. Ez adatbázisban azt jelentené, hogy minden egyes kézben található lapnál le kell cserélni a külső kulcsokat az új játékosokra. Ennek az áthidalására lettek bevezetve a kapcsolótáblák. Ezekkel a játékosok egy-egy kapcsolatban állnak, és a körök végén a játékosokon található kezekre mutató kulcsokat

kell csak cserélgetni. Emellett a kártyák lekérdezését sem lassítja, mivel azokat egy külön kérésben a kéz azonosítójának segítségével kérdezi le.

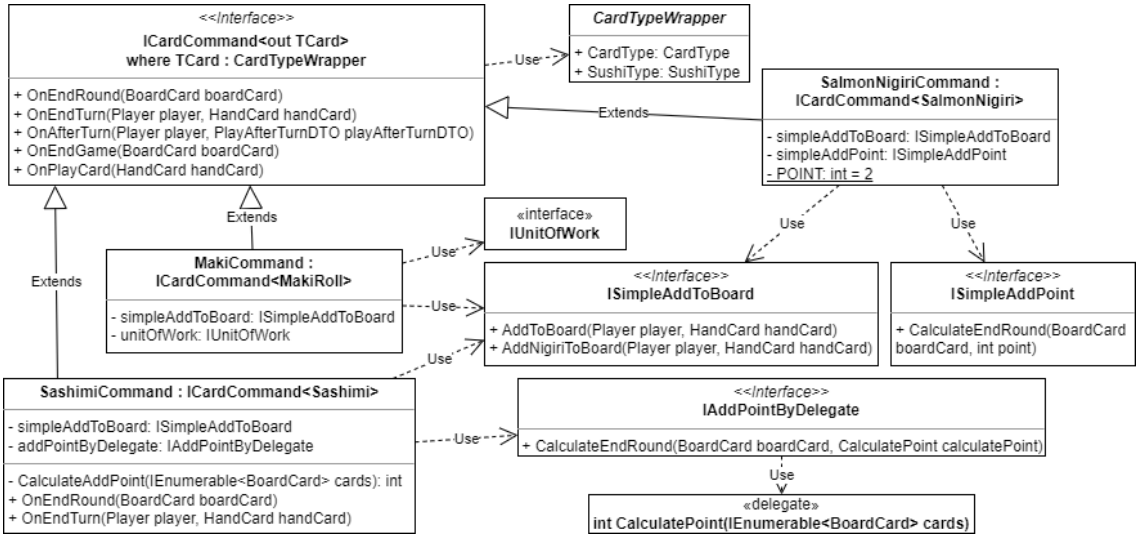
4.7.2 Üzleti logika

A játék logikájának megvalósításánál a legnagyobb akadály a kártyák működésének megvalósítása volt, mivel számos és változatos implementációt igényelnek. Miattuk is lettek bevezetve a fázisok a játékban, hogy valamilyen rendszerezett logika szerint tudjanak működni.

4.7.2.1 ICardCommand

Fontos szempont volt, hogy a kártyákhoz köthető logikák típustól függetlenül egységesen lehessenek felhasználhatóak. Ehhez lett bevezetve egy központi struktúra, ami generikusan megfogalmazza, hogy az egyes események során a kártyákkal mi a teendő. Itt a generikus érték a kártyatípusokat becsomagoló wrapper osztály, aminek a segítségével könnyedén tudunk típustól függően új kártya implementációkat hozzáadni.

Az ilyen események a körök, menetek vagy játék végén a kiértékelés, vagy a kijátszás és extra lépés során a típustól függő opcionális implementáció. Ezzel a megoldással könnyedén tudjuk meghívni minden kártyára a saját szabályát az egyes események során.



24. ábra ICardCommand struktúra 3 példával

Mivel a kártyák használatakor gyakran vannak olyan esetek, amikor megegyező vagy bizonyos logikával eltérő műveleteket hajtanak végre, ezért pár implementáció különálló segédkomponensekbe lett kihelyezve. Ezeket a komponenseket injektálással

tudják felhasználni tetszőlegesen az egyes kártya kezelő osztályok, így könnyedén bővíthetőek további funkciókkal vagy komponensekkel. Három ilyen segédkomponens van implementálva. Egy, ami egy adott kártyának az asztalra helyezését valósítja meg, egy, ami adott pontot ad hozzá egy játékoshoz, és egy, ami egy paraméterül kapott függvény alapján ad hozzá pontot egy játékoshoz. Az olyan kártyáknál, amik ezeken túlmutató komplex logikát igényelnek a UnitOfWork injektálásával könnyedén meg tudják valósítani a saját módosító és lekérdező logikájukat.

Az asztalra kihelyező segédkomponens ki lett egészítve egy külön Nigiri típusú kártyák lehelyezésére szolgáló AddNigiriToBoard függvénnyel, amiket a Nigiri kezelő osztályok használnak fel. Ennek az oka, hogy ha egy játékos lerak maga elé egy Wasabi kártyát, akkor a következő Nigiri kártyája dupla annyi pontot ér. Ennek az ellenőrzését és a lerakott kártyán való jelölését hajtja végre külön ebben a függvényben.

Ahhoz, hogy a wrapper osztály segítségével tudjuk példányosítani a kezelő osztályokat, emellé szükség volt egy segédfüggvény implementálására is. Ez a függvény az IServiceProvideren fogalmazza meg a GetCommand kérést, ami az alkalmazásba regisztrált osztályok közül generikusan létrehozza a keresett kezelő osztályt, ha létezik. Ez a segédfüggvény akármelyik parancs során meghívható. Használati példa:

```
var command = _serviceProvider.GetCommand(card.CardType.GetClass());
if (command != null)
{
    await command.OnEndRound(card);
}
```

4.7.2.2 CardCommandHandler

Feladata a kártyák funkcióinak kezelése. Ebbe beletartozik a kártya kézből lerakása, vagy egy játékos extra lépése vagy annak kihagyása. Egy kártya lerakása során a legtöbb kártya esetén még csak annyi történik, hogy megjelöljük, hogy a kör végén ezt a kártyát fogjuk felfordítani, és továbblépünk a következőre.

A kérés során először történik egy validáció, amivel leellenőrzi, hogy a megfelelő játékos a megfelelő fázisban választott lapot. Ezután megjelöli a lapot, és felrakja az opcionális kapott plusz információkat az AdditionalInfo blokkba, végül meghívja az OnPlayCard függvényt a kártya kezelő osztályán, ami többségében opcionális logikát tartalmaz. A kártya kirakása végén kiszámolja, hogy ki a következő játékos. Ilyenkor akár fázist is lép, ha extra lépések köre kezdődne vagy vége van a körnek. A folyamat legvégén pedig elküldi a szükséges eseményeket a feliratkozói felé.

Az extra lépések logikája hasonlóan épül fel, mint a kártya kiválasztása. Különbségek vannak a validált fázis és a kártya kezelő osztályon meghívott függvény között. A kihagyás funkciója pedig az extra lépés helyett a validáció után csak a következő játékost keresi meg és állítja be.

4.7.2.3 GameCommandHandler

Elsődleges feladata a játékok létrehozása és törlése, emellett az egyes fázisok végén a kiértékelések lefolytatása. A játék létrehozása során a kapott pakli típus szerint kever egy paklit és létrehozza a játékosokat. Ezután a kevert pakliból rögtön szétosztja a játékosok között a felső lapokat, a kiegészítő információkkal együtt. Az egyes játékosokhoz létrehozza az asztalukat, kialakítja a játékosok közötti sorrendet és beállítja a kezdő és aktuális azonosítót a játék elmentése előtt. A legvégén küld egy jelet a RabbitMQ-n keresztül a felhasználókezelőnek, hogy a játék elkészült.

Az első kiértékelő parancs a körök végén történik, ahol az egyes játékosok kiválasztott lapjainak a lekezelése történik. Minden kiválasztott kártyán megtörténik az OnEndTurn meghívása, ahol a legtöbb kártya az ISimpleAddToBoard egyik függvényét hívja meg. Azután, hogy megtörtént az egyes kártyák kiértékelése, a játék leszimulálja a paklik körbeadását a HandId azonosítók lecserélésével. Ha ez volt az utolsó kör, tehát nincs több kártya, amit körbe tudnának adni, akkor pedig átlép a menet végi fázisba, amiről eseményt is küld.

A második kiértékelő parancs során veszi az asztalokon megtalálható lapokat, amiknek a már említett kártyakezelő osztályon keresztül kielemzi melyik mennyi pontot ér. Itt az elemzőfüggvények elég komplikáltak tudnak lenni, mivel függhetnek más lapoktól is, legyen az a sajátunk vagy más játékos asztalán. Ezért is segítség a kiemelt kártyakezelő implementáció, mivel letisztult kezelést ad az elemzésnek. Az elemzések után letisztítja az asztalokat a desszert típusokon kívül, mivel azok a játék végén pontozódnak. Menet végén pedig, ha nem az utolsó menet, akkor újrakeveri a paklit és kiosztja a lapokat, mielőtt elkezdené a következő kört. Ha az utolsó menet volt, akkor pedig a játék végi fázisba állítva küldi el a megfelelő eseményeket.

A játék végi elemzés során beszámításra kerülnek a desszert lapok is, és letakarításra kerülnek az egyes asztalok. Mivel a korábbi pontok fokozatosan kerültek beszámításra a menetek mentén, így ezután az egyetlen teendője, hogy átállítsa a fázist az eredmények megjelenítésére.

Mindhárom kielemző parancs elején található validáció, amivel kimondja, hogy a köröket kezdő játékosnak kell továbblépnie velük. Viszont ez elég kényelmetlen lehet minden alkalommal egy játékosnak, ezért bevezetésre került, hogy ha egy idő után nem lép tovább, akkor a szerver automatikusan elvégzi a Hangfire segítségével.

4.7.2.4 GameQueryHandler

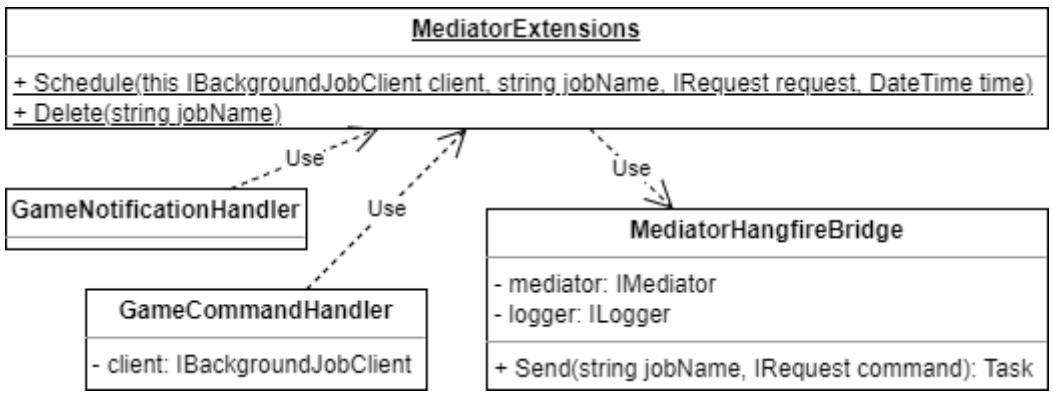
Feladata a játékosoknak a saját játékuknak és kezüknek a szolgáltatása. A kérések gyorsítótárazva vannak, amiket az események segítségével tart naprakészen. Mivel a játékosok szabadon megtekinthetik mások asztalait, ezért a játék lekérdezése asztalok kártyáit is tartalmazóan mélyen történik meg. A kezek lekérdezése viszont külön történik meg, és csak a saját kártyáinkat tartalmazzák, így biztosan csak mi ismerjük a tartalmukat.

4.7.2.5 GameNotificationHandler

Az osztály egyik feladata, hogy elkapja az egyes játékoknak a frissítési vagy törlési eseményét, amik alapján a gyorsítótár kezelését valósítja meg a játékok azonosítóival címezve. Másik feladata, hogy a kiértékelő fázisokba lépésnek az eseményét kapja el. Ezekben létrehoz egy-egy új parancsot, ami a kiértékeléseket imitálja olyan esetekben, amikor a kezdő játékos nem nyomja tovább időben. Ezeket a parancsokat pedig elküldi a Hangfire-nek, hogy hívja meg 30 másodperc múlva.

4.7.2.6 Hangfire

Háttérben futó feladatok kezeléséhez az előzőekből fakadóan a Hangfire segédkönyvtár lett bevezetve. Ehhez a komponenshez különálló adatbázis és adminisztrációs dashboard is tartozik a játék kezelő konténeren. Lehetőséget ad konkrét időpont szerinti vagy ismétlődő feladatok elvégzésére is, de az alkalmazásban egyedül az aktuális időtől 30 másodperc múlva lefutó feladatok elvégzésére volt kihasználva.



25. ábra Hangfire használata

A Hangfire alapértelmezetten függvények futtatására ad lehetőséget, de az kevésbé átlátható lenne a parancs objektum alapú implementációk mellett. Erre lett bevezetve a MediatorHangfireBridge összekötő struktúra, aminek át tudjuk adni a meghívandó parancsokat. Az eseménykezelő osztály egy MediatorExtensions segédfüggvény segítségével indítja el a feladatokat. A parancskezelő ugyanúgy a segédfüggvény segítségével keresi ki és állítja le a parancsokat, ha a szerveroldali időzített futtatása előtt a játékos kezdeményezi a parancs hívását.

4.7.3 Felület

A játék felületének kialakításakor a legfontosabb szempont volt, hogy a játékosok minél könnyebben elérhessék a játék állásáról az információkat. Mivel egy webes felületen limitáltabbak a lehetőségek, mint egy kiterített asztalon, így különösebb figyelmet kapott, hogy hol és mi található meg.

26. ábra Játék felülete

A játék felületének kialakításakor két alkomponens lett bevezetve. Van egy komponens, ami a kártyák megjelenítéséért felel. Mivel egységes, de komplex saját logikát követel, mint például a pontszámos kártyák különválasztása, ezért jobban megéri őket egységesen külön elemként felhasználni.

Ezenkívül külön lett választva a játékosok kezét megvalósító komponens, mivel a háttérlogikája külön lekérdezést igényel, így letisztultabb marad a kialakítás. A játékosok kezét az oldal alján megjelenő sáv reprezentálja. Itt egymás mellett, lapozhatóan jelennek meg a kártyák, egy alap animációval kiegészítve. Mivel a kártyák akcióinál gyakran szükség van az asztalra kirakott lapjainkra is, ezért a sáv mellett van két gomb, aminek a segítségével lehet váltogatni a kéz és asztal nézetet, így az akciók során is könnyedén lehet választani kézből és asztalról is.

A kártyák kezelésénél egy segédszolgáltatás feladata, hogy figyelje a játék és a játékos kezének az állását, ami alapján kiszámolja, hogy milyen lépéseket tehet meg a felhasználó. Röviden egy bonyolultabb kliensoldali validáció központi elvégzése. Ennek az eredményeként jelenik meg a pálya egyes részei külön megvilágított stílussal, jelezve a felhasználónak, hogy ezt és ezt tudja megtenni. Ez azoknál az eseteknél is hasznos, amikor egyes kártyák akciói bonyolultabb kliensoldali logikát igényelnek. Például lerakáskor választani kell egy lapot az asztalról, amit lemásoljon, vagy extra lépés során ki kell cserélni egy lappal a kézből. Ezek használatára a plusz stílussal ad útmutatót a játékosnak, így kevés ismerettel is gyorsan ki lehet igazodni.

A kéz felett található a játéktér. Tetején, a barát ikonnal egyvonalban mutatja az aktuális menetet és fázist. A felső sáv és a kéz között található maga a játék. Ezen a területen jobboldalt egy görgethető listában vannak összeszedve a játékosok a kezdőbetűjükkel és profilképükkel megkülönböztetve. Ezen a listán egy játékost kiválasztva jelenik meg középen az adott játékos asztala. Ezáltal az egyes játékosok asztalait a játékosok szerint tudjuk böngészni. Az egyes játékosok gombjai alatt emellett azt is látjuk, hogy az adott játékos eddig mennyi pontot gyűjtött össze.

A játék során az asztal frissítésének eseményét a szervertől elkapva módosul a tábla. Ilyenkor gyakran változik az aktuális játékos is, ezért, hogy kicsit gyorsabbnak tűnjön a játék, egy egyszerű animáció is be van vezetve, ami leszimulálja a játékosok sorrendjének az újra rendezését, amivel mindig az aktuális van legfelül. Emellett át is rakja az aktívan megtekintett játékos tábláját az aktuálisra. Ilyenkor az egyes játékosok a saját lapuknak a listáját is frissítik, ami egy becsúszó fokozatosan késleltetett animációt kapott. Tehát az asztal állását közösen kapják meg az eseményen keresztül, de a kezük tartalmát önmaguknak frissítik a játékosok.

A játék végén az eredmények fázisban a kiürített játék felett egy összesítő eredménytáblát jelenít meg a játékosoknak, ahol a helyezéseket és pontokat látják, és a

tulajdonosnak van joga a végső törlésére. Magát a játékot akármikor tudja a tulajdonos törölni, de akkor nem számít bele a játék az egyes játékosok pontjai és előzményei közé.

4.8 Tesztelés

4.8.1 Unit tesztek

A szerver konténereihez unit tesztek is elkészültek, amik különálló projektekként lettek létrehozva az egyes konténerek mentén. Az elsődleges feladatuk az üzleti logika tesztelése volt, így a Repository komponensek használata a Moq segédkönyvtár segítségével lettek helyettesítve. Potenciálisan hosszan lehetne tesztelni az elkészült alkalmazást, de a projektnek nem fő témája a tesztelés, így csak részlegesen készült el. Tesztelve lettek a felhasználókezelő parancsok és a kártya kezelő osztályok.

4.8.2 E2E tesztek

End-to-end tesztek segítségével lett letesztelve a kliensoldali alkalmazás, ami a Cypress alapú tesztek során egy felhasználó lépéseit szimulálja le. Az alkalmazásban részletesen le lettek tesztelve a felhasználói funkciók és a várótermek kezelése.

A tesztek kialakításához 3 kiegészítő függvény lett kialakítva. Két függvény, ami a bejelentkezés és regisztráció elküldését valósítja meg a kliensoldalt megkerülve, így annak tesztelése nélkül is tökéletesen lefut. Egy további függvény valósult meg a tesztadatok generálására. Ebben a függvényben vesz egy fixture fájlt, és annak a tartalmát felülírja a szükséges értékekkel, amiknek a végére random generált azonosítót helyez el. Ezáltal mindig egyedi felhasználókkal és várótermekkel tesztel. Ez azt jelenti, hogy nem fordulhat elő, hogy az egyes elindított tesztelések összecsúsznak. Ezt a generálást elvégzi a teszt egységek before blokkjában, tehát egységekként egyszer kell megtörténnie.

A felhasználói tesztek során le lett tesztelve a belépési és regisztrációs felületének használata. A tesztek részletesen körbejárták az egyes validációs lehetőségeket és potenciális elkövethető hibákat. Mivel a tesztelés során létrehoz több felhasználót is, ezért a végén minden esetben elvégzi ezeknek a törlését, letakarítását.

A várótermek tesztelésénél a kezelésüknek az alapvető logikája lett megvizsgálva. Tehát a termeknek a készítése, csatlakozása vagy elhagyásának és törlésének a variációi. Mivel nem a felhasználókat teszteli, ezért a before blokkban előkészíti előre a teszteléshez használható felhasználókat, amiket az egység lefutásának a végén töröl is.

5 Összefoglaló

A diplomaterv elkészítése során megvalósítottam egy nagyobb projekt implementációját, ami nemcsak funkciókban, de technológiákban is gazdag volt. Az így elkészült szoftver nemcsak funkcióknak és technológiáknak is gazdag listáját valósította meg. A fejlesztés során mélyebb ismeretet sajátíthattam el olyan technológiákban és módszerekben, amikben vagy volt korábbi projekten szerzett tapasztalatom, vagy a diplomaterv során találtam és ismertem meg. Ezáltal az architektúra és az egyes tervezési minták használata során is nagy befolyása volt a korábbi munkáimnak és önálló kutatásaimnak.

Az alkalmazás kialakításában nagy szerepe volt magának a tervezési fázisnak is. A dolgozat témája magának a játéknak a megvalósítására koncentrált, így a segédfunkciók és a hozzájuk köthető konténerek kialakítása nagyban saját ötleteim alapján lettek implementálva. Itt arra figyeltem, hogy a felhasználói élményt milyen funkciók tudják növelni, emellett milyen funkciók bevezetésével használhatóak fel változatos technológiák és tervezési minták.

A legnagyobb hangsúly a játék logikai elemein volt, ami egy komplex megoldást igényelt. A megvalósításukban megpróbáltam egy hatékony és átlátható struktúrát kialakítani az általam ismert tervezési minták felhasználásával. Fontos volt az implementációban, hogy olyan szerkezet készüljön el, ami a jövőben könnyedén bővíthető további kártyákkal és azoknak a logikájával.

A projekt komplexitásából adódik, hogy az elkészült funkcióknak szerteágazó továbbfejlesztési lehetőségei vannak. A felhasználókezelés komplexitása és hatékonysága is tud növekedni. Emellett a barátkezelés is csak segédfunkcióként készült el, amin van továbbfejlesztési lehetőség, mint például chat lehetőség vagy váróterem integráció kialakítása. Bolt funkcionalitása fejlődhet akár valódi pénz kezelése vagy a kínálat bővítésének az irányába is.

A várótermeknek is növelhető a hatékonysága. Funkciókban például a pakliválasztás bonyolítása, bevonva a többi játékost, vagy kevés játékos esetén más termekkel összevont keresés megvalósítása. A játék implementációja is sok további lehetőséget ad. Bevezethető lenne játékon belüli csevegés, időkorlátok kezelése vagy mesterséges intelligencia alapú belső automata játékosok használatának lehetősége is.

6 Irodalomjegyzék

[1] *Docker alapozó [Online]:*
https://thebojda.medium.com/docker-alapoz%C3%B3-b8efb6aa68e9

[2] *ASP.NET CORE dokumentáció [Online]:*
https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-7.0

[3] *Ocelot dokumentáció [Online]:*
https://ocelot.readthedocs.io/en/latest/index.html

[4] *MediatR wiki [Online]:*
https://github.com/jbogard/MediatR/wiki

[5] *SignalR dokumentáció [Online]:*
https://learn.microsoft.com/hu-hu/aspnet/core/signalr/introduction?view=aspnetcore-7.0&WT.mc_id=dotnet-35129-website

[6] *Redis dokumentáció [Online]:*
https://redis.io/docs/

[7] *RabbitMQ dokumentáció [Online]:*
https://www.rabbitmq.com/documentation.html

[8] *IdentityServer4 dokumentáció [Online]:*
https://identityserver4.readthedocs.io/en/latest/

[9] *Hangfire dokumentáció [Online]:*
https://docs.hangfire.io/en/latest/

[10] *Angular Material dokumentáció [Online]:*
https://material.angular.io/

[11] *NGX-Translate dokumentáció [Online]:*
https://github.com/ngx-translate/core

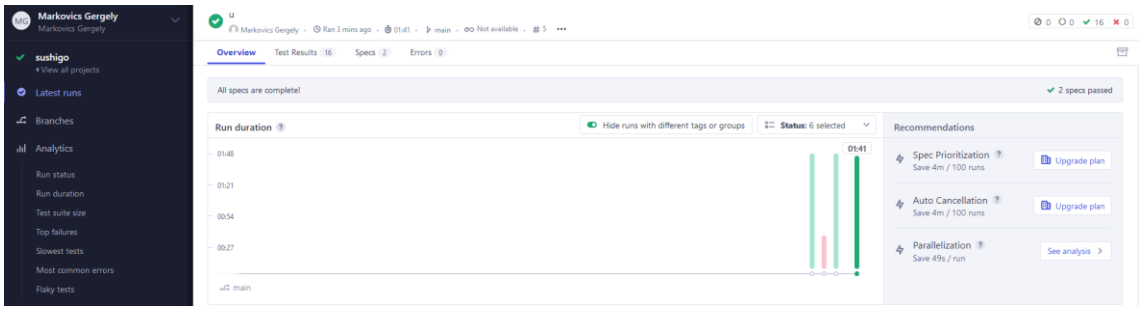
[12] *Cypress dokumentáció [Online]:*
https://docs.cypress.io/guides/overview/why-cypress

[13] *Sushi Go Party társasjáték kellékei és szabályzata*

7 Függelék

27. ábra Felhasznált kártyák

28. ábra Felhasznált paklik



29. ábra Cypress Cloud felület