

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Markovics Gergely

**Kártyajáték  
megvalósítása mikroszolgáltatás alapokon**

Konzulens

Dr. Kővári Bence András

BUDAPEST, 2023

Tartalom

[Összefoglaló 5](#_Toc149643389)

[Abstract 6](#_Toc149643390)

[1 Bevezetés 7](#_Toc149643391)

[1.1 Böngészős- és társasjátékok 7](#_Toc149643392)

[1.2 Motiváció 7](#_Toc149643393)

[1.3 Sushi Go Party! 8](#_Toc149643394)

[1.4 Felhasználói élmény 9](#_Toc149643395)

[1.5 Megvalósítás 9](#_Toc149643396)

[2 Technológia 11](#_Toc149643397)

[2.1 Bevezetés 11](#_Toc149643398)

[2.2 Szerveroldali technológiák 11](#_Toc149643399)

[2.2.1 Docker 11](#_Toc149643400)

[2.2.2 ASP.NET CORE 12](#_Toc149643401)

[2.2.3 Ocelot 12](#_Toc149643402)

[2.2.4 MediatR 13](#_Toc149643403)

[2.2.5 SignalR 13](#_Toc149643404)

[2.2.6 Redis 14](#_Toc149643405)

[2.2.7 RabbitMQ 14](#_Toc149643406)

[2.2.8 IdentityServer4 15](#_Toc149643407)

[2.2.9 AutoMapper 15](#_Toc149643408)

[2.2.10 Hangfire 15](#_Toc149643409)

[2.2.11 xUnit 16](#_Toc149643410)

[2.3 Kliensoldali technológiák 16](#_Toc149643411)

[2.3.1 Angular Material 16](#_Toc149643412)

[2.3.2 SCSS 16](#_Toc149643413)

[2.3.3 SignalR 17](#_Toc149643414)

[2.3.4 NGX-Translate 17](#_Toc149643415)

[2.3.5 Dotenv 17](#_Toc149643416)

[2.3.6 Cypress 18](#_Toc149643417)

[3 Tervezés 19](#_Toc149643418)

[3.1 Bevezetés 19](#_Toc149643419)

[3.1.1 Felhasználókezelés 19](#_Toc149643420)

[3.1.2 Bolt rendszer 20](#_Toc149643421)

[3.1.3 Váróterem rendszer 21](#_Toc149643422)

[3.1.4 Játék rendszer 22](#_Toc149643423)

[3.2 Architektúra 23](#_Toc149643424)

[3.2.1 Áttekintő 23](#_Toc149643425)

[3.2.2 Adatbázis 24](#_Toc149643426)

[3.2.3 Szerveroldal 25](#_Toc149643427)

[3.2.4 Kliensoldal 25](#_Toc149643428)

[4 Önálló munka bemutatása 26](#_Toc149643429)

[4.1 Bevezetés 26](#_Toc149643430)

[4.2 Szerveroldali funkciók 26](#_Toc149643431)

[4.2.1 Adatelérési réteg 26](#_Toc149643432)

[4.2.2 Üzleti logikai réteg 30](#_Toc149643433)

[4.2.3 API réteg 34](#_Toc149643434)

[4.2.4 API Gateway 39](#_Toc149643435)

[4.3 Kliensoldali funkciók 40](#_Toc149643436)

[4.3.1 Dotenv 40](#_Toc149643437)

[4.3.2 Témák 41](#_Toc149643438)

[4.3.3 Nyelvesítés 41](#_Toc149643439)

[4.3.4 WebSocket 41](#_Toc149643440)

[4.3.5 Interceptor 41](#_Toc149643441)

[4.3.6 Guard 41](#_Toc149643442)

[4.3.7 Directive 41](#_Toc149643443)

[4.4 Felhasználókezelés 41](#_Toc149643444)

[4.5 Bolt kialakítása 42](#_Toc149643445)

[4.6 Váróterem megvalósítása 42](#_Toc149643446)

[4.7 Játék 42](#_Toc149643447)

[5 Összefoglaló 43](#_Toc149643448)

[6 Irodalomjegyzék 44](#_Toc149643449)

[7 Függelék 45](#_Toc149643450)

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, 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. 05. 28.

...…………………………………………….

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.

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

A szoftver implementációjában ezenkívül kiemelt szerepet kapott, hogy a szerveroldali komponens mikroszolgáltatás alapokon működjön, azt 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 socialising.

One disadvantage of card games is that they require physical presence. No matter how enjoyable the game, players need to come together to use it, which is often difficult in today's world. The software implementation of the game is a way to overcome this problem, allowing players to enjoy the game's features globally.

The software implementation of the game will be in a browser-based format, allowing players to connect to the game and enjoy its features regardless of platform. In order to provide a true social user experience, features beyond the game, such as managing friends, shopping and text chat, were implemented during development.

In addition, the implementation of the software has focused on ensuring that the server-side component is microservice based, actively and efficiently exploiting the server-side component.

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

# Bevezetés

## Böngészős- és társasjátékok

Társasjátékok évszázadok óta nyújtanak szórakoztatást és kikapcsolódást családoknak és barátoki társaságoknak. Ezek az interaktív, változatos játékok nem csak szórakoztatást nyújtanak, de különböző készségeket is fejlesztenek, mint 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. Ezek lehetőséget biztosítanak arra, hogy igényektől függően bárhol és bármikor élvezhessük a játékot.

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

## Motiváció

Ez a diplomaterv a Sushi Go Party! nevű változatos kártyajátékot hivatott megvalósítani interneten keresztül. Az interneten keresztül való használatával a játéknak nagyobb célközönséget tud teremteni magának.

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.

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 vagy gazdasági lehetőségeket is.

## 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 először 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 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 képességüktől 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 velük a legtöbbet szerezzék meg. Például a nigiri kártyák annyi pontot érnek, amennyi szerepel rajtuk, további logika nélkül. Viszont a tempura esetén minden második lerakott után lehet pontokat szerezni.

Egy körnek akkor van vége, ha minden lap elfogyott a játékosok kezéből. Vannak kártyák, amiknek a játéknak ennek a szakaszában van akciójuk. A kör végén, hacsak nem ilyen akciójú (desszert típusú) a lap, minden lerakott kártya visszakerül a pakliba a pontja beszámítása után. Egy játék 3 ilyen körből áll, és az utolsó kör végén a legtöbb összesített ponttal rendelkező játékos nyer.

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 program keretében is.

## Felhasználói élmény

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

A játékban tehát a felhasználókezelési funkciók és annak megvalósítása is fontos volt. Az egyes felhasználóknak regisztrálniuk kell, hogy igénybe vegyék a játék adta funkciókat és azon felüli lehetőségeket is igénybe vegyenek. Ilyen például, hogy egymást barátnak tudják jelölni, amivel megtalálják egymást könnyedén 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, 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 várakoznak.

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

## 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 egyéb képeknek a linkjei is.

Szerveroldalon ezenkívül a perzisztens relációs adatbázis mellett egy temporális cache-elésre 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 barátunk 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 kedvenc kiválasztott 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.

# Technológia

## 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.

## 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.

### Docker

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

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 esetleg kommunikációnak a lehetőségét, ami az izolált rendszerek közötti logikai kapcsolatot hozza létre.

### 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. Tehát 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.

### Ocelot

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

Az Ocelot az üzenetek átirányítása során többféle stratégiát vagy transzformációt alkalmazhat. Emellett átlátható integrációt valósít meg az „IdentityServer” -rel, ami a szerver felhasználókezelő komponense. Ezáltal könnyedén lekezeli a felhasználók authentikációját, ha a helyzet úgy adja.

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.

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.

### 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 lehetőséget ad az esemény feliratkozás minta használatára is, ami a cache adatbázis naprakészen tartásában adott segítséget.

### SignalR

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

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 is tudunk értesítéseket küldeni a megfelelő felhasználóknak. Erre egy példa, ha egy játékos kirak egy lapot, akkor arról csak 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 alkalmazás felhasználói számára.

### 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 lekérdezések és parancsok környékén volt szerepe. Bizonyos lekérdezéseknél be lett állítva, hogy a lekérdezés tovább küldése 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.

### 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 gondoskodni kell arról, hogy az egyes konténerek tudjanak egymással kommunikálni amikor szükséges. Itt van szerepe a RabbitMQ-nak, hogy bizonyos eseményekre parancsokat futtassunk le más-más konténerekben, mint amiken meghívtuk a kéréseket.

### 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.

Maga az IdentityServer4 egy 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 átadni 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 biztosít a szerver többi komponensével, mint az Ocelot vagy SignalR könyvtárak.

### AutoMapper

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

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

### 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.

### 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.

## Kliensoldali technológiák

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

### Angular Material

Az Angular Material [9] 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.

### 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.

### 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.

### NGX-Translate

Az ngx-translate [10] 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 Asset fájlokba, ahol a könyvtárban beállított nyelv segítségével választja ki melyik Asset segítségével állítsa be.

A könyvtárnak több más segédfunkciója van, ami sokban elősegíti a 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 mondatoknak a tárgyai is ugyanígy könnyen megadható kívülről.

### 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 fel is használhatják ellenünk. Emellett az Asset-ek közé se akarjuk őket helyezni, mert akkor bárki kiolvashatja kívülről.

Ilyenkor jön szerepbe a dotenv könyvtár, 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.

### Cypress

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

# Tervezés

## Bevezetés

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

### 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 elfogadhatják vagy elutasíthatják. Barátok látják egymást egy listán, mellettük egy indikátorral jelezve 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.

### 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.

A képen szöveg, képernyőkép, hold, kör látható

Automatikusan generált leírás

2. ábra Boltrendszer használati esetei

Ezzel további motivációt adunk a játékosoknak, hogy többet használják az alkalmazást. Emellett nagyobb változatosságot is ad, meg döntési lehetőséget is, hogy milyen pakli tetszett meg nekik.

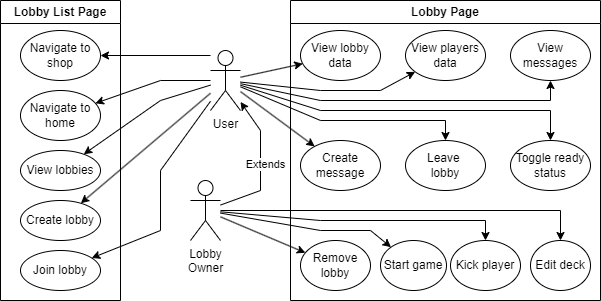
Minden játékos a játék alap Sushi Go verziójának paklijával tud kezdetben játszani, és elég pontot szerezve tudja magának feloldani a boltban vásárlási lehetőséget. Ezzel kicsit elrejtve a plusz ponthasználat szükségességét, de mégis újdonságot adva.

### 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 megtekinthetői. Tehát ha valaki elindít egy szobát, akkor azt egy másik felhasználó rögtön lát 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 belépett. A szobában többféle információval találkozunk, kezdve a szoba nevével és jelszójával, hogy további játékosokat hívhassunk 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.

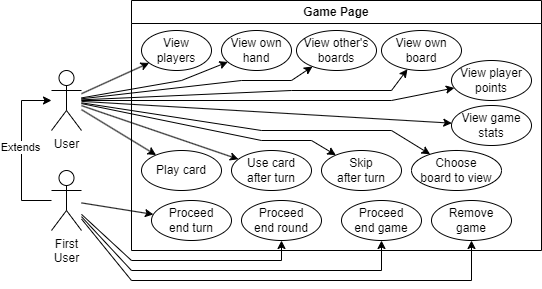
Egy további funkció a szobában, hogy a belépett játékosok tudnak élőben chatelni egymással, míg a játék indítására várakoznak. Ez időrendben megjelenik a szoba alján, ahol látják ki és mit írt a szobában.

### 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.

## Architektúra

### 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

### 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 sémáját az Entity Framework Code-First technikája segítségével, az adatelérési rétegből fogalmazhatóak meg. Itt megadható az egyes modellek, amiket adatbázisba táblákra lehet fordítani. Továbbá megadhatóak táblák közötti kapcsolatok és 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.

### 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 hasonló, vagy 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.

### 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.

# Önálló munka bemutatása

## 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.

## Szerveroldali funkciók

### Adatelérési réteg

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

#### Repository

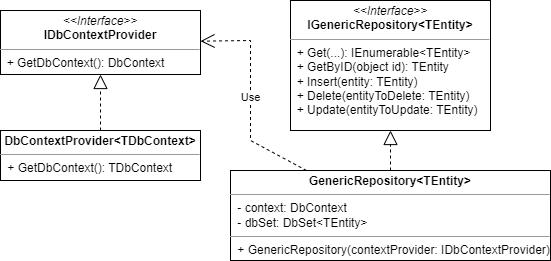
Az adatbázis elérése elsősorban a repository minta segítségével van kialakítva. Ennek a mintának az elsődleges feladata, hogy egy absztrakciós réteget biztosítson az adatbázis fölé az üzleti logikai rétegnek.

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.

#### 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 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>));

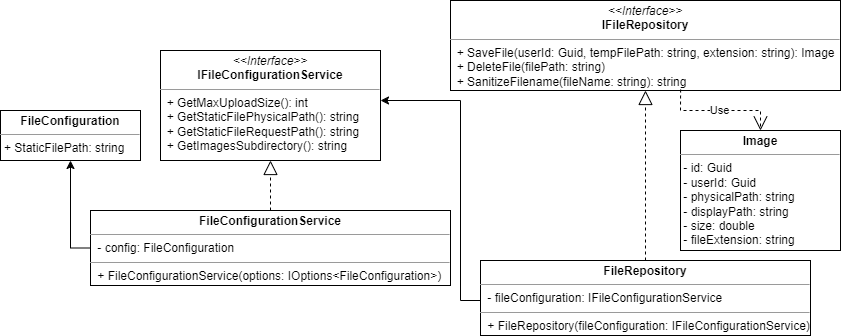
Így az egyes repository komponensek a saját konténerük adatbázisával tudnak dolgozni a közös implementáció során is.

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.

#### 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 képei, felhasználó által kezelt 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("FileConfiguration"));

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()}"

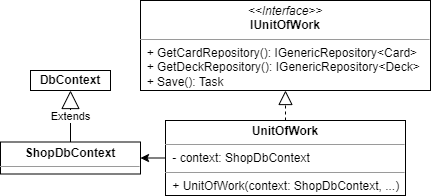
});

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

#### UnitOfWork

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

Az implementációban a szerepe az egyes generikus repository-k betöltése függőség injektálással és getterek 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 gettert. Így az üzleti logikai rétegben egyedül ezt a komponenst kell kezelni és használni. Ezáltal könnyen karbantartható és skálázható további entitásokkal az implementáció.

#### 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 az adott generikus értéket 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, amikor vagy szöveges vagy enumerációs kulcs segítségével Dictionary változóban kezelünk adatokat. Ezeket az adatbázisba való mentés előtt szöveges JSON objektumba mentjük, és lekérdezés során olvassuk ki belőle.

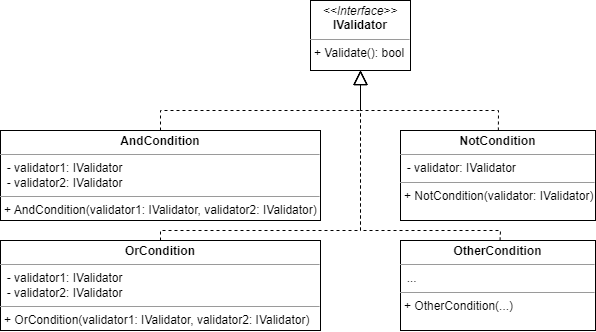
### Üzleti logikai réteg

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.

#### 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, így a saját feltételeket összevonva. 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));

}

#### 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 a pipeline-okat az egyes konténerek a többi függőség injektálással együtt tudja beépíteni az alkalmazásba az IPipelineBehavior típus alatt, generikus kérés osztály megadása nélkül is:

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

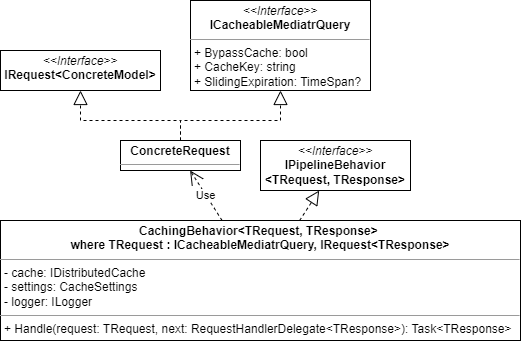
#### LoggingBehavior

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.

#### 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 getterrel 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.

#### Exceptions

A szoftverben saját hibaosztályok vannak implementálva. Kevés jelentőségük van azonkívül, hogy 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.

#### 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;

}

### 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.

#### 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);

#### 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.

#### 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.

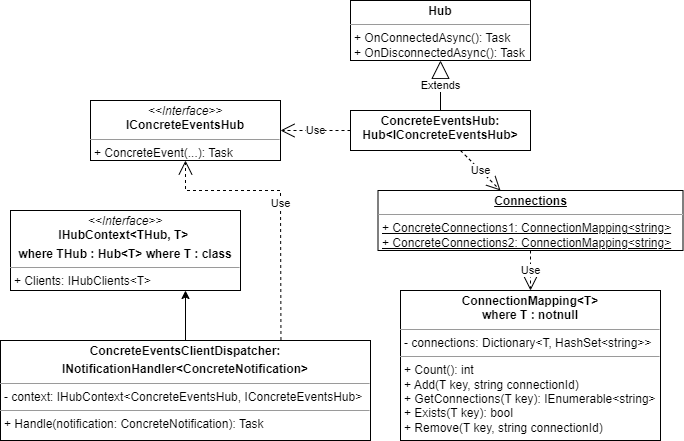
Ebből az ősosztályból kapunk két fontos függvényt. Egyik egy kliens csatlakozásának az eseménye, a másik pedig a lecsatlakozása. Mindkettőre tudunk implementációt kötni, kapcsolatot csoportba rendezni. A csatlakozás eseményén a kontextusból kapunk egy ConnectionId azonosítót, ami az adott kapcsolatot 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.

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 eseményeikben.

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.



. 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.

#### 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 továbbítására közöttük, viszont fontos szerepet töltenek be.

A jelek továbbításának megvalósítása a MediatR eseményeihez hasonlóan a publish-subscribe mintát követi. Tehát egyes funkciók elindítanak egy eseményt, amire más konténerek feliratkoznak a saját implementációikkal. Itt fontos- hogy egy-egy kommunikációról van szó, tehát minden eseményre egyetlen konténer iratkozik fel, ellentétben a MediatR-tól.

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 lettek megadva, ahol közösen lehet szerkeszteni vagy bővíteni őket mindkét félnek.

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

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 funkcióját használjuk ki. Viszont sokszínű kialakítást tesz lehetővé. A kapcsolat kialakítása nagyvonalakban így néz ki:

services.AddMassTransit(options =>

{

options.AddConsumer<LobbyRemoveConsumer>();

options.SetKebabCaseEndpointNameFormatter();

options.UsingRabbitMq((context, cfg) =>

{

cfg.Host("rabbitmq", "/", h =>

{

…

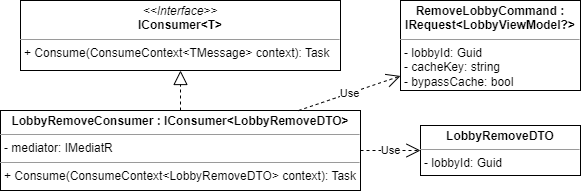
});

cfg.ConfigureEndpoints(context);

});

});

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



### API Gateway

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

A JSON fájlban konténeren belüli útvonalat és konfigurációt a downstream értékek jelölik, és a külvilág felé irányított értékeket pedig az upstream. Például a „DownstreamHostAndPorts” tartalmazza, hogy pontosan melyik konténer felé irányítjuk.

#### SwaggerForOcelot

Egy praktikus segédkönyvtár, aminek a segítségével swagger felületet tudunk biztosítani az ocelot által kezelt interfésznek. Mivel az interfész mindenféle konténerből tartalmaz végpontokat, ezért lehetőség van saját elválasztást adni a konfigurációknak. 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 mappában található 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. Ebben a fájlban az értékek tartalmazzák a kulcsot, amit a kérésekbe rakunk, és a swagger felület beállításait. Például a felhasználókezelő konténer értéke:

{

"Key": "user",

"Config": [

{

"Name": "User API",

"Version": "v1",

"Url": "http://user.api/swagger/v1/swagger.json"

}

]

}

## Kliensoldali funkciók

### Dotenv

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

baseUrl=http://localhost:5000/api

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

Ezáltal az alkalmazás indítása előtt betölti a környezeti változókat, ha szükséges.

### 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 használ fel.

Az alkalmazás alapértelmezett témája a világos, amit a sötéttel tudunk felülírni Cookie-val vezérelve. Ez a vezérlés úgy működik, hogy a body tagnek a css osztályát állítjuk light-theme-re vagy dark-theme-re. Ezáltal tudjuk a témát css osztályokon belül kezelni. 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 színeket átállítani.

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

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

A változókat egy mixin segítségével töltjük be, ami létrehoz egyszerre egy hex alapú és egy rgb alapó változót is:

@mixin addColor($color, $value) {

--color-#{$color}: #{$value};

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

}

Ehhez a mixinhez a változókat egy listából olvassuk 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(…);

}

}

}

### Nyelvesítés

A nyelvesítés megvalósításához az ngx-translate könyvtár volt használva. Az alkalmazásban két nyelv található, magyar és angol, de tetszőlegesen bővíthető. A megvalósítás részeként az Asset 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 }}

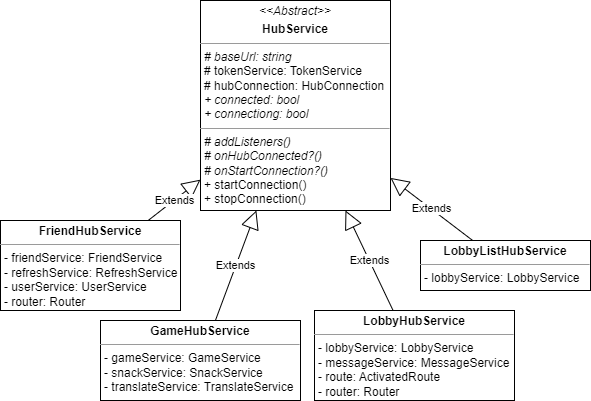
A felhasználó által kiválasztott nyelv Cookie-val van vezérelve, aminek az alkalmazás megnyitásakor az alapértelmezett értéke angol. A Cookie-val vezérelt nyelv így böngésző specifikusan elmenti a felhasználó preferált nyelvét, a stílus beállítással hasonló módszerrel.

### WebSocket

A SignalR alapú kommunikáció kiépítése elkészült kliensoldalon is. Az implementáció ki lett szervezve a komponensektől független szolgáltatásokba. Mivel a kapcsolat kiépítésének a hubtól függetlenül hasonló a módszere, 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.

Egy másik megvalósítandó függvény a leszármazottaknak a szerver eseményeire való feliratkozások regisztrációja. Itt tudjuk elkapni azokat az eseményeket, amiket a szerverben megadott interfészen határoztunk meg. Ezért az ottani interfészekkel párhuzamosan van kialakítva az itteni feliratkozások a nevekkel és paraméterek modelleivel.



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

### Interceptor

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

Ennek az AuthInterceptor osztálynak az a feladata, hogy a kérések fejlécébe felhelyezze az authorization értékbe a bearer tokent az alkalmazás Cookie-jaiból. A felhelyezés mellett további feladata, hogy elkapja a hibákat azután, hogy elküldte a kérést. Az elkapott kérésben, ha 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.

### Guard

Olyan komponens, aminek a segítségével eldönthető, hogy egy útvonal 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.

#### AclGuard

Ennek a guardnak a szerepe, hogy kliensoldali authorizációt hajtson végre az oldalakra lépéskor. A feladata, hogy egy konfigurációs ACL modellből olvassa ki, hogy adott kulcsokhoz milyen felhasználói Claimekre van szükség. Ezek a kulcsok tetszőleges szöveges értékek lehetnek.

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

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

#### LoginGuard

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

#### 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.

#### GameGuard és LobbyGuard

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

### Directive

## Felhasználókezelés

## Bolt kialakítása

## Váróterem megvalósítása

## Játék

# Összefoglaló

# 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>

# Függelék