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

Horváth Gergely

MICROSERVICES ARCHITEKTÚRÁJÚ VENDÉGLÁTÓHELY KERESŐ RENDSZER FEJLESZTÉSE

KONZULENS

Jánoky László Viktor

BUDAPEST, 2019

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Horváth 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, 2019. 12. 22.

...…….
Horváth Gergely

Összefoglaló

Napjainkban egyre elterjedtebbé válik a szoftverfejlesztésben a microservices architektúra. Ez részben a felhőszolgáltatások elterjedésének és a szoftverek növekvő komplexitásának köszönhető. A diplomamunkám célja, hogy egy microservices architektúrájú alkalmazást fejlesszek és a fejlesztés során megismerkedjek a szükséges technológiákkal és módszerekkel. Ezek közé tartoznak a Docker konténerizációs és a Kubernetes konténer kezelő eszközök.

A célom egy vendéglátóhely kereső alkalmazás elkészítése, ami segíti a felhasználóit az alkalmas vendéglátóhelyek kiválasztásában. A helyek közötti böngészés mellett, olyan funkciókat is meg kell valósítania, amik hasonló megoldásokban nem elérhetők. Ilyenek például a helyekhez kapcsolódó nemvárt események megtekintése vagy az ajánlás funkciók. Az elkészítendő szoftverrel szemben követelmény, hogy jól karbantartható és lazán csatolt legyen, valamint képes legyen dinamikusan változó felhasználóbázis kiszolgálására.

A feladat tehát, hogy megismerjem és bemutassam a microservices architektúrát. Ismertessem a fejlesztés lépéseit a tervezéstől kezdve, az implementáción át, a tesztelésig. Az elkészült alkalmazásnak három részből kell állnia, egy szerver oldali programból, egy webes adminisztrációs felületből és egy mobil kliensből. A dolgozatnak ki kell térnie a futtatáshoz szükséges infrastruktúrák bemutatására és felhasználásuk módjaira. Végül tesztekkel igazolnom kell, hogy az elkészített szoftver teljesíti a követelményeket és jól skálázható.

A dolgozat végén, majd összefoglalom és ismertetem az elkészült munkát és a továbbfejlesztési lehetőségeket. Végezetül pedig levonom a személyes tapasztalataimat.

Abstract

These days, in software development the microsercies architecture becoming more and more popular. This is partly due to the popularity of cloud services and the increasing complexity of softwares. The aim of my thesis is to develop a software with microservices architecture and during the development get acquainted with required technologies and methods. Including Docker container and Kubernetes container managing technologies and tools.

My goal is to create a catering unit search application which is helping the users to find the most appropriate places. Besides browsing between the caterings, the application has to realize functions what are not available in similar solutions. Such as showing unexpected events related to the caterings or recommending places. Important requirements for the software are to be easily maintainable, loosely coupled and be able to serve dynamically changing user bases.

So the task is to get to know and present the microservices architecture. Present the steps of the development, starting from the design, through the implementation and finally the testing. The final application has to include three parts: a server side program, a web based administration interface and a mobile client. The thesis has to describe the infrascturcure required to run the application and the way how it can be used. Finally I need to verify that the implemented software fulfills the requirements and well scalable.

At the end of my thesis I am going to summarize the final outcome and describe the possibilities for further improvements. Lastly I deduct my personal experience.

1 Bevezetés

A szórakozni vagy kikapcsolódni induló emberek meglepően sokszor szembesülnek azzal a problémával, hogy az általuk kiválasztott vendéglátóhelyhez érve ott telt ház van, vagy órákig kell sorban állni. Számos esetben az sem könnyű kérdés számukra, hogy egy vacsorához, céges csapatépítőhöz vagy baráti találkozóhoz melyik helyet érdemes választaniuk. Az általam fejlesztendő rendszer célja, hogy ezekben a kérdésekben biztosítson segítséget a felhasználói számára.

1.1 Célkitűzés

Az elkészítendő alkalmazás célja, hogy vendéglátóhelyek adatait tárolja, tegye elérhetővé és kereshetővé a felhasználók számára. További cél, hogy élő adatokat szolgáltasson a foglaltságról, sorban állás hosszáról vagy rendkívüli zárva tartásról. A szoftverrel szemben támasztott elvárás, hogy képes legyen nagy mennyiségű felhasználó kiszolgálására és megbirkózzon azok dinamikusan változó számával. A jelenlegi trendeket követve szükséges egy mobiltelefonos alkalmazás, amely segítségével bárhol elérhető a szolgáltatás. Elvárás, továbbá egy webes felület is, az adminisztráció megkönnyítése érdekében. Ezen a felületen a tulajdonosok regisztrálhatják vendéglátóhelyeiket és frissíthetik azok adatait.

2 Követelmények

Ez a fejezet bemutatja a rendszer által megoldandó problémákat és a használati eseteket. Pontos képet ad arról, hogy milyen elvárásokat kell megoldania az elkészült szoftvernek.

2.1 Használati esetek leírása

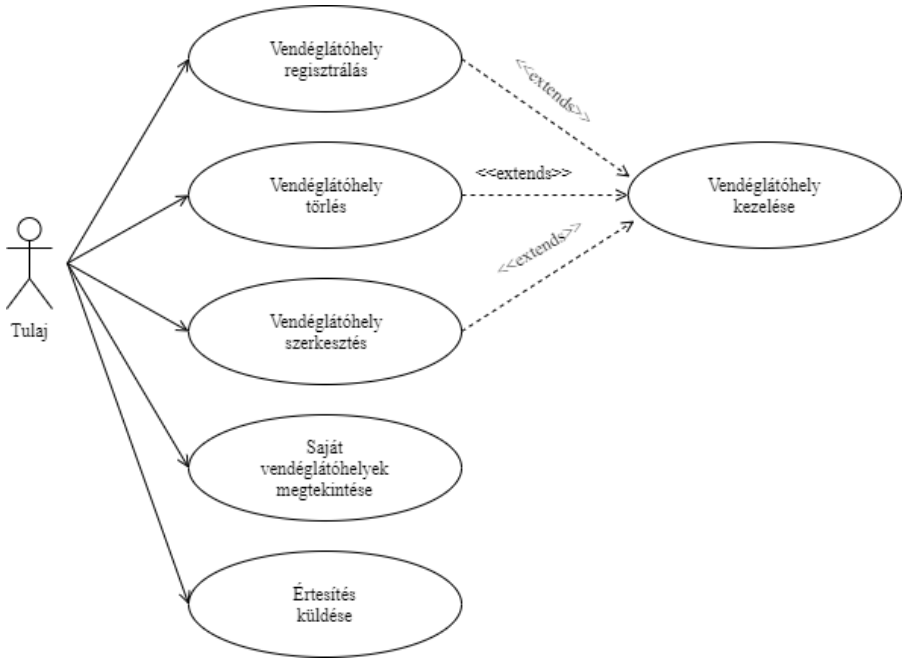
A rendszer három típusú felhasználót különböztet meg aszerint, hogy kinek mi a feladata és milyen funkciókat kíván használni. A három típus a következő:

- **Vendéglátóhely tulajdonos** *(Lásd később: Tulaj)*
- **Egyszerű felhasználó** *(Lásd később: Felhasználó)*
- **Rendszer adminisztrátor** *(Lásd később: Adminisztrátor)*

2.1.1 Használati esetek a *Tulaj* szempontjából

A *Tulaj* típusú felhasználó bejelentkezés után feltöltheti a rendszerbe a vendlátóhelyét vagy helyeit. A feltöltött adatokat később szerkesztheti és törölheti, illetve értesítést küldhet a feliratkozott felhasználók számára. A saját adatait is megadhatja és szerkesztheti.

A részletes használati eseteket az alábbi 1. ábra mutatja be.

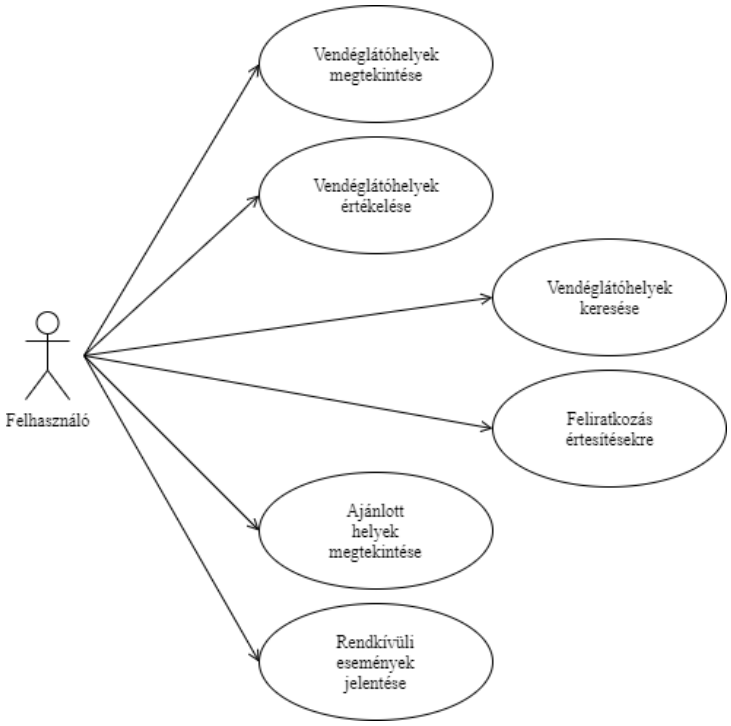


1. ábra Tulaj használati eset diagram

2.1.2 Használati esetek a *Felhasználó* szempontjából

Az általános *Felhasználó* megtekintheti a *Tulajok* által feltöltött vendéglátóhelyeket és kereshet azok között. A keresés célja, hogy gyorsan és egyszerűen szűkíthesse le a választékot. Lehetősége van a helyek értékelésére, és ezen értékelések módosítására. Feliratkozhat az intézmények értesítéseire, hogy emailben kapjon tájékoztatást az aktuális információkról. A többi felhasználó számára megjelölheti, hogy milyen aktuális, esetleges nemvárt körülmény tartozik egy adott helyhez.

A részletes használati eseteket az alábbi 2. ábra mutatja be.

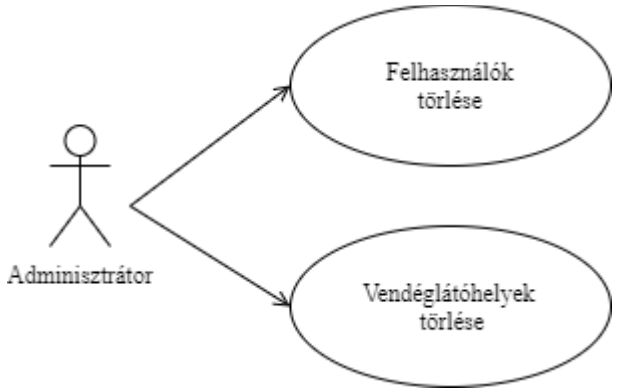


2. ábra Felhasználó használati eset diagram

2.1.3 Használati esetek az *Adminisztrátor* szempontjából

Az *Adminisztrátor* felhasználó célja, hogy kezelni lehessen a rendszer olyan adatait, amihez másnak nincs hozzáférése. A feladata lehet a nemkívánatos felhasználók törlése a rendszerből.

A részletes használati eseteket az alábbi 3. ábra mutatja be.

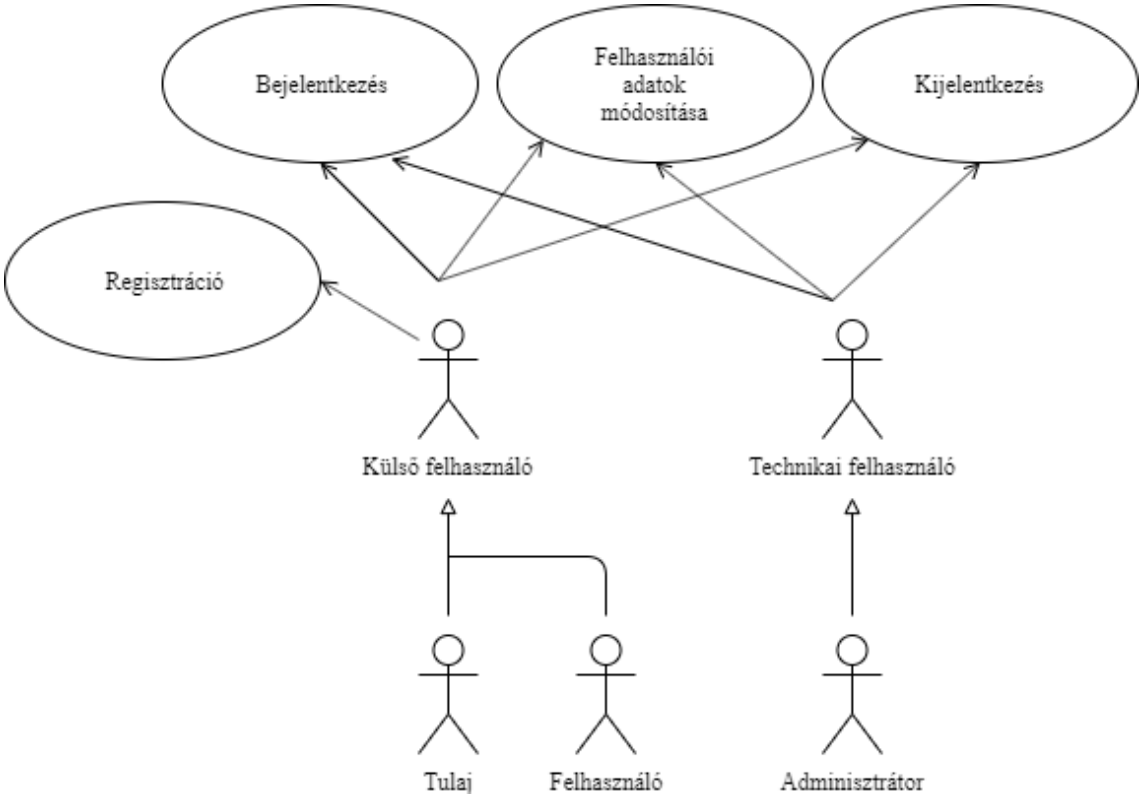


3. ábra Adminisztrátor használati eset diagram

2.1.4 Használati esetek az összes felhasználó szempontjából

A rendszerben vannak olyan esetek is, amelyek több felhasználói csoport számára is elérhetők. A regisztráció, bejelentkezés és felhasználói adatok kezelése tartoznak ide.

A részletes használati eseteket az alábbi 4. ábra mutatja be.



4. ábra Közös használati eset diagram

2.2 Funkcionális követelmények

Ebben a fejezetben részletesebben ismertetem a használati esetek mentén kirajzolódott követelményeket. A leírást a 2.1 fejezetben bevezetett felhasználói szerepkörök mentén adom meg.

2.2.1 Minden felhasználót érintő követelmények

1. **A felhasználóknak biztosítani kell, hogy regisztrálhassák magukat a rendszerbe.**

   Az új felhasználók bejelentkezés nélkül regisztrálhatnak egy saját fiókot a rendszerben. Ez alól kivétel az *Adminisztrátor* felhasználó, mert ez a felhasználó előre definiált. A regisztrációhoz meg kell adni egy rendszerszinten egyedi felhasználónevet és egy tetszőleges jelszót, valamint

ki kell választani, hogy *Felhasználó* vagy *Tulaj* szerepkörben kerüljön rögzítésre a fiók.

2. **A regisztrált felhasználóknak be kell tudniuk jelentkezni a rendszerbe.** A sikeresen regisztrált kliens a felhasználóneve és jelszava segítségével bejelentkezhet a alkalmazásba. Sikeres belépés esetén egy egyedi biztonsági kulcsot kap, mellyel azonosíthatja magát.

3. **A bejelentkezett felhasználó kijelentkezhet a rendszerből.**

   A már bejelentkezett felhasználók kijelentkezhetnek a rendszerből, így nem férnek hozzá az alkalmazáshoz a következő bejelentkezésig. Ebben az esetben a biztonsági kulcsuk törlődik.

4. **A Felhasználók módosíthatják személyes adataikat.**

   A bejelentkezett felhasználók megadhatják és módosíthatják személyes adataikat. A következő paramétereket adhatják meg: teljes név, város, email cím, születési dátum és nem.

2.2.2 *Tulaj* felhasználókat érintő követelmények

Az alábbi funkciók mindegyike kizárólag bejelentkezett felhasználók számára elérhető.

1. **A *Tulaj* felhasználók regisztrálhatják saját vendéglátóhelyeiket.**

   A tulajok felvehetnek új szórakoztató egységeket a rendszerbe. Meg kell adniuk a hely nevét, ami rendszer szinten egyedi kell legyen, továbbá egy rövid leírást, a nyitva tartást, a hely címét és egyedi kategória paramétereket kulcs-érték formában. Lehetőségük van képek feltöltésére is.

2. **A *Tulajok* frissíthetik a saját helyeik adatait.**

   A tulajok később szerkeszthetik a vendéglátó egységek adatait. A név paraméter kivételével az összes attribútum felülírható.

3. **A *Tulajok* törölhetik a saját vendéglátóhelyeiket a rendszerből.**

   A tulajok törölhetik saját helyeiket a rendszerből. Ebben az esetben az összes hozzá kapcsolódó információ is elveszik.

4. **A *Tulajok* megtekinthetik saját vendéglátóhelyeiket.**

A tulajok megtekinthetik egy lista formájában az általuk regisztrált összes helyet, valamint minden hely részletes adatait is megnézhetik.

5. **A *Tulajok* értesítést küldhetnek a *Felhasználók* számára.**

   A tulajok értesítést küldhetnek az általuk regisztrált helyek nevében azon felhasználók számára, akik feliratkoztak az kiválasztott hely értesítéseire és rendelkeznek email címmel.

2.2.3 A *Felhasználókat* érintő követelmények

Az alábbi funkciók mindegyike kizárólag bejelentkezett felhasználók számára elérhető.

1. **A *Felhasználók* megtekinthetik a vendéglátóhelyeket.**

   A felhasználók megtekinthetik egy listában az összes helyet, ami a rendszerben található. Továbbá minden hely esetén megtekinthetnek egy részletes nézetet, ami az összes a tulajdonosok által megadott információt tartalmazza és megjeleníti a kapcsolódó értékeléseket és nemvárt eseményeket.

2. **A *Felhasználók* értékelhetik a vendéglátóhelyeket.**

   A felhasználók minden helyhez rendelhetnek egy saját értékelést. Az értékelés egy 1-től 5-ig terjedő skálán adható meg, továbbá kiegészíthető egy szöveges véleménnyel is. Minden értékelés később módosítható.

3. **A *Felhasználók* kereshetnek a vendéglátóhelyek között.**

   A felhasználók egy keresőmezőbe írhatják a keresési feltételeiket, amivel leszűkíthetik a helyek listáját. Kereshetnek a helyek neveiben, címeiben, leírásaiban vagy kategória paramétereiben.

4. **A *Felhasználók* feliratkozhatnak értesítésekre.**

   A felhasználók feliratkozhatnak egyes helyek értesítéseire. Ekkor, ha a hely tulajdonosa értesítést küld, akkor a feliratkozott kliensek erről emailt kapnak, amennyiben helyesen be van állítva a saját email címe.

5. **A *Felhasználók* megtekinthetik a rendszer által számukra ajánlott helyeket.**

A felhasználók számára a rendszer az általuk kedvelt helyek alapján jelenít meg ajánlatokat. Azokat a helyeket részesíti előnyben a program, melyeket más hasonló ízlésű kliensek is kedveltek.

6. **A *Felhasználók* rendkívüli eseményeket vehetnek fel a rendszerbe.**

   A felhasználók az adott helyekhez rendkívüli eseményeket jelenthetnek be. A helyek azonosításához a készülékük által szolgáltatott helyadatok nyújtanak segítséget. Az így rögzített események a jelentéstől számított néhány óráig elérhetőek más felhasználók számára.

2.3 Nem funkcionális követelmények

A nem funkcionális követelmények felállítása során arra törekedtem, hogy egy jól karbantartható és kis egységekre tagolt rendszert követeljen meg. Emelet az egyik legfontosabb szempont, hogy az alkalmazás skálázható legyen és könnyen lehessen bővíteni.

2.3.1 Használhatósági követelmények

1. A szolgáltatásnak elérhetőnek kell lennie Android operációs rendszerű mobiltelefonról, amennyiben biztosított a folyamatos internetkapcsolat.
2. A mobiltelefonos alkalmazás telepítés után használható.
3. A mobil kliens esetén engedélyezni kell a helyadatokhoz való hozzáférést.
4. A szolgáltatás adminisztrációs felülete elérhető kell legyen egy weboldalon, tetszőleges modern webböngészőben. Ebben az esetben külön telepítés nem szükséges.

2.3.2 Megbízhatósági követelmények

1. Az alkalmazásnak egyes szolgáltatások leállása esetén is elérhetőnek kell maradnia. Ilyen esetben elvárható, hogy leállás nélkül egy ugyanazt a funkciót ellátó új szolgáltatás álljon a leállt párja helyébe. Ez alól kivételt képez a perzisztens adattároló.

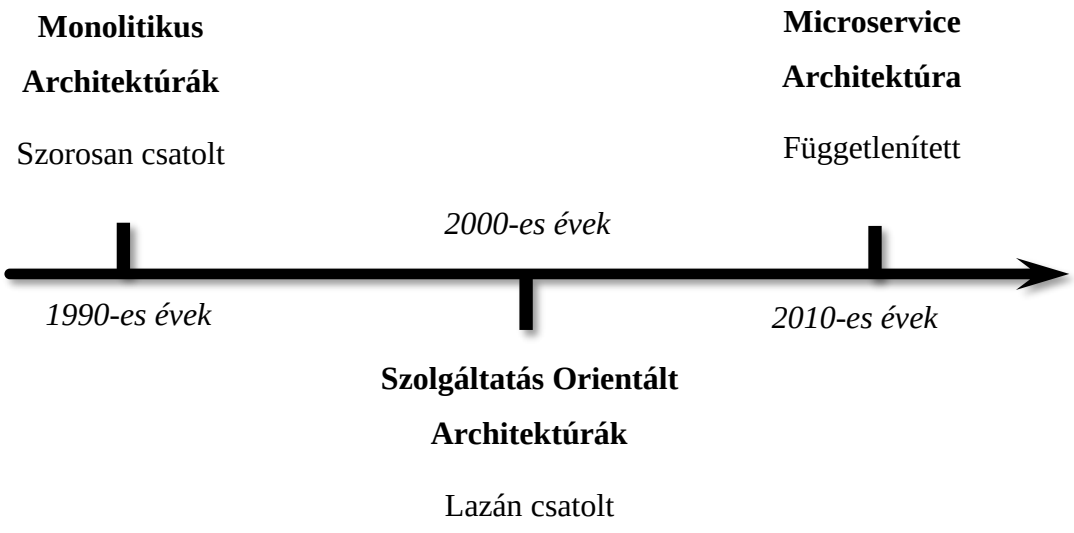
2.3.3 Hatékonysági követelmények

1. A rendszernek minden kérésre maximum néhány másodpercen belül válaszolnia kell. Ezt abban az esetben is teljesítenie kell, ha a felhasználók száma megemelkedett.
2. Amennyiben a rendszert kevés felhasználó használja, képesnek kell lennie a használt erőforrásokat minimalizálni. Ezzel csökkentve az üzemeltetés költségeit.
3. A rendszernek egyszerűen telepíthetőnek kell lennie. Lehetőség szerint minél kisebb mértékben függjenek a szolgáltatások a futtató környezetüktől.

3 Technológiai áttekintés

Napjainkban egyre népszerűbb a microservices architektúra. A legnagyobb technológiai konferenciákon állandó téma ez a tervezési minta. [1] Nagyban segítette az elterjedését, hogy számos vezető cég alkalmazza sikeresen úgy, mint az Amazon vagy a Netflix csak, hogy néhányat említsek.

Ez az architektúra a monolitikus és szolgáltatás orientált (SOA – Service Oriented Architecture – Szolgáltatásorientált architektúra) megoldásokat követte. Az 5. ábra mutatja be az általam említett minták elterjedésének sorrendjét és azt, hogy mely korszakokra voltak jellemzők. Természetesen ez nem azt jelenti, hogy napjainkban kizárólag a legújabb trendeknek megfelelő megoldások a legelterjedtebbek vagy a legjobbak. Mindegyik megoldásnak megvan az saját előnye és hátránya különböző szempontokat figyelembe véve. Ezekre a tulajdonságokra a 3.1-es fejezetben részletesebben kitérek. A következő fejezetben pedig egy alapos összehasonlítást adok, ami segít megérteni a megoldások közti különbségeket.



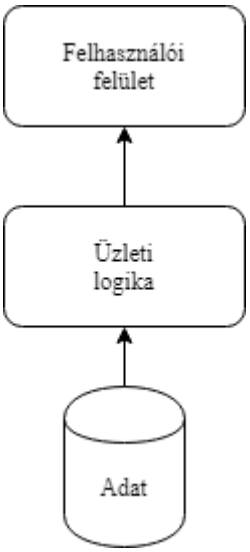
5. ábra Architektúra evolúció

3.1 Architektúrák összehasonlítása

Ebben a fejezetben összehasonlítom a bevezetőben megemlített három architektúrát és ismertetem hátrányaikat, valamint előnyös tulajdonságaikat. A bemutatás időrendben történik.

3.1.1 Monolitikus architektúra

A monolitikus architektúra nevezhető a klasszikus megközelítésnek a vállalati alkalmazások fejlesztésében. A kód általában három rétegű felépítést követ, aminek a legalsó szintjén az adat réteg található, ezt követi az üzleti logikai réteg és végül a felhasználói felület, ahogyan ezt a 6. ábra mutatja.



6. ábra Három rétegű felépítés

Az adatokat jellemzően egy központi adatbázisban tárolják és minden logikai modul ehhez fér hozzá. A rétegeken belül a funkciók mentén modulokra bontott a felépítés, így növelhető az áttekinthetőség és karbantarthatóság. A koncepció lényege, hogy a szoftver különböző komponenseit egy alkalmazássá fogja össze, ami egy szimpla alkalmazás példányként fut jellemzően egy alkalmazás szerveren.

3.1.1.1 Monolitikus architektúra előnyei:

Egyszerűbb fejlesztés:

A fejlesztés során egy kódbázist kell csak kezelni. Ezt általában egy egységes verziókezelő rendszerben tartják, így nem kell minden modul esetén új kódbázist letölteni

és azon megkezdeni a fejlesztést. Ezen felül a többször felhasználható eszközök is a teljes kódban könnyen elérhetőek.

Egyszerűbb release management és verziózás:

A teljes alkalmazást egy futtatható fájlként vagy csomagként lehet kiadni. Ezt a kiadást egy folyamaton belül el lehet végezni és ehhez egyértelmű verzió rendelhető. Nem okoz problémát a különböző funkciókat lefedő szoftverrészek inkompatibilitása és ezek egyben való tesztelése.

Egyszerű telepítés:

A telepítés során általában a szoftvert egy csomagként kezelik és egy alkalmazás példányként fut. Ez azt jelenti, hogy jellemzően rövidebb időt vesz igénybe a telepítése, mint egy microservices architektúrájú alkalmazás esetén.

Kevesebb üzemeltetési többletköltség:

Mivel egyetlen alkalmazás példány fut, ezért elegendő egy alkalmazás számára biztosítani naplózást, monitorozást és tesztelést.

Rövidebb válaszidő:

Ha a kód jól felépített, akkor a monolitikus programok jobban kihasználhatják az erőforrásokat, a microservices megoldásokkal szemben. Ez annak köszönhető, hogy a microservices felépítés esetén egy-egy hívás kiszolgálása érdekében sok esetben számos API (Application Programming Interface – Alkalmazásprogramozási Interfész) hívást kell indítani, ami növeli a válaszidőt. Ezzel ellentétben a monolitikus megoldás esetén gyorsabb lehet a komponensek közti kommunikáció a megosztott kódnak és memóriának köszönhetően. [2]

3.1.1.2 Monolitikus architektúra hátrányai:

Nehezebben megérthető kódbázis:

A monolitikus felépítésű kódok a fejlesztés előrehaladtával könnyen bonyolulttá és komplexé válhatnak. Ez végül egy nagyon nehezen áttekinthető kódbázist eredményezhet. Ennek hátránya, hogy egy javítás vagy funkció bevezetése sok időt vehet igénybe. Egy új fejlesztő bevonása a munkába nehézkessé válthat, mert hatalmas kódmennyiséget kell az új csapattagnak áttekintenie, így a betanulási idő drasztikusan megnő.

Limitált agilitás:

Ebben a struktúrában minden apró frissítés egy teljesen új verzió kiadását vonja maga után. Emiatt gyakori, hogy a csapattagoknak másokra kell várniuk, hogy elvégezhessék a saját feladataikat. Ez lassítja a fejlesztést és rontja az agilitást.

Nehézkes az új technológiák bevezetése:

Egy új technológia bevezetése a teljes szoftver kódját érinti. Rengeteg helyen törheti el a meglévő működést. Az adoptáció pedig sok időt igényel, valamint rengeteg változtatást von maga után.

Lassú reakció a visszajelzésekre:

Az, hogy a programot egyben kell kiadni megnehezíti annak a lehetőségét, hogy naponta, akár több verzió is kiadható és kipróbálható legyen az alkalmazásból. Egy komponens megváltoztatása teljes újratelepítést igényel. Ez a nehézség könnyen útjába állhat az újonnan felmerülő igények gyors kiszolgálásának.

Limitált horizontális skálázhatóság:

Abban az esetben, amikor nagy igénybevételt kell kiszolgálnia a rendszerünknek, előbb-utóbb elérkezik az a pont, amikor a vertikális skálázhatóság már nem elegendő. Ilyenkor szükséges a horizontális skálázás, ami ebben az esetben azt jelenti, hogy a teljes alkalmazást több példányban kell futtatni. Nem lehetséges csak a szükséges modulok skálázása, amivel költséghatékonyabban üzemelhetne a rendszer.

Alacsony hibatűrés:

A monolitikus felépítésben egyetlen modul hibája, akár a teljes rendszer leállását is okozhatja, akkor is, ha a rendszer többi eleme ettől a hibától függetlenül még képes lenne a helyes működésre. Ilyenkor a teljes alkalmazást meg kell javítani és újraindítani, ami a már fentebb említett tulajdonságok miatt hosszú időt vesz igénybe. Ez idő alatt a teljes rendszer elérhetetlenné válik.

3.1.2 Szolgáltatásorientált architektúra

A szolgáltatásorientált architektúra (SOA) lényege, hogy a szoftver bizonyos részeit szolgáltatásokra bontja, melyek egy hálózaton belül helyezkednek el, így különböző modulokban is újra felhasználhatók. Ezek a szolgáltatások külön-külön

frissíthetők, megújíthatók és egymástól függetlenül képesek működni. Ezzel egy gyengén csatolt megoldás felépítését teszi lehetővé.

3.1.2.1 Szolgáltatásorientált architektúra előnyei:

Újra felhasználható szolgáltatások:

Az önálló és gyengén csatolt szolgáltatásoknak köszönhetően ezek a komponensek többször felhasználhatóak különböző helyeken anélkül, hogy befolyásolnák egymás működését.

Jobb karbantarthatóság:

A független szolgáltatásoknak köszönhetően ezek az egységek külön-külön karbantarthatók és frissíthetők.

Magasabb megbízhatóság:

A különálló szolgáltatások párhuzamosan fejleszthetők és tesztelhetők szemben a monolitikus megoldással, ahol a teljes rendszert egyben kellett kezelni. Emellett egy- egy ilyen szolgáltatás hibája nem feltétlenül okozza a teljes rendszer leállását.

Párhuzamosan fejleszthető:

Egy masszív fejlesztői csapat számára nem elhanyagolható szempont, hogy a szolgáltatásorientált architektúrában a független kódrészek párhuzamosan is fejleszthetők, így jobban kihasználható és skálázható egy nagy létszámú csapat munkája.

3.1.2.2 Szolgáltatásorientált architektúra hátrányai:

Komplex menedzsment:

Ezek a rendszerek nagyobb komplexitásúak a monolitikus elődjeiknél. A komponensek közötti kommunikáció komoly kihívásokat jelenthet. Továbbá a teljes szoftver összeállítása is nagyobb kihívás. A párhuzamosan dolgozó csapatok összehangolása egy nehéz feladattá válhat.

Lassabb válaszidő:

Azzal, hogy egy adott funkció kiszolgálása több komponensen is végig haladhat és ezek között különböző kommunikációs protokollok biztosítják a kapcsolatot, megnövekedhet a kiszolgálási idő.

3.1.3 Microservice architektúra

A microservices architektúrához nehéz jó definíciót találni, mert jellemzően többféle megközelítést alkalmaznak a gyakorlatban. Az egyik legáltalánosabb a következőképpen hangzik [3]:

A microservicek kicsi és autonóm szervizek, amelyek együttműködnek.

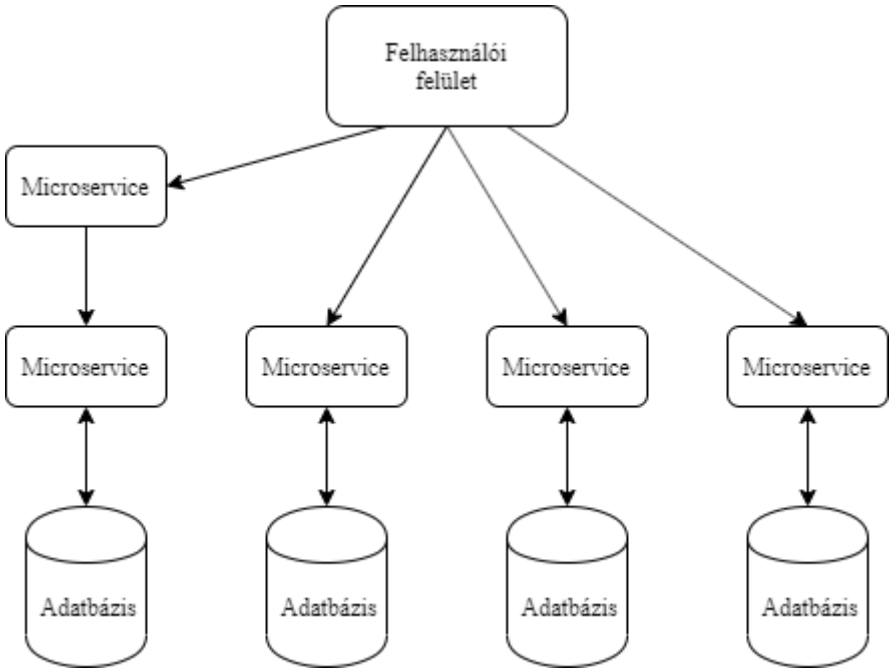
/Sam Newman/

Ezen tervezési minta célja, hogy egy komplex rendszert lebontson kisebb, független és önálló szervizekké. Ennek a függetlenségnek a szerepe kiemelten fontos. A szétbontás határait jellemzően az üzleti logika köré csoportosítják és az adott funkcionalitások határozzák meg, hogy a szoftver mely részei tartoznak egybe. Ehhez egy jó megközelítés az egységes felelősség elve (Single Responsibility Principle), melynek célját az alábbi idézet jól leírja [4]:

Egyben tartani azokat a dolgokat, amelyek ugyanazon okokból változnak és szétválasztani azokat, amik különböző okokból változnak.

/Robert C. Martin/

Ezeknek a módszereknek az használata jellemzően nagy méretű alkalmazásokban előnyös, ugyanis egy bizonyos méret felett nehezen áttekinthetővé válik a kódbázis és ez megnehezíti a továbbfejlesztését és a karbantartását. Emellett további szempont lehet, hogy egy monolitikus megközelítéssel készült megoldás sokkal törékenyebbé válhat az idők során, nehezebben cserélhető le benne egy-egy technológia és új fejlesztők betanítása is sok időt vehet igénybe. Az architektúra általános magasszintű felépítését a 7. ábra mutatja.



7. ábra Microservice architektúra

3.1.3.1 Microservice architektúra előnyei:

Független szolgáltatások:

A többi architektúrával szemben a microservices megvalósítás egyik legnagyobb előnye, hogy kicsi és önálló szolgáltatásokból épül fel. Ennek köszönhetően egymástól teljesen függetlenül fejleszthetők, fordíthatók, tesztelhetők, kiadhatók és telepíthetők. Egy szolgáltatás kiadását nem korlátozza, ha egy másik szolgáltatás még nem készült el, amennyiben annak interfésze ismert és helyettesíthető.

Technológiai heterogenitás:

A szolgáltatások között nincs szoros technológiai függőség. Ennek köszönhetően a különböző komponensekben különböző technológiák használhatók. Ezzel elérhető, hogy a különálló fejlesztőcsapatok a számukra legkézenfekvőbb megoldásokat használják. Akár a programozási nyelvet is szabadon megválaszthatják. A technológiai szabadság mellett, azonban a kommunikációs protokollok és interfészek kötöttek kell maradjanak, ezzel biztosítva, hogy a rendszer képes lesz együtt működni.

Könnyebb érthetőség:

Annak következtében, hogy a rendszerünk több kisebb részre van szétbontva, egy komplex alkalmazás részei könnyebben átláthatóvá válnak. Egy szolgáltatás esetén

jelentősen kevesebb kódot kell karbantartani, mint ha a teljes szoftver kódját egyben kezelnénk. Az új csapattagok betanítási ideje ezzel jelentősen megrövidülhet.

Agilis és párhuzamos fejlesztés támogatása:

A különálló szolgáltatásoknak köszönhetően párhuzamosan több funkció is fejleszthető a rendszerben anélkül, hogy a másik csapatra kéne várakozni. A fejlesztés és üzemeltetés során a felelősség egy adott csapathoz rendelhető, amely csapat felelős a fejlesztéstől a tesztelésen át a telepítésig.

Magas rendelkezésreállás és hibatűrés:

Az egymástól független mikroszolgáltatásoknak köszönhetően egy komponens hibája nem okozza a teljes rendszer leállását. Az is megoldható, hogy a szolgáltatásokból több példány fusson egyidejűleg és egy terhelés elosztó mögé téve ezeket, egy komponens leállása esetén is minden funkció elérhető marad. Ebben az esetben a redundáns példány veszi át a meghibásodott párja szerepét.

Jól skálázható:

A szolgáltatásokból egymástól függetlenül több példány is indítható, így megoldható a rendszer horizontális skálázhatósága. Mivel a program nem egy processzként fut, elegendő a túlterhelt funkciók kiszolgálását végző komponensek felskálázása. Ennek köszönhetően csökkenthető az üzemeltetési költség, mivel kevesebb hardveres erőforrással kiszolgálhatók a rendszer igényei. Alacsonyabb rendszerterhelés mellett könnyen visszaskálázható a rendszer, így ilyen esetben minimálisra szorítható a szoftver erőforrásigénye.

Gyorsabb reagálás:

A felmerülő hibák javítása kevesebb időt vesz igénybe annak köszönhetően, hogy csak az érintett modulban kell kijavítani a hibát és elvégezni a tesztelést. A mikroszolgáltatások verziói külön-külön kiadhatók és telepíthetők. Egy szolgáltatás könnyen visszaállítható egy korábbi verzióra, ha a legfrissebb kiadás esetén valamilyen újonnan felmerülő hiba lép fel. Az is megoldható, hogy egy szolgáltatásból egyszerre több verzió is fusson, így fokozatosan vezethetők be új funkciók az alkalmazásba.

3.1.3.2 Microservice architektúra hátrányai:

Összetett funkciók megvalósítása komplexebb:

Olyan funkciók megvalósítása, amik több szolgáltatás használatát igénylik nagyon komplex lehet. A fejlesztő csapatoknak szorosan együtt kell működniük, hogy megfelelően megtervezzék a rendszer ezen részeit.

Megnövekedett rendszerszintű komplexitás:

A szoftver kisebb egységekre tagolása a lebontott komponensek bonyolultságát lecsökkenti és áttekinthetőbbé teszi a kódokat. A teljes rendszer komplexitása, azonban nagymértékben megnő ennek következtében. Sokkal több elemet kell egyszerre kezelni. Egy ilyen szoftver megtervezése nagy körültekintést igénylő kihívás, sok erőforrást és magas szakértelmet igényel. Az egyik legnagyobb kihívás, hogy minden szolgáltatás a saját adatmodelljében tárolja az adatait, amikre nehéz feladat garantálni a konzisztenciát és a tranzakcionalitást.

Eltérő szolgáltatások:

Habár előnyként jelenik meg, hogy a komponensek, akár különböző programozási nyelveken és más-más technológiákkal készülhetnek ez a következő problémához vezethet. Egy-egy csapat nem tudja gyorsan átvenni a munkát egy másik csapattól. Emellett a telepítés is nehezebbé válhat, mert a különböző technológiáknak különböző környezetekre vannak szükségeik.

Bonyolultabb menedzsment:

Mivel a rendszer sok apró építőelemből áll össze, amik egymástól függetlenül jelenhetnek meg és mindegyik saját verzióval rendelkezik, ezért egy összetett feladattá válik, hogy ezekből a szolgáltatásokból egy együtt is jól működő rendszert lehessen építeni. Érdemes egy kompatibilitás mátrixot karbantartani, amiből kiderül, hogy mely verziók képesek egymással helyesen működni.

Nehezebb üzemeltetés:

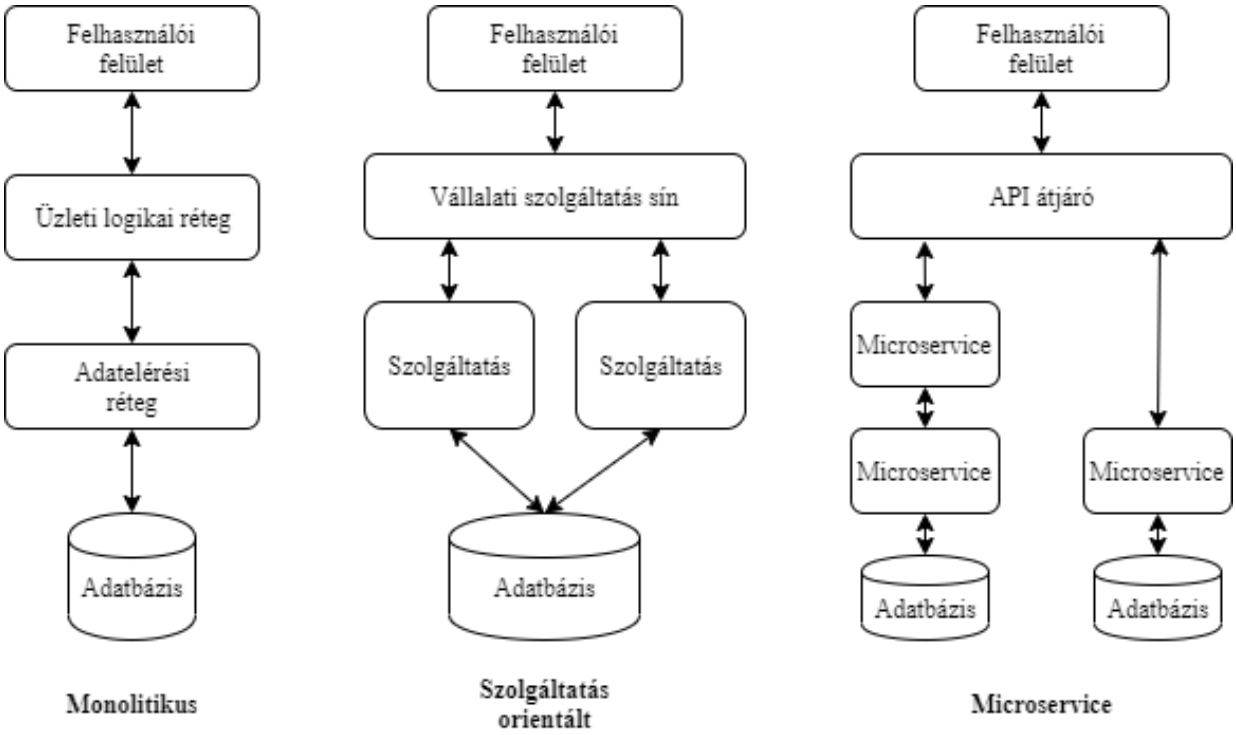
Az üzemeltetési feladatok is nehezednek egy microservice architektúra esetén. Mivel nem egy processzként fut a rendszerünk, hanem adott esetben több tíz vagy akár több százként, ezeket valahogy kezelni kell. A rendszer telepítése is nagyobb kihívás lesz. Ennek köszönhetően érdemes valamilyen automatizmust bevezetni ezekbe a folyamatokba, hogy kevesebb emberi erőforrást emésszen fel az üzemeltetés. Az is egy elterjedt megközelítés, hogy az egyes szolgáltatásokért a hozzárendelt csapatok felelnek és a telepítésről is ők gondoskodnak, azaz szoftverük teljes életciklusát végig kísérik.

Magas szintű tesztek futtatása:

Az előnyök között megtalálható, hogy könnyű a szolgáltatásokat tesztelni, mert kisebb egységekért kell csak felelni. Ezzel szemben eljön az a pont, amikor az elkészült összetett szoftvert egyben szeretnénk tesztelni integrációs vagy rendszer tesztekkel. Az ilyen jellegű tesztek futtatása a monolitikus rendszerekhez képest bonyolultabb feladat, mert komplexeb a tesztrendszer összeállítása.

3.2 Összefoglalás

Összegzésként elmondható, hogy mindhárom alkalmazás architektúrának megvan a létjogosultsága. Az általános felépítésük a 8. ábraán látható. Balra a monolitikus felépítés a központi adatbázissal és az egybe csomagolt üzleti logikai réteggel. Középen helyezkedik el a szolgáltatás orientált architektúra, ahol megfigyelhető a szolgáltatások tagolása. Jobb oldalon, pedig a microservices minta látható, ahol a szolgáltatások önálló adatbázissal rendelkeznek és több kisebb komponensből áll össze az alkalmazás.



8. ábra Monolitikus, Szolgáltatás orientált és Microservice architektúrák általános felépítése

Az összehasonlítás eredményeként elmondható, hogy egyszerű alkalmazások fejlesztéséhez, kimondottan kis csapatok esetén érdemes lehet a monolitikus architektúrát választani. Ezzel szemben egy összetett és komplex megoldás fejlesztéséhez egy jó alternatívát nyújt a microservices felépítés.

Az én célom a diplomamunkám során, hogy egy microservices architektúrájú alkalmazást készítsek és ezen keresztül ismerkedjek meg a fejlesztéshez szükséges technológiákkal és módszerekkel. A munkám eredményeképp pedig egy skálázható, hibatűrő és rugalmas rendszert hozzak létre.

4 Tervezés

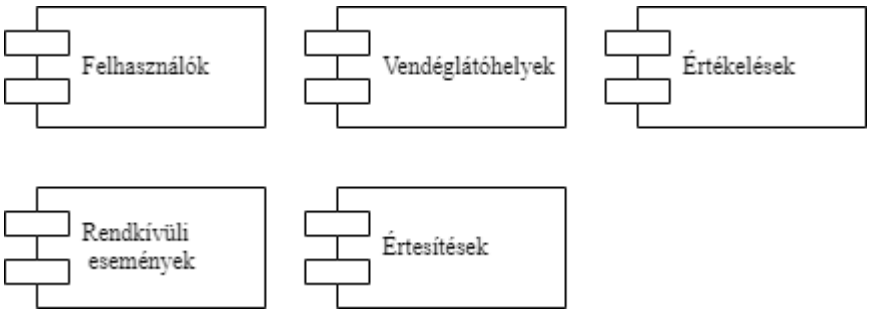
Ebben a fejezetben ismertetem az alkalmazás tervezésének folyamatát és a tervezés során hozott döntések okait. A cél, hogy az elkészült terv alapján egyértelműen implementálható legyen a szoftver.

4.1 Architektúra

Kiindulási pontként a követelmények leírása szolgált számomra. Ezek mentén igyekeztem azonosítani az entitásokat és meghatározni, hogy milyen elvárt bementek és kimenetek jelennek meg a rendszerben. Ennek eredményeként első körben a következő felbontást készítettem:

- Felhasználók
- Vendéglátóhelyek
    - Értékelések
    - Rendkívüli események
- Értesítések

Ebben a megközelítésben a vendéglátóhelyekhez szorosan hozzátartoznak az értékelések és a rendkívüli események. Jobban végig gondolva ezek az entitások tartozhatnának, akár a felhasználók alá is, mivel mindkét entitást felhasználók hozhatják létre. Figyelembe véve az egységes felelősség elvét, mind az értékelések, mind a rendkívüli események megjelenhetnek fő entitásokként. Ennek oka, hogy sem a felhasználók, sem a vendéglátóhelyek adatainak módosítása nincs hatással ezekre az adatokra leszámítva, ha törölnek a rendszerből egy felhasználót vagy vendéglátóhelyet. Ennek eredményeképp az elkészítendő rendszeremet a 9. ábraán látható komponensekre bontottam.



9. ábra Komponensek

Ezek után meghatároztam, hogy milyen funkciók tartoznak az adott komponensek körébe. Ezen a ponton nem volt célom a funkciók részletekbe menő kibontása, inkább a csoportosításra helyeztem a hangsúlyt. Ennek segítségével ellenőriztem, hogy minden követelményt figyelembe vettem-e a tervezés során.

Felhasználók:

- Regisztráció / Felhasználó törlése
- Bejelentkezés / Kijelentkezés
- Felhasználói adatok tárolása / módosítása / törlése

Vendéglátóhelyek:

- Vendéglátóhelyek regisztrálása / szerkesztése / törlése
- Vendéglátóhelyek lekérdezése / keresése

Értékelések:

- Vendéglátóhelyek értékelése / értékelések módosítása / lekérdezése
- Helyek ajánlása más felhasználók értékelései alapján

Rendkívüli események:

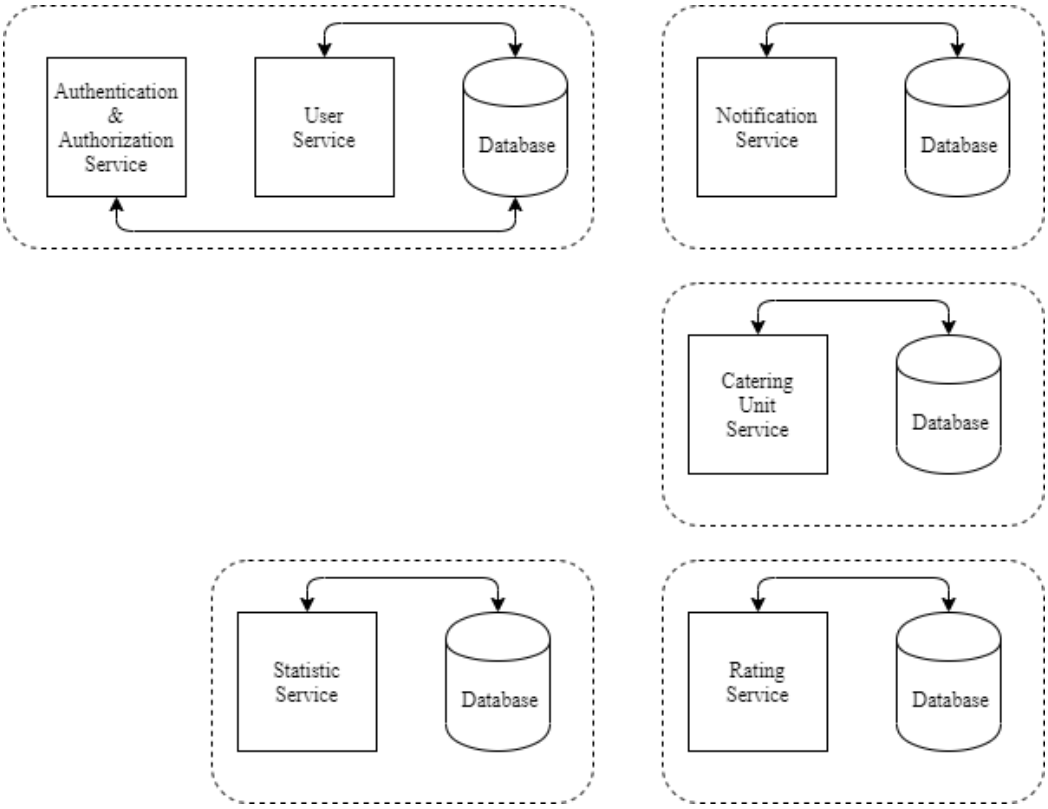
- Rendkívüli események jelentése
- Jelentések lekérdezése

Értesítések:

- Email értesítések küldése a feliratkozott felhasználók számára
- Feliratkozás / leiratkozás értesítésekre

4.1.1 Rendszer szintű architektúra

Ezután elkészítettem egy rendszerszintű architektúrát, ami a megvalósítandó szolgáltatásokat azonosítja. A következőkben számos komponensre angol elnevezést fogok bevezetni. Ennek oka, hogy a fejlesztés nyelve angol, így könnyebb az itt leírtakat az adott kódrészekkel összekapcsolni.



10. ábra Azonosított szolgáltatások

A 10. ábra az eddig azonosított szolgáltatásokat vázolja fel. A felhasználók komponens mentén két szolgáltatás megvalósítása a cél. Az *Authentication and Authorization Service* a felhasználók azonosításáért felelős, aminek részletes folyamatát egy későbbi fejezetben ismertetem. A *User Service* a felhasználói adatokat kezeli. Az ábrán a szaggatott vonal jelöli a szolgáltatások határait. Látható, hogy ez a két szerviz egy határon belül helyezkedik el. Ennek oka, hogy a felhasználók azonosításához szükség van a felhasználói adatokra és ezeket célszerűnek tartottam egy helyen tárolni, továbbá mindkét szolgáltatást tartalmazó program mérete még így is kicsinek mondható. A *Notification Service* az értesítésekért és feliratkozásokért felelős. A *Catering Unit Service* a vendéglátóhelyek adatait kezeli. A *Rating Service* a helyek értékeléseit tárolja és az értékelések alapján készült ajánlásokat szolgáltatja. A *Statistic Service* a korábban rendkívüli események kapcsán azonosított funkcionalitást látja el.

A 10. ábra láthatóan nem tartalmazza a szolgáltatások közötti kommunikációt, a webes kliens kapcsolatát és a mobil kliens helyzetét sem. Ehhez szükséges bevezetni a következő technológiákat és fogalmakat.

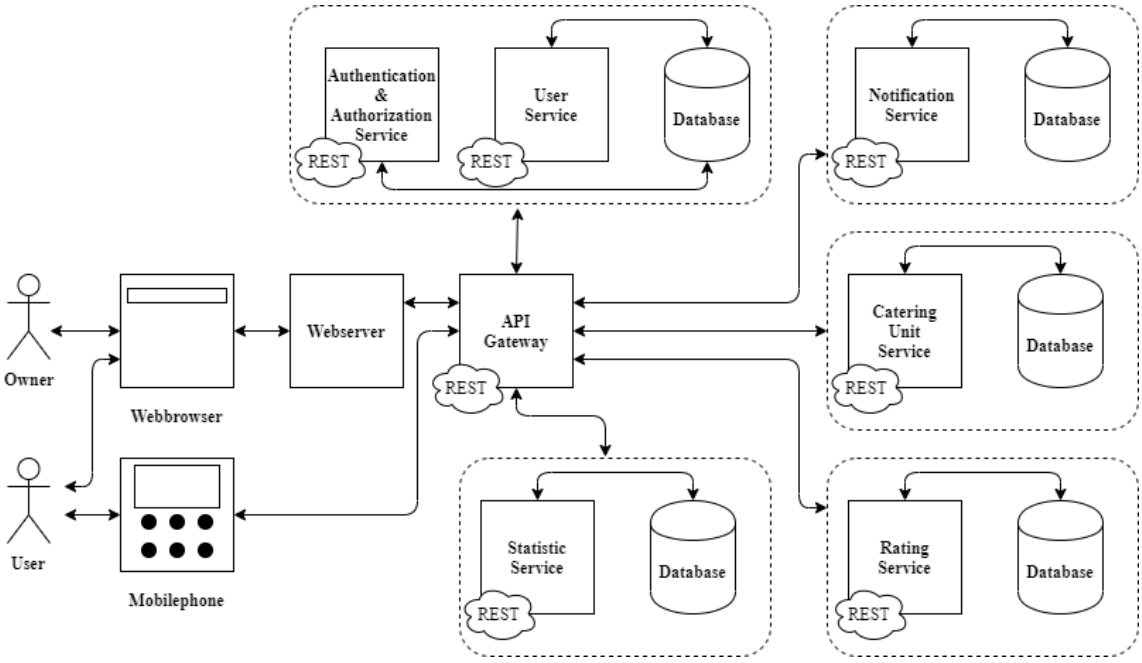
A kommunikációs protokoll kiválasztása során két lehetséges technológiát vettem figyelembe. Az első opció az RMI (Remote Method Invocation – Távoli Metódus Hívás)

volt, a második pedig a REST (Representational State Transfer) [5]. A választásom a REST konvenciókat követő HTTP API-ra esett, mert webes kliensek kiszolgálására ez egy alkalmas és elterjedt megoldás. A szolgáltatások közti kommunikációt mindkét verzió képes kiszolgálni, de mivel a kliensek felé egyértelműen a REST technológiát célszerű használnom, így az alkalmazáson belüli kommunikációhoz is ezt a megoldást választottam. Emelet nagy előnye, hogy technológia és platform független, valamint napjainkban igen elterjedt megoldásról van szó.

A következő megoldandó probléma, amely microservices alkalmazások esetében minden esetben felmerül, hogy a kliensek hogyan érik el az egyes szervizeket. Erre ad megoldást az API Gateway (Application Programming Interface Gateway – Alkalmazásprogramozási felület átjáró) minta [6]. Ez a megoldás egyetlen elérési pontot biztosít a kliensek számára, hogy hozzáférjenek a szerver oldali alkalmazás minden szolgáltatásához. Egy ilyen szolgáltatás irányítja a webes kéréseket és válaszokat a megfelelős szervizek számára.

Ezen felül az elkészítendő weboldal kiszolgálására szükséges egy webszerver, ami HTTP protokollon keresztül elérhetővé teszi a helyileg tárolt weboldalakat.

Ezek után az architektúra a 11. ábraán látható módon változik és lesz teljes abban az értelemben, hogy tartalmazza a jelenleg ismert követelmények kiszolgálásához szükséges komponenseket. A microservicek közötti kommunikációt a 4.1.2-es fejezetben ismertetem.



11. ábra Rendszer architektúrális nézete

A 11. ábra a teljes rendszer magasszintű architektúráját mutatja be. A szolgáltatások esetén a *REST* jelzés a kommunikációs protokollt definiálja. A kliensek a szervizeket az *API Gateway*-en (API átjárón) keresztül érik el, amellyel közvetlenül kommunikál a mobiltelefonos alkalmazás (*Mobilephone*) és a *Webserver* (Webszerver). A *Webbrowser* (Webböngésző) a webszerveren keresztül éri el a szolgáltatásokat és tölti be a HTML oldalakat. Az *Owner* aktor a tulaj felhasználókat reprezentálja, még a *User* aktor az egyszerű felhasználókat.

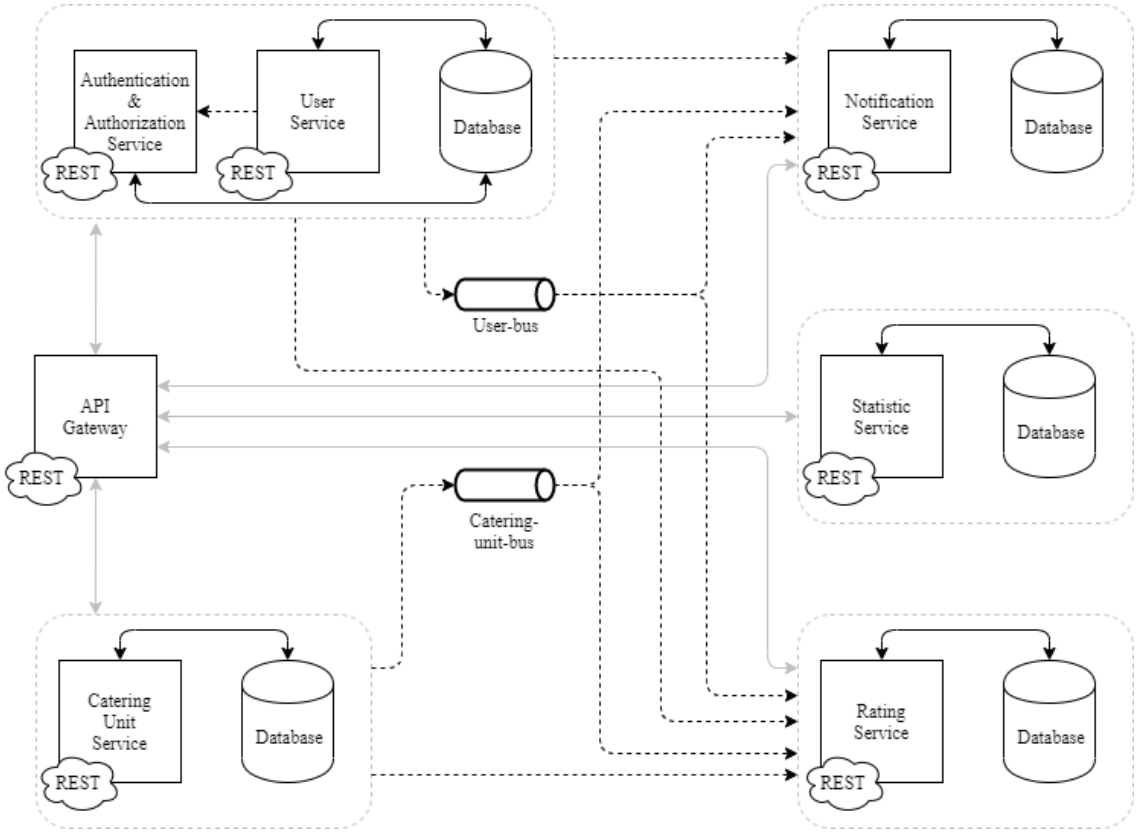
4.1.2 Szerver oldali alkalmazás architektúra

Ebben a fejezetben részletesebb felbontásban ismertetem a szerver oldali alkalmazás architektúráját. Meghatározom azokat a helyeket, ahol szükség van a szolgáltatások közötti kommunikációra.

A *Notification Service* felel az értesítések kiküldéséért és ahhoz, hogy tudja milyen címre kell küldenie az üzeneteket szüksége van arra, hogy kiolvassa a felhasználók adatait a *User Service*-ből. Ezen felül a feliratkozásokat is ez a szolgáltatás kezeli azonban, ha törölnek a rendszerből egy felhasználót vagy vendéglátóhelyet, akkor az ezekhez tartozó feliratkozásokat is törölni kell. A *Rating Service* tárolja a helyek értékeléseit, ebben az esetben validációs célból célszerű lehet, ha a *User Service*-ben ellenőrizni tudja a felhasználók meglétét, illetve a *Catering Service*-ben ellenőrizheti, hogy regisztrálva vannak-e az adott vendéglátóegységek. Ebben az esetben is igaz,

hogyha törölnek a rendszerből egy felhasználót vagy helyet, akkor a hozzájuk kapcsolódó értékelést is törölni kell a rendszerből.

Amikor egyik szerviz olvas ki adatot egy másik szervizből, akkor használható a korábbiakban bevezetett REST interfész és szinkron kommunikációval kielégíthetők a felmerülő igények. A bonyolultabb probléma, amikor egy műveletnek, jelen esetben a törlésnek több komponensre is hatása van. Erre a problémára ad megoldást az aszinkron eseményvezérelt megközelítés, amit a 4.1.2.2-es fejezetben részletesen ismertetek.

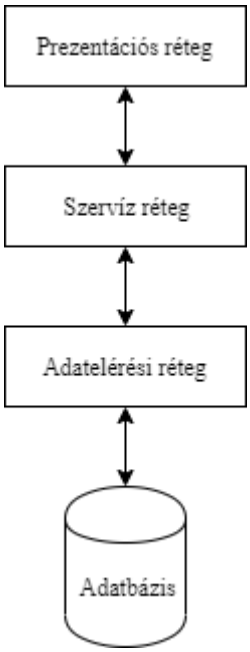


12. ábra Szerver oldali szolgáltatások kommunikációja

A 12. ábra tartalmazza a szervizek közti kommunikáció részleteit. A képen a szaggatott vonalak jelölik a kommunikációs csatornákat és a nyilak iránya az adatok áramlásának irányát határozza meg. A *User-bus* és a *Catering-unit-bus* jelölik az aszinkron eseményvezérelt kommunikációt. Ezeken a csatornákon utaznak a felhasználók és vendéglátóhelyek törlését jelző események.

4.1.2.1 Mikroszolgáltatások architektúrális felépítése

Ebben az alfejezetben bemutatom, hogy az egyes szervizeknek mi a jellemző felépítése.



13. ábra Mikroszolgáltatások architektúrája

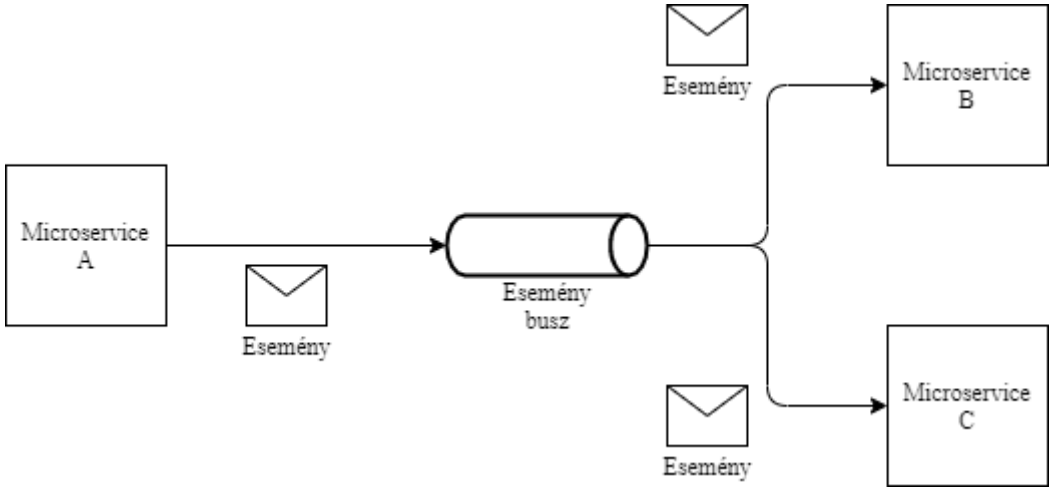
A 13. ábraán látható egy mikroszolgáltatás általános felépítése. Lentről felfelé haladva a felépítés a következő. A legalsó szinten helyezkedik el az adatbázis, ahol az adatok perzisztens tárolása valósul meg. Erre épül az adatelérési réteg, ami a program és az adatbázis közötti kapcsolatért felelős. Középen található a szerviz réteg, ami az alkalmazás működéséhez szükséges logikát tartalmazza. Végül legfelső a prezentációs réteg, ami a REST interfészek implementációit foglalja magában, ez a réteg felelős a kommunikációért.

4.1.2.2 Aszinkron eseményvezérelt kommunikáció

Az aszinkron eseményvezérelt kommunikáció lényege a microservices architektúrában, hogy egy szolgáltatás valamilyen változásról szeretné értesíteni a többi szolgáltatást a rendszerben. Ezt úgy teszi meg, hogy a változás hatására egy úgynevezett eseményt készít, ami leírja változás típusát és annak tartalmát. Például egy esemény típusa lehet egy felhasználó törlése, annak tartalma pedig a törölt felhasználó azonosítója.

Ha szinkron módon hajtaná végre a többi komponens frissítését, akkor meg kéne várnia, amíg minden komponens reagál a változásra és végrehajtja azt. Ezzel szemben az aszinkron megközelítés esetén, elküldi az értesítést és a többi szolgáltatásra bízza, hogy reagálnak-e rá, és ha igen akkor milyen módon.

A legelterjedtebb megoldás erre egy Message broker (Üzenet bróker) használata, ami gondoskodik az üzenetek fogadásáról, tárolásáról, kézbesítéséről, hibatűrésről és skálázhatóságról [7].



14. ábra Aszinkron eseményvezérelt kommunikáció

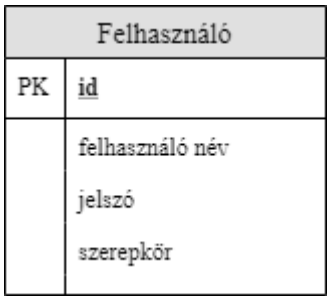
A 14. ábra demonstrálja, a publish/subscribe (publikálás/feliratkozás) eseménybusz működését, melynek lényege, hogy egy szolgáltatás elküld egy eseményt az üzenetkezelő bróker felé, majd az adott eseményekre feliratkozott szolgáltatások megkapják az elküldött üzenetet. Ennek a megoldásnak az előnye, hogy lazán csatolt és könnyen bővíthető, továbbá a küldőnek nem kell figyelembe vennie, hogy az adott komponensek hogyan dolgozzák fel az általa elküldött értesítéseket.

Ez a megközelítés megfelel a Eventual consistency (előbb-utóbb konzisztens) elvnek, melynek lényege, hogy a teljes rendszer nincs állandóan konzisztens állapotban, de előbb-utóbb minden szerviz konzisztens állapotba kerül. Ez a helyzet áll fent például abban az esetben, amikor törölnek az alkalmazásban egy vendéglátóhelyet, ekkor törlődik a vendéglátóhely a *Catering Service*-ben és kiküldésre kerül az esemény, ami leírja ezt a változást. Erre az eseményre reagálva a helyhez tartozó értékeléseket majd törölni fogja az ezért felelős komponens. Ebben az esetben keletkezik egy olyan idősáv, amikor az egyik adatbázisban már nem szerepel az adott vendéglátóhely, de a hozzátartozó értékelések még perzisztálva vannak egy másik komponens adatbázisában. Ez azért engedhető meg, mert az alkalmazás működése szempontjából ez a művelet nem kritikus és nem okoz hibát a rendszer működésében. Ezzel a módszerrel az alkalmazás jól skálázható marad és csökkenthető bizonyos műveletek válaszideje.

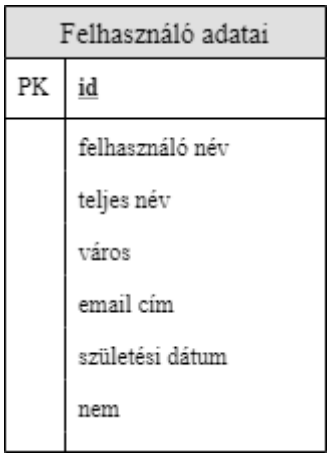
Egy másik probléma, amit felvet az eseményvezérelt megközelítés, hogy az üzenetküldő brókerek a legtöbb esetben nem tudják garantálni a pontosan egyszer küldést. Ennek fő oka, hogy az esetleges bróker leállások után újra kell küldeniük az üzeneteket, de nem tudnak arról meggyőződni biztosan, hogy az események már feldolgozásra kerültek-e a hiba előtt. Ezzel szemben garantálni tudják a legalább egyszer küldést, ami azt jelenti, hogy az esemény biztosan megérkezik a fogadóhoz, de lehet, hogy többször is kézbesítésre kerül. Erre a problémára kell felkészíteni az esemény feldolgozókat. A megoldás, ha a fogadó oldalon az üzeneteket idempotens módon kezeljük, ami azt jelenti, hogy egy művelet többszöri végrehajtása a rendszerben ugyanahhoz az eredményhez vezet, mintha egyszer hajtanánk végre. Például, ha egy törlés üzenet kétszer érkezik meg egy szolgáltatáshoz, akkor a második feldolgozás során nem okozhat hibát a rendszerben, hogy a törlendő elem már nem létezik, mert az első feldolgozás során már törölték.

4.1.2.3 Adatbázis terv

A tervezés következő lépése az adatmodellek megtervezése, amihez egyed-kapcsolat (ER – Entity Relationship) diagrammokat használtam. Az entitásokat sorban minden szolgáltatáshoz meghatároztam, így a következő modellek készítettem.



15. ábra Felhasználó ER diagramm



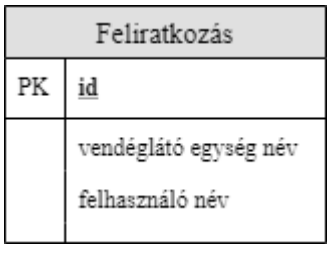
16. ábra Felhasználó adatai ER diagramm

Az alkalmazásban a felhasználó kezelésért és azonosításért a *User Service* felel. A szükséges entitásokat a 15. ábra és 16. ábra mutatja be. A **Felhasználó** entitás reprezentálja a regisztrált felhasználókat. A mezőinek a jelentése a következő:

- id: Egyedi azonosító. Automatikusan generált.
- felhasználó név: A felhasználó kitalált neve a rendszerben. Alkalmazás szinten egyedinek kell lennie.
- jelszó: A felhasználó regisztrációkor megadott jelszava. Később a felhasználónevével és jelszavával azonosíthatja magát.
- szerepkör: Meghatározza a felhasználó típusát továbbá, hogy mely funkciókat használhatja. Lehet tulaj, egyszerű felhasználó vagy adminisztrátor.

A **Felhasználó adatai** a felhasználók személyes adatait írja le, ezen adatok megadása az alkalmazásban opcionális. A mezői a következő jelentésekkel bírnak:

- id: Egyedi azonosító. Automatikusan generált.
- felhasználó név: A felhasználó regisztrációkor megadott neve.
- teljes név: A felhasználó saját neve.
- város: Az a város, ahol a felhasználó a legtöbbet kívánja használni az szolgáltatást.
- email cím: A kliens email címe, ahol a tulajok által küldött értesítéseket fogadja.
- születési dátum: A regisztrált személy születési dátuma.
- nem: Férfi vagy nő.

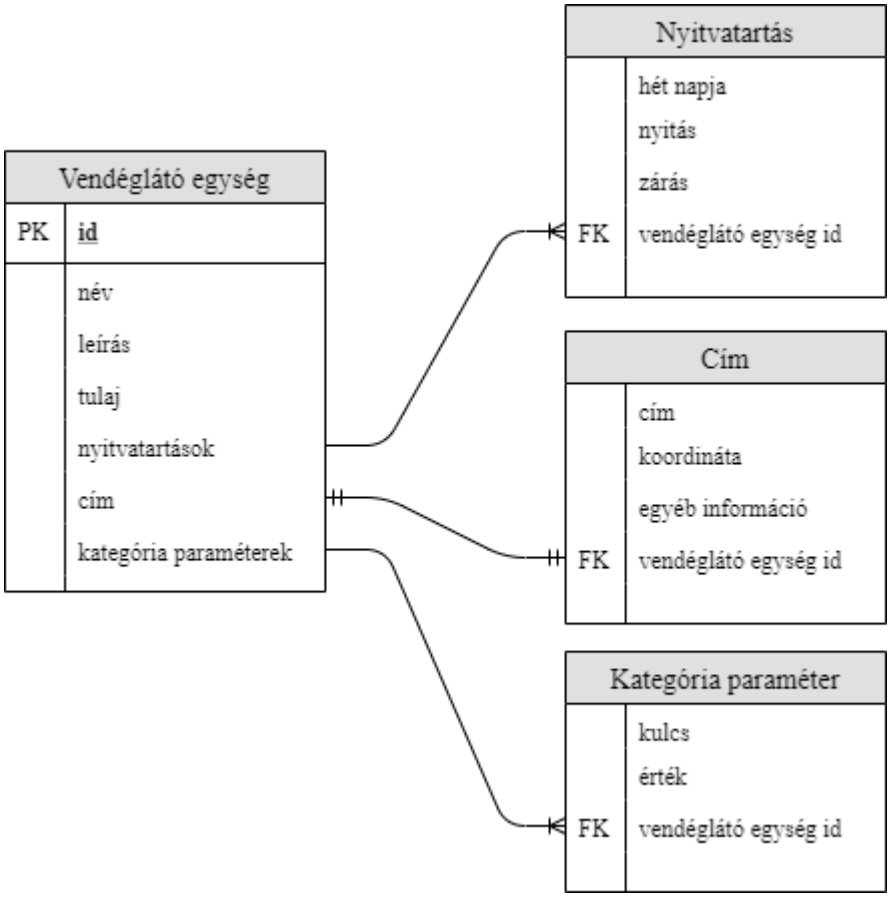


17. ábra Feliratkozás ER diagramm

A *Notification Service* feladata, hogy kezelje az email értesítések kiküldését. A tulajok a vendéglátóegységek részéről üzeneteket küldhetnek azok számára, akik megjelölték, hogy szeretnének értesítést kapni, így egyszerűen tájékozódhatnak a

szórakozni vágyók például arról, hogy az általuk kedvelt hely zárva tart. Azért, hogy a szerviz tudja, hogy kik számára kell üzenteket küldenie szükség van a **Feliratkozás** entitásra. Ez az entitás a következő attribútumokkal rendelkezik:

- id: Generált egyedi azonosító.
- vendéglátó egység neve: Annak a vendéglátó egységnek a neve, ahonnan szeretne értesítést kapni a kliens.
- felhasználó név: A feliratkozott felhasználó regisztrációkor megadott neve.

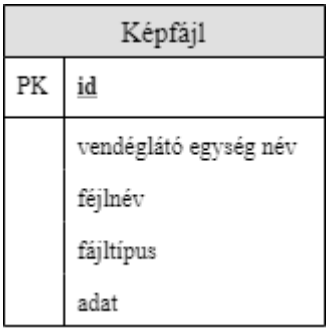


18. ábra Vendéglátó egység ER diagramm

A *Catering Unit Service* felel a vendéglátóhelyek adatainak kezeléséért. A 18. ábra mutatja be a **Vendéglátó egység** entitást az attribútumaival és kapcsolataival. A megtervezett entitások és paramétereik a következők:

- id: Generált egyedi azonosító.
- név: A hely neve. Alkalmazás szinten egyedinek kell lennie.
- leírás: A helyhez tartozó szabad szöveges leírás.

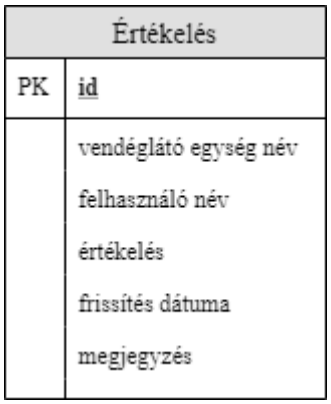
- tulaj: A tulaj felhasználó regisztráláskor megadott neve.
- nyitvatartások: A hét napjaira lebontva tartalmazza a nyitvatartás adatait. A kapcsolódó entitás felépítése a következő:
    - hét napja: A hét napja, amihez a nyitvatartási adat tartozik.
    - nyitás: Az adott napi nyitás időpontja.
    - zárás: A zárás időpontja.
    - vendéglátó egység id: Idegen kulcs a vendéglátó egység entitás felé.
- cím: A hely címét tárolja egy kapcsolódó entitáson keresztül, mely attribútumai a következők:
    - cím: A hely postacíme szöveges formában megadva.
    - koordináta: A vendéglátó egység földrajzi koordinátái hosszúsági és szélességi fokokban megadva.
    - egyéb információ: A címmel kapcsolatos kiegészítő információ. Például leírható, hogy honnan érdemes megközelíteni a helyet.
    - vendéglátó egység id: Idegen kulcs a vendéglátó egység entitás felé.
- kategória paraméterek: Egy gyűjtemény egy adott hely szabadon megválasztható paramétereiről. Megadható például egy hely típusa, mint étterem. A kapcsolódó entitás felépítése:
    - kulcs: A paraméter neve.
    - érték: A paraméter értéke.
    - vendéglátó egység id: Idegen kulcs a vendéglátó egység entitás felé.



19. ábra Képfájl ER diagramm

A regisztrált vendéglátóhelyekhez feltölthetők képek is a rendszerbe, hogy előre meglehessen tekinteni a kiválasztott helyet. Ezeknek a képeknek a tárolására szolgál a **Képfájl** entitás, amit a 19. ábra mutat. Magyarázat az ábrához:

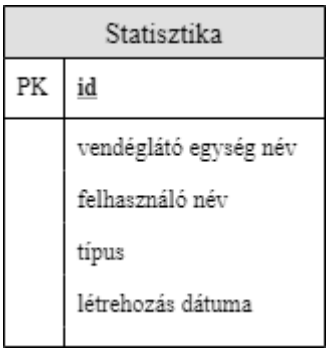
- id: Generált egyedi azonosító.
- vendéglátó egység név: Annak a vendéglátóhelynek a neve, amihez az adott képet feltöltötték.
- fájlnév: A feltöltött képfájl neve.
- fájltípus: A feltöltött állomány kiterjesztése.
- adat: A lementett fájl byte sorokba szerializálva.



20. ábra Értékelés ER diagramm

A *Rating Service* felelős a vendéglátóhelyekhez tartozó értékelések tárolásáért, továbbá az értékelések alapján ajánlásokat készít a felhasználók számra. A 20. ábra írja le tároláshoz szükséges **Értékelés** entitást, amely paraméterei a következők:

- id: Generált egyedi azonosító.
- vendéglátó egység neve: A vendéglátóhely beregisztrált neve.
- felhasználó név: Az értékelést adó felhasználó regisztrációkor megadott neve.
- értékelés: Az értékelés pontszáma 1-től 5-ig terjedő skálán.
- frissítés dátuma: Az értékelés utolsó módosításának dátuma.
- megjegyzés: Szöveges vélemény az értékeléshez.



21. ábra Statisztika ER diagramm

A *Statistic Service* tárolja az úgynevezett nemvárt eseményeket, amiket a felhasználók adhatnak meg a rendszerben, hogy ezzel tájékoztatást nyújtsanak társaik számára. Az adatok tárolására a **Statisztika** entitást terveztem, amit a 21. ábra demonstrál az alábbi paraméterekkel:

- id: Generált egyedi azonosító.
- vendéglátó egység név: A vendéglátóhely regisztrált neve.
- felhasználó név: A jelentést leadó felhasználó regisztrációkor megadott neve.
- típus: A jelentett esemény típusa. Például telt ház vagy hosszú sorban állás várható.
- létrehozás dátuma: A jelentés leadásának ideje.

4.1.2.4 REST interfész terv

Az általam választott kommunikációs konvenció a REST. Ebben a fejezetben ismertetem az alkalmazáshoz tartozó interfészek tervét, amik szorosan kapcsolódnak a tárolandó adatokhoz és a megvalósítandó funkciókhoz. A tervet a szervizek szintjére bontva ismertetem és egy rövid leírást adok az egyes végpontok elvárt működéséről.

Az adatok JSON (JavaScript Object Notation) formátumban utaznak a kommunikáció során.

Az *Authentication & Authorization Service* felel a felhasználók azonosításáért, ennek megfelelően a következő végpontok szükségesek:

Metódus típusa	Elérési útvonal	Rövid leírás
POST	/api/v1/authenticate/login	Felhasználó bejelentkeztetése.

POST	/api/v1/authenticate/logout	Felhasználó kijelentkeztetése.
POST	/api/v1/authenticate/refresh	Biztonsági token frissítése.

A *User Service* felelőssége a felhasználói adatok kezelése. A feladata ellátásához az alábbi végpontok megvalósítása szükséges:

Metódus típusa	**Elérési útvonal**	**Rövid leírás**
POST	/api/v1/users/sign-up	Felhasználó regisztrálása.
DELETE	/api/v1/users	Bejelentkezett felhasználó törlése.
DELETE	/api/v1/users/{username}	A *username* paraméterként megadott felhasználó törlése.
GET	/api/v1/users/exists/{username}	Megadja, hogy a *username* paraméterként definiált felhasználó létezik-e a rendszerben.
POST	/api/v1/users/userinfo	Felhasználói adatok rögzítése a bejelentkezett felhasználó számára.
PUT	/api/v1/users/userinfo	Felhasználói adatok frissítése a bejelentkezett felhasználó számára.
GET	/api/v1/users/userinfo	Felhasználói adatok lekérdezése a bejelentkezett felhasználó számára.
GET	/api/v1/users/userinfo/{username}	Felhasználói adat lekérdezése a *username* paraméterként megadott felhasználó számára.
GET	/api/v1/users/userinfo/bulk	Csoportos felhasználói adatlekérdezés.

A *Catering Unit Service* a rendszer legnagyobb szolgáltatása, ennek megfelelően ez a komponens felel a vendéglátóhelyek kezeléséért. A funkciói halmaza a következő módókon lesz elérhető:

Metódus típusa	Elérési útvonal	Rövid leírás
GET	/api/v1/cateringunit/all	Az összes vendéglátóhely lekérdezése.
GET	/api/v1/cateringunit /owned	A tulajok lekérdezhetik a saját helyeiket.
DELETE	/api/v1/cateringunit/{id}	Az *id* azonosítóval rendelkező hely törlése.
POST	/api/v1/cateringunit	Új vendéglátó regisztrálása az alkalmazásba.
PUT	/api/v1/cateringunit/{id}	Az *id* azonosítóval rendelkező hely adatainak frissítése.
GET	/api/v1/cateringunit/{id}	Az *id* azonosítóval rendelkező hely adatainak lekérdezése.
GET	/api/v1/cateringunit/ exists/{cateringUnitName}	Visszaadja, hogy a névvel (*cateringUnitName*) rendelkező vendéglátó egység létezik-e a rendszerben.
GET	/api/v1/cateringunit/search	Egy megadott keresési paraméter alapján keres a helyek a között.
GET	/api/v1/cateringunit/nearest	Visszaadja a megadott koordinátákhoz legközelebb található helyeket.

A szolgáltatás egy másik funkciója, hogy kezelje a vendéglátóhelyekhez feltöltött képeket. Ehhez a következő funkciók szükségesek:

Metódus típusa	Elérési útvonal	Rövid leírás
POST	/api/v1/cateringunit/image/ upload/{cateringUnitName}	Kép feltöltése a rendszerbe, a *cateringUnitName* paraméter által megadott hely számára.
GET	/api/v1/cateringunit/image/	A megadott *fileId* azonosítóval rendelkező kép letöltése.

	download/{fileId}	
GET	/api/v1/cateringunit/image/ getIds/{cateringUnitName}	Visszaadja a *cateringUnitName* paraméterben megadott helyhez tartozó képek azonosítóit.
DELETE	/api/v1/cateringunit/image/{fileId}	Törli a *fileId* azonosítóval rendelkező képet az adatbázisból.

Az értesítések és feliratkozások kezelését a *Notification Service* látja el. A feladat kiszolgálásához szükséges végpontok az alábbiak:

Metódus típusa	**Elésrési útvonal**	**Rövid leírás**
PUT	/api/v1/notification/ subscribe/{cateringUnitName}	A bejelentkezett felhasználó feliratkozhat a *cateringUnitName* paraméterben megadott hely értesítéseire.
PUT	/api/v1/notification/ unsubscribe/{cateringUnitName}	A bejelentkezett felhasználó a feliratkozáshoz hasonlóan le is iratkozhat az értesítésekről.
GET	/api/v1/notification/ isSubscribed/{cateringUnitName}	Ez a végpont megadja, hogy a bejelentkezett felhasználó fel van-e iratkozva a *cateringUnitName* paramétéként átadott hely értesítéseire.
POST	/api/v1/notification/ email/subscribed/{cateringUnitName}	A tulajok ezen a végponton keresztül küldhetnek értesítést a felirakozók számára a *cateringUnitName* paraméter által megadott vendéglátóhely nevében.

A vendéglátó egységek értékeléseit a *Rating Service* kezeli, továbbá ez a szolgáltatás felelős az ajánlott helyek meghatározásáért. A szervizhez kapcsolódó interfész terv a következő:

Metódus típusa	**Elérési útvonal**	**Rövid leírás**
PUT	/api/v1/rating/rate	Értékelés elmentése a bejelentkezett felhasználó nevében a megadott vendéglátóhely számára.
GET	/api/v1/rating/ catering/{cateringUnitName}	A *cateringUnitName* paraméterben megadott helyhez tartozó értékeléseket adja vissza.
GET	/api/v1/rating/specific	Visszaadja a bejelentkezett felhasználó és egy megadott helyhez tartozó értékelést.
GET	/api/v1/rating/recommended	A bejelentkezett felhasználó számára szolgáltat ajánlott vendéglátóhelyeket.

A nemvárt események bejelentését a *Statistic Service* kezeli. A funkciók kiszolgálásához a szolgáltatásnak az alábbi interfészre van szüksége:

Metódus típusa	**Elérési útvonal**	**Rövid leírás**
GET	/api/v1/statistic/{cateringUnitId}	Visszaadja a *cateringUnitId* azonosítóhoz tartozó vendéflátóhelyek statisztikáit.
POST	/api/v1/statistic/ {type}/{cateringUnitId}	Rögzít egy *type* típusú bejelentést a *cateringUnitId* azonosítóval ellátott helyhez.

4.2 Biztonság

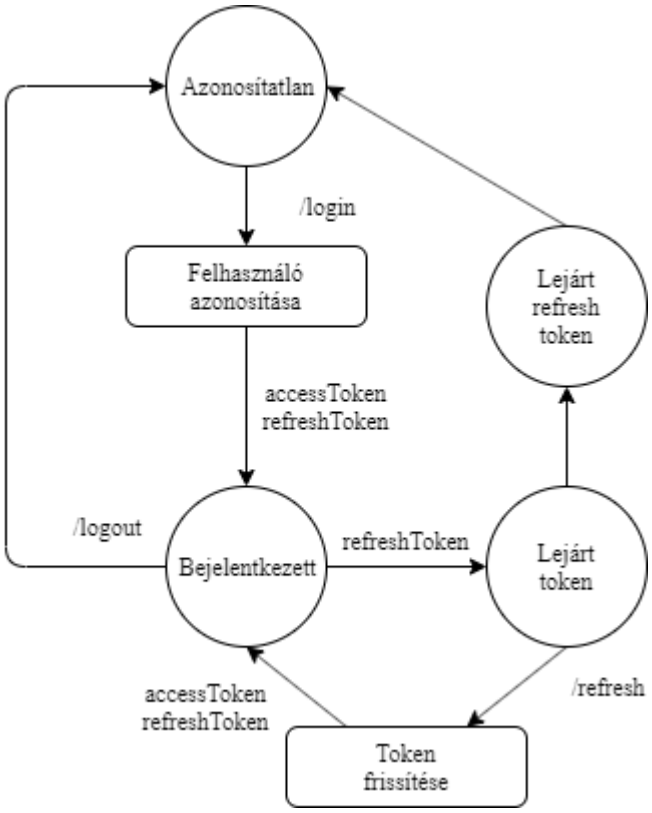
Az alkalmazás a nyílt hálózaton elérhető lesz, ezért szükség van annak szabályzására, hogy kik jogosultak a használatára. Megoldandó kihívás a felhasználók azonosítása és annak kezelése, hogy mely típusú felhasználónak mihez van joga a rendszerben.

A probléma megoldására az alkalmazásban token alapú azonosítást fogok használni. Egészen pontosan a JWT (JSON Web Token) ipari szabványt, mely elterjedt módja a biztonságos információmegosztásnak egy alkalmazás résztvevői között [8]. A kiállított token tartalmazza az azonosításhoz szükséges adatokat úgy, mint a felhasználó neve és jogosultságai.

A JWT három részből áll. Az első része az úgynevezett Header rész, ami tartalmazza a token típusát és a kódolási algoritmust. A Payload-ban találhatók a felhasználó szükséges adatai és kiegészítő információk (claims). A harmadik rész az aláírást tartalmazza.

4.2.1 Felhasználók azonosítása

Ebben a fejezetben ismertetem a JWT alapú azonosítás folyamatát és technikai részleteit.



22. ábra Felhasználó azonosítása

A 22. ábra mutatja be a felhasználó azonosításának folyamatát. A kiinduló állapot, hogy a kliens *Azonosítatlan*, azaz ilyen esetben nem használhatja az alkalmazást. Ezután a 4.1.2.4-es fejezetben bemutatott */login* végpont használatával azonosíthatja magát. Ehhez meg kell adnia a regisztrált felhasználónevét és a jelszavát. Sikeres bejelentkezés esetén a rendszer biztosít egy *accessToken*-t, ami egy egyszerű JWT rövid érvényességi idővel és egy *refreshToken*-t, ami egy hosszabb érvényességű token arra a célra, hogy új *accessToken*-t lehessen igényelni anélkül, hogy újra meg keljen adni a felhasználónév és jelszó párost. Ezután kerül valaki *Bejelentkezett* állapotba, amikor használni tudja a szoftvert. Ebből az állapotból a */logout* funkció segítségével kijelentkezhet és újra alapállapotba kerülhet a kliens. Egy másik szcenárió, ha az *accessToken*-nek lejár az érvényességi ideje és a *Lejárt token* tulajdonságot veszi fel egy felhasználó. Ebben az esetben, ha érvényes *refreshToken*-nel rendelkezik, akkor a */refresh* funkció segítségével újra *Bejelentkezett* állapotba kerülhet egy új token páros birtokában. Egy másik eset, ha a *Lejárt token* esetén a frissítéshez szükséges kulcs is érvénytelenné válik, ekkor a kliens újra alapállapotba kerül.

A bejelentkezett felhasználónak az alkalmazás használata során minden HTTP kérés esetén el kell küldenie az *accessToken*-t a kérés fejlécében (Header). Ehhez az

úgynevezett Bearer sémát választottam, ami azt jelenti, hogy az Authorization Header-ben utazik a token és az értéke előtt szerepel egy Bearer előtag.

Authorization: Bearer <accessToken>

A biztonság szempontjából fontos, hogy a kérések során nem kerül elküldésre a frissítéshez használt token, így ez nehezebben hozzáférhető illetéktelen személyek számára. Továbbá, ha illetéktelen személyek megszerzik az *accessToken*-t annak rövid élettartama miatt csak korlátozott ideig élhetnek vele vissza.

4.2.2 Jogosultságok kezelése

Egy másik feladat a felhasználók azonosítása során, hogy szabályozni lehessen a funkciókhoz való hozzáférést a rendszerben. Ehhez szükség van jogosultságok bevezetésére, amik a felhasználó szerepköröket reprezentálják az alkalmazásban. Az általam bevezetett jogosultságok az alábbiak:

- Tulaj felhasználó szerepköre a rendszerben: ROLE_OWNER
- Egyszerű felhasználó szerepköre a rendszerben: ROLE_USER
- Adminisztrátor szerepköre: ROLE_ADMIN

Egy kliens azonosítása után a kiállított JSON Web Token-ben kerül eltárolásra a felhasználó jogosultsága. A HTTP hívások kezelése során az alkalmazásban szerepelnie kell egy kérésmegszakítónak, ami hitelesíti a biztonsági tokent, amit a kérés fejlécéből olvas ki, továbbá megvizsgálja, hogy a felhasználónak van-e jogosultsága egy adott végpont használatához.

5 Implementáció

Ebben a fejezetben bemutatom a fejlesztéshez felhasznált főbb technológiákat és fontosabb tulajdonságaikat. Prezentálom az elkészült szoftvert és annak felépítését. Részletesen ismertetem a szerver oldali alkalmazást, a webes klienst és a mobil klienst is.

5.1 Szerver oldali alkalmazás elkészítése során használt technológiák

A szerveroldali alkalmazásfejlesztéshez a JAVA nyelvet választottam, mert ebben a technológiában volt a legtöbb meglévő tapasztalatom. A függőségek kezeléséhez, kód fordításához, szoftver csomagolásához és csomagok kezeléséhez a Maven eszközt használtam. Ennek előnye, hogy egyetlen fájlból, az úgynevezett *pom.xml*-ből menedzselhető egy szoftverprojekt, így könnyen áttekinthető és módosítható. Egy másik kiemelten fontos szempont, hogy hordozhatóvá teszi a kódbázist.

Az alkalmazás fejlesztéséhez a Spring keretrendszer elemeit használtam. A microservicek Spring Boot alkalmazásokként futnak, így a futtatásukhoz elegendő egy webkonténer és nincs szükség teljes webszerverre [9]. A szükséges modulok a lefordított szoftver mellé vannak csomagolva, így a szervizek egyetlen JAR (Java Archive) fájlként kezelhetők. Az általam felhasznált specifikus Spring eszközöket az implementáció ismertetése közben említem majd meg.

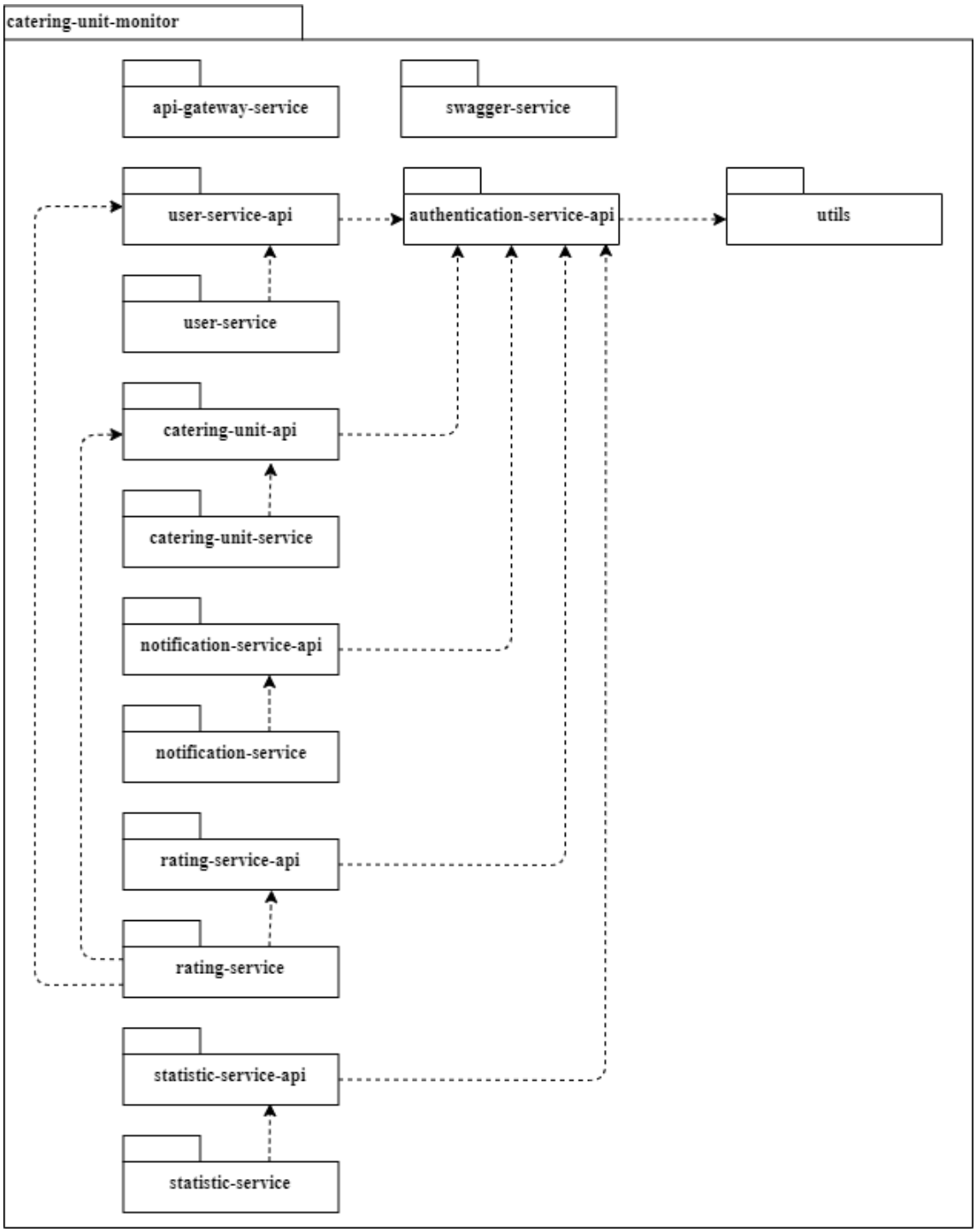
Az adatok tárolására a MySQL relációs adatbázis szervert választottam, ami napjaink egyik legelterjedtebb adatbázis kezelője. A tesztek során a H2 In-Memory adatbázist használtam, mert beágyazott módban és gyorsan futtatható, ami a tesztek esetén kézenfekvő.

Az üzenetkezelés feladatának ellátásához az AMQP (Advanced Message Queuing Protocol) protokollra esett a választásom. Egy másik felmerülő lehetőség lett volna a JMS (Java Message Service) szabvány használata, de ez elsősorban JAVA nyelvhez készült és nem szerettem volna ennek mentén esetleges korlátozást vinni a rendszerembe. Üzenet bróker funkcióit a RabbitMQ látja el, ami egy széleskörben elterjedt nyílt forráskódú üzenet bróker [10].

5.2 Szerver oldali alkalmazás felépítése

Ebben a fejezetben bemutatom a szerver oldali szoftver kódjának felépítését. Ismertetem a felhasznált technológiákat és módszereket.

A fejlesztés során a kódbázisomat GIT verziókezelő rendszerben tároltam, így könnyebben követhetők voltak a változások és akárhonnan hozzáfértem a szoftverem kódjához. A microservicek kódját egy központi mappában helyeztem el és a szervizeket különböző Maven modulokként valósítottam meg. A gyakorlatban gyakran alkalmazzák azt a megközelítést, hogy minden mikroszolgáltatást külön tárolnak, így egyértelműen szétválaszthatók és külön-külön verziózhatók. Az én esetemben a kód egyben tartásának célja az egyszerűség elve volt, mivel egyedül dolgoztam a projekten és a modul szintű szeparáció kielégítette a felmerülő igényeket. A projekt struktúrája a következő 23. ábraán látható, ahol az elkészült szoftvercsomagokat és a szaggatott nyilakkal a köztük lévő függőségeket jelöltem.



23. ábra Projekt struktúra

A *catering-unit-monitor* az úgynevezett szülő projekt, amely összefogja az alá tartozó modulokat és tartalmazza a minden szolgáltatás szempontjából szükséges függőségeket. A struktúrában az *-api* végű modulok tartalmazzák a microservicek használatához szükséges interfészek leírásait. A *-service* végű modulok pedig a konkrét mikroszolgáltatásokat valósítják meg. A következőkben a 11. ábra által meghatározott komponensek mentén csoportosítom a fent bemutatott modulokat.

Az *Api Gateway* komponens kódját az *api-gateway-service* modul tartalmazza. Ennek célja a kérések és válaszok irányítása a szolgáltatások és kliensek között, ezáltal egy központi elérési pont szerepét tölti be. A szerviz megvalósításához a Spring Cloud Gateway könyvtárat használtam, amely kimondottan erre a célra készült. A forgalomirányításhoz szükséges adatok a konfigurációs fájlban megadhatók, ami az én esetemben az *application.yml* fájl. Ezen felül definiálhatók egyedi kérésmegszakítók is, amik minden kérés esetén lefutnak és módosítják azokat.

Az *Authentication & Authorization Service* interfészének leírását tartalmazza az *authentication-service-api* modul. Az interfészt leíró modulok minden esetben tartalmazzák a DTO (Data Transfer Object) leírásokat, amik a kommunikáció során küldött objektumok formátumát határozzák meg. Ezen felül itt kerülnek definiálásra a szervizek működése során használt egyedi exception objektumok, valamint a prezentációs rétegben szereplő kontroller osztályok interfészei is. A leírt REST kontroller interfészek és a Spring Cloud OpenFeign könyvtár felhasználásával a szolgáltatások végpontjai egyszerűen hívhatókká válnak más szervizek számára. A Feign egy deklaratív REST kliens megoldás, ami natív kód szintű hívások mögé rejti a rendszerben lezajló HTTP kommunikációt. Ezzel mentesíti a fejlesztőt a végpontok elérési útjának kezelésétől, a kérések és válaszok dekódolásától és a bonyolult konfigurációtól. Az *authentication-service-api* a többi api leíró modultól eltérően tartalmazza a Feign használatához szükséges konfigurációs fájlt és egy egyedi kérésmegszakítót, aminek célja, hogy amikor egy szerviz hívást indít egy másik szolgáltatás felé, akkor az úgynevezett Security Context (Biztonsági Kontextus) alapján hozzáadja a kéréshez az azonosított felhasználó biztonsági tokenjét és megjelölje a hívás fejlécében, hogy ez egy alkalmazáson belüli hívás. További tartalma a modulnak a biztonsági tokent feldolgozó kérésmegszakító, ami minden HTTP kérés esetén ellenőrzi a JWT tokent és hitelesíti a felhasználót a rendszer számára. Ennek implementálásához felhasználtam a Spring Security könyvtárat. Ezek a plusz kódrészek minden szolgáltatás számára elérhetők, így növeltem az újra felhasználhatóság lehetőségét.

A *User Service* komponenshez tartozó interfészleírást a *user-service-api* modul tartalmazza. A szerviz megvalósítása a *user-serive* modulban található és magában foglalja az *Authentication & Authorization Service* implementációját is. Egy-egy microservice kódjának felépítésére a következő struktúra jellemző:

- configuration

Ez a csomag tartalmazza az alkalmazás konfigurációjához szükséges kódrészeket úgy, mint a biztonságért felelős beállítás és az üzenet brókerhez kapcsolódó konfiguráció.

- controller

  A REST interfészek implementációit tartalmazó csomag. A 13. ábra prezentációs rétegként hivatkozik a szoftver ezen részére. Minden esetben a szerviz réteggel kommunikál és kiszolgálja a beérkező kéréseket.

- dao

  Az adatbázis entitások leírásait tartalmazza.

- repository

  Ebben a könyvtárban találhatók az adatbázis elérésért felelős kódrészek. A 13. ábra adatelérési rétegét jelöli. Az implementáció során a Spring Data JPA (Java Persistence API) könyvtárat használtam, ami támogatást nyújtott az adatbáziskapcsolatok kezelésében és elfedte az adatmanipulációhoz szükséges SQL kódokat.

- service

  A 13. ábra szerviz rétegének megfelelő implementációt tartalmazó csomag. Az adatelérési és prezentációs rétegek között teremt kapcsolatot és itt található az alkalmazások logikai részei.

- util

  Számos microservice esetén készítettem egy util nevű csomagot, ami az általános több helyen felhasználható kódokat gyűjti össze. Ilyen például az entitás osztályok és DTO osztályok konvertálását végző kód.

Minden mikroszolgáltatásban található egy *Application* elnevezésű osztály, ami a futtatáshoz szükséges *main()* metódust tartalmazza és itt kerülnek beállításra a szoftver alapvető konfigurációi. Ezen felül a *resources* könyvtárban helyezkedik el az *application.properties* fájl, ami az alkalmazás konfigurációját tartalmazza. Ez a konfiguráció az alkalmazás indítása során felülírható.

A *Catering Service*-hez tartozó kódrészek a *catering-unit-api* és a *catering-unit-service*. Ez a szerviz felel a vendéglátóhelyek adatainak kezeléséért.

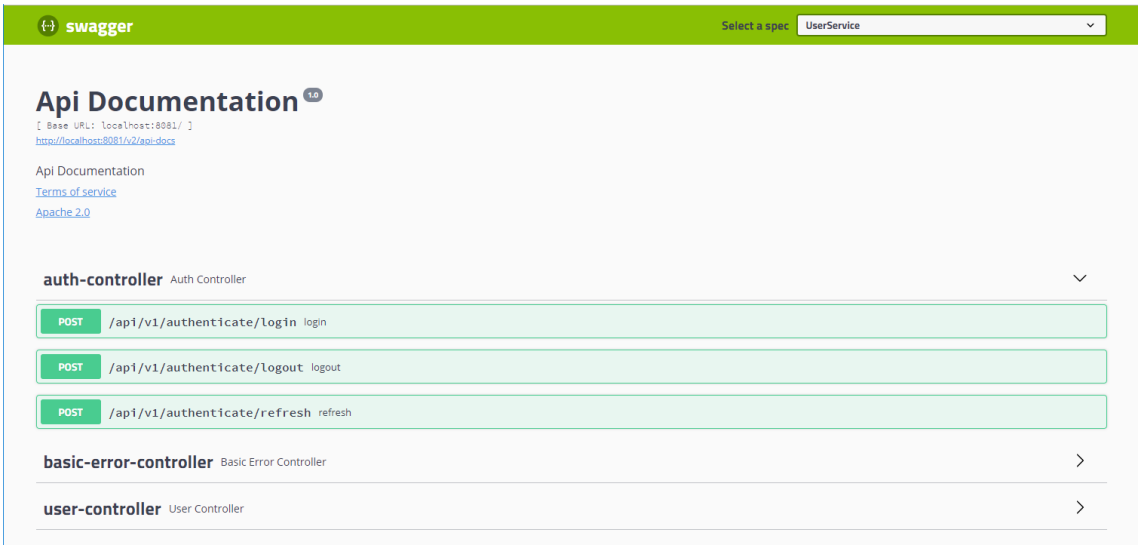
A *Notification Service* implementációjáért a *notification-service-api* és *notification-service* kódokrészek felelnek. A microservice működéséhez szükség van egy SMTP (Simple Mail Transfer Protocoll) szerverre, amin keresztül elküldhetők az email értesítések a felhasználók számára. Az email küldés implementációjához a JavaMailSender külső Spring könyvtárat használtam, ami leegyszerűsítette a kommunikációt az üzenetküldő szerverrel.

A *Rating Service* megvalósítása a *rating-service-api* és *rating-service* mappákban található az ismertetett struktúrába rendezve. Ez a szolgáltatás felel a helyek értékeléseiért és ajánlásokat készít a kliensek számára az eltárolt vélemények alapján.

A *statistic-service-api* és *statistic-service* könyvtárakban található a *Statistic Service*-hez kapcsolódó kódbázis. Ez a mikroszolgáltatás felel a nemvárt események tárolásáért. A szerviz a bejelentéseket perzisztálja és egy időzített feladat során 5 percenként ellenőrzi az elmentett statisztikákat és törli a 2 óránál régebbi bejegyzéseket.

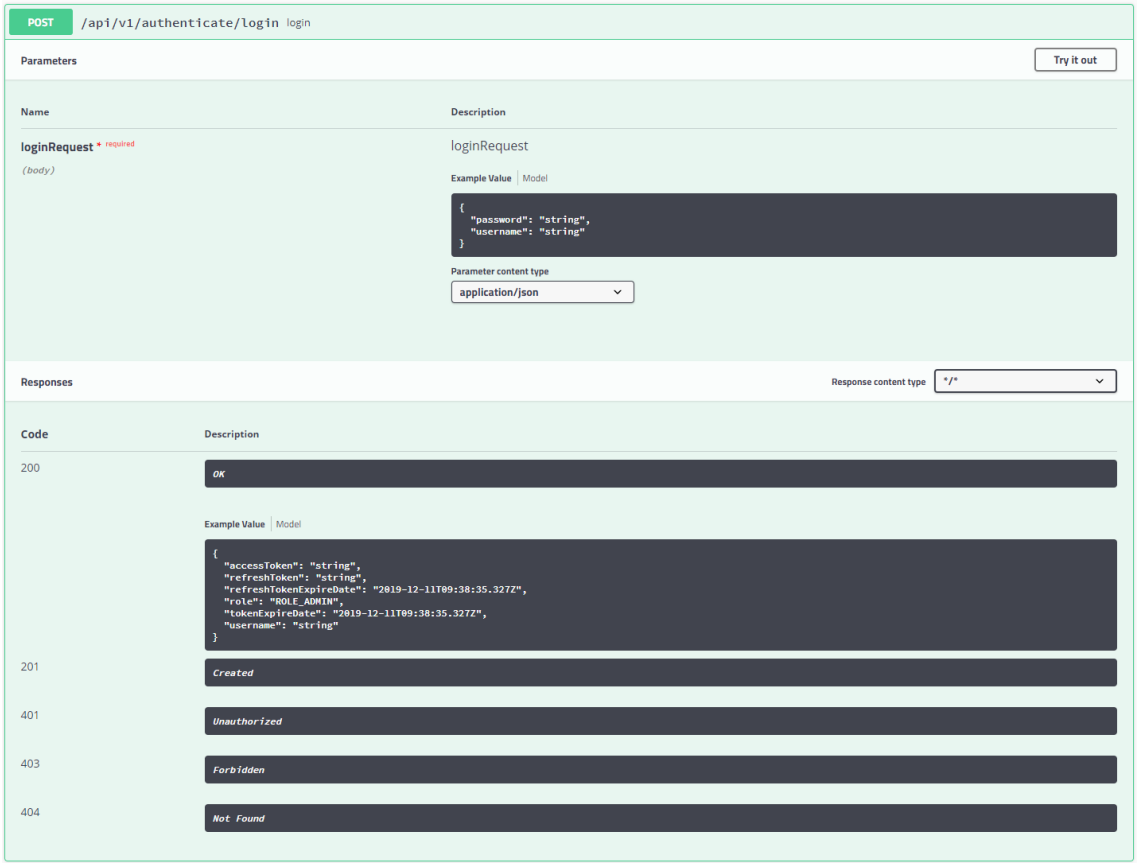
A *utils* csomagban minden szolgáltatás számára megosztott újra felhasználható kódrészek találhatók. Itt definiáltam az eseménykezelő által használt eseménytípusokat és az üzenetek felépítését. Megtalálhatók emellett az authentikáció folyamatához használt azonosítók és konstans értékek is.

A *swagger-service* az API dokumentációért felel, ehhez a Swagger eszköztárat használja fel, ami egy REST API leírására szolgáló eszköz. A dokumentáció generáláshoz felhasználtam a Springfox Swagger2 könyvtárát és a megjelenítéshez a Springfox Swagger UI-t. Az *Application* osztályon elhelyezett *EnableSwagger2* annotáció engedélyezi a program számára, hogy összegyűjtse a kontroller osztályokat és leírást készítsen hozzájuk. Az általam elkészített szerviz azt a célt szolgálja, hogy egyetlen felületen elérhetővé tegye az összes microservice API dokumentációját, ami a gyakorlatban a 24. ábraán látható.



24. ábra Swagger UI API dokumentáció

A 24. ábra jobb felső sarkában a *Select a spec* opció segítségével kiválaszthatók a microservicek. A képernyő közepén felsorolva megtalálhatók a REST kontrollerek és egy lenyíló menüben láthatóvá válnak a hozzájuk tartozó végpontok. A végpontokhoz részletesebb leírás is tartozik, amit a 25. ábra mutat be.



25. ábra Swagger UI végpont dokumentáció

Ezen a felületen megtekinthetők az elérési utak, kérések struktúrái és a lehetséges válaszok, amik a kliensek fejlesztése során nyújtanak segítséget.

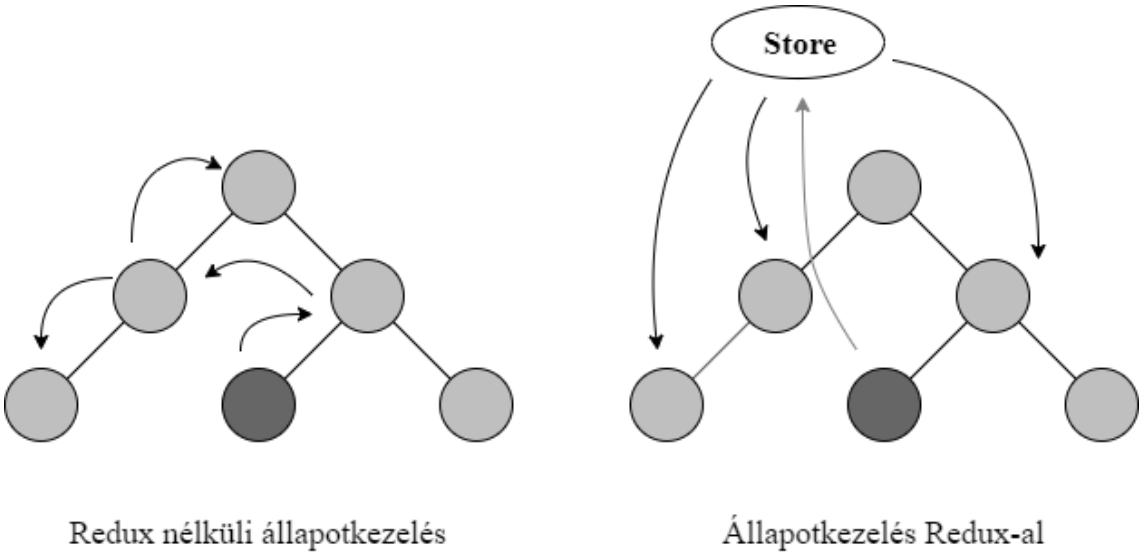
5.3 Webes kliens elkészítése során használt technológiák

A webes kliens célja, hogy egy adminisztrációs felületet biztosítson a felhasználóknak az adataik szerkesztéséhez és a tulajoknak lehetővé tegye, hogy kezeljék a vendéglátóhelyeik adatait.

Az elkészült kód nyelvének a JavaScriptet választottam, ami egy objektumorientált lehetőségeket támogató szkript nyelv. A nyelvről fontos megemlíteni, hogy gyengén típusos és elsősorban weboldalak elkészítéséhez használják. Napjainkban a legelterjedtebb webes fejlesztéshez használt technológia, ezért kézenfekvő volt számomra a választás. Népszerűségének köszönhetően számos oktatóanyag elérhető hozzá és nagyon aktív közösség áll mögötte [11].

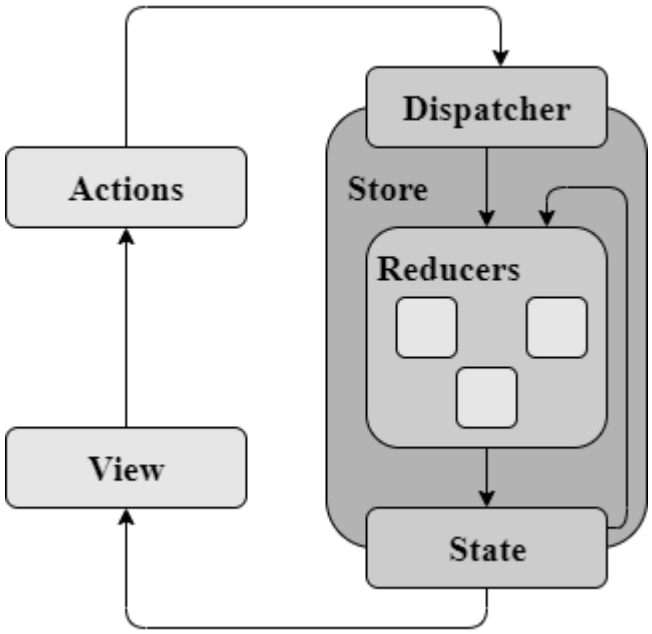
A felhasználói felület elkészítéséhez felhasználtam a React JavaScript könyvtárat. A munkám során szerettem volna betekintést nyerni ennek a népszerű eszköznek a működésébe. A komponens alapú támogatással egyszerűen készíthettem újra felhasználható elemeket és a renderelési technológiájának köszönhetőn egy-egy elem frissítése során nem volt szükség a teljes oldal újra betöltésére, ezzel simább felhasználói élmény volt kialakítható [12].

Az állapotkezelés megkönnyítése érdekében a Redux JavaScript könyvtárat vettem igénybe. Ennek célja az volt, hogy az alkalmazásom számára kiszámítható és központosított állapottárolóként szolgáljon. A 26. ábra bal oldalán látható, hogy a Redux használata nélkül bonyolulttá válhat az állapotok eljuttatása több komponensen keresztül. Ezzel szemben a jobb oldalon látható, hogy a Redux használatával az állapotok egy központi tárolóban (*Store*) helyezkednek el és minden komponens ezen keresztül fér hozzájuk. A képen sötéttel jelölt elemek reprezentálják a változást kezdeményező programrészeket.



26. ábra Redux állapotkezelés demonstráció [13]

A 27. ábra részleteiben mutatja be hogyan zajlik az állapotkezelés az alkalmazáson belül. A nyilak iránya az adatfolyamot írja le, ebből láthatóvá válik, hogy ez a minta az egyirányú adatáramlást támogatja. A *View* elem jelképezi a megjelenített képernyők kódjait. Ezen nézetek a változások hatására úgynevezett *Action* eseményeket készítenek a változás leírására, amelyet a *Dispatcher* eljuttat a központi *Reducer* számára. A *Reducer* célja, hogy egy bemeneti állapotból és a változás leírásából egy új frissített állapotot hozzon létre. Az ábrán látható, hogy a *Reducers* bemente a *State*, ami az aktuális állapotot írja le és a *Dispatcher* által továbbított esemény. A központi *Reducer* a feladatot szétoszthatja más-más a funkciót ellátó komponensek között. A folyamat végén a *State* változásáról a *View* elemek értesülnek és ennek megfelelően frissíthetik nézeteiket. A következő fejezetben ismertetem az elkészült kódomban, melyik réteget hogyan valósítottam meg.



27. ábra Redux adatfolyam [14]

Az Axios HTTP klienst választottam a REST interfészek eléréséhez. Az alap Fetch megoldás funkciói mellett, további hasznos lehetőségeket biztosított számomra. Ilyen funkció többek közt az automatikus JSON adat transzformáció vagy a kérés és válasz interceptorok használatának lehetősége. A kérésmegszakítók használata a biztonsági tokenek kezelésében nyújtott számomra segítséget. A megvalósítás részleteire az 5.4 fejezetben térek ki.

A Reactstrap könyvtár előre elkészített Bootstrap 4 dizájnnal rendelkező komponenseket tartalmaz. Ennek köszönhetően gyorsan és minimális dizájnt leíró kód segítségével készíthető esztétikus megjelenésű weboldal. A webes kliensem elkészítése során felhasználtam ezt a könyvtárat.

5.4 Webes kliens felépítése

Az elkészült kliens kódja a *catering-app-client* mappában található. A függőségek kezeléséhez a Node Package Managert használtam, ennek megfelelően a projekt gyökerében található *package.json* fájl tartalmazza a szükséges függések felsorolását és verzióit.

Az alkalmazás kódja az *src* almappában található. A *components* mappa írja le a megjelenítendő képernyőket és a hozzájuk tartozó elemeket. Ezek a nézetek a 27. ábra *View* rétegének felelnek meg. Az állapotkezelés működését végig kísérve a következő

lépés az *action_creators* almappa, amiben azok a komponensek találhatók, amik a Redux eseményeket (action) állítják elő. Az volt a célom ebben az esetben, hogy az API hívásokat kimozgassam a nézetek kódjából és így jobban áttekinthető legyen a szoftver. Ezeket az API hívásokat minden esetben a nézetek kezdeményezik és az általuk végrehajtott műveletek eredménye egy változást leíró eseményként jelenik meg a rendszerben. Az esemény típusok leírása *constants* mappa *actionTypes.js* fájljában találhatók. Egy action két részből áll, az első része az esemény típusa, a másodig része pedig a változáshoz kacsolódó adat, az úgynevezett payload. A 27. ábra *Store* objektuma a *store* mappában kerül inicializálásra. A *Reducers* komponens implementációi a *reducers* mappában találhatók. Látható, hogy különálló reducer funkciót készítettem a feladataik szerint csoportosítva. Ezeket az *index.js* fájlban megírt *combineReducers()* funkció regisztrálja a gyökér reducer számára, ami a beérkező eseményeket a megfelelő reducer-ekhez irányítja a működés során. A nézetek az állapotváltozásokról a *componentWillReceiveProps()* funkción keresztül értesülnek és ennek megfelelően frissítik belső állapotukat és a megjelenített nézetet.

Az 5.3-as fejezetben bevezettem az általam használt Axios HTTP klienst, valamint megemlítettem, hogy a JWT token kezelése során kérés és válasz interceptorokat használtam fel. Ennek a megvalósítása a *utils* mappa *api.js* bejegyzésében található.

```
const instance = axios.create(...);
```

Először példányosítottam az általam használni kívánt HTTP klienst és megadtam a szerver oldali alkalmazás elérési útját. Ezután két féle interceptort valósítottam meg. Az első egy kérésmegszakító.

```
instance.interceptors.request.use(
    config => {
        const token = localStorage.getItem(authConstants.JWTTOKEN);
        if (token) {config.headers['Authorization'] = 'Bearer ' + token; } else
{ config.headers['authorization'] = ''; }
        return config;
    }, error => { Promise.reject(error); });
```

Ennek célja, hogy a felhasználó bejelentkezése után a *localStorage* tárolóba elmentett tokenek közül kiválassza az *accessToken*-t, ami az azonosításhoz szükséges és ezt minden

kérés fejlécéhez hozzáadja. Az azonosítás folyamatáról egy részletes képet adok a 4.2-es fejezetben. A válaszok érkezése során is lefut egy interceptor kód, ami a következőképp néz ki.

```
instance.interceptors.response.use((response) => { return response;}, error =>
{   const originalRequest = error.config;
    if (error.response.status !== 403) { return Promise.reject(error);}
    if (originalRequest._retry === undefined) {
        originalRequest._retry = true;
        const username = localStorage.getItem(authConstants.USERNAME);
        const refreshToken = localStorage.getItem(authConstants.REFRESHTOKEN);
        return axios.post(baseUrl + '/api/v1/authenticate/refresh',
            {
                userName: username,
                refreshToken: refreshToken
            })
            .then(res => {
                localStorage.setItem(authConstants.JWTTOKEN, …);
                localStorage.setItem(authConstants.REFRESHTOKEN, …);
                error.response.config.headers['Authorization'] = 'Bearer '+ …;
                return axios(error.response.config);
            }).catch(error => {
                unsetLocalStorage();
                return Promise.reject(error);
            });}
    return Promise.reject(error);});
```

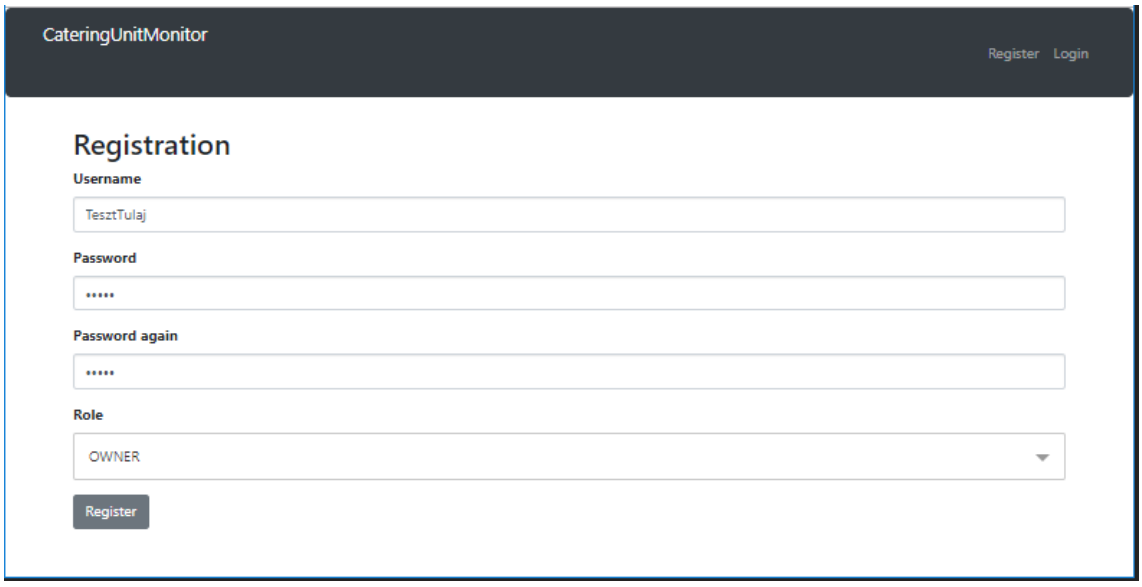
Sikeres válasz esetén nem hajt végre plusz műveletet, azonban hibás válasz esetén, ha a válasz státuszkódja 403, azaz az elérni kívánt erőforrás rejtett, akkor egy próbát tesz arra, hogy a *localStorage*-ban eltárolt *refreshToken* használatával új token párost igényeljen a felhasználó számára.

5.4.1 A webes kliens bemutatása

A technológiai áttekintés után bemutatom az elkészült webes kliens működését és az elérhető funkciókat. A 28. ábra a webes felület nyitóoldalát mutatja, ahonnan a felhasználó választhatja a regisztrálás (Register) vagy a bejelentkezés (Login) opciót.

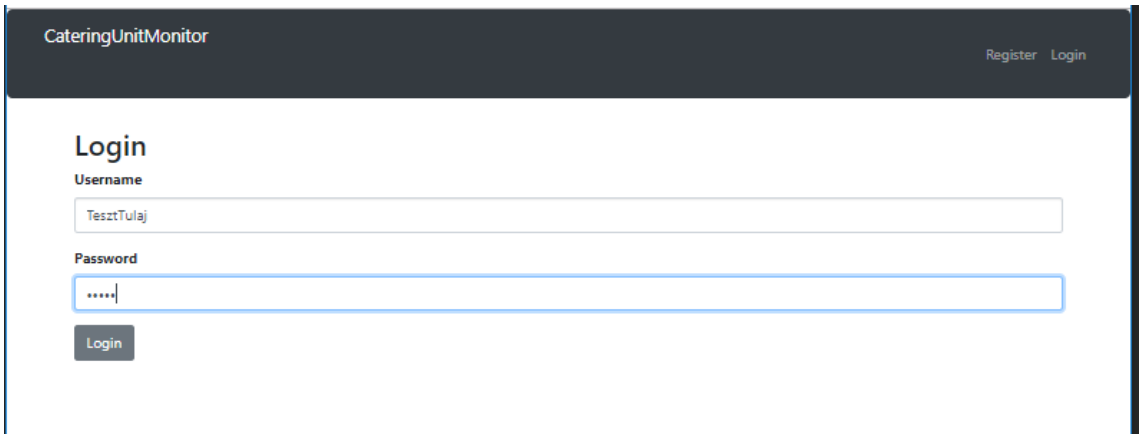
28. ábra Webes kliens nyitóoldal

A regisztrálás során meg kell adnia egy felhasználónevet és egy jelszót, amit kétszeri megadással kell érvényesíteni, továbbá ki kell választani, hogy tulaj vagy egyszerű felhasználó szerepkörben szeretné használni az alkalmazást. A regisztrációs felületet a 29. ábra mutatja be.



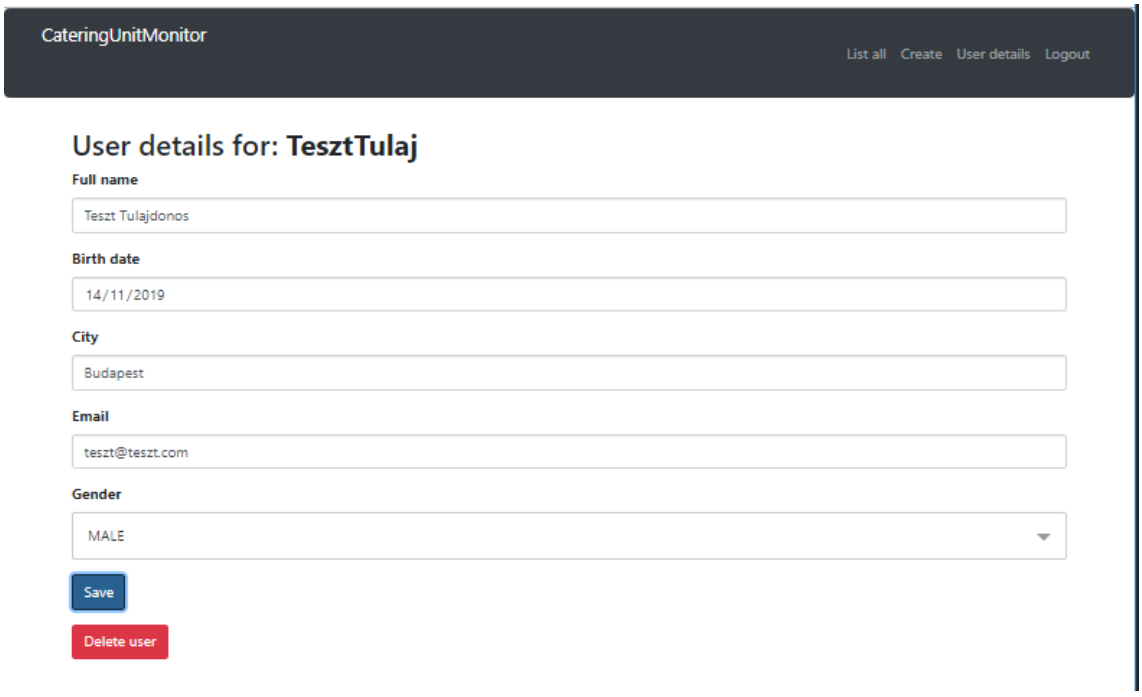
29. ábra Webes kliens regisztráció

A bejelentkezés felülete a 30. ábraán látható, ahol a regisztráció során megadott felhasználónév és jelszó páros segítségével lehet belépni a webes felületre.



30. ábra Webes kliens bejelentkezés

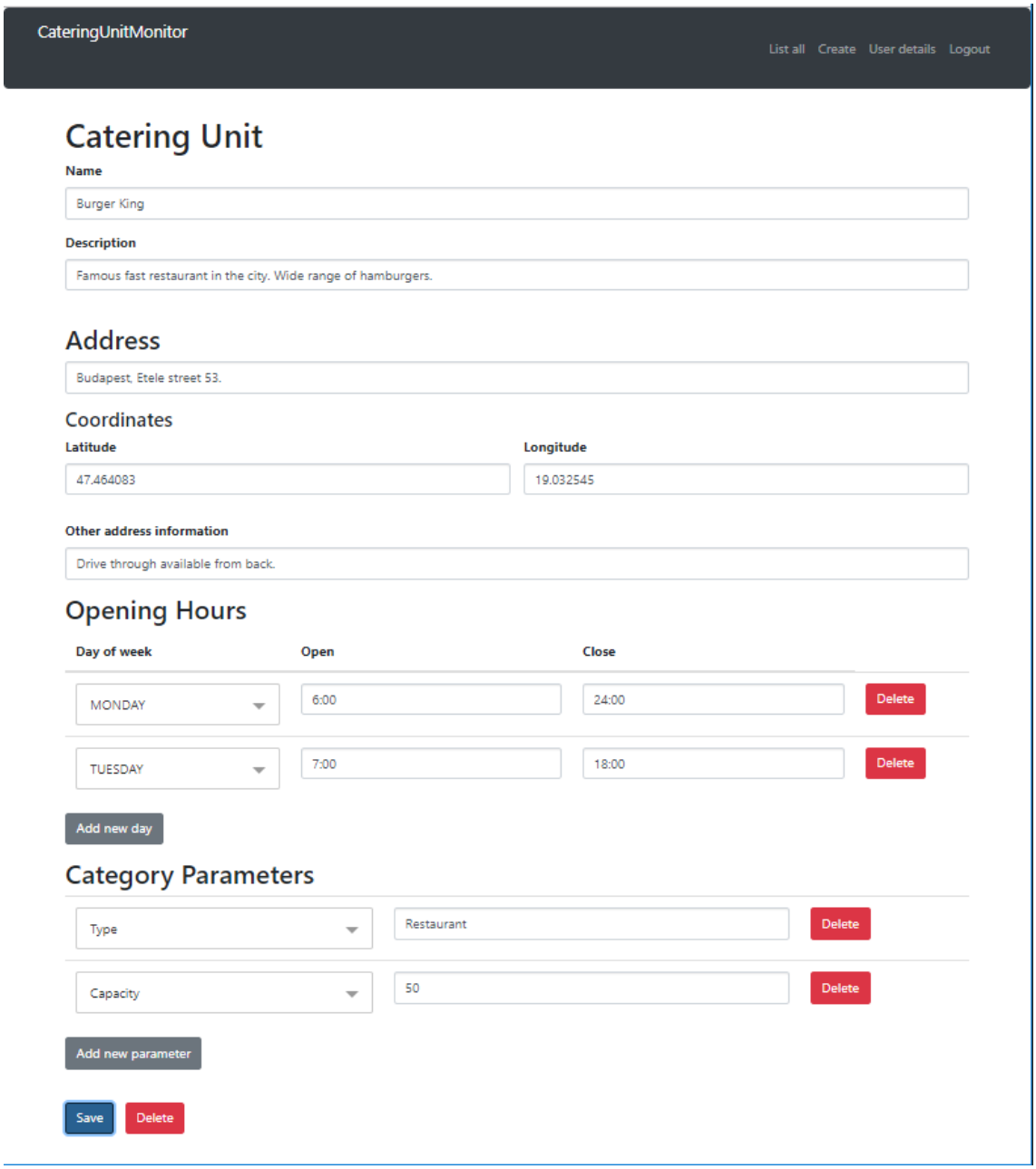
Bejelentkezés után a felhasználó számára a fejlécben elérhetővé válnak a program által biztosított funkciók, amik a tulajok esetében rendre az összes saját vendéglátóhely megjelenítése (List all), vendéglátóhelyek regisztrálása (Create) és a felhasználói adatok kezelése (User details). Az egyszerű felhasználók ezen a menüsoron kizárólag a felhasználói adatok szerkesztése menüpontot érik el.



31. ábra Webes kliens felhasználói adatok szerkesztése

A 31. ábra a felhasználói adatok szerkesztésének menetét szemlélteti, ahol megadható sorrendben a személy teljes neve, születési ideje, városa, email címe és a neme. Ezen a felületen, ezen felül lehetősége van a kliensnek törölnie magát a rendszerből a *Delete user* gomb használatával. Az első alkalommal a *Save* gomb regisztrálja a

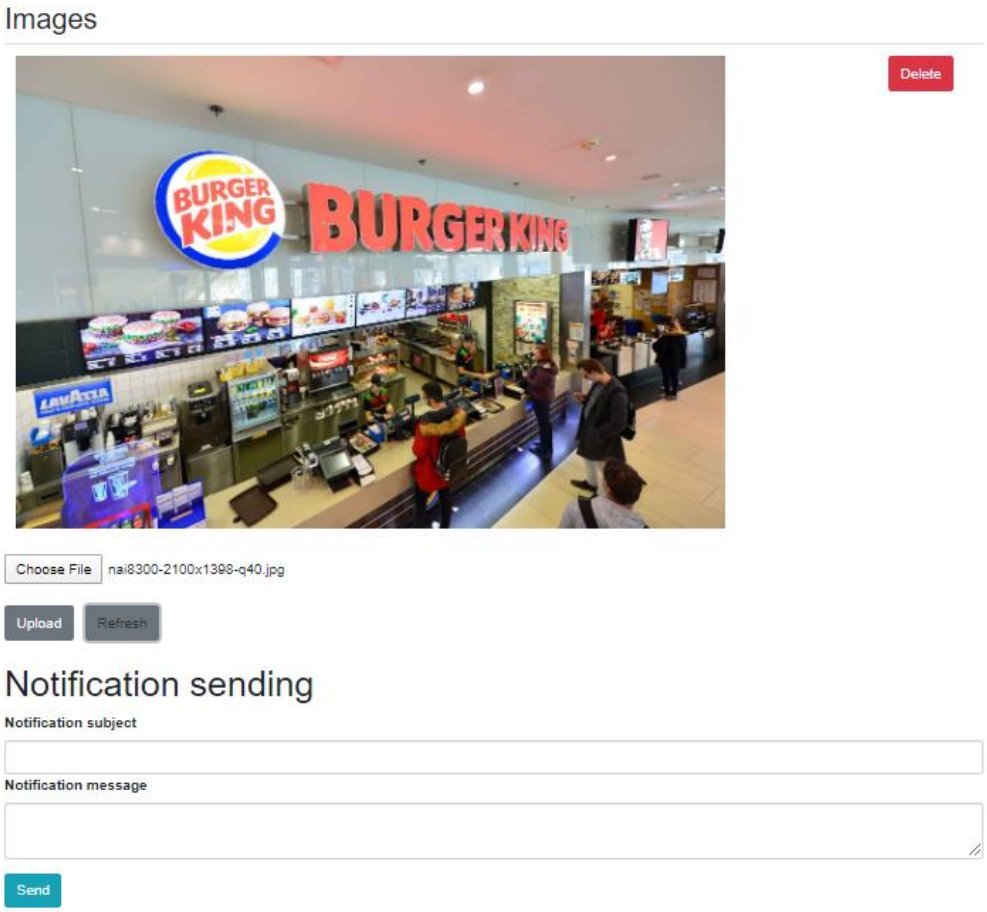
felhasználók adatait a rendszerben és minden későbbi esetben ezek az adatok módosíthatók és a *Save* gomb megnyomásával felülírhatók.



32. ábra Webes kliens vendéglátóhely regisztrálása

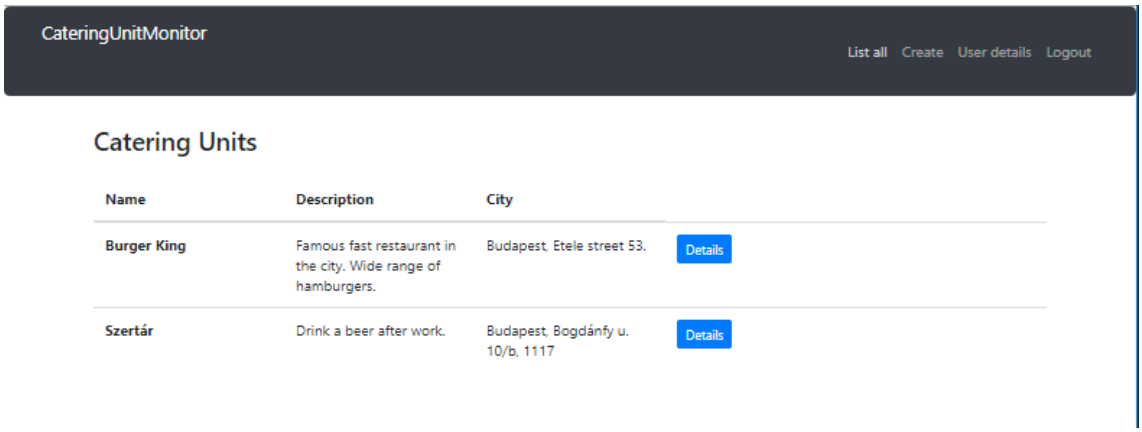
A webes felület legösszetettebb képernyője a vendéglátóhelyek regisztrálásáért és szerkesztéséért felel. A 32. ábra egy új hely regisztrálásának első lépését mutatja be. A képen fentről lefelé haladva megadható a hely neve, leírása, címe, a címhez tartozó földrajzi koordináták, a címhez tartozó kiegészítő információk, továbbá a nyitva tartás napokra bontva és a kategória paraméterek úgy, mint a hely típusa vagy kapacitása. Abban az esetben, ha a felhasználó a *Create* opció használatával navigál erre az oldalra,

akkor minden esetben egy új vendéglátó egységet vehet fel a rendszerbe, de ugyanezt a képernyőt használhatja az adatok szerkesztéséhez is vagy a *Delete* gomb segítségével törölheti a rendszerből az adott helyet.



33. ábra Webes kliens vendéglátóhely képfeltöltés és értesítés küldés

A már létrehozott vendéglátó egységek esetén lehetősége van a tulajoknak képeket feltölteni és email üzenetek küldeni a feliratkozók számára. Ezeket az opciókat mutatja be a 33. ábra, amin látható, hogy az *Images* szekció alatt képek feltölthetők és törölhetők és a *Notification sending* részben pedig az üzenetek bevitele után a *Send* gomb megnyomásával email értesítések küldhetők.



34. ábra Webes kliens tulaj vendéglátóhelyei

A következő 34. ábra tartalmazza a tulajok számára kilistázott vendéglátóhelyeket és erről a felületről a *Details* opció használatával a korábban bemutatott szerkesztőfelületre navigálja őket az alkalmazás, ahol megtekinthetik és frissíthetik az adatokat.

5.5 Mobil kliens elkészítése során használt technológiák

A mobil kliens célja, hogy az általános felhasználók igényeit kiszolgálja, ehhez a 2.3.1-es fejezetben leírtak szerint készítettem egy Android platformot támogató mobil alkalmazást. Ebben a fejezetben bevezetem a fejlesztés során felhasznált technológiákat és módszereket.

Az általam elkészített alkalmazás programozási nyelve a JavaScript, amit a webes kliens implementálásakor is használtam. A klienst a React Native keretrendszer felhasználásával készítettem el, aminek segítségével natív alkalmazások készíthetők Android és iOS platformokra a React keretrendszer felhasználásával. Habár a szoftveremet Android platformra készítettem és nem volt követelmény az iOS támogatása mégis ezt a keretrendszert választottam, mert szerettem volna ismeretet szerezni a technológiáról, aminek a térhódítása egyre jelentősebb a piacon.

A függőségek kezeléséhez a Node Package Manager eszközt használtam. A projekt létrehozása és kezelése során az Expo nyílt platformot hívtam segítségül, ami leegyszerűsítette számomra a projekt vázának összeállítását, a futtatását és a hibakeresést.

A kliensben található állapotok kezeléséhez a React Hooks megoldást választottam, ami a React 16.8-as verziójának egyik új eszköze. Ennek segítségével a 26. ábra által bemutatott problémát a Redux használata nélkül oldottam fel és nem volt

szükségem külső könyvtárak használatára, valamint áttekinthetőbb implementációt tett lehetővé.

A HTTP hívások kezeléséhez a 5.3-as fejezetben ismertetett Axios-t használtam fel és a biztonsági tokenek kezelése a mobil alkalmazás esetében megegyezik a webes kliensben bemutatott megoldással.

A megjelenítés összeállításához a NativeBase könyvtár elemeit választottam. Ebben alap dizájnnal rendelkező UI (User Interface – Felhasználói Interfész) komponensek találhatók, amik ellentétben a webes kliens esetén választott ReactStrap könyvtárral, úgynevezett cross-platform elemeket tartalmaz, ami azt jelenti, hogy közös kóddal írható le a megjelenítés az iOS, Android és webes kliensek esetében is.

Az applikáció futtatásához a fejlesztés során az Android Studio fejlesztőeszköz által biztosított emulátort használtam. Ennek köszönhetően fejlesztés közben a számítógépemen tudtam tesztelni a szoftver elkészült funkcionalitását.

5.6 Mobil kliens felépítése

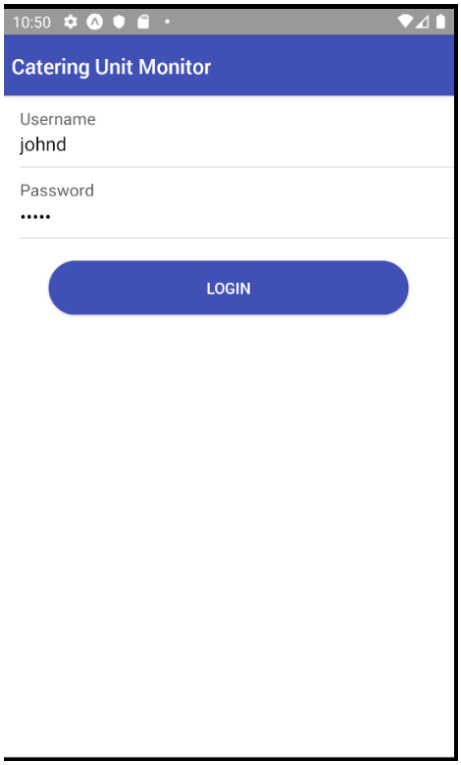
Az aktuális fejezetben ismertetem az elkészült mobil kliens projekt struktúráját, továbbá a forráskód felépítését.

A szoftver kódja a *mobil-client-app* mappában található, aminek a tartalma a következő. A *screens* alkönyvtárban helyezkednek el a megjelenített képernyőket leíró fájlok és a hozzájuk tartozó API hívások és állapotok. A *components* mappában azok a UI elemek találhatók, amik többször felhasználhatók. Az *actions* mappa magában foglalja az authentikációhoz szükséges kódsorokat és a Redux esetén ismertet mintához hasonlóan action objektumokat hoz létre. A létrehozott action objektumokat a *reducers* könyvtárban elkészített feldolgozó funkciók kezelik és ezek alapján állítják be az alkalmazás globális állapotát. Az állapotváltozásokról később a megjelenítésért felelős komponensek értesülnek és végrehajtják a szükséges frissítéseket. A konstans értékek tárolásáért felel a *constants* csomag. A navigáció leírása és a szükséges navigátor funkciók a *navigation* mappában kaptak helyet. Az újra felhasználható kódrészeket a *utils* alkönyvtárban helyeztem el.

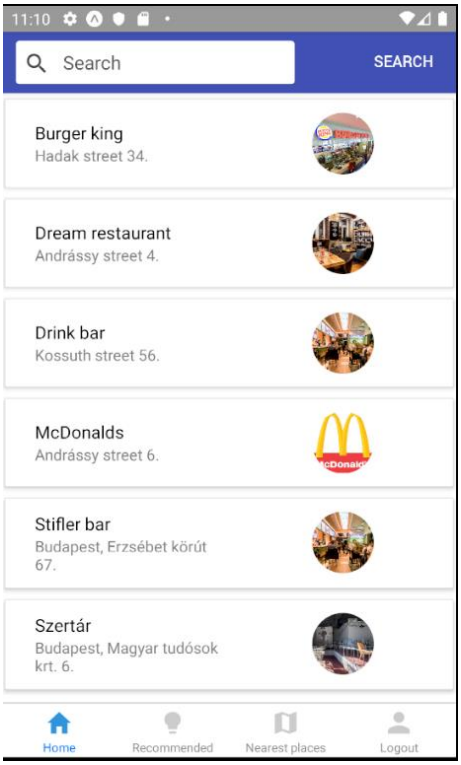
5.7 Mobil kliens bemutatása

Ebben a fejezetben bemutatom az elkészült mobil klienst és alkalmazás működését. A leírást az adott képernyőkön végig haladva ismertetem.

Az alkalmazást elindítva a felhasználót egy bejelentkezési képernyő fogadja, ahol a regisztrációkor megadott felhasználónevével és jelszavával beléphet a rendszerbe. A képernyő a 35. ábraán látható.

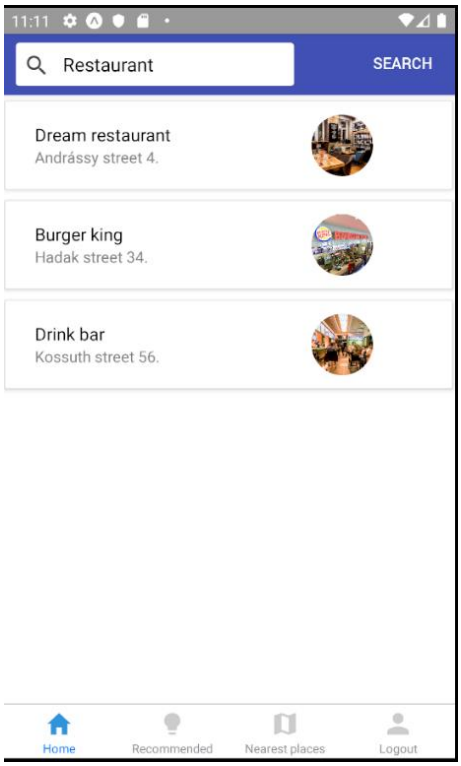


35. ábra Mobil kliens bejelentkezés

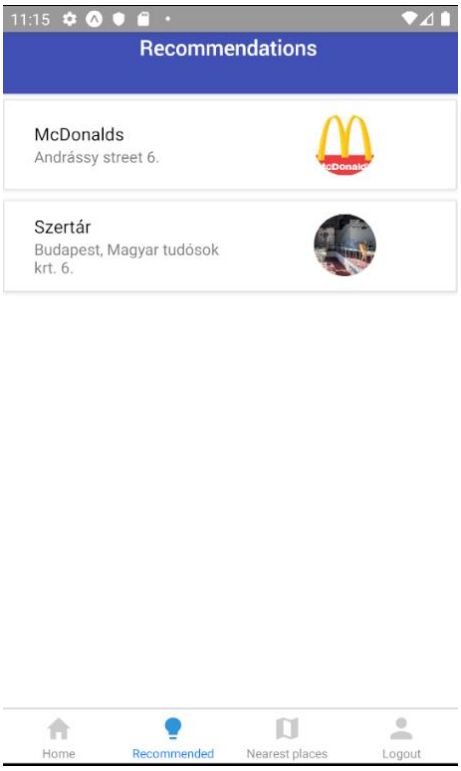


36. ábra Mobil kliens fő képernyő

A bejelentkezés után a felhasználót egy kezdő oldal fogadja, amit a 36. ábra mutat. Ezen az oldalon megtekinthető az összes vendéglátóhely ábécé sorrendben. Az egyes helyekről látható egy-egy kép, a hozzájuk tartozó név és cím. A képernyő felső részén található egy keresőmező, aminek segítségével szűkíteni lehet az eredményeket. Erre mutat példát a 37. ábra, ahol éttermek láthatók. A keresés a helyek neveibe, leírásaiban, címeiben és paramétereiben történik.



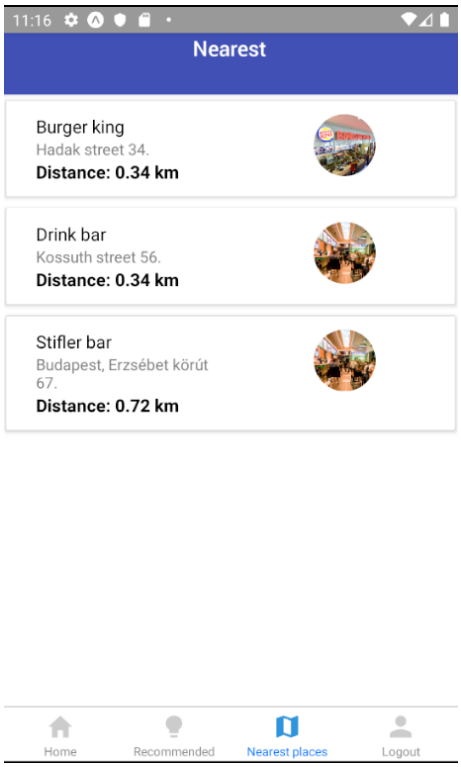
37. ábra Mobil kliens keresés



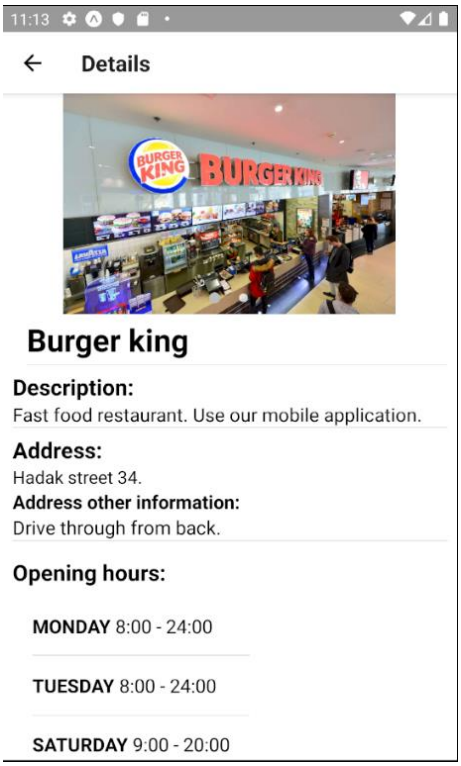
38. ábra Mobil kliens ajánlott helyek

A képernyő alsó részén található a navigációs sáv, ahonnan elérhetők az ajánlott helyek (*Recommended*) és közeli helyek funkciók (*Nearest*), továbbá kijelentkezhet a felhasználó az alkalmazásból (*Logout*). Az ajánlott helyek a 38. ábraán látható módon tekinthetők meg, hasonlóan a kezdőképernyőhöz. A rendszer a hasonló ízlésű felhasználók által kedvel helyeket jeleníti meg az ajánlatok között.

A közeli helyek szűrése a mobilkészülék földrajzi pozíciója alapján történik, ehhez a kliensnek engedélyeznie kell a helyadatokhoz való hozzáférést az alkalmazás számára.



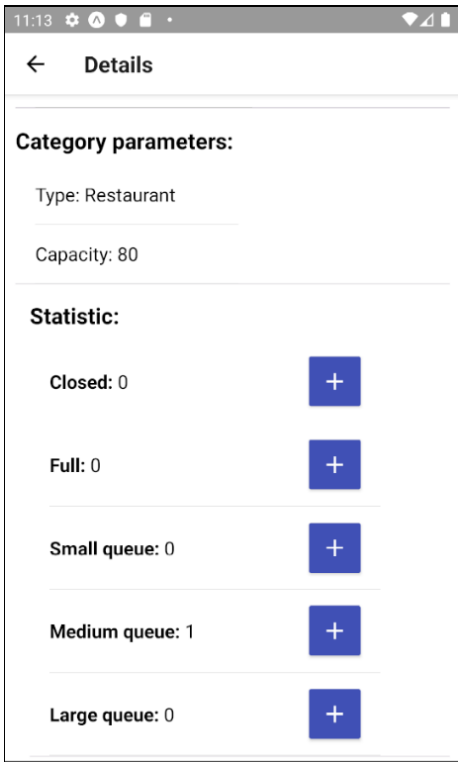
39. ábra Mobil kliens közeli helyek



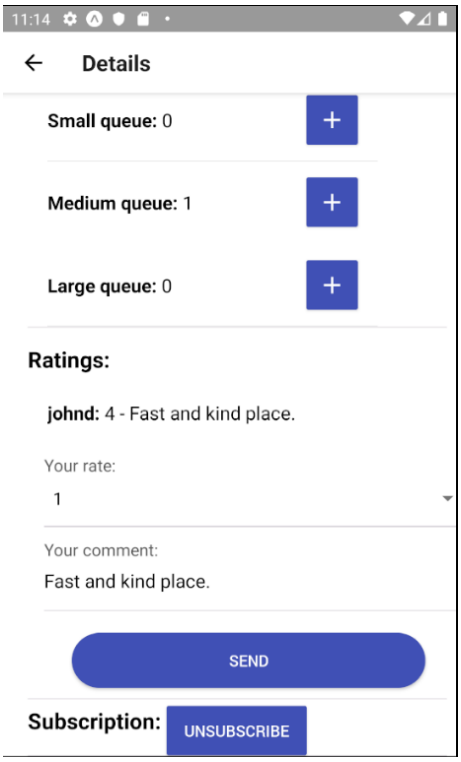
40. ábra Mobil kliens részletes nézet

A helyadatok alapján történő szűrés eredményét a 39. ábra mutatja be. Ebben az esetben megjelenik a felhasználó és a vendéglátóhely közti távolság, ami a lista rendezésének paraméterét adja.

Bármelyik képernyőn egy vendéglátó egység kártyájára kattintva megjelenik a hozzá tartozó részletes nézet, ami az összes szükséges információt tartalmazza. Ezek a 40. ábraától kezdve a következők. A legelső elem egy képgaléria, ahol az összes kép megnézhető a vendéglátóhelyről. Ez alatt található a hely neve, leírása. címe és a nyitvatartási idő. Az adott információk a címekre kattintva elrejthetők, hogy a nézet jobban éttekinthető legyen. Lefelé görgetve a képernyőn további részletek láthatók.



41. ábra Mobil kliens részletes nézet közepe



42. ábra Mobil kliens részletes nézet vége

A 41. ábra mutatja a kategória paramétereket (*Category Parameters*) és a nem várt események statisztikáját. Láthatóvá válik, hogy hány felhasználó jelenti egy adott helyről, hogy az zárva van, megtelt vagy hosszan kell sorban állni. A plusz gomb megnyomásával újabb bejelentés adható hozzá, de minden felhasználó számára kizárólag egy. A részletes nézet utolsó összetevője a 42. ábraán látható, ahol láthatóvá válnak a vendéglátóhelyhez tartozó értékelések (*Rating*) és a felhasználó saját értékelést is feltölthet a rendszerbe. Az utolsó elérhető funkció pedig az értesítésekre való feliratkozás (*Subscribe*) vagy leiratkozás (*Unsubscribe*). A felhasználó a fejlécben található nyíl segítségével navigálhat vissza a fő képernyőre, ahol pedig a menü sáv *Logout* funkciójával kiléphet a rendszerből.

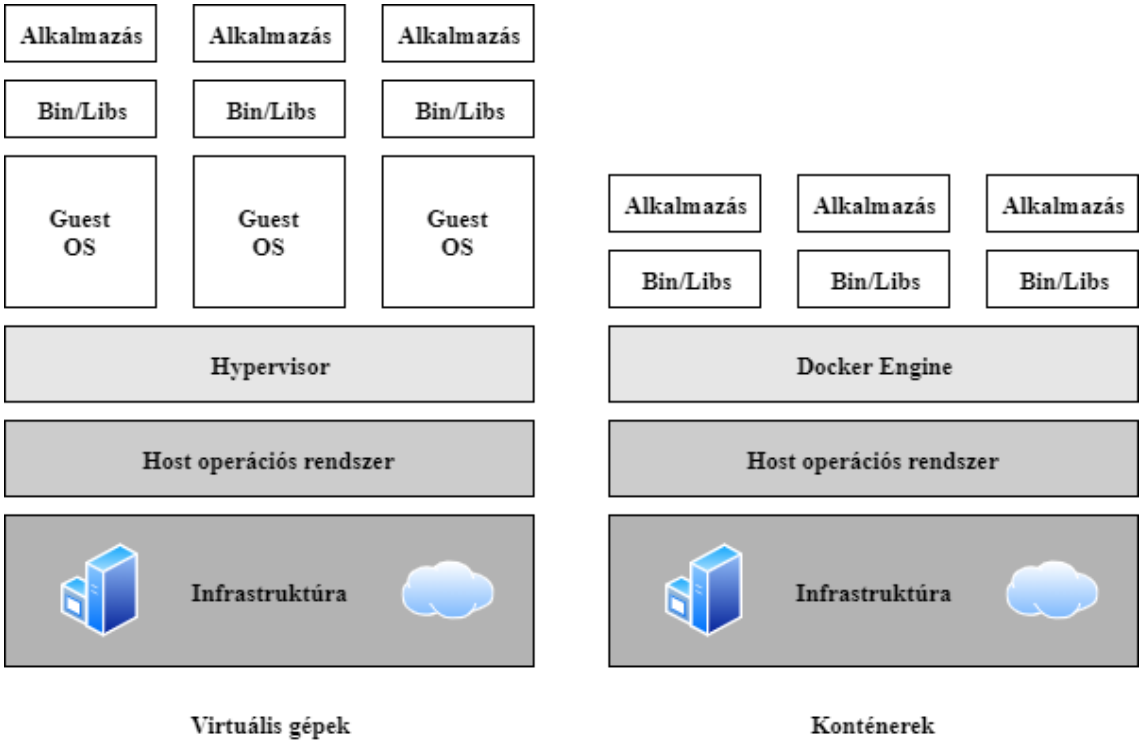
5.8 Infrastruktúra

Ebben a fejezetben ismertetem a fejlesztés során felhasznált infrastruktúrát és bemutatom a végleges infrastruktúra eszközeit. Többek közt bevezetem a konténertechnológia és a felhő platform fogalmát.

5.8.1 Docker infrastruktúra

A fejlesztés során a Docker infrastruktúra segítségével telepítettem és futtattam az alkalmazásom. Ez a konténerizációs eszköz megkönnyítette a sok szolgáltatásból álló programom futtatását és kezelését.

A konténer technológia a klasszikus virtualizációs megközelítést hivatott helyettesíteni és egyszerűbbé tenni. A technológia alapja, hogy a fejlesztők egyetlen csomagba gyűjthetik az alkalmazásukat és a szükséges könyvtárakat, függőségeket vagy fájlokat. Később ezeket a csomagokat egy egységként telepíthetik és konténerek szintjén kezelhetik. A konténerek egymástól izoláltan futnak és dedikált tárhellyel, hálózattal és processz kezeléssel rendelkeznek. [15]



43. ábra Virtuális gépek és konténerek összehasonlítása [16]

A 43. ábra illusztrálja a különbséget a hagyományos virtuális gépeket használó megoldások és a konténer technológiák közt. A virtuális gépek esetén a host operációs rendszer felett szükség van egy virtualizációs rétegre, amit a *Hypervisor* jelöl a képen. Ezen felül látható, hogy az alkalmazások alatt külön-külön operációs rendszerek futnak, amit a *Guest OS* szimbolizál. Ezzel szemben a Docker a host operációs rendszert használva izolálja a rendelkezésre álló erőforrásokat. Ezt a feladatot a *Docker Engine* látja

el és felel a konténerek menedzsmentjéért. A konténereken belül nincs szükség teljes operációs rendszerre, hanem elegendő a program által használt *Bin/Libs* források megléte.

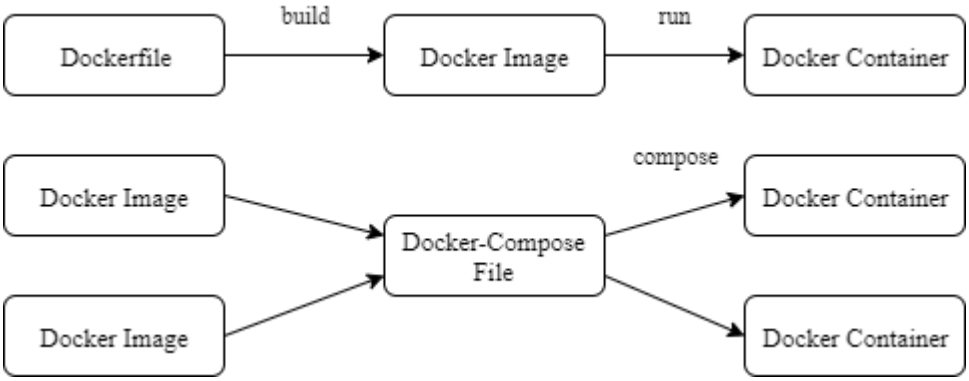
A fejlesztés során a szolgáltatásokat, képfájlokba (docker image) csomagoltam. A képfájlok definícióit a komponensek kódja mellett helyeztem el az úgynevezett *Dockerfile* állományokban. Ezek az állományok leírják, hogy hogyan kell felépülnie egy adott képfájlnak. Például a *Catering Service* leírása a következőképpen néz ki:

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
EXPOSE 8082
COPY  ./target/CateringUnitService.jar /home/CateringUnitService.jar
ENTRYPOINT ["java", "-jar", "/home/CateringUnitService.jar"]
```

Megadtam egy alap image-t és ezt kiegészítve belemásoltam a programomat egy JAR file formájában. Leírtam, hogy az alkalmazást hogyan kell elindítani és hogy milyen hálózati portot fog használni.

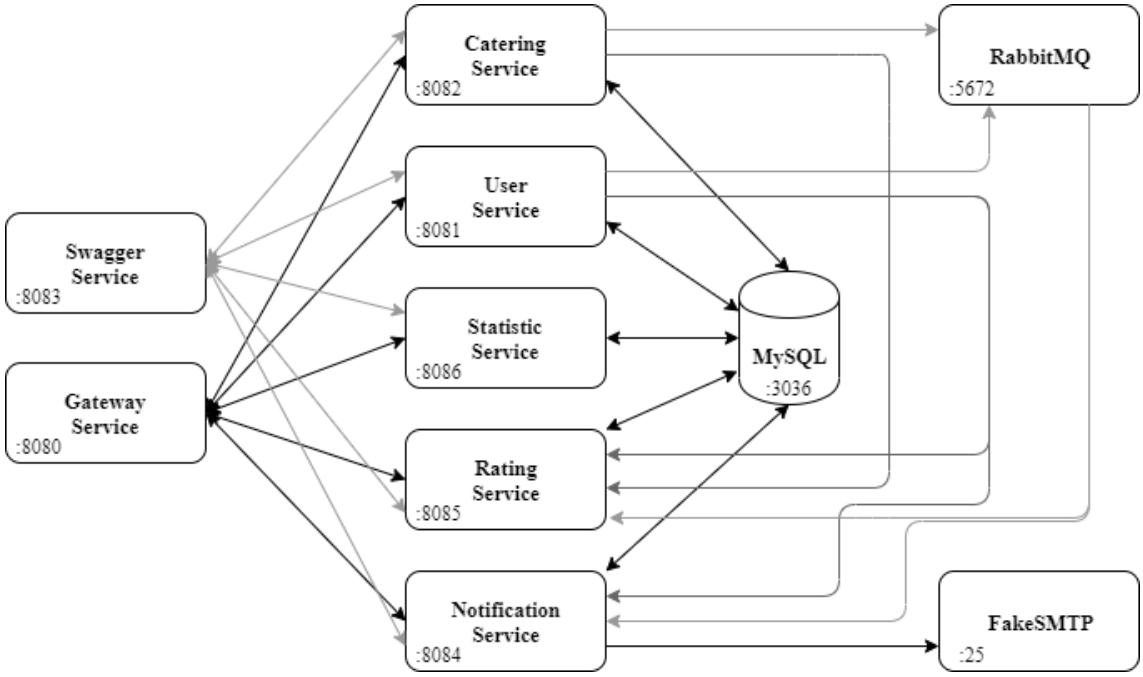
A 44. ábra bemutatja, hogy milyen lépések szükségesek az alkalmazás telepítéséhez. A *Dockerfile*-ok leírásai alapján a *build* során elkészülnek a képfájlok (*Docker Image*). Ezen folyamat felgyorsítása érdekében felhasználtam a Spotify által fejlesztett *Dockerfile-Maven-Plugin*-t, ami a projektben elhelyezett leíró fájlok alapján a Maven build folyamat végén, a lefordított és becsomagolt alkalmazásokból elkészíti a szükséges képfájlokat. A konténerek elindítása lehetséges a *run* parancs segítségével, amivel egyesével kezelhetjük a komponenseinket, de komplexebb telepítések esetén érdemes az következő módszert használni.

A Docker infrastruktúra része az úgynevezett Docker-Compose eszköz, aminek célja, hogy egy teljes rendszer leírható egyetlen fájlban és az összes komponens egyszerre kezelhető. Ehhez a *docker_deploy* mappában elkészítettem a *docker-compose.yml* leírást. Ez tartalmazza az általam használt konténerek paramétereit és elvárt állapotait és a futtatáshoz szükséges egyéb szolgáltatásokat úgy, mint az adatbázis vagy az üzenetkezelő bróker.



44. ábra Docker telepítés folyamata

A *docker-compose* fájl felhasználásával a teljes rendszer egyetlen utasítással elindítható és leállítható. Az elkészített alkalmazás konténer szintű nézetét a 45. ábra mutatja be.

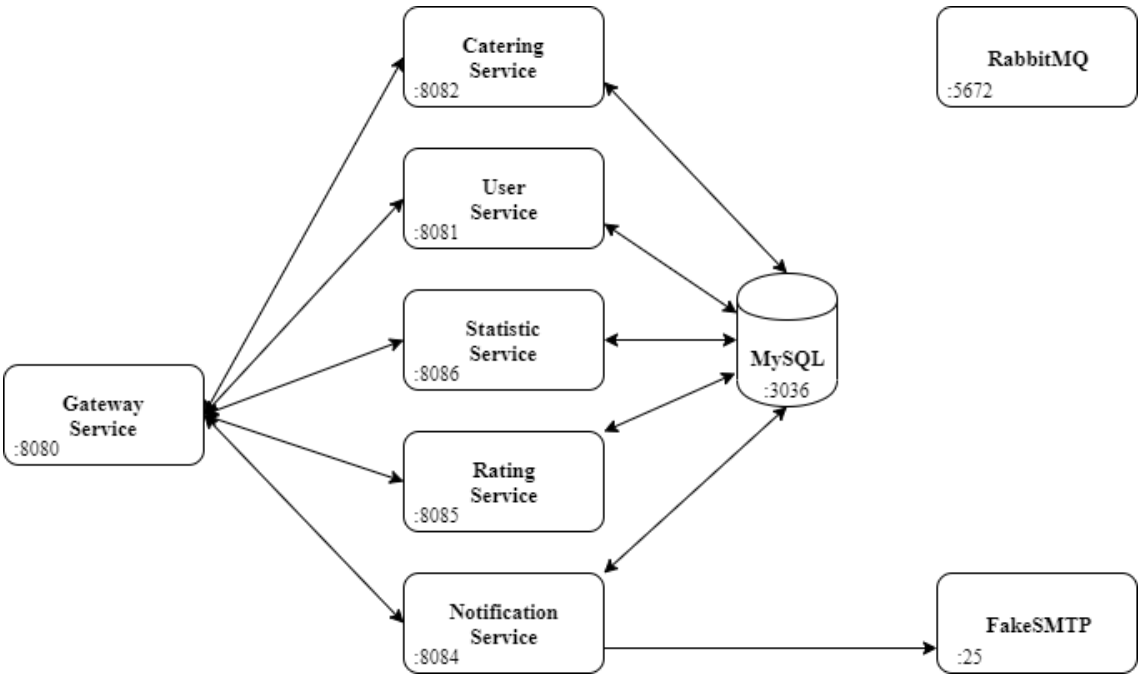


45. ábra Docker infrastruktúra

Minden a képen látható elem egy-egy Docker konténer, amik felelősek a korábban ismertetett szolgáltatások futtatásáért. Az ábrán jelöltem a konténerek által használt hálózati portokat és a köztük lévő kapcsolatokat. A *FakeSMTP* szolgáltatás ebben az esetben az email küldő szervert helyettesíti és a beérkező üzeneteket egy fájlba menti, így a tesztelés során ellenőrizhető a rendszer helyes működése. Mivel a Docker-t a szoftver készítésének átmeneti szakaszában használtam, a szolgáltatások számára egyetlen központi adatbázist biztosítottam. Ezt azért tehettem meg, mert minden microservice kizárólag a saját adatbázistáblájába ír és onnan olvas és ezek minden esetben különböző

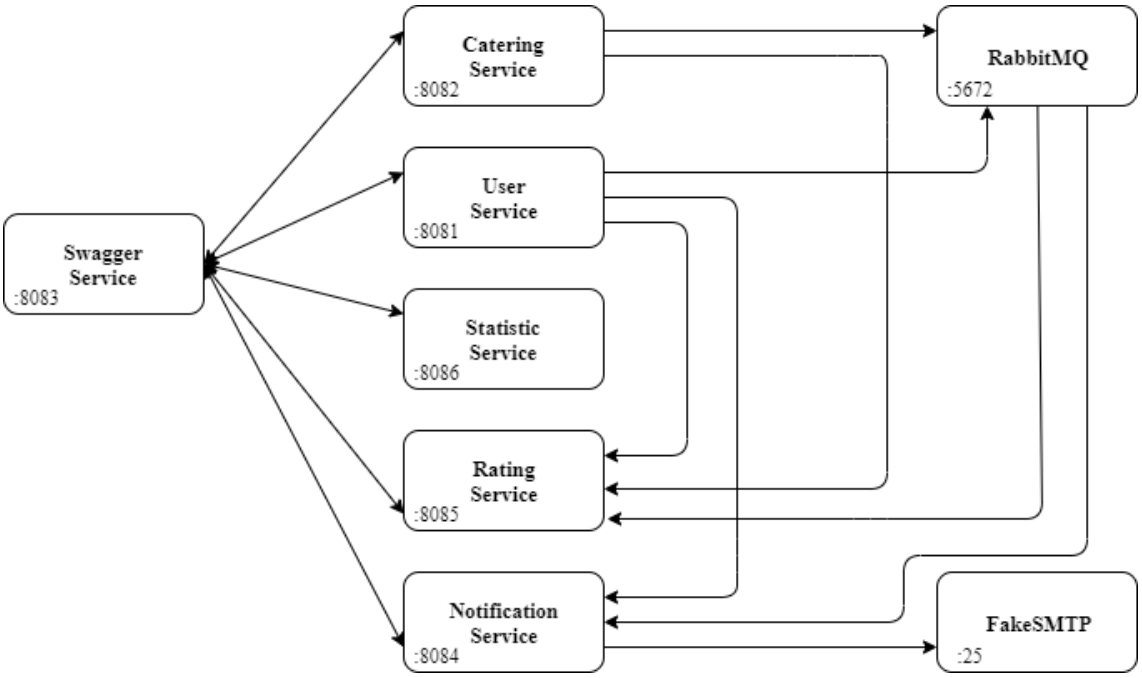
névvel rendelkeznek. Ebben az esetben a teljesítmény szempontok nem voltak kritikusak számomra.

A 45. ábra célja, hogy áttekinthetőbbé tegye a szolgáltatások közötti kapcsolatokat és csak az adatbázis (*MySQL*), *Gateway-Service* és *FakeSMTP* konténerek kapcsolatát emelje ki.



46. ábra Egyszerűsített Docker infrastruktúra

Az előző képhez hasonlóan a 47. ábra célja szintén az áttekinthetőség biztosítása és a *Swagger-Service*, *RabbitMQ* és mikroszolgáltatások közötti kommunikáció bemutatása.

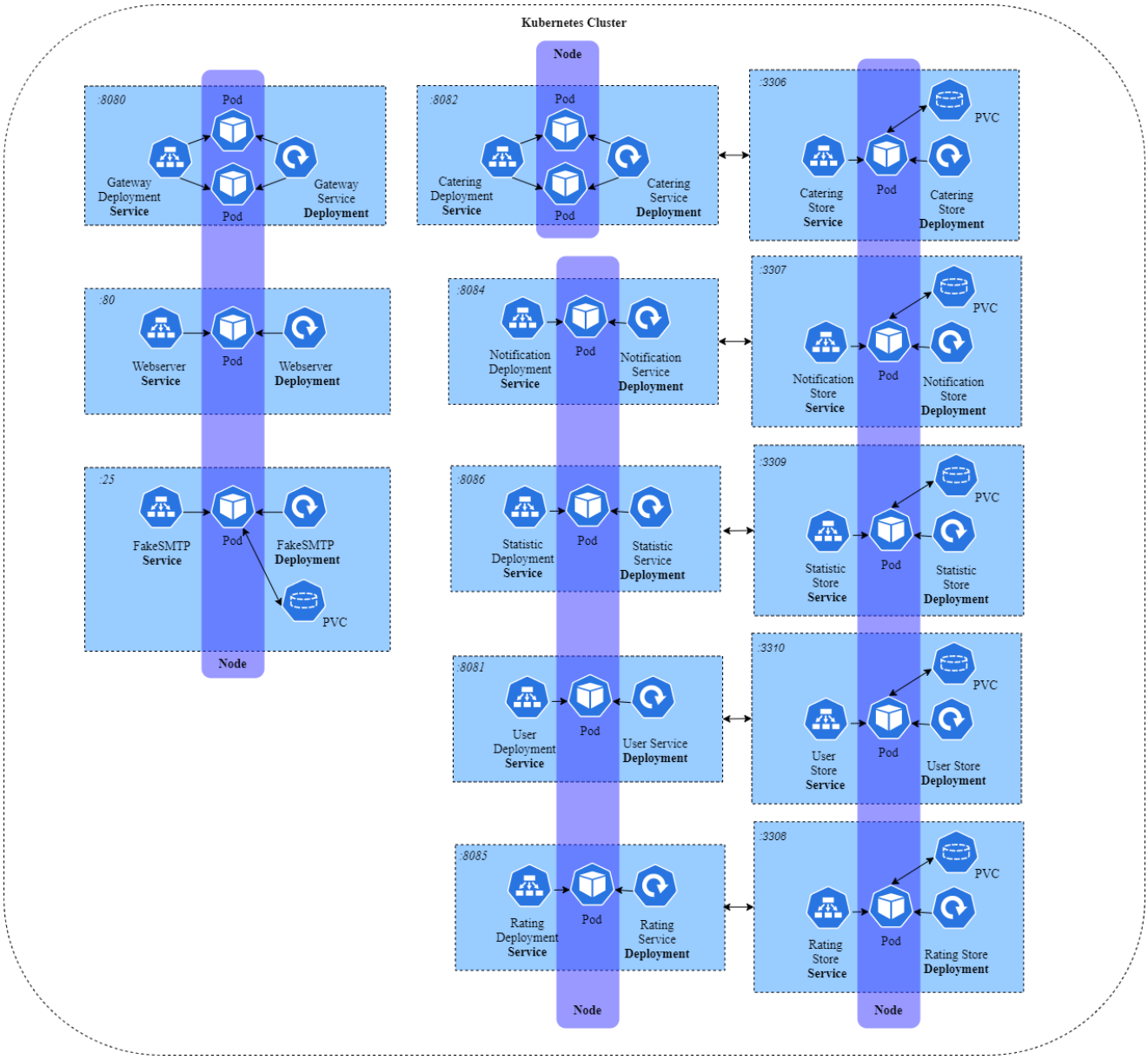


47. ábra Egyszerűsített Docker infrastruktúra folytatása

A fejlesztés során rendkívül hasznos volt számomra, hogy a szoftverem minden szükséges elemét egyszerűen és gyorsan tudtam telepíteni. A szükséges változtatások végrehajtása esetén elegendő volt a leíró fájlok szerkesztése és nem kellett az operációs rendszerre a futtatáshoz szükséges eszközöket telepítenem. Ezen felül ez a megoldás felépítésben és technológiában közel állt ahhoz az infrastruktúrához, amit a végleges futtatás során használtam és a következő fejezetben részletesen ismertetek.

5.8.2 Kubernetes infrastruktúra

Az alkalmazás üzemeltetése során az egyik legfőbb megoldandó probléma a konténerek kezelése. Egy-egy példány leállása esetén szükség lehet annak automatikus újraindítására és a növekvő rendszerterhelés során biztosítani kell, hogy több példány futhasson. Ezen problémák kezelésére használtam a végleges telepítés során a Kubernetes konténer kezelő infrastruktúrát. Ennek lényege, hogy képes például Docker konténereket futtatni és kezelni köztük a kapcsolatokat, segíteni a telepítést és a frissítést. A telepítéshez Google Cloud felhőszolgáltatást választottam, ahol menedzselt szolgáltatásként elérhető a Kubernetes, így a rendszer üzemeltetésével nem kellett foglalkoznom, ezen túl szabadon igényelhettem hardver erőforrásokat. Az elkészített alkalmazás felépítése a 48. ábraán látható. [17]



48. ábra Alkalmazás felépítése Kubernetesben

A projekt alapja a Kubernetes Cluster és a futtatáshoz szükséges minden elem ezen belül helyezkedik el. Az erőforrások vezérlését a Kubernetes Master végzi, ami a Google Cloud platform esetén egy menedzselt szolgáltatás, így nem kell a futtatásáról külön gondoskodni. A telepítés alapegysége a Deployment, ami egy alkalmazás elvárt állapotát írja le. Ezek a leírások a *deployment* mappa alatt találhatók és *-deployment* végződéssel ellátott fájlok tartalmazzák őket. Egy ilyen fájlban leírásra kerül, hogy az alkalmazást melyik képfájl tartalmazza, milyen metaadatokkal kell címkézni és melyik hálózati portot használja. Itt írható le az úgynevezett ReplicaSet is, ami azt mondja meg, hogy egy komponensből hány példány fusson. Ennek segítségével növelhető a rendszer hibatűrése és skálázható az alkalmazás. A konténerek Pod-okban futnak és mivel az

állapotkezelés a Pod-ok szintjén történik érdemes minden konténert külön Pod-ban elhelyezni. Ezek kiszolgálását a Node-ok végzik, amik a hardveres erőforrásokat biztosítják. Ahhoz, hogy egy Deployment a hálózaton elérhető legyen biztosítani kell számára egy szolgáltatást (Service), ami több típusú lehet. Én a LoadBalancer és ClusterIP típusokat használtam. Az első a szolgáltatások számára egy publikus IP (Internete Protocol) címet biztosít, még a második a Clusteren belül teszi elérhetővé a telepítéseket. Az adatok tárolásához a PersistentVolumeClaim (PVC) elemeket használtam, amik a tárhelyeket reprezentálják. A végső telepítés során minden microservice számára külön adatbázist készítettem, így elkerülve, hogy ezek szűk keresztmetszettet jelentsenek. Az adatbázisokhoz is tartoznak Deploymentek, amiknek a leírását a *database* mappa tartalmazza és az ehhez tartozó szolgáltatások a *databaseservice* mappában vannak definiálva. A *datastore* könyvtár magában foglalja az adatbázisokhoz kapcsolódó tárhelyek leírásait.

A 48. ábra tartalmazza a szolgáltatásokhoz kapcsolódó hálózati portszámokat is. A szolgáltatások és adatbázisok kapcsolatát jelöltem, de a szolgáltatások közötti kapcsolatokat nem ábrázoltam az áttekinthetőség érdekében, de ezek megegyeznek a Docker esetén látottakkal. Az ábrán a Node-ok jelölése jelképes, mert a gyakorlatban 8 Node-ot használtam és ezek között a Kubernetes Master tetszőlegesen osztotta el a Pod-okat és más erőforrásokat.

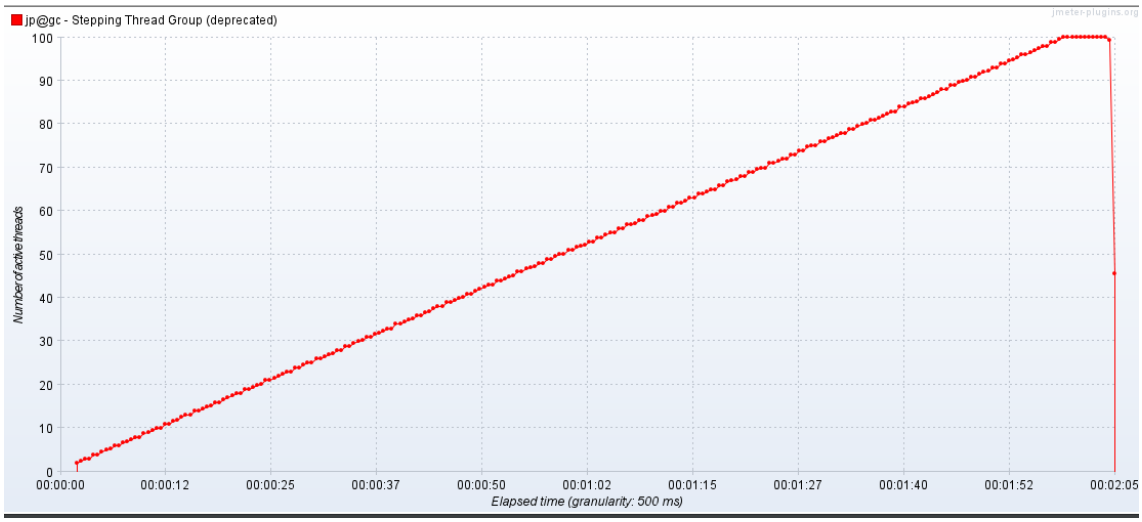
6 Tesztelés

Ebben a fejezetben ismertetem, hogy milyen módszerekkel és technológiákkal teszteltem az elkészített alkalmazásomat. A fejezet végén pedig bemutatom a dolgozat szempontjából lényeges teszteredményeket.

A szerver oldali alkalmazás fejlesztése során az elkészített szerviz rétegek funkcióihoz egység teszteket (*Unit test*) írtam. Az volt a célom, hogy a bonyolultabb logikát megvalósító kódrészekről bizonyítsam, hogy helyesen működnek és későbbi változtatások során nem sérülnek. A tesztek futtatásához a Spring Test keretrendszert használtam és az adatok tárolására In-Memory (Memóriában tárolt) adatbázist választottam. Az email szervert Greenmail tesztszerverrel helyettesítettem, így megbizonyosodhattam róla, hogy az üzenetek helyesen elküldésre kerülnek.

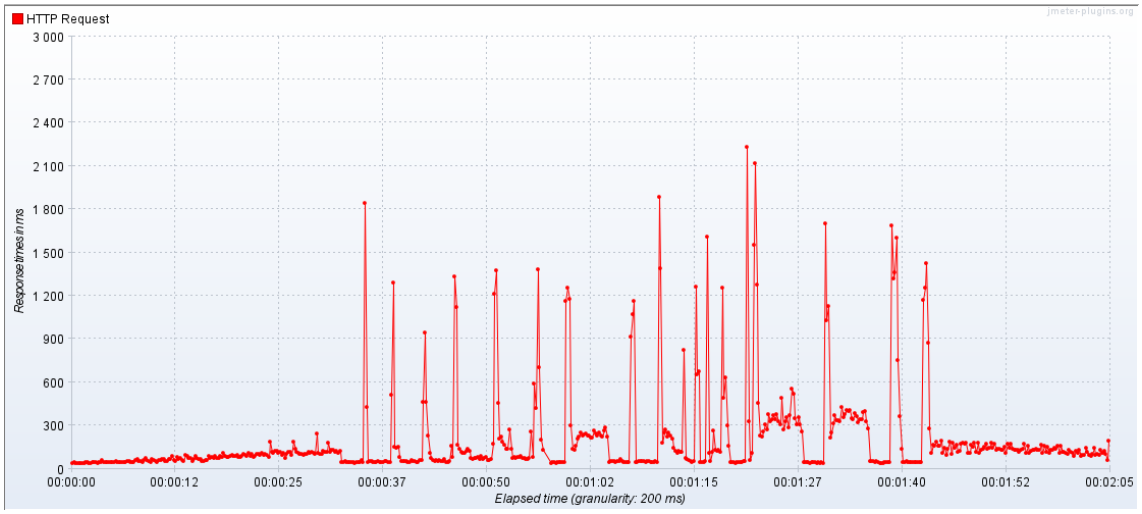
Az elkészült alkalmazást a fejlesztés során a Docker segítségével futtattam és a rendszertesztek végrehajtásához a Postman programot használtam. Ennek segítségével a REST végpontokat valós adatokkal hívhattam meg és ellenőrizhettem az eredményeket. A rendszertesztek során bizonyítottam, hogy a szolgáltatások együtt használva is helyesen működnek. A tesztesetek leírását a *postman_tests* mappában helyeztem el. Ezen felül természetesen végeztem manuális tesztelést is, ugyanebben a környezetben, de ezek hátránya az volt, hogy nem automatikusan futottak és az egyes változtatások során sok időt vett igénybe az újabb végrehajtásuk, ezért igyekeztem minél több esetet automatizálni.

A fejlesztés végén a szoftvert a Kubernetes segítségével a Google felhő platformra telepítettem és teljesítménytesztet végeztem. Megvizsgáltam, hogyan viselkedik a program, ha nem használok horizontális skálázást és miként változik a működése, ha igen. A tesztek futtatásához a JMeter teszt eszközt választottam, aminek segítségével meghívhatók az alkalmazás REST végpontjai magas terhelés mellett. A méréshez a közeli vendéglátóhelyeket szolgáltató funkciót választottam, mert ebben az esetben képzelhető el a legnagyobb terhelés valós körülmények között. A teszt dinamikusan emelkedő terhelést szimulál. A kezdeti állapotban egyetlen szálról hajt végre folyamatos hívásokat és a szálak számát egyenletesen emeli egészen 100-ig. A teljes futás időtartama két perc. A terhelés képét a 49. ábra mutatja.



49. ábra Teljesítményteszt terhelés

A képen megfigyelhető, hogyan emelkedik a végrehajtások száma. A vízszintes tengelyen az idő osztás található, a függőleges tengely pedig a párhuzamos szálakat mutatja. Az első tesztek során szembesültem azzal a korláttal, hogy a felhőszolgáltató által biztosított tárhelyhez tartozó IO (Input Output) limit a mérettel egyenesen arányos, ezért a kezdetleges 5 GB méretet meg kellett emelnem. Végül 1 TB-ra változtattam a méretet, hogy semmiképp se jelentsen szűk keresztmetszetet. Az első esetben minden szolgáltatásból egy példány (Pod) futott és az eredményeket az 50. ábra demonstrálja.

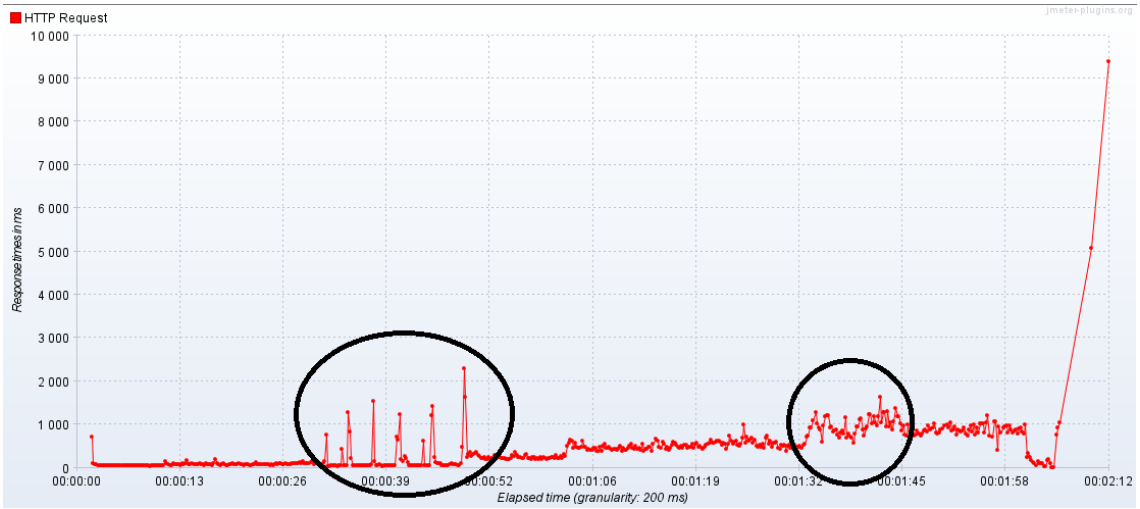


50. ábra Teljesítményteszt skálázás nélkül

A grafikon vízszintes tengelyén található az idő, a függőleges tengelyen pedig az adott pillanat béli válaszidő milliszekundumban kifejezve. Megfigyelhető, hogy a rendszer 30 másodperc környéként igen instabillá válik és a válaszidő jelentősen megnő. Külön ki

kell emelnem, hogy az eredmény végén látható alacsony válaszidő annak a következménye, hogy az alkalmazás rengeteg hibás választ adott.

A második esetben megvizsgáltam, hogyan viselkedik a rendszerem, ha a Kubernetes AutoScaling (Automatikus skálázás) opcióját használom. Ehhez meg kellett adnom, hogy melyik Deployment esetén szeretném alkalmazni a funkciót, ami az adott tesztek esetén a Catering-Service volt. Minimális Pod számnak kettőt választottam. Felső limitnek pedig kipróbáltam a tízet és a húszat is. Elmondható, hogy a rendszer mindkét esetben hasonlóan viselkedett, de következő képen a húszas limittel készült eredmény lesz látható. További paraméterként megadtam, hogy mekkora processzor terhelés esetén skálázzon fel a rendszer. Esetemben ez az érték 55% volt.



51. ábra Teljesítményteszt automatikus skálázással

Az 51. ábra alapján az eredmények a következők. Látható, hogy a rendszer ebben az esetben is instabillá vált 30 másodperc környékén, amit az első bekeretezett rész jelöl. Rövid idő elteltével megfigyelhető, hogy a rendszer újra stabilizálódott és eközben újabb Pod-ok indultak. Fontos információ, hogy a Kubernetes 30 másodpercenként ellenőrzi a terhelést és eszerint skálázza a rendszert, ezért szerepelnek a kiugró értékek huzamosabb ideig. A második bekeretezett részen újra megfigyelhető a jelenség és az átlagos válaszidő minimálisan csökkenni kezd. A terhelés végeztével a Pod-ok száma fokozatosan visszaállt az alap értékekre, tehát megállapítható, hogy az alkalmazásom helyesen kezelte a magas igénybevételt és az alacsony terheléshez is alkalmazkodott.

7 Összefoglalás

A munkám célja az volt, hogy megismerkedjek a microservices architektúrával és ebben a szemléletben elkészítsek egy alkalmazást. Az elkészült alkalmazás vendéglátóhelyek keresésére szolgál. A felhasználók számára tájékoztatást ad a helyekhez kapcsolódó esetleges nemvárt eseményekről és ajánlásokat nyújt a hasonló ízlésű emberek értékelései alapján.

Az elkészült programom három fő részből áll. Egy szerver oldali alkalmazásból, egy webes adminisztrációs felületből és egy mobil kliensből. A dolgozatomban részletesen bemutattam a fejlesztés folyamatát, a tervezéstől, az implementáción át, a tesztelésig. A fejlesztés során számos új technológiával és módszerrel ismerkedtem meg. Az egyik legnagyobb kihívás számomra a követelmények pontos és reális megfogalmazása volt. A szerver oldali alkalmazás fejlesztése során több technológiai kihívást is lekellett küzdenem és adott esetben a felhasznált keretrendszerek hibáira vagy korláira kellett megoldást találnom. Számomra a felhasználói felületek elkészítése egy ismeretlen terület volt, ezért külön örülök, hogy a munkám során betekintést nyerhettem ezen technológiákba és sikeresen elkészítettem a szükséges applikációkat. Az elkészült szoftvert végül két különböző módszer felhasználásával is futtattam. Az első a Docker infrastruktúra volt, ahol megismerkedtem a konténer alapú alkalmazás üzemeltetéssel. Emellett kipróbáltam a felhőszolgáltatásokat is, ahol végül egy Kubernetes deploymentet készítettem, aminek segítségével megvizsgáltam a szoftverem horizontális skálázhatóságát. Az elvégzett tesztek sikeresek voltak és kielégítették a bevezetőben meghatározott követelményeket.

Konklúzióként elmondhatom, hogy rengeteget tanultam a munkám során és a megszerzett tudást a jövőben könnyen felhasználhatom a szakmámban. A microservices architektúrával kapcsolatban az a tapasztalatom, hogy komplex és teljesítménykritikus alkalmazások fejlesztéséhez, talán napjaink egyik legalkalmasabb módszere. Kis csapatban történű egyszerű szoftverek fejlesztése esetében, azonban még mindig helytálló megközelítés a monolitikus felépítés.

7.1 Továbbfejlesztési lehetőségek

Ebben a fejezetben kitérek az esetleges továbbfejlesztési lehetőségekre, amik nem készültek el a munkám során vagy nem volt célja a feladatomnak.

A megvalósított email küldő szolgáltatás mellett célszerű lenne egy szolgáltatás elkészítése, ami Push Notification formájában képes értesítéseket küldeni a felhasználók mobil készülékeire.

Érdemes lenne az elkészített két kliens funkcióinak összevonása és egy krossz platform alkalmazás elkészítése, amire a felhasznált React Native technológia alkalmas, így egyetlen kódbázisban kezelhető lenne a webes kliens és a mobil kliens, valamint a felhasználók számára is könnyebbé válna a szolgáltatás használata.

Célszerű a mobil kliensben egy térkép nézet bevezetése, ahol megtekinthető az összes vendéglátóhely, így könnyebben lehetne keresni közöttük és növelhető lenne a felhasználói élmény.

8 Rövidítések

API - Application Programming Interface – Alkalmazásprogramozási Interfész

API Gateway - Application Programming Interface Gateway – Alkalmazásprogramozási felület átjáró

DTO - Data Transfer Object

ER - Entity Relationship - Egyed-Kapcsolat

HTTP – HyperText Trasfer Protocol

JAR - Java Archive

JSON - JavaScript Object Notation

JWT - JSON Web Token

IP - Internete Protocol

REST - Representational State Transfer

RMI - Remote Method Invocation – Távoli Metódus Hívás

SMTP - Simple Mail Transfer Protocoll

SOA - Service Oriented Architecture – Szolgáltatásorientált architektúra

UI – User Interface – Felhasználói felület

9 Irodalomjegyzék

[1] A. KWIECEIN, „10 companies that implemented the microservice architecture and paved the way for others,” 02 08 2019. [Online]. Available: https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/. [Hozzáférés dátuma: 10 11 2019].

[2] A. D., „Best Architecture for an MVP: Monolith, SOA, Microservices, or Serverless?,” 18 04 2019. [Online]. Available: https://rubygarage.org/blog/monolith-soa-microservices-serverless. [Hozzáférés dátuma: 16 11 2019].

[3] R. M. M. M. &. M. A. Irakli Nadareishvili, „What are microservices? Don't I already have them?,” in *Microservice Architecture*, O'REILLY, 2016, p. 6.

[4] S. Newman, Building Microservices, United States of America: O'REILLY, 2015.

[5] K. Lange, The Little Book On REST Services, Copenhagen, 2016.

[6] S. Newman, „API Composition,” in *Building Microservices*, United States of America, O'Reilly Media, 2015, p. 71.

[7] B. W. M. R. Cesar de la Torre, „Implement event-based communication between microservices,” Microsoft, 10 02 2018. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications. [Hozzáférés dátuma: 30 11 2019].

[8] S. E. Peyrott, „The JWT Handbook,” Auth0 Inc., 2016.

[9] C. Walls, Spring Boot in Action, United States of America: Manning Publications Co., 2016.

[10] D. Dossot, RabbitMQ Essentials, Birmingham, UK.: Pack Publishing Ltd., 2014.

[11] D. Flanagan, „Introduction to JavaScript,” in *JavaScript - The Deifinite Guide*, Sebastopol, CA 95472, O'Rielly Media Inc., 2006, pp. 1-3.

[12] A. B. &. E. Porcello, „Welcome to React,” in *Learning React*, Sebastopol, CA 95472, O'Reilly Media Inc., 2017, pp. 1-6.

[13] S. Weck, „Developing modern offline apps with ReactJS, Redux and Electron,” 12 03 2017. [Online]. Available: https://blog.codecentric.de/en/2017/12/developing-modern-offline-apps-reactjs-redux-electron-part-3-reactjs-redux-basics/. [Hozzáférés dátuma: 10 10 2019].

[14] A. Gupta, „Introduction To Redux,” 13 07 2017. [Online]. Available: https://medium.com/@abhayg772/introduction-to-redux-using-react-native-a8f1e8778333. [Hozzáférés dátuma: 20 11 2019].

[15] K. M. &. S. P. Kane, Docker Up & Running, Sebastopol, CA 95472: O'Reilly Media Inc, 2015.

[16] „What is a Container?,” [Online]. Available: https://www.docker.com/resources/what-container. [Hozzáférés dátuma: 15 11 2019].

[17] B. B. &. J. B. Kelsey Hightower, Kubernetes Up & Running, Sebastopol, CA 95472: O'Reilly Media Inc., 2017.

Függelék

A diplomatervhez tartozó melléklet tartalmazza az elkészített alkalmazás kódját és a felhasznált fájlokat. A melléklet elérhető https://github.com/hgeri95/diplomamunka címen.