



Budapesti Műszaki és Gazdaságtudományi Egyetem  
Villamosmérnöki és Informatikai Kar  
AUT Tanszék

Pintér Tamás

**SKÁLÁZHATÓSÁG VIZSGÁLATA  
MIKROSZOLGÁLTATÁSOKRA  
ÉPÜLŐ KONTÉNER-ALAPÚ  
KÉPMEGOSZTÓ  
ALKALMAZÁSON**

KONZULENS

Dr. Forstner Bertalan

BUDAPEST, 2022

# Tartalomjegyzék

<b>Összefoglaló .....</b>	<b>5</b>
<b>Abstract.....</b>	<b>6</b>
<b>1 Bevezetés .....</b>	<b>7</b>
1.1 Az alkalmazás inspirációja .....	7
1.2 A feladatkiírás értelmezése .....	8
<b>2 Felhasznált technológiák .....</b>	<b>9</b>
2.1 Választott technológiák és indokltságuk.....	9
2.1.1 Spring framework különböző moduljai .....	9
2.1.2 JWT autentikáció és autorizáció RSA kulcspárral .....	10
2.1.1 Adatbáziskezelés - PostgreSQL és a Spring JPA .....	11
2.1.2 AWS.....	11
2.1.1 Kafka.....	12
2.1.2 Docker.....	14
2.1.3 Kubernetes .....	15
2.1.4 Feign Client.....	16
2.2 Választott platformok .....	17
2.2.1 Android .....	17
2.2.2 REST API alapú web .....	18
<b>3 A rendszer megtervezése .....</b>	<b>19</b>
3.1 Magasszintű architektúra .....	19
3.2 Adatmodell, E-K diagramm.....	22
3.3 Adatbázis választása .....	23
3.4 A monolitikus és mikroszolgáltatás alapú alkalmazások közti különbség .....	24
3.5 Logolás.....	26
3.6 Szolgáltatások szétválasztása szerepek / felelősségek szerint .....	28
3.7 A szolgáltatások közti, valamint a szerver és a kliens közti kommunikáció módja .....	29
3.8 Autorizáció, Autentikáció és az API gateway kapcsolata .....	31
3.9 Load balance Eureka és Kubernetes esetében .....	32
3.10 Képek letöltése AWS S3 tárhelyről .....	35
3.11 Konténerizálás Docker-rel .....	36

<b>4 Megvalósítás és üzemeltetés .....</b>	<b>40</b>
4.1 Hostolás az IBM Cloud-ra .....	40
4.2 Load test.....	45
4.3 Képek az elkészült alkalmazásról .....	50
<b>5 Az eredmények értékelése .....</b>	<b>51</b>
5.1 Az elkészült munka értékelése.....	51
5.2 Továbbfejlesztési lehetőségek .....	51
5.2.1 Milyen további funkciókkal egészíteném még ki a rendszert?.....	52
5.2.2 Milyen komponenst lenne érdemes cserélni? .....	53
5.2.3 A téma más alkalmazási területei .....	55
<b>Köszönetnyilvánítás.....</b>	<b>57</b>
<b>Irodalomjegyzék.....</b>	<b>58</b>
<b>Függelék.....</b>	<b>59</b>

# HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2022. 12. 08.

.....  
Pintér Tamás

# Összefoglaló

Az elmúlt néhány évtizedben felfoghatatlan sebességgel növekedett az internet mérete, ezzel a felhasználóinak száma is, ami azt eredményezte, hogy az itt elérhető alkalmazások terhelhetősége egyre kritikusabb kérdés lett a fejlesztések során.

A kezdetekben még annyi ember fért hozzá informatikai megoldásokhoz, amit manapság egy személyes felhasználásra szánt számítógép ki tudna szolgálni, viszont ahhoz, hogy a jelenlegi felhasználóknak akár egy apró töredékét kiszolgálhassuk, közel sem elég ennyi erőforrás, az alkalmazásokat serverparkokban kell futtatni, ezeknek megfelelő infrastruktúrát biztosítani, arról nem is beszélve, hogy az sem mindegy, hogy milyen architektúrával alakítottuk ki őket.

Ez a dolgozat az olyan konténerizált, mikroszolgáltatás alapú, automatikusan skálázódó backend alkalmazás megtervezéséről és megvalósításáról szól, amely képes nagy felhasználói terhelés kiszolgálására is különösebb késleltetés nélkül.

Ez a dolgozat bemutatja, hogy hogyan lehet egy olyan képmegosztásra alkalmas közösségi média alkalmazás szerver és kliensoldali részét megalkotni, ami a fentebb említett igényeknek megfelel. Az alkalmazás szerver oldali kódja Spring keretrendszer használatával, kliens oldala pedig Kotlin nyelven lett írva.

A dolgozat kitér a mikroszolgáltatás alapú és a monolitikus alkalmazások közti különbségekre, arra, hogy ezek a szolgáltatások hogyan épülnek fel, hogyan kommunikálnak egymással és milyen megoldásokat nyújtanak különböző problémákra, valamint az ezekhez szükséges technológiákat olyan szinten, amennyire az a megoldás megértéséhez szükséges.

# Abstract

In the last couple of decades, the internet's size increased with an inconceivable speed which made its user base grow as well. This resulted in the issue of application load becoming more important during developments.

In the beginning, not many people had access to IT solutions, these days a personal computer can serve that many users. Because of this rapid increase, to serve even a small fraction of the current user base, we would have to have way more resources, optimally run these applications in huge server parks, optimize them with the correct infrastructure, not to mention we would have to design a suiting architecture for these applications.

This thesis is about the design and implementation of a containerized, microservice-based, automatically scaling backend application that can serve large number of users without any delay.

This thesis shows how to create the server and client side of a social media application suitable for image sharing that meets the above-mentioned needs. The server-side code of the application was written using the Spring framework, and the client-side code was written for Android, in Kotlin language.

This thesis covers the differences between microservice-based and monolithic applications, how these services are structured, how they communicate with each other and what solutions they provide to various problems, as well as the technologies required for them, at a level that is necessary to understand the solution.

# 1 Bevezetés

## 1.1 Az alkalmazás inspirációja

A témám bemutatásához az utóbbi időben igencsak elterjedt kép megosztó közösségi média alkalmazást, a BeReal-t választottam. Ennek az a különlegessége a versenytársaihoz képest, hogy minden nap egyszer, egy adott pillanatban kap az összes felhasználó egy értesítést arról, hogy itt az ideje megosztani azt, hogy mit csinálnak, ehhez az előlapi és hátlapi kamerát is egyaránt használva. Ezek után pedig lehetőségük van reakciókat feltölteni a barátaik képeihez és kommentálni azokat.

Ahogy ebből érződik is, ide valami olyan megoldást kell találni, ami egyidejűleg képes kezelni minél több felhasználót, esetünkben opcionálisan az összeset anélkül, hogy bármelyiküknek különösebben sokat kellene várakoznia, mire a posztja felkerül és neki kezdhet a barátai posztjaival való interakcióknak.

A választásom azért erre az alkalmazásra esett, mivel mikor elkezdtem használni, csak azon járt az agyam, hogy hogyan képesek megoldani azt a fejlesztők, hogy egyszerre ekkora terhelést kapnak a szervereik. Néha megesett, hogy rengeteget kellett várni a képek feltöltésére, ez pedig elindított egy gondolatot bennem, hogy vajon meg tudnám-e ezt úgy csinálni, hogy ne legyen késleltetés a képek feltöltése után és ha igen, akkor hogyan.

Nem csak az eredeti alkalmazás funkcionalitásait, de a kinézetét is ugyan úgy akartam megalkotni, hogy minél autentikusabb másolata lehessen neki az én változatom.

Az eredeti alkalmazás célja az, hogy leszoktassa a fiatal generációt a folyamatos telefon nyomkodásról és a közösségi médiától való függőségről azzal, hogy naponta csak egyszer kell megnyitnunk az appot, végig nézni, hogy a barátaink éppen mit csinálnak meg reagálni rájuk, majd be is csukhatjuk aznapra. Ezért is nevezték el BeReal-nek, azaz, hogy „légy valós”. Én az alkalmazásomat BeFake-nek neveztem el, hiszen az én változatom csak egy másolat, így találó ez a név.

## 1.2 A feladatkiírás értelmezése

A dolgozat során be fogom mutatni, hogy a mikroszolgáltatás alapú architektúrák hogyan teszik lehetővé a skálázhatóságot, hogyan képesek az egymás közti kommunikációra, továbbá, hogy miért robosztus egy mikroszolgáltatás alapú program és milyen további lehetőségeket nyújt a fejlesztések során.

Ismertetem a feladat megvalósításához használt technológiákat olyan szinten, amennyire a megoldás megértéséhez az szükséges és azt, hogy melyiket milyen megfontolásból választottam.

Ezek után el fogom magyarázni, hogy hogyan épül fel az alkalmazásom, milyen tervezési folyamatokon vittem keresztül, milyen problémákba ütköztem, ezeket hogyan és milyen megfontolások szerint oldottam meg. Beszélek még arról is, hogy milyen eszközöket használtam a tervezés véghezviteléhez és ahhoz, hogy jelenlegi formájába kerülhessen az alkalmazás.

Miután az olvasó már látta az alkalmazás architektúráját, megmutatom azt is, hogy hogyan néz ki az elkészült alkalmazás, hogy az is érthető legyen teljes egészében, hogy min dolgoztam a téma során és mit látna ebből egy felhasználó, ha éles környezetben megjeleníteném.

Legvégül kitérek arra is, hogy milyen eredményeket ért el az én alkalmazásom, hány felhasználót sikerült egyidejűleg kiszolgálni és ez hogyan viszonyul a kezdeti elvárásaimhoz, ami az volt, hogy csináljak egy, a BeReal-lel versenyképes klón alkalmazást.

A dolgozatomat azzal zárom, hogy összeszedek néhány olyan megoldást, újabb funkciót, vagy eszközt, amiket szívesen beleraknék még az alkalmazásomba továbbfejlesztés gyanánt, valamint írok arról, hogy ezek milyen előnyökkel járnának.



## **2 Felhasznált technológiák**

### **2.1 Választott technológiák és indokoltságuk**

#### **2.1.1 Spring framework különböző moduljai**

##### **2.1.1.1 Spring Boot**

A Spring Boot egy Java alapú framework, aminek segítségével könnyedén írhatunk önálló alkalmazásokat. Fő tulajdonsága az, hogy automatikus konfigurációkat hozhatunk vele létre, ami azt jelenti, hogy ha nem írunk felül bizonyos beállításokat, akkor azokat alapértelmezett módon próbálja létrehozni a hozzáadott .jar függőségek alapján. [1] Például, ha a HSQLDB az osztályútvonalon van, és a fejlesztő nem állít be kézzel egyetlen adatbázis-kapcsolati komponenst sem, akkor az automatikus konfiguráció beállít egy memórián belüli adatbázist. [2]

A Spring Bootnak vannak további csomagjai, amiket a projektbe importálás után (maven projekt esetén a pom.xml, gradle projekt esetén a build.gradle-höz hozzáadva) használhatunk. Ezek közül a legtöbbet a Web-et használtam. Ez a Controller-ek annotálását teszi lehetővé, aminek köszönhetően HTTP szervereket tudunk írni és endpoint-okat definiálni.

Használok még a Thymeleaf és a Mail csomagok által kínált lehetőségeket is ahhoz, hogy email küldéskor a felhasználók személyes, a felhasználónevüket tartalmazó, formázott leveleket kaphassanak.

##### **2.1.1.2 Spring Cloud**

A Spring Cloud eszközöket biztosít a fejlesztők számára az elosztott rendszerekben elterjedt minták gyors felépítéséhez (pl. konfigurációkezelés, szolgáltatáskeresés, megszakítók, intelligens útválasztás, stb). Az elosztott rendszerek koordinációja „boiler plate” mintákhoz vezet, a Spring Cloud használatával a fejlesztők pedig gyorsan elkészíthetik azokat a szolgáltatásokat és alkalmazásokat, amelyek megvalósítják ezeket a mintákat. Bármilyen elosztott környezetben jól működnek, legyen az lokális futtatás, „bare metal” adatközpont, virtuális környezet, mint az AWS EC2, vagy akár felügyelt platform, mint az AWS Lambda. [3]

Én a projekt során a Spring Cloudnak a Cloud Gateway, Eureka Naming Server, Feign Client, CircuitBreaker és Sleuth csomagjait használtam, amiknek java részét a későbbi pontokban fogok bemutatni külön-külön.

### 2.1.1.3 Spring Security

A Spring Security egy hatékony és nagymértékben testreszabható hitelesítési és hozzáférés-felügyeleti keretrendszer. Ez a de facto szabvány a Spring alapú alkalmazások biztosítására. A Spring Security egy olyan keretrendszer, amely a Java alkalmazások hitelesítésének és engedélyezésének biztosítására összpontosít. Mint minden Spring projekt, a Spring Security igazi ereje abban rejlik, hogy nagyon egyszerűen bővíthető, hogy megfeleljen az egyéni követelményeknek. [4]

Az alkalmazásom ezt a keretrendszert használja a JWT tokenek létrehozására és validálására az OAuth 2.0 JOSE modul segítségével, valamint a Spring Security Config modult az autentikáció és autorizációhoz.

### 2.1.2 JWT autentikáció és autorizáció RSA kulcspárral

A felhasználók autorizációját és autentikációját JSON Web Tokeneket használva valósítottam meg [5]. Ezek a tokenek arra szolgálnak, hogy adatokat szállítsunk aláírt állapotban, ezzel a forrásuknak a validitását garantálva. Három részből állnak, a fejlécből (header), a hordozott adatból (payload) és az aláírásból (verify signature). [6] A fejléc tartalmazza a token titkosításának algoritmusát és típusát, a payload-ban pedig a hordozni kívánt adatok „utaznak” JSON formátumban. A szerver miután feltölti a tokent adatokkal, aláírja a választott módon (esetemben mivel RSA256 algoritmust választottam a titkosításhoz, egy privát és publikus kulcs kellett ehhez), így készül el a végleges token. Egy példa arra, hogy hogyan néz ki egy token, alább látható.

eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJCUZha2UiLCJzdWUiOiJUb21pIiwiaXhwIjoxNjY3NTA5NTY3LCJpYXQiOiJlE2Njc0TE1NjcsInVzZXJJZCI6IjEiEiFQ.TcTGSC1HdDziPfQWmGmWm2W4MySEF-KQ8joQMPJCG2-UW\_ONH4YxDKYTxxCtXm9t8dL19pRGAPIAAUPfWwhBvyGoUWBZWGq8uduNsh9LZI1s93uZHakWN-5YjHIHCoe77-VENTmJn1EGe\_d0myeYeZLJs9h3wbUQk3ATy88q\_UTCVU\_T2w7H00W3709LSA5U0-6CKUZemCE2\_bd4ZV3WUCWYWTt1B86hXjAmin\_EJIWk3fgTabJzbwpLJIWCiHy0scf2glMMAIM-39PHA0H3uv\_dFAeU3qv8uzT1q13zCjR090DAT0okYB7GI9NCsaSuXki7WRA4dS85sBzePp1yw

Megfigyelhetjük, hogy a karakterek között két ' ' van, ezek segítségével határozhatjuk meg a token három részét.

A tokenek tervezésekor nagyon fontos arra figyelni, hogy ne rakjunk beléjük érzékeny, vagy bármi olyan adatokat, amit illetéktelenek rossz célból fel tudnának használni, mivel ezeket a tokeneket olvasni lehet, csak a validitásuk ellenőrzéséhez kell az aláíráshoz használt kulcsokat ismerni. Ezt ki lehet próbálni a fentebb látható tokennel is, ha beillesztjük az alábbi oldalon található szövegbeviteli mezőbe [6].

Az aláíráshoz egy RSA kulcspárt használtam, amiről azt kell tudni, hogy egy privát és egy nyilvános kulcsból áll. A privát kulcsot digitális aláírások generálására, az RSA nyilvános kulcsot pedig a digitális aláírások ellenőrzésére használják.

### **2.1.1 Adatbáziskezelés - PostgreSQL és a Spring JPA**

Mivel az adatokat egy viszonylag merev, strukturált formában tárolom, ezért az adatbáziskezeléshez az SQL-t választottam, ezzel kihasználva, hogy az adataim között nem fog inkonzisztens állapot kialakulni (tekintve, hogy támogatja az ACID tranzakciókat), mint ahogy az megtörténhetne, ha a NoSQL mellett döntöttem volna.

Ahhoz, hogy a kódból módosíthassam a táblákat, a Spring JPA-t használtam, mivel ez támogatja az Object-Relation Mapping (ORM) alkalmazását, aminek segítségével adatbázis táblákon tudok CRUD (Create, Read, Update, Delete) műveleteket végezni, Java objektumokhoz való hozzárendelésük után.

### **2.1.2 AWS**

Az Amazon Web Services (AWS) egy rendkívül széleskörű szolgáltatáskészlettel rendelkező, felhőalapú platform, aminek a projekt során több szolgáltatását is használtam.

#### **2.1.2.1 AWS S3**

Az Amazon S3 (Simple Storage Service) egy AWS által kínált fájlok tárolására használható felhő szolgáltatás, amit a projektem során arra használtam, hogy a profilképekhez, posztokhoz és reakciókhoz tartozó képeket tároljam rajta. Ezt olyan módon csinálom, hogy miután a felhasználó készített egy fotót, azt MultipartFile formátumúvá alakítom, majd elküldöm a megfelelő mikroszolgáltatásnak form-data-ként, amit pedig a szerver validál típus és méret szerint. Ha megfelelő formátumú és méretű képet küldött a kliens, akkor az elkészített objektum fájlnev tulajdonságába mentem el az S3-ban kulcsként használt értéket, ennek segítségével pedig vissza tudom kérni az ott tárolt képet. Erről a képek betöltése résznél 3.10 írok részletesebben is.

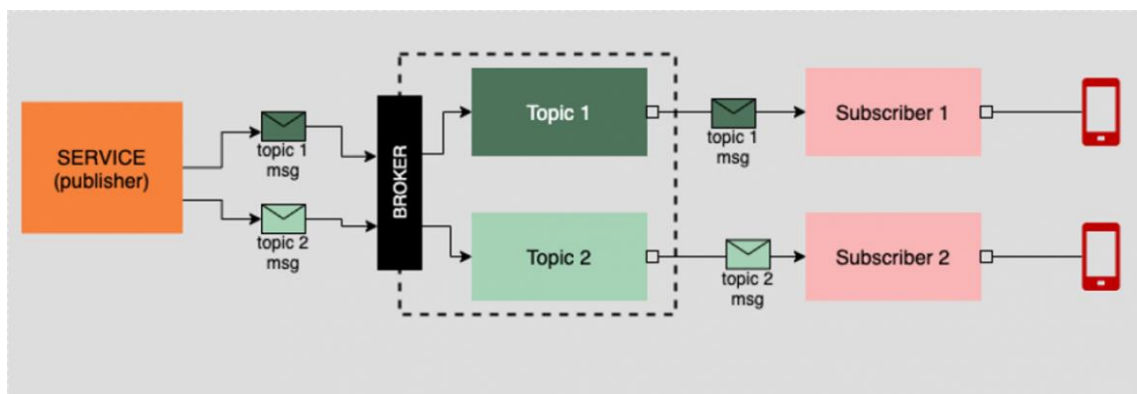
### 2.1.2.2 AWS SES

Az Amazon SES (Simple Email Service) egy felhőalapú e-mail szolgáltató, amely bármely alkalmazásba integrálható tömeges e-mail küldéshez. Én ezt az alkalmazásomban ahhoz használtam, hogy a Thymeleaf -fel megformázott email-eket a Spring mail segítségével ezen a szolgáltatáson keresztül küldjem ki, így kaphatok küldői statisztikákról szóló jelentéseket és megbizonyosodhatok arról, hogy minden email célba ért.

### 2.1.1 Kafka

Az Apache Kafka egy nyílt forráskódú elosztott eseményfolyam-platform, amit arra használok, hogy a kritikus fontosságú üzenetek, esetemben az értesítések megérkezzenek. Ehhez a Kafka egy nagyon megbízható, skálázható megoldást nyújt, amivel témákra küldhetnek üzeneteket a szolgáltatásaim, ezekre pedig figyelhet a notification-service és reagálhat rá az üzenetnek megfelelően. A Kafka akkor is megőrzi a beérkezett üzeneteket, ha valami hiba okán a notification-service működése megszakadna, újraindulása után pedig utólagosan kiküldi azokat, ez pedig jól jött nekem, mivel az alkalmazásomat hibatűrésre szerettem volna felkészíteni. Ezek mellett az is cél volt, hogy minél több felhasználót tudjon kiszolgálni egy időben, emiatt pedig szintén találó volt ez a technológia, mivel rajta keresztül trilliós nagyságrendű üzenetet lehet küldeni, átlagosan csupán 2ms késleltetéssel. [9]

Ugyan még számos egyéb alkalmazási módja és előnye van a Kafkának, én nem használtam ki teljesen az általa kínált lehetőségeket, így most többnyire csak arról fogok írni, hogy az a funkcionalitása, amit én használtam, hogyan működik. Ezt az alábbi ábra nagyon jól mutatja be magas szinten.

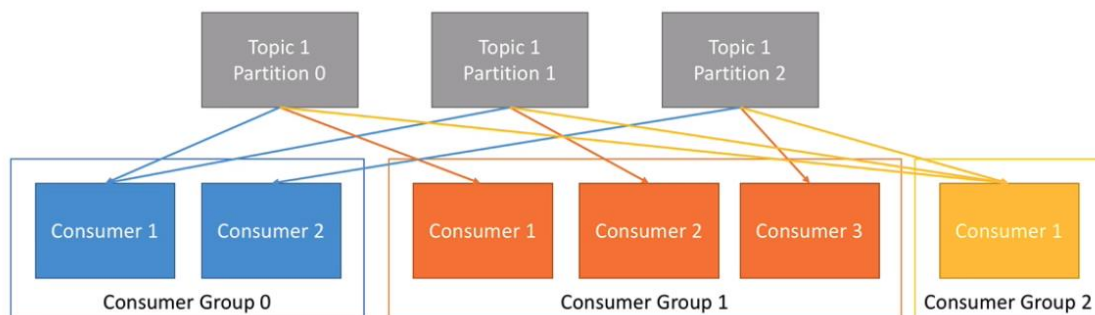


2.1:1 A Kafka üzenetek publish-subscribe munkafolyamata [10]

Ahhoz, hogy üzenetek küldését hozhassam létre, a következőkre volt szükségem: Producer-ek (néha Publisher néven is találkozhatunk velük), akik az adatokat küldik, Consumer-ek (esetenként Subscriber-ek), akik az adatokat a topic-okból kiolvassák, majd feldolgozzák, Kafka broker-ek, amik kezelik a rájuk beérkező üzenetek Producer-ek és Consumer-ek közötti tranzakcióját, topic-ok, amik gyakorlatilag az üzenet célpontját határozzák meg és a készülékek, amikre az üzeneteket szeretném küldeni.

Minden topic-ra feliratkozhat akár több Consumer is, esetemben viszont csak a notification-service volt feliratkozva az összesre, az üzeneteket pedig a time-service, user-service, post-service, interaction-service és a friend-service küldték. Így bármilyen üzenet érkezett a neki megfelelő topic-ra, azt az értesítéseket kezelő szolgáltatásom le tudta kezelni az annak megfelelő módon.

Ahhoz, hogy az üzenetek feldolgozása gyorsabban mehessen, lehet használni particionálást, ami azt jelenti, hogy a témákat felosztjuk több részre, ezekre pedig külön tudnak figyelni azonos Consumer csoportba tartozó Consumer-ek, viszont ebben elég nagy szabadságot ad a Kafka és olyan módon is lehet gyorsítani az adatátvitelt, ahogy azt az alábbi ábra demonstrálja. [10]



**2.1:2 Consumer csoportok**

Itt azt láthatjuk, hogy a Topic 1 nevű témát 3 partícióra bontottuk, amikre 3 különböző Consumer csoport tagjai vannak feliratkozva (a nyilak mutatják, hogy a Consumer csoport adott Consumer-e a téma melyik partíciójára beérkező adatokra figyel). Különböző témáknak a partícióikkal vett összességét egy Kafka Cluster-nek nevezzük.

A topic-okra érkező üzeneteket a jobb elérhetőség érdekében különböző replikákba lehet tenni, ami segítségével azok több brokeren lesznek tárolva.

Ahogy azt már írtam, a Kafka egy elosztott rendszer, így a brokerek egymással való kommunikációját is kezelni kell valahogy, amire a Zookeeper-t szokás használni. Ez a szolgáltatás lehetőséget nyújt a Kafka topic-jainak konfigurációjában, meta információinak tárolásában és elosztott feladatainak központosított irányításában, mint például a topic leaderek kezelését és még sok mást. Minden téma partíciói közül választ egy vezetőt, amire az üzenetek érkeznek, majd a többi ennek a követőjeként viselkedik, azaz a vezetőtől kapja az adatokat, de nem ad át információt a Consumer-eknek.

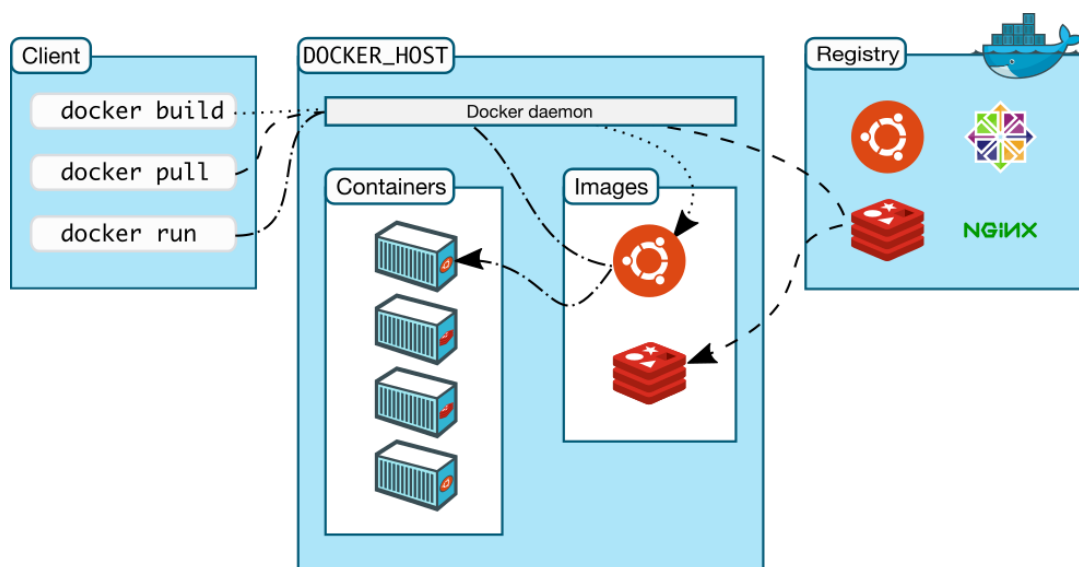
A docker-compose.yaml fájlban a Kafka Cluster-nek a topic-jait az alábbi módon hoztam létre.

```
KAFKA_CREATE_TOPICS:  
"befake:2:3,registration:2:3,post:2:3,comment:2:3,reaction:2:3,friend:2:3"
```

Ezzel a kódsorral azt konfigurálom, hogy milyen néven, hány partícióval és replikával hozza létre a topic-okat.

### 2.1.2 Docker

A Docker egy operációs rendszer szintű virtualizációt nyújtó szolgáltatás, aminek segítségével alkalmazásokat lehet úgynevezett image-ekbe csomagolni, ezzel elérve azt, hogy bármilyen környezetben ugyan úgy fusson a belesomagolt program. Ezzel egy hatalmas problémát küszöböl ki, ami az lenne, ha a fejlesztők megírnák a kódot a Windows-t, vagy MacOS-t futtató laptopjukon, ahol hibamentesen fut a program, de amikor éles környezetbe tennék ki, akkor már egy Linux-ot futtató szerver gépen kellene ugyan úgy futnia, ami viszont már nem triviális, hogy működni fog-e.



2.1:3 A Docker architektúrája

A fenti ábra azt szemlélteti, hogy a Docker architektúrája hogyan épül fel. Van egy Docker daemon, ami a Docker API kéréseket kezeli, mint például a `docker build`, `pull`, vagy `run` parancsot, valamint a Docker objektumokat (image, container, network, volume) kezeli. Vannak Docker kliensek, akik a daemon-nak küldik a parancsokat, ezeket a fejlesztők egyszerűen megtehetik a Docker Desktop alkalmazást használva. Az elkészített image-eket elérhetővé tehetjük másokkal egy általunk választott Docker registry-n keresztül, ilyen például a Docker Hub. Ezeket az image-eket mások felhasználhatják a saját alkalmazásaikhoz, én is ezt tettem például a Kafka esetében, így nem kellett a különböző környezetekben való előkészítésével annyit bajlódni.

A Docker rendkívül jól működik együtt a Kubernetes-szel, együtt lehetővé teszik a konténerek skálázását, erről viszont a következő pontban írok részletesebben.

### 2.1.3 Kubernetes

Röviden összefoglalva, a Kubernetes egy nyílt forráskódú szolgáltatás, aminek segítségével automatizálhatjuk konténerizált alkalmazások kitelepítését, skálázását és karbantartását.

Ahhoz, hogy elmondjam kicsit részletesebben, mire is képes a Kubernetes, bevezetem a konténerek fogalmát. Ezek gyakorlatilag számítógépek erőforrásait virtualizálják, egy virtuális géphez hasonlóan, majd ezekből különböző példányokat hozhatunk létre és futtathatunk bennük alkalmazásokat, az előző pontban említett Docker image-eket használva. Annyiban viszont különböznek a virtuális gépektől, hogy ezek kisebb méretűek és néha csak egy rövid kis időre hozzuk őket létre, ami azt a problémát okozza, hogy lehetetlen manuálisan létrehozni, törölni, rendszerezni meg monitorozni ezeket a konténereket.

Erre a problémára nyújt megoldást a Kubernetes. Ezt használva automatikusan tudom létrehozni meg törölni a konténereket, be tudom állítani, hogy melyik konténert milyen metrika szerint skálázza, de akár egyszerre is megtehetem ezt, több service csoportosításával. Ezen kívül még sok más feladatot is el tudok végezni az általa kínált automatizálási funkciókkal, például, hogy tárolhatom a jelszavakat, titkos kulcsokat és érzékeny adatokat rajta biztonságosan. Talán az egyik legfontosabb képessége, hogy abban az esetben, ha egy konténer meghibásodik, tud automatikusan a helyébe tenni egy újat, ha viszont egy konténer nem válaszol, akkor egy idő után ki tudja azt törölni, hogy ne pazarolja az erőforrásokat. Ahhoz, hogy el tudja dönteni, hogy egy konténer

meghibásodott, folyamatosan figyeli ezek „egészségét” (health check). Fontos még megemlíteni, hogy a konténerek elérésének módját DNS nevekkel és IP címekkel szabályozza, valamint load balance képességgel is rendelkezik, amiről a terhelés elosztókról szóló pontban 3.9 részletesebben is fogok majd írni.

Lehetőségem van még olyanra is, hogy egy frissítés közben megtartom a futó konténereknek egy részét a régi verzióval és csak egy részüket frissítem, ezzel megőrizve a rendszer elérhetőségét a leállítása nélkül, valamint, ha a frissítésbe hiba csúszik és a konténer health check-je problémát jelez, akkor azt automatikusan visszaállítja az előző verzióra. Ennek a technológiának a lényege az, hogy bármilyen komplexitású, a rendszerünkre illeszkedő infrastruktúrát ki tudjunk alakítani. A CI/CD folyamatoknak viszont nem része, a fejlesztésnek egy későbbi lépésében kerül képbe.

### 2.1.4 Feign Client

A FeignClient a Spring egy nagyon hasznos szolgáltatása, aminek segítségével könnyedén megvalósíthattam a mikroszolgáltatásaim közti kommunikációt. Először engedélyeztem a Feign Kliensek használatát az alkalmazáson, majd a megvalósításhoz a proxy tervezési mintát alkalmaztam, ami miatt van egy proxy interface-em, amiben definiálom a hívni kívánt szolgáltatásnak az endpointját, a kérés típusát és visszatérési értékét és annotáltam ezt a megfelelő módon.

```
@FeignClient(name = "user-service", fallbackFactory = UserFallback.class)
public interface UserProxy {
    @GetMapping("/user/user-by-username/{username}")
    ResponseEntity<User> findUserByUsername(@PathVariable String username);
}
```

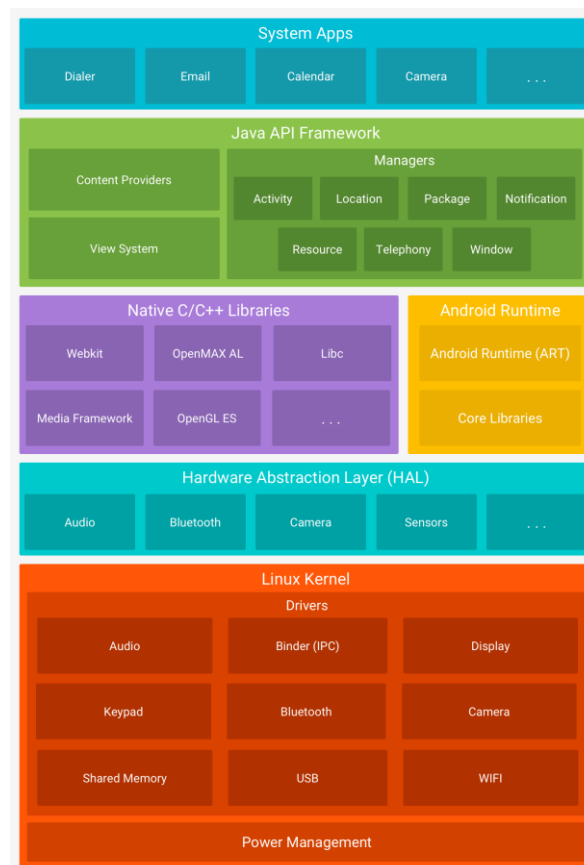
A kódrészletből az látszik, hogy a hívást a user-service-re küldöm, ha viszont nem kapok onnan választ, akkor egy UserFallback nevű osztályban definiált hibakóddal térítem vissza. A bemutatott sor egy felhasználónév alapján adja vissza azt, hogy melyik felhasználó tartozik hozzá.



## 2.2 Választott platformok

### 2.2.1 Android

Az Android egy nyílt forráskódú, alapvetően Linux-alapú szoftvercsomag, amelyet elsősorban érintőképernyős mobileszközökhöz, például okostelefonokhoz és táblagépekhez terveztek. A következő diagram az Android platform főbb összetevőit mutatja be.



2.2:1 Az android szoftvercsomag [7]

Az android platform alapját képezi a Linux Kernel. Az Android Runtime (ART) erre támaszkodik az olyan funkciók használatához, mint például a szálkezelés, vagy az alacsony szintű memóriakezelés. Mivel az Android Linux Kernelen alapszik, kihasználja annak legfontosabb biztonsági funkciók előnyeit és lehetővé teszi az eszközgyártók számára, hogy hardver-illesztőprogramokat fejlesszenek a már jól ismert kernelhez.

A Linux Kernel felett helyezkedik el a Hardware Abstraction Layer (HAL), ami olyan standard interfészeket biztosít, amiken keresztül az applikációk kommunikálhatnak a hardverekkel, mint a kamera, vagy a Bluetooth.

Az android 21-es API-szinje fölött az alkalmazások képesek a különböző process-eiket a saját Android Runtime (ART) példányukban futtatni. Ez arra való, hogy különböző virtuális gépeket futtathassunk vele anélkül, hogy ha az egyiknek baja esne (például lefagyna), akkor az egy másik folyamatra, vagy az operációs rendszerre kihatással lenne.

Mivel jónéhány alapvető Androidos komponens (mint például az ART vagy a HAL is) olyan natív könyvtárakon alapszik, amik C és C++ nyelven lettek írva, ezért ezeket is futtatnia kell a kernelnek. Ezekre a könyvtárakra láthatunk példákat a lilával megjelölt rétegben.

Az Android operációs rendszer szolgáltatáskészletének teljes egésze a Java API Framework rétegen keresztül érhető el, ez tartalmazza az alkalmazások létrehozásához szükséges építőelemeket, mint például a felhasználói felület készítéséhez szükséges View System, az értesítésekhez használt Notification Manager, az Activity Manager, ami az alkalmazások életciklusáért és a navigation back stack kezeléséért felel.

A rendszeralkalmazások rétegében helyezkednek el azok a beépített alkalmazások, mint például a számológép, SMS, telefon alkalmazás vagy az időjárás app.

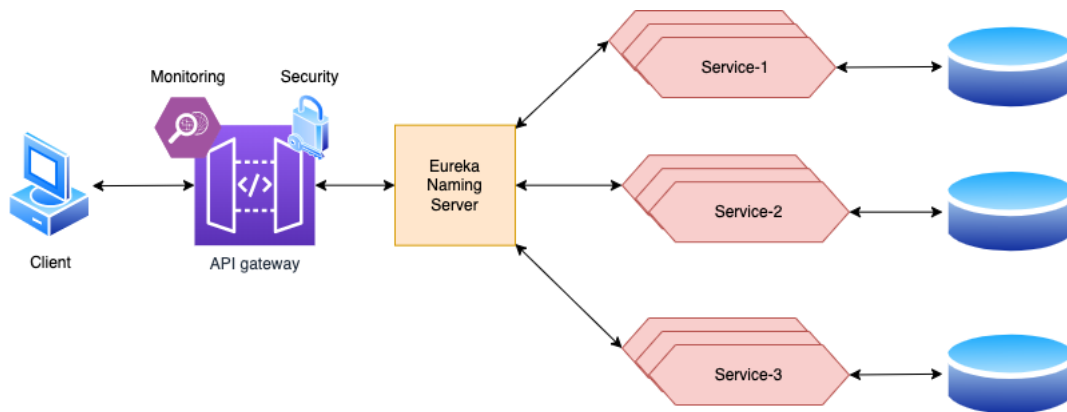
### **2.2.2 REST API alapú web**

A mikroszolgáltatásaim alapvetően 3 részből állnak össze. A Service-ek, amikben az üzleti logika van, a Repository, amiben az adatbázis kommunikációhoz szükséges kód található és a Controllerek, amikben pedig a Service réteg függvényei kerültek meghívásra, valamint a kliensektől fogadott majd nekik visszaküldött hálózati funkcionalitást kezelem. Itt találhatóak meg az olyan kódok, amik definiálják, hogy egy végpont milyen paramétereket vár, milyen lehetséges visszatérései vannak és ezek milyen HTTP státusz kóddal kerülnek visszaküldésre. Ennek, a közel 50 endpoint-nak az összessége szolgálja ki a klienseket és alkotja az alkalmazásom API rétegét.

A REST, vagy „REpresentational State Transfer”, egy architektúrális stílus web alapú szolgáltatások tervezéséhez. A RESTful egy olyan web szolgáltatás, amely a REST architektúrális stílusának megfelelően van tervezve, a HATEOAS, vagy „Hypermedia as the Engine of Application State” pedig egy REST megköttés, ami lehetővé teszi a rendelkezésre álló erőforrások felfedezését és navigálhatóságát a RESTful API-nak, anélkül, hogy annak szerkezetét előzetesen ismernénk. Emiatt frissíteni lehet az API-t úgy, hogy a korábbi verziókat használóknál is változatlanul működik.

## 3 A rendszer megtervezése

### 3.1 Magasszintű architektúra



3.1:1 Architektúra felépülése Eureka naming szerverrel és API gateway-t használva

A fenti képen az látható, hogy az Eureka-t és API gateway-t használva hogyan épül fel az alkalmazás architektúrája. Bal oldalon a Client reprezentálja a felhasználókat, akik szeretnék használni az alkalmazást. Ők ekkor az API gateway-re küldenek egy kérést, ami sikeres autentikáció után átirányítja őket egy, a kérés URL-jétől függő szolgáltatáshoz. Ennek a kivitelezéséről az Eureka gondoskodik, hiszen oda van bejegyezve minden szolgáltatás minden futó példánya, ezt mutatja a Service 1-től 3-ig beszámozott képek csoportja, mellettük a hozzájuk tartozó adatbázissal. Az, hogy melyik szolgáltatás, melyik másikkal kommunikál és milyen kapcsolatok vannak köztük, a 3.7:1 ábrán látható.

A szolgáltatások belül is kommunikálnak egymással, ahogy említettem már, erre a Spring framework Feign Client megoldását használtam. A service-ek kapcsolatának „térképe” a kommunikációik alapján a fenti ábrán látható módon néz ki.

Mivel a dolgozatban bemutatott alkalmazást egy felhő szolgáltatásban akartam futtatni, kicsit át kellett dolgozni az architektúráját, mivel már nem Docker, hanem Kubernetes környezetben szerettem volna, hogy működjön.

Az viszont, hogy az általam írt microservice-ek hogyan vannak kezelve a Kubernetes cluster-ben, közel sem egyszerű, mivel rengeteg megvalósítás létezik és nehéz megtalálni azt, ami számomra a leginkább megfelelő és kivitelezhető. Az általam választott Kubernetes cluster felépítése a következő elemekből áll. A legalsó szinten a

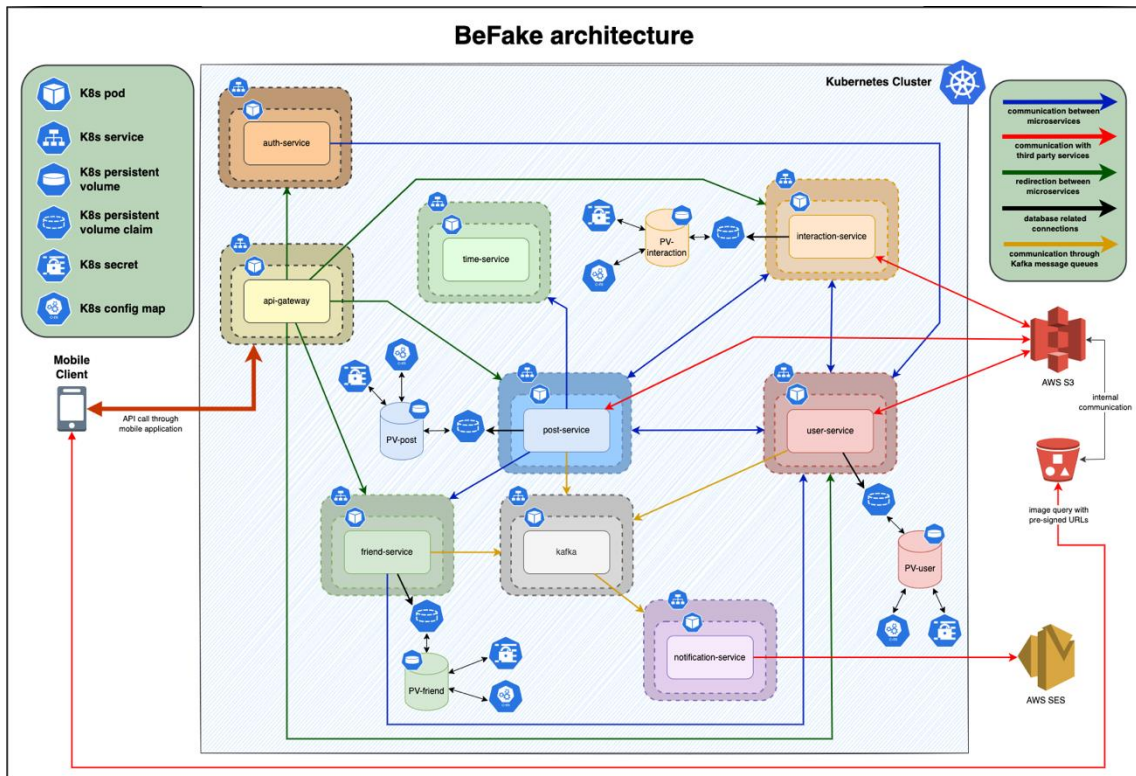
futtatni kívánt alkalmazás, esetemben a mikroszolgáltatások egy Docker image-be csomagolva állnak. Ezekhez a Docker image-ekhez készítettem konfigurációs fájlokat, amikben azt adtam meg, hogy az adott szolgáltatás kitelepítése (Deployment) milyen szabályok szerint történjen. Ennek a Deployment-nek a segítségével hoz létre automatikusan a (későbbiekben néha K8s-ként emlegetett) Kubernetes úgynevezett pod-okat, amikben futnak a konténerekbe tett image-ek. A Deployment konfigurációban adtam meg olyan szabályokat, hogy legalább és legfeljebb hány pod fusson, hány legyen elérhető, skálázódjon-e vagy sem, ha igen, mi alapján, valamint a konténerekhez itt adtam meg a környezeti változókat, amiknek segítségével például az Eureka szervert ki tudtam kapcsolni. A Kubernetes megoldást nyújt arra is, hogy titkos és kevésbé titkos beállításokat kezeljek, erre a ConfigMap (kevésbé titkos, például adatbázis neve) és Secret (titkos, például jelszavak) beállításokat használtam. Mivel a fentebb említett pod-ok ugyan azon az IP címen szolgálják ki a kéréseket, ezért, ha mindegyiket a maga portján használnám, összeakadnának. Erre az a megoldás, hogy összefogom őket egy LoadBalancer típusú Service-szel, ami elosztja a rá bejövő kéréseket a pod-ok között, valamint a Deployment-tel is együtt működik, hiszen, ha az éppen felskálazza a pod-ok számát, akkor már az újabb pod-oknak is tudja továbbítani a kéréseit.

Ahhoz, hogy a rendszer belsejét elérjem, az API gateway-emet elérhetővé tettem a külvilág számára, így, ha például Postman-t használva, vagy a mobil alkalmazásból az API gateway external IP címét címezem meg, akkor használni tudom a rendszert rajta keresztül.

Nagy nehézséget okozott a fejlesztés során, hogy a szolgáltatások közti service discovery-t Kubernetes környezetben megoldjam, ugyanis az Eureka-t szerettem volna lecserélni egy olyan megoldásra, ami K8s környezetben is megfelelően működik és bevett szokás is a fejlesztők között. Alapvetően erre a problémára több megoldás is létezik, de mindegyiknek megvannak a sajátosságai. Használhattam volna az Eureka naming szerver-t továbbra is, vagy a hozzá hasonló Ribbon Client-et, viszont amikor tanulmányoztam a problémát, azt állapítottam meg, hogy érdekesebb lenne kettő másik opció közül választanom, amik a Kubernetes natív service discovery megoldása és a Spring Cloud Kubernetes-ben lévő Discovery Client [14]. Az előbbi azt tudja, hogy meg tudom címezni például a Feign Client-ből a többi szolgáltatásomat olyan módon, hogy

```
{service-name}.{namespace}.svc.{cluster}.local:{service-port}
```

Itt gyakorlatilag kihasználjuk, hogy az összes service-t a K8s automatikusan elérhetővé teszi a fenti formátumban látható domain-nel. A problémám ezzel az volt, hogy ha bármit megváltoztatok az elnevezéseimben, akkor kezdetem volna előlről a microservice-ek deploy-olását, mivel át kellett volna írni az összes FeignClient implementációt a kódban. Az utóbbi viszont nagyon megtetszett, ugyanis csak egy plusz függőséget kellett felvenni a pom.xml fájlokban, engedélyezni a felfedezést, majd a FeignClient-ben megadott „name” érték alapján már elérhetőek is lettek a szolgáltatások.

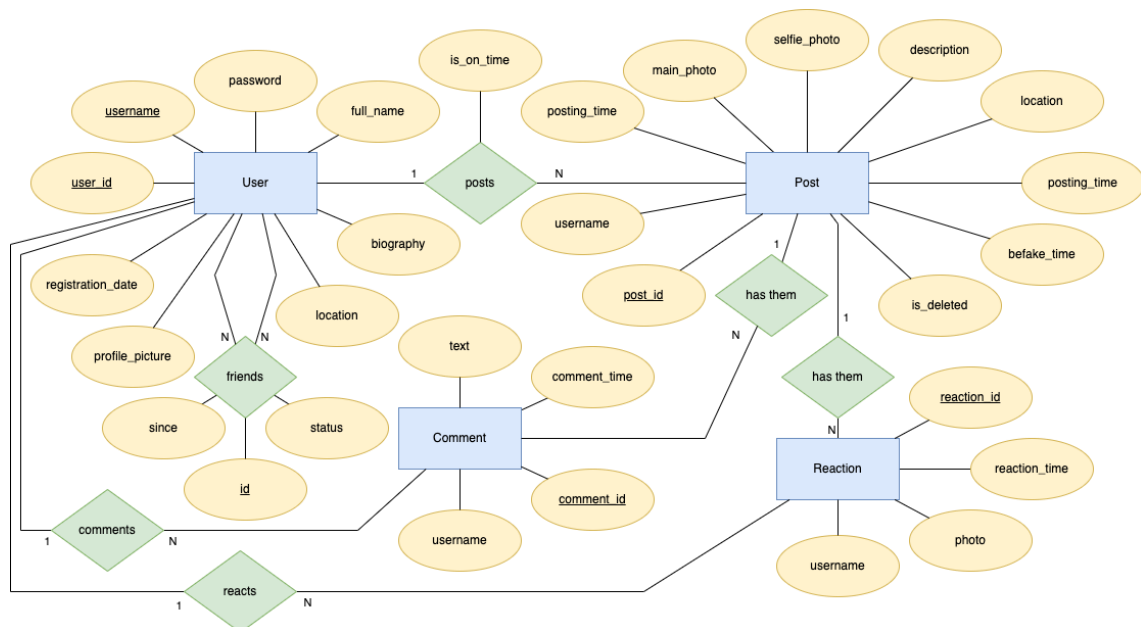


3.1:2 Az alkalmazás architektúrája Kubernetes környezetben

A fenti ábra a Kubernetes-re áttérés utáni architektúráját mutatja be az alkalmazásnak. A színes dobozok a microservice-ek pod-okban futását mutatja, amiket a service-ek terhelés elosztással kezelnek. A 4 mikroszolgáltatás, ami adatbázis kapcsolattal működik össze van kötve perzisztens adattárolókkal egy PVC-n keresztül, mivel ezen keresztül kér hozzáférést a PV-hez, amin az adatok tárolva vannak. Ezeket egy saját postgres:latest Docker image-et futtató service-en keresztül teszik meg, viszont a kép leegyszerűsítése miatt azokat kihagytam az ábrából. A másik két Kubernetes-hez köthető kapcsolat a ConfigMap és a Secret, amikről már fentebb írtam részletesebben. Az ábrán látható összeköttetések jelentését a jobb felső sarokban lévő jelmagyarázat alapján lehet értelmezni. A nyilak kezdő és végpontjának indoklása az, hogy a microservicek

küldik a hívást egy adott Kubernetes service-nek, ami a benne futó pod-okból egynek kiosztja a feladatot a terhelés elosztást figyelembe véve, majd a választott pod a benne futó mikroszolgáltatásnak „továbbítja” azt, ami pedig elvégzi a feladatot. A third party kommunikációknál azért mennek a nyilak direktben a mikroszolgáltatásokhoz, mivel ott nem a service dönti el a célpontot, hanem a hívó mikroszolgáltatás kapja csak vissza a választ a third party szolgáltatástól. Az imént bemutatott koncepciót az 5.2:2 ábra kicsit változtatott formában ábrázolja, csak ott egy ingress irányítja a kéréseket, nem pedig az egyik service, esetemben az API gateway.

## 3.2 Adatmodell, E-K diagramm



3.2:1 Ábra az E-K diagrammról

Az egyed kapcsolat diagram tervezésekor különféle szempontokat kellett figyelembe venni, amiknek eredményeképpen a fenti ábra jött létre.

A megfontolás emögött a következő volt. Mivel az alkalmazásom a felhasználók kezeléséhez azoknak modelljeit perzisztensen fogja tárolni, kellett, hogy legyen egy User egyedem. A felhasználók az adataikat is meg tudják adni, vagy változtatni, így ezeket tulajdonságokként felvettem az entitásra.

Az alkalmazás lényegi része az, hogy lehet posztokat készíteni, így létrehoztam egy Post entitást is a hozzájuk tartozó tulajdonságokkal. Mivel az alkalmazás úgy működik, hogy a felhasználók minden nap egy képet posztolhatnak, egy kép viszont

mindig egy felhasználóhoz tartozik, így közéjük felvettem egy 1-N kapcsolatot, valamint azt, hogy mikor került ki a poszt és mikor volt itt az ideje az aznapi posztnak.

A posztokra lehet kommentelni és reagálni is, minden poszthoz tartozhat több komment és több reakció is, viszont ezekhez egyértelműen megadható a felhasználó, aki kommentelt / reagált, emiatt köztük és a felhasználó entitás között is egy 1-N kapcsolatot alakítottam ki.

A felhasználóknak lehetőségük van egy barátlista kialakítására, ezért létrehoztam a User entitáson saját magával egy kapcsolatot, amin jeleztem, hogy lehet N-N kapcsolat a felhasználók között, azaz egy felhasználónak lehet több ismerőse is, ahogy az ismerősének is lehetnek más ismerősei. Ennél a kapcsolatnál feltüntettem, hogy hozzá tartozik egy státusz, azaz, hogy milyen fázisban van a barátságuk (elfogadásra vár, vagy már aktív, de a továbbiakban akár ki is lehet egészíteni a blokkolt státusszal is, ha meg akarja valamelyik felhasználó tagadni egy másik előtt azt, hogy az a baráti kérelem elutasítása után újra küldhesse azt) és egy since, ami azt jelzi, hogy mióta aktív a barátságuk (mikor fogadta el a másik fél a barát jelölést).

Ez az adatbázisban így jelenik meg: USER1\_ID, USER2\_ID, STATUS, SINCE. Ezt a status tulajdonságot használom fel arra, hogy a bejövő barátkéréseket megjelenítsem, még hozzá egy szűréssel, aminél, ha a user2id a felhasználóé és a status pending, akkor az egy bejövő barátkérést jelent, de ugyan ilyen könnyen az is kitalálható, hogy ki küldte eredetileg a barát kéréseket, kinek.

### **3.3 Adatbázis választása**

A mikroszolgáltatásoknak meglehetősen fontos tulajdonsága az, hogy teljesen izoláltan működnek egymástól olyan értelemben, hogy saját adatbázissal rendelkeznek, nem írnak egymás tábláiba, ezzel nem váratták meg egymást. Ezért is választottam szét az alkalmazásomban őket, hogy minden mikroszolgáltatáshoz tartozzon egy külön adatbázis.

Ha valamelyik szolgáltatásnak szüksége van egy másiknak az adataira, akkor azt egy REST kéréssel kapja meg attól, de nem ír bele annak a táblájába, ezzel esetleges fennakadásokat alkalmazva. Ugyan manapság már az adatbázisok megoldják önmagukban a skálázódást, de szerettem volna tartani magamat az iparágban bevett felépítéshez, ezért döntöttem a megoldásom mellett.

Annyi hátránya van ennek a megoldásnak, hogy mivel például egy törlés művelet több szolgáltatás, különböző adatbázisainak írásával valósul meg, ahhoz, hogy garantálhatjuk a hiba mentes tranzakciókat, saját tranzakciókezelést kell megvalósítani, ha nagyon biztonságossá szeretnénk tenni az alkalmazást. Ez azért kell, mert ha az egyik adatbázisban már kitöröltük az összes kívánt adatot, de a másik meg valami hiba folytán nem elérhető, vagy csak történik egy váratlan hiba, akkor megszakad a folyamat és az adatok egy része megmarad, a többi meg már törlésre került, így egy nem várt állapot áll elő.

### **3.4 A monolitikus és mikroszolgáltatás alapú alkalmazások közti különbség**

A két megoldás közül nincsen egy jobb és egy rosszabb, de vannak mindkettőnek előnyei is és hátrányai is, amiket most bemutatok és leírom, hogy miért döntöttem végül a mikroszolgáltatás alapú megoldás mellett. Ugyan mindkét megoldással robosztus megoldásokat lehet készíteni, de ha nem választunk megfelelő körültekintéssel, akkor jelentősen megnehezíthetjük a dolgunkat a fejlesztés során.

A monolitikus alkalmazásokról ismert, hogy egy nagy kódbázisból állnak, tartalmazzák a backend és frontend kódot is egyaránt, valamint a konfigurációs fájlokat is minden esetben. Mivel az alkalmazás méretétől függően változik az alkalmazás komplexitása is, ezért a továbbfejlesztése kihívásokkal járhat. Ennek ellenére mégis nagyon sok cég használja, ugyanis könnyebben tesztelhető, hiszen csak egyetlen kódbázist kell kezelni. A fejlesztésük is sokkal egyszerűbb a mikroszolgáltatás alapú alkalmazásokhoz képest, de nem csak amiatt mert egy kódbázis van, hanem mert nem kell specifikus architektúráis tudást elsajátítaniuk a fejlesztőknek, ami különös készséget igényelne (Itt olyanra gondolok, mint például a szolgáltatások közti kommunikáció). Ugyanezeket összevetve az üzembe helyezés is egy gyorsabb folyamat, azonban technológiai korlátai vannak ennek az architektúrának, hiszen ha frissíteni akarnám bármelyik komponensét, akkor biztosítanom kellene, hogy az az apró változtatás nem fogja elrontani bármelyik másik részét az alkalmazásnak, amit viszont nagyon nehezen lehet garantálni, ugyanis a mikroszolgáltatásokkal szemben itt fennáll a „single point of failure” fogalma a kód szoros kapcsolódása miatt, azaz ha egy valami elromlik a kódban, akkor az az egész alkalmazást elronthatja.



A mikroszolgáltatás alapú alkalmazások a monolitikusan felépülőknél jó pár hibáját javítják, de ez néha kellemetlenséggel jár. Ennek a koncepciónak a lényege abban rejlik, hogy az alkalmazást szét kell bontani különböző kódbázisokra (mikroszolgáltatásokra) azok feladataitól vagy felelősségeitől függően, ezzel az egymástól való függetlenséget megteremtve. Mivel már nem egy kódbázisból áll az alkalmazás, nem áll fent a korábban említett „single point of failure”, igény szerint lehet változtatni a kódbázisokat anélkül, hogy ez a többi futását akadályozná, vagy újabb mikroszolgáltatásokat hozzáadni különösebb nehézségek nélkül. Ez több szempontból is előnyt jelent, az egyik, hogy akár eltérő programnyelveken is írhatjuk ezeket a szolgáltatásokat, vagy frissíthetjük egyenként őket, addig, amíg az egymással való kommunikáció módját nem változtatjuk meg. Ennek a módja általában az API-n keresztüli kommunikáció, így alkotják meg közösen az egész alkalmazás teljes funkcionalitását.

Hatalmas előnye ennek az architektúrának, hogy szolgáltatásonként lehet skálázni, szóval, ha egy bizonyos funkció nagyobb terhelést kap, akkor elég csak az azt megvalósító szolgáltatás méretét növelni, ezzel csökkentve a fenntartási költséget. Ezzel viszont óvatosan kell bánni, mivel akár hátránya is lehet a mikroszolgáltatás alapú alkalmazásoknak, ha túl sok, keveset használt szolgáltatásunk van, ugyanis ezeket mind futtatni kell valahol, az meg erőforrás igényes, azaz magas plusz költségekkel jár.

Ugyan csökkentjük az egyes modulok komplexitását, viszont az alkalmazás egészének jelentősen megnőhet a bonyolultsága olyan módon, amit a monolitikus alkalmazásoknál nem tapasztalunk, hiszen ott nem kell ilyen problémákkal foglalkozni. Ahogy a monolitikus alkalmazásoknál már előnyként írtam ezt, itt hátrányként jelenik meg az, hogy a tesztelés rendkívül komplexsége tud válni, mivel itt nehezebb a hibák okainak visszafejtése. Ennek enyhítésére ajánlott használni valamilyen logger-t, ezért én is választottam egy elosztott rendszerek tesztelésére valót, a Zipkin-t 3.5, amiről még később fogok írni.

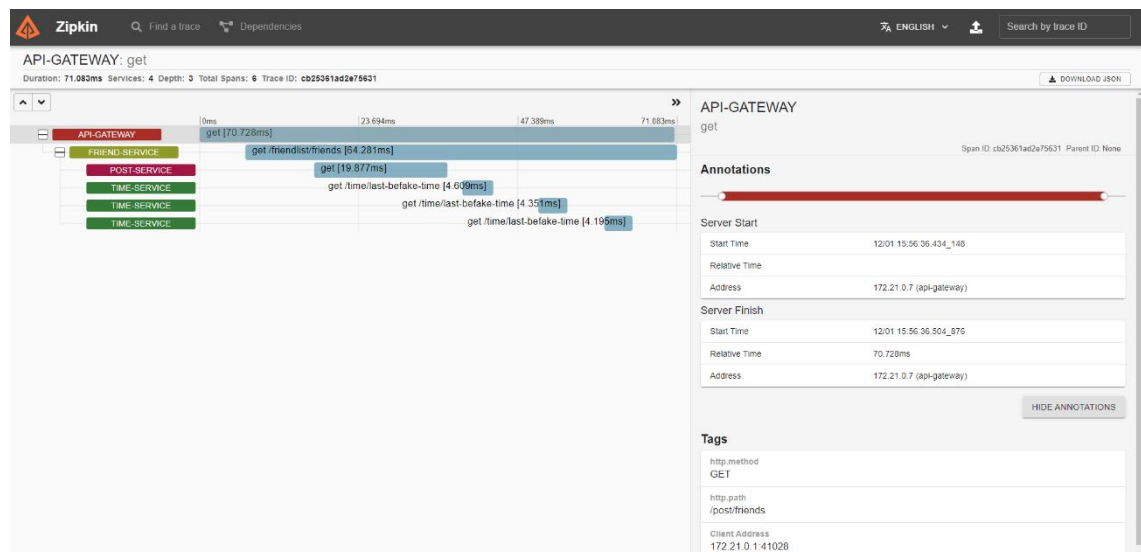
Ahhoz, hogy a megfelelő alkalmazás architektúrát válasszam ki és ne kövessek el olyan hibákat, amik később megnehezítenék a projekt tovább fejlesztését, a következő szempontokat vettem figyelembe. Szerettem volna egy olyan alkalmazást fejleszteni, aminek a komplexitása nem nő attól, hogy a kódbázist növelem, könnyű karban tartani, igény esetén továbbfejleszteni és ami pedig elhanyagolhatóan szempont volt, az az, hogy könnyen skálázódjon, hiszen a projektem célja ennek a demonstrálása.

## 3.5 Logolás

Én a projekt során azSlf4j által nyújtott logoló funkciókat használtam, ami egyrészt az összes, API gateway filterén keresztül menő kérést logolja, hogy könnyebben lehessen megállapítani, hogy hol akadt el egy kérés, valamint alkalmanként használtam nagyobb funkcióknál is, hogy pontosabb képet kapjak a hiba okáról. Ezek mellett az IntelliJ beépített debug funkciója is sokat segített a hibák megtalálásában. Az API gateway-ben lévő logger így néz ki:

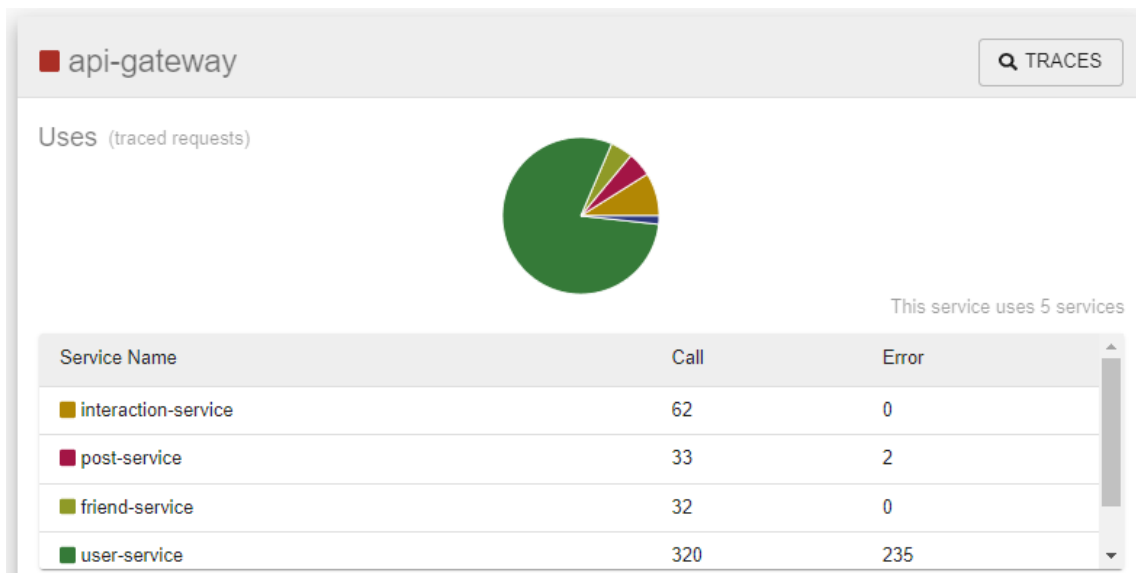
```
@Slf4j
@Component
public class LoggingFilter implements GlobalFilter {
    @Override
    public Mono<Void>filter(
        ServerWebExchange exchange,
        GatewayFilterChain chain) {
        log.info("Path of the request received -> {}",
            exchange.getRequest().getPath());
        return chain.filter(exchange);
    }
}
```

Ahhoz, hogy igazán jól lássam, hogy egy kérés milyen úton megy végig és hol akad el, a Zipkin nevű elosztott követő rendszert (distributed tracing system) használtam, amit minden követni kívánt service függőségeihez hozzáadtam. Ezek segítségével akár grafikusan is láthattam, hogy a kérések honnan-hova mentek, milyen hibakóddal akadtak el és mennyi időbe telt ez anélkül, hogy bármennyi kódot kellett volna írni. A Zipkin mélyen beépül a Spring alkalmazásba és olyan eseményeket naplóz ki, amiket a kódból meglehetősen nehéz és bonyolult lenne megoldani.



3.5:1 A Zipkin-ben látható hívások egyike

Azért ezt a fenti képet 3.5:1 választottam a bemutatásához, mivel ez a kép tökéletesen demonstrálja, hogy milyen hibákat lehet vele találni az alkalmazásomban. A bal oldalt látható service-ek nevei mellett láthatjuk, hogy mennyi idő alatt szolgált ki bizonyos kéréseket a szerver, milyen endpoint lett hívva és melyik service-ből. Látható, hogy itt egy már bejelentkezett felhasználó készüléke éppen betölteni próbálja a barátainak a képeit (jelen esetben 1 képet), viszont ahhoz, hogy azt kiszámolja, hogy a posztjának elkészítése és a központi posztolás ideje között mennyi idő telt el, elég hátrányos módon, háromszor is lekéri azt. Ezt azért jó, hogy látom az ábrából, mert így már tudom, hogy a mobil kliensem nem a legeffektívebb módon kezeli az időeltérések számítását, ezt pedig tudom majd javítani. Még egy hasznos statisztika, amit látni lehet a Zipkin felhasználói felületén az az, hogy arányaiban mennyi kérés ment, melyik service-hez, ezek közül pedig hány volt sikeres és hány sikertelen.



**3.5:2 A Zipkin vizualizációja, amiből a szolgáltatások hívásait és hibáit elemezhetjük**

Az ábrán azt látjuk, hogy a user-service nagyon nagy arányban produkál sikertelen válaszokat, aminek pedig két fő okozója van. Az egyik, a ritkább, hogy a felhasználók sikertelenül jelentkeznek be, így az authorization-service-nek hibakódot küld vissza. A másik, amit az előző képen 3.5:1 bemutatott módon elemezve találtam meg pedig az, hogy azoknak a felhasználóknak is lekérdezem a profilképük linkjét, akik nem állítottak még be egyet, emiatt a szerver nem találja meg azokat az adatbázisban és hibával tér vissza. A Zipkin-nek még van több nagyon hasznos tulajdonsága is, ezekből lesz olyan, amiről később még írok.

### 3.6 Szolgáltatások szétválasztása szerepek / felelősségek szerint

A microservicek szétválasztásánál a szempont a felelősségi körök és a feltehető terhelésük mértéke volt a szempont. Alább a különböző szolgáltatásokat fogom felsorolni felelősségeik szerint és néhánynál indoklással támasztom alá döntéseimet.

- user-service: Felhasználók regisztrálása, profil és profilkép szerkesztése, valamint a felhasználók adatainak kezelése, többi szolgáltatás számára elérhetővé tétele
- authorization-service: Autorizáció kezelése, azaz JWT tokenek generálása a bejelentkezett felhasználók számára, ezek aláírása és megfelelő információkkal feltöltése. Az autentikációt és autorizációt azért választottam szét, mert egy nagyobb alkalmazásnál lehet többféle bejelentkezési metódus is, ráadásul az autentikációnak nincsen köze a felhasználó személyes adataihoz. Ugyan így fordítva is igaz, hogy a felhasználót kezelő servicenak nincsen köze a bejelentkezés folyamatához és a tokenek generálásának folyamatához.
- post-service: Itt lehet készíteni, lekérni, szerkeszteni, törölni a posztokat. Feltehetően ebből fog majd a legtöbb példány futni, mivel ez lesz a leginkább leterhelve az értesítést követően.
- interaction-service: Itt lehet kommenteket és reakciókat létrehozni, törölni.
- friend-service: Ez a szolgáltatás kezeli a barátok listáját. Lehet bejelölni, törölni, jelölést elfogadni vagy törölni, ezek mellett pedig le lehet kérdezni a függőben lévő, vagy már aktív barátságok listáját.
- notification-service: Ez a szolgáltatás felelős az eseményekről való értesítések küldéséről a mobilok felé.
- naming-server: Ez a szolgáltatás egy Eureka naming server-t valósít meg, ami a manuálisan skálázott szolgáltatásokat regisztrálja be magához és osztja el a terhelést köztük.
- api-gateway: Ez a szerver valósítja meg a routing-ot, azaz dönti el, hogy bizonyos kéréseket melyik mikroszolgáltatásoknak kell kiszolgálniuk. Ezt úgy csinálja, hogy be van hozzá regisztrálva az összes microservice és amikor akarunk küldeni egy kérést az egyikre, nem kell megkeresnünk, hogy az most éppen melyik porton fut (ráadásul mivel több példányban van, azt is meg kellene tudnunk, hogy melyik van éppen kevésbé terhelve), hanem hívhatjuk a gateway portján a megfelelő

URI-t, ami utána tovább irányítja azt a megfelelő helyre. Ezek mellett még az autentikációról is ez a szolgáltatás gondoskodik.

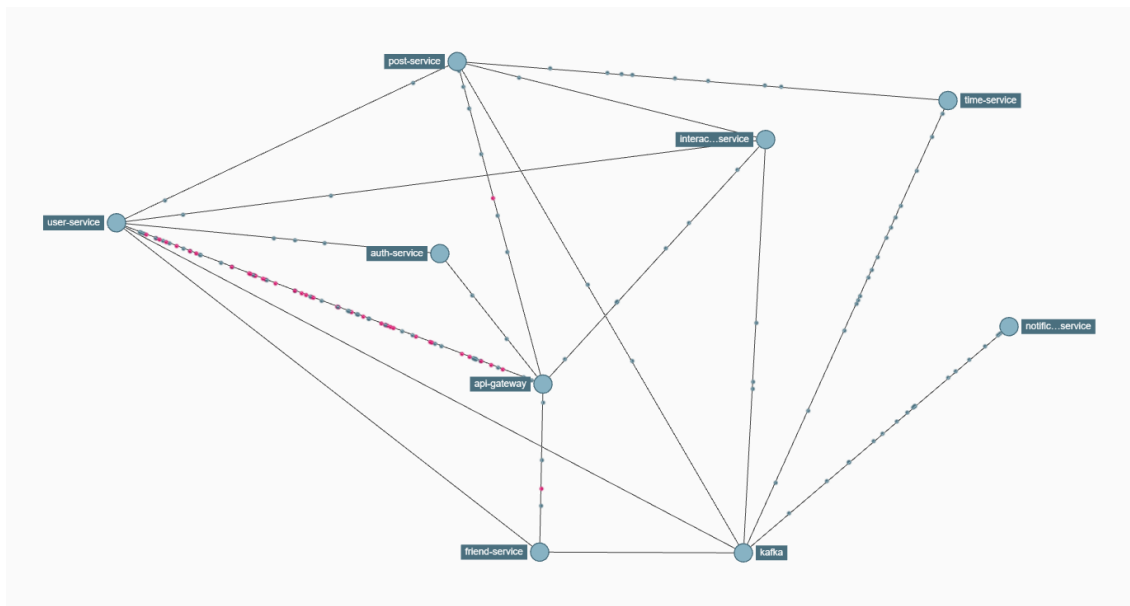
- **time-service:** Ez a szolgáltatás felel a random generált időért, amiből kifolyólag adódik belőle egy probléma. Mivel jelenlegi működése szerint memóriában tárolja ezt a véletlenszerűen kitalált időt, ezért szűk keresztmetszetnek számít az alkalmazásban. Ennek az az oka, hogy ha több példány futna belőle, nem lehetne eldönteni, hogy melyik változat generálta az igazi időt, viszont mivel a posztoláskor használva van ez a szolgáltatás, jelentősen függ tőle a kiszolgálás sebessége. Ennek a kiküszöböléséről később még fogok írni a fejlesztési lehetőségek kapcsán. A másik probléma ezzel a fajta megoldással az az, hogy ha ennek a szolgáltatásnak az egyetlen példánya elérhetetlenné válik, akkor nem lesz másik, ami kiszolgálja a többi szolgáltatást, így a felhasználók nem fogják tudni, hogy mikor kellett volna kirakni posztot, ebből kifolyólag azt sem, hogy mit kellene látni, hiszen az a napi poszt attól függ, hogy már volt-e véletlen idő, vagy sem. Ez véleményem szerint meglehetősen függővé teszi az alkalmazást ettől a szolgáltatástól, így mindenképpen szeretném a későbbiekben ezt a problémát orvosolni.

### **3.7 A szolgáltatások közti, valamint a szerver és a kliens közti kommunikáció módja**

A felhasználók a regisztrációjuk sikerességéről, az új posztokról, kommentekről, reakciókról és barát jelölésekről értesítéseket kapnak. Mivel az, hogy ezek az értesítések megérkezzenek, rendkívül fontos, kellett találni egy ehhez megfelelő eszközt, ami egyben egy megbízható szolgáltatás. A választásom az üzenetsor, vagyis „message queue” elnevezésű technológiára esett, azon belül pedig a Kafka-ra. Erről már korábban írtam a választott technológiák részben. 2.1.1 Azért megfelelő az esetemben ez a választás, mert ha a notification-service elérhetetlenné válik, attól még a Kafka tárolni fogja a beérkezett üzeneteket, a notification-service pedig amint visszakapcsolódik, azonnal megkapja ezeket, így a felhasználók nem maradnak le semmiről, legrosszabb esetben is csak kicsit később kapnak értesítést az eseményekről.

Alapvetően a szolgáltatásaim a saját API-jukat szolgálják ki, amiket a mobil kliens retrofit [16] segítségével hív meg. Mivel a legtöbbször nem kimondottan kritikus az elérhetőség, így nem tartottam fontosnak, hogy mindenhol message queue-kat

alkalmazzak a kommunikációhoz, így, ha ezek egymással akarnak kommunikálni, akkor ahhoz a Spring Cloud által nyújtott, korábban már kifejtett Feign Client-et használom.



**3.7:1 A Zipkin-ben látható kommunikációk a szolgáltatások között**

Ahogy már említettem korábban, a Zipkin-nek vannak még hasznos funkciói, ezek egyike az, hogy a fenti ábrán 3.7:1 látható módon vizualizálja a microservice-ek közti API hívásokat. Az ábrán látható gráf csúcsai a szolgáltatások, élei pedig a lehetséges kommunikációs csatornák a két szolgáltatás között, azaz amelyik két service között van él, azok kommunikálnak egymással. Az éleken utazó pontok egy-egy kérést reprezentálnak, színük pedig a hívás sikerességét indikálja, szürke színnel a sikeres, pirossal pedig a sikertelen hívások láthatóak. Ugyan a Zipkin nem jelenít meg minden élet automatikusan, így, hogy tökéletes képet alkothassunk az alkalmazásról, minden funkciót ki kell próbálni, csak utána fog látszódni az ábrán. Ahhoz viszont, hogy bemutassam a szolgáltatások közti kommunikációt, megfelelő eszköznek bizonyult. Hasznos lehet olyan elemzésekhez is, amikben azt vizsgálom, hogy melyik szolgáltatás mennyire van terhelve (ezt az éleken utazó pontok mennyiségéből tisztán láthatjuk), ami egy elég jól bemutatja, hogy melyik szolgáltatásokat kell skálázni, esetleg vissza skálázni, de üzleti elemzésekhez is lehet használni nagyobb alkalmazásoknál. Erre példa, ha egy-egy funkciót elenyésző mennyiségű felhasználó használ, akkor el lehet gondolkozni rajta, hogy van-e értelme még futtatni miatta egy külön service-t. Ez mind Docker környezetben, mind Kubernetes környezetben felesleges erőforrásokkal jár, így az ilyen döntések nagyon sokat segíthetnek az alkalmazás hatékonyságának növelésében.

### 3.8 Autorizáció, Authentikáció és az API gateway kapcsolata

Erre a problémára rengeteg megoldási lehetőség létezik, én viszont úgy döntöttem, hogy a megoldásomban az API gateway fogja végezni az autentikációt, az auth-service pedig az autorizációt.

Az API gateway ahogy azt már korábban is kifejtettem, egy routing-ot valósít meg, azaz, ha a felhasználó regisztrálni, vagy bejelentkezni akar, akkor azt tovább irányítja az ennek megfelelő endpoint-ra, máskülönben csak akkor engedi tovább a felhasználót, ha az autentikálta magát egy tokennel.

Az, hogy melyik mikroservice szolgálja ki a bejövő kérést, az alábbi kódrészletből látható.

```
@Bean
public RouteLocator gatewayRouter(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("user-service", r->r.path("/user/**")
            .filters(f->f.filter(filter))
            .uri("lb://user-service"))
        .route("auth-service", r->r.path("/auth/**")
            .filters(f -> f.filter(filter))
            .uri("lb://auth-service"))
    (...)
    .build();
}
```

Ebből azt lehet látni, hogy a különböző kéréseket filterezem az alapján, hogy mi az URI-juk eleje, ugyanis minden microservice kontrollerének adtam egy saját előtagot a lehető legráilóbb módon, például a user-service endpointjai /user/\*\* mintájúak, így, ha az API gateway lát egy ilyet, azt a load balance-olt user-service-nek küldi tovább.

Azért döntöttem emellett a megoldás mellett, mert az én megértésemben az API gateway egyfajta jegy ellenőr, aki megnézi, hogy érvényes-e, vagy hogy egyáltalán van-e jegye a beérkező felhasználónak és ha nincs, akkor csak a jegypénztárhoz, azaz az én esetemben a register/login-hoz engedi tovább, azt viszont nem tartom a felelősségének, hogy JWT tokeneket provizionáljon a felhasználóknak meg kikeresse az adatbázisból, hogy jók-e a bejelentkezési adatok. Ezt a feladatot szerintem egy külön service-nek kell ellátnia, így szétoszlik a terhelés is és kontrasztosabbak a felelősségek is.


Ez a megoldás viszont előhozza azt a problémát, hogy mivel külön szerver generálja a kódot és egy másik validálja, más secret-ekkel is fognak dolgozni, ami miatt nem fogják elfogadni a megadott tokeneket mert nem bíznak meg bennük. Erre azt a

megoldást találtam, hogy egy aszinkron RSA kulccsal generálok egy privát és egy publikus kulcspárt és ezekkel a kulcsokkal titkosítom és dekódolom a tokeneket, ezek a kulcsok pedig minden szerveren ott vannak így garantáltan ugyan azt fogja használni az összes service. Ennek előnye, hogy a program működik, hátránya pedig az én esetemben, hogy nem generálódik magától újra minden szerver indításkor a kulcs és mivel a fájlrendszerből elérhető, így csökkenti az alkalmazás biztonságát. Ennek a biztonsági kockázatnak a lehetséges elhárításáról írok majd a továbbfejlesztési lehetőségeknél.

### **3.9 Load balance Eureka és Kubernetes esetében**

Az alkalmazásomat mivel nagy mennyiségű felhasználó kiszolgálására szerettem volna felkészíteni, így egyes mikroszolgáltatásokból több példánynak kell futnia. Azonban, hogy ez működhessen, valahogy el kell döntenie azt, hogy éppen melyik service szolgálja ki az éppen beérkező kérést. Amíg fejlesztettem az alkalmazást, ennek a következő volt a folyamata. Először egy példány futott minden szolgáltatásból, majd, amikor fejleszteni akartam, futtattam még egyet belőlük egy másik porton, majd amelyiket akartam használni, annak a portján hívtam meg a HTTP kérést. Ebből viszont világosan látszik, hogy nagy alkalmazásoknál nem működik, hiszen nem fogja tudni minden felhasználó, hogy éppen melyik szolgáltatás van kevésbé terhelve, hova kellene küldeniük a kérést. Ezért is kezdtem el használni az Eureka szerveret, amire be tudnak regisztrálni a szolgáltatások, majd az nyomon tudja követni, hogy egyes service-ekből hány példány és melyik portokon fut, valamint a legfontosabb funkciója a service discovery, aminek segítségével a különböző service-ek elérhetik egymást, mivel azok különböző példányait összefogja egy DNS név alá az Eureka, így ha pl a post-service használni akarja a user-service-t, akkor nem kell ismernie az összes példányát és portját annak, csak a user-service néven hívhatja a szolgáltatást, innentől pedig az Eureka eldönti, hogy melyik instance-nek továbbítsa a kérést. Ezt az alábbi ábra szemlélteti.





HOME    LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2022-12-01T15:10:33+0000
Data center	default	Uptime	00:20
		Lease expiration enabled	true
		Renews threshold	22
		Renews (last min)	44

DS Replicas

[localhost](#)

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - <a href="#">2b5af5c59f5a:api-gateway-8295</a>
AUTH-SERVICE	n/a (2)	(2)	UP (2) - <a href="#">917067790f0c:auth-service-8082</a> , <a href="#">d5b712c77832:auth-service-8082</a>
FRIEND-SERVICE	n/a (1)	(1)	UP (1) - <a href="#">3a80dea4dec6:friend-service-8003</a>
INTERACTION-SERVICE	n/a (1)	(1)	UP (1) - <a href="#">bc9ec184f39c:interaction-service-8002</a>
NOTIFICATION-SERVICE	n/a (1)	(1)	UP (1) - <a href="#">d29e9e4d705:notification-service-8101</a>
POST-SERVICE	n/a (3)	(3)	UP (3) - <a href="#">3acbbd53bd94:post-service-8001</a> , <a href="#">e2966adda02:post-service-8001</a> , <a href="#">cfe30bee031c:post-service-8001</a>
TIME-SERVICE	n/a (1)	(1)	UP (1) - <a href="#">35648f83bb3a:time-service-8081</a>
USER-SERVICE	n/a (2)	(2)	UP (2) - <a href="#">6cab1f649627:user-service-8000</a> , <a href="#">3751bb561e1d:user-service-8000</a>

### 3.9:1 Az Eureka naming server kezelői felülete

A képen látható, hogy az authorization-service-ből és a user-service-ből futtattam 2 példányt, a post-service-ből pedig 3-at, amiről az Eureka kezelői felületén kaptam visszajelzést. Ahogy látszik, olyan információkat kaphatunk a szolgáltatásokról, mint hogy hány elérhető példány fut, ezekből melyiknek mi az egyedi neve, belül milyen porton érhető el, de még sok mást is lehet látni, amik nincsenek rajta a képen, mint például a futtató környezet erőforrásairól adatokat stb.

Onnantól kezdve, hogy néhány service-ből elindítottam több példányt, az Eureka ezeket számontartotta, de mégis kellett találni egy módot arra, hogy a kérések beérkezésekor el legyenek egyenletesen osztva azok minden service között. Erre a problémára nyújt megoldást a Spring Cloud Load Balancer, ami a Spring Gateway csomagban elérhető és az Eureka-val együttműködve, automatikusan irányíthatjuk az éppen aktuálisan legkevésbé terhelt szolgáltatásra a kérést, ezzel megspórolva a többi túlterhelését. Arról, hogy ezt a megoldást hogyan használtam a gyakorlatban, az API gateway-ről szóló részben található kódrészletből látható.

Ugyan ez a megoldás nagyon jól elosztja a terhelést az éppen futó, de statikus mennyiségű szolgáltatás között, azonban, ha egy szolgáltatást nem használunk ki eléggé, felesleges a terhelést elosztani több példányra között, hiszen az is elképzelhető, hogy egyszerre csak egy szolgáltatás fogja kiszolgálni a beérkező néhány kérést, az összes többi pedig tétlen állapotban lesz, valamint az is hátránya, hogy valakinek figyelnie kell az szolgáltatások aktuális terhelését és annak megfelelően kézzel skáláznia az erőforrásokat. Ez érezhetően egy meglehetősen macerás megoldás, viszont orvosolható

azzal, ha nem manuálisan, hanem dinamikusan skálázzuk az alkalmazásunkat. Ehhez a megoldáshoz érdemes egy erre kifejlesztett eszközt használni, a Kuberneteset.

Az automatikus skálázásra váltás azt eredményezi, hogy az Eureka szolgáltatására már nincs szükség, mivel amikor egy alkalmazást kirakunk egy Kubernetes cluster futtatására alkalmas felhőszolgáltatás-ra, akkor az adott szolgáltató a legtöbb esetben lehetőséget ad a saját load balancer megoldásának használatára, ezzel megspórolva rengeteg időt a fejlesztőknek, valamint a szolgáltatás felfedezés (service discovery) is más módon zajlik K8s környezetben, mint Eureka esetén.

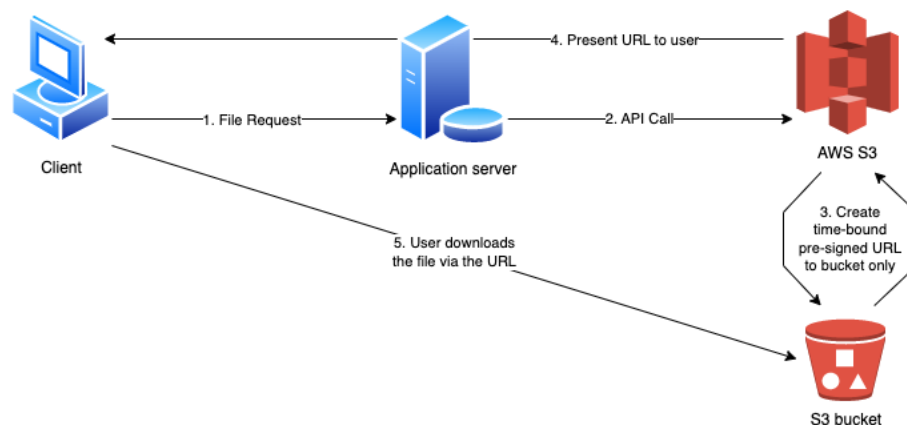
Ahhoz, hogy a Kubernetes-en lévő pod-ok elérésének módjait és a Kubernetes load balancer-ének működését bemutassam, ismertetem a Service-ek fontosabb típusait. Az első a ClusterIP, aminek segítségével a Service-t el tudom érni a Cluster saját hálózatán belül. Ezt olyankor érdemes használni, ha a külvilág felé nem szeretném elérhetővé tenni a Service által kezelt pod-okat, de azt szeretném, hogy a többi Service belül kommunikálhasson vele. Egy másik gyakran használt, a NodePort. Ez a Service csinál egy iptables szabályt a worker node-okon, ami miatt a szolgáltatásom portja elérhetővé válik egy magasabb, 30000-32767 közé eső porton. Ezt a Service típust olyankor érdemes használni, ha kívülről akarunk hozzáférni a Cluster belsejében futó pod-okhoz. Úgy tudjuk ezt megtenni, hogy a worker publikus IP címén az elérhetővé tett porton hajtjuk végre a hívást. A következő, LoadBalancer típus is lényegében így működik, azzal a különbséggel, hogy ebben az esetben az összes worker node-on elérhetővé teszi az adott NodePort-os portot, majd a saját megoldásával végzi el a load balance-ot úgy, hogy közben biztosítja a redundanciát a pod-ok között. Amikor beérkezik a master node-ra egy kérés, olyankor az a worker node-oknak kiosztja a feladatokat egy iptables alapján, azok pedig amint tudják, elvégzik a nekik kiosztott feladatokat. Ha valamelyik véletlenül elérhetatlenné válik, akkor sem veszik el a feladata, így amint visszacsatlakozik, újra el tudja kezdeni ezeknek az ellátását.

A Kubernetesnek köszönhetően, ha egy Service megcímez egy másikat olyan módon, hogy a konfigurációban feltüntetett nevét használja (például user-service), akkor az egy egyszerű névfeloldás után elkerül a megcélzott Service-hez, majd az továbbítja a load balance alapján éppen választott pod felé a kérést, ott pedig a megfelelő porton elért konténerben futó microservice kiszolgálja azt.

### 3.10 Képek letöltése AWS S3 tárhelyről

A képek letöltésével kapcsolatban először egy olyan megoldást alkalmaztam, ami az API hívás alkalmával letölti a kért képet a szerveren keresztül az AWS S3 bucket-ből, viszont ez két problémát okozott. Az egyik az, hogy mobil oldalon nem akarom a felhasználókat rengeteg fájl letöltésére kényszeríteni, vagy legalábbis nem olyan módon, mint ahogy ez ebben a formában történne, hogy a mobil letölti a képet a belső tárhelyre és utána azt megjeleníti, hanem valamilyen caching segítségével. A másik ok pedig az, hogy az összes képnek keresztül kellene mennie a szervereimen, ami nagyon nagy sávszélesség terhelést okozna, jobb lenne egyből az AWS S3 szervereire küldeni a kérést kliens oldalról is. Ezzel annyi a probléma, hogy a képeim elérése titkosítva van, így, ha csak megnyitom az URL-jüket, egy Access Denied oldalra kerülök. Azt a megoldást találtam erre, hogy készítek olyan előre, a szerver által aláírt URL-eket, amik egy ideig érvényesek csak, utána lejárnak.

Így most a szerverre csak egy kérés fog menni, amire az kér egy előre aláírt URL-t a szerveren tárolt AWS-es Access key segítségével, az S3-ról pedig visszakapja és továbbítja a kliensnek, aki ennek segítségével érheti el a bucket-ből a kívánt képet egy rövid ideig. Ennek előnye, hogy most egy stringet ad vissza a szerver egy több megabájtos kép helyett.



**3.10:1 AWS S3-ról kép letöltés előre aláírt URL használatával**

Ezt a megoldást nagyon jól mutatja be a fenti ábra, azonban az én megvalósításomban nem a lehető leghatékonyabb módon alkalmazom ezt a metódust. Az ideális változata a képek ilyen fajta letöltésének az lenne, hogy a lejáratí időt nagyon kevés, nagyjából pár másodpercre állítom be, ami ahhoz elegendő, hogy még egy lassú kliens is letölthesse addig.

Ez azért kell, hogy így legyen, mivel a kliens onnantól kezdve, hogy letöltötte a képet, el tudja cache-elni, ami pedig megkíméli a szerveret is és az S3-at is bármiféle felesleges kérésektől.

Ebben a cache-elési megoldásban a Kotlin-ban elérhető Picasso, vagy Glide könyvtár segítenek. Ezek a könyvtárak úgy működnek, hogy a betölteni kívánt URL szerint döntenek el, hogy a képet már betöltötték-e és ha még nem, akkor letöltik azt, ha viszont igen, akkor az újbóli letöltés helyett betöltik memóriából a kívánt képet.

Ezt a fajta cache-elést az én alkalmazásomban sajnos csak körülményesen lehetett alkalmazni, mivel a képeimnek csak a nevét tárolom az adatbázisokban, az elérési útvonalat pedig az imént leírt előre aláírt URI módszerével generáltatom, amik viszont minden alkalommal változnak. Ami nem változik az a képek neve, mivel egyedi módon neveztem el őket. Ezért is csináltam egy saját cache-t a mobil kliensen, ami a képek neveihez tárolja az első URL-t egy Room adatbázisban, így, ha megtalálja a kép nevét a saját cache-ben a kliens, akkor az ahhoz tartozó URI-val, a Picasso által már elmentett képet tölti be, nem generáltat újat. Ez megoldja azt a problémát, hogy a szervernek folyamatosan új képeket kelljen generálnia és Picasso minden alkalommal cache-eljen egy újabb képet, ezzel csökkentve az alkalmazás hatékonyságát.

Azonban ezzel is volt egy probléma, hogy az app újraindítása után a Picasso cache-e kitörlődik, így az elmentett képeim már nem lesznek betölthetőek a segítségével, mivel az URL-ek lejártak, a Picasso pedig le akarja őket újra tölteni az internetről, mert nem találja ezeket, mint kulcsokat a cache-ben. Erre azt a megoldást találtam ki, hogy hoztam egy kompromisszumot és felemeltem az URL-ek lejáratát idejét két napra, mivel ezek a képek úgyis a nap végéig, plusz a következő napi BeFake ideig lesznek kint az oldalon, ami maximum két nap lehet, utána újat fognak posztolni a felhasználók. Arról, hogy ennél mi lenne még ideálisabb, a továbbfejlesztési lehetőségeknél írok majd.

### **3.11 Konténerizálás Docker-rel**

A fejlesztés során több környezetben is kellett futnia az alkalmazásomnak. Eleinte, amíg még kezdetlegesek voltak a szolgáltatások, lokálisan futtattam azokat, majd áttértem arra, hogy Docker konténerekből fussanak. Ezt eleinte kézzel kellett csinálnom, ami abból állt, hogy futtattam egy `maven clean install` parancsot a munkakörnyezetben, majd egy Dockerfile-t készítettem, ami az alkalmazást „becsomagolta” és készített belőle egy Docker image-et. Az alábbi kód egy példa a Dockerfile-jaimra.

```
#Build stage
FROM maven:3.6.3-openjdk-17-slim AS build
COPY src /home/app/src
COPY pom.xml /home/app
RUN mvn -f /home/app/pom.xml clean package

#Package stage
FROM openjdk:17-alpine
MAINTAINER pintertamas
COPY target/api-gateway-latest.jar api-gateway-latest.jar
EXPOSE 8765
ENTRYPOINT ["java", "-jar", "/api-gateway-latest.jar"]
```

A kódból az látható, hogy két fázison megy keresztül az image-ek készítése. Az elsőn az openjdk 17-es verziójával futó maven-t használom a projekt build-eléséhez, a másodikban pedig készítek egy Docker image-et az előző fázisban elkészített build-ből, ami egy .jar kiterjesztésű fájlt fog futtathatóvá tenni az image-n keresztül a felhasználók számára. Ahhoz, hogy az alkalmazás portjait el lehessen érni majd az image-en keresztül, meg kell határozni azokat az EXPOSE kulcsszó után, esetemben ez a 8765-ös port, amin keresztül a gateway a bejövő API hívásokat irányítja. Az image-eket a Dockerfile mappájában futtatott

```
docker build --tag <service neve:verziószám> .
```

paranccsal lehet létrehozni.

Ahhoz, hogy felgyorsítsam ezt és ne kelljen mikroszolgáltatásonként megcsinálni ezt a folyamatot, készítettem egy egyszerű batch fájlt, ami automatizálja ezt a folyamatot és miután letörölt minden nem használt image-et és konténert, rekreálja az összeset a fent említett módon.

Miután kész lettek az image-ek, létrehoztam egy docker-compose.yaml fájlt, ami arra való, hogy a létrejött image-ekből személyre szabott konfigurációkkal konténereket hozzon létre, amikben futhatnak a különböző image-ekbe csomagolt programjaim. Ezt a következő kódrészletben mutatom be.

```

version: '3.8'
services:
  (...)
    user-service:
      image: pintertamas/user-service:latest
      restart: always
      ports:
        - "8000:8000"
      networks:
        - befake-network
      depends_on:
        - naming-server
        - api-gateway
        - kafka
      environment:
        EUREKA.CLIENT.SERVICEURL.DEFAULTZONE:http://naming-
server:8761/eureka
        SPRING.ZIPKIN.BASEURL:http://zipkin-server:9411
        SPRING.DATASOURCE.URL:jdbc:postgresql://host.docker.interna
l:5432/user-service
        SPRING.DATASOURCE.USERNAME: postgres
        SPRING.DATASOURCE.PASSWORD: postgres
        KAFKA.URL: kafka:9092
  (...)

```

A konfiguráció azzal kezdődik, hogy definiáljuk a konténer nevét, hogy milyen image alapján készüljön el és az újraindítási szabályát, ami esetemben annyit csinál, hogy ha a konténer leállna, azonnal megpróbálja újraindítani, így mihamarabb helyre áll a stabil működés. Ezek után megnyitok egy portot a külvilág felé olyan módon, hogy a 8000-es porton elérhetővé teszem az alkalmazás 8000-es portját, azaz, ha a konténeremen hívom ezt a portot, akkor az olyan lesz, mintha a benne lévő image-be csomagolt mikroszolgáltatás 8000-es portján tettem volna meg ezt. Ez lehet, hogy triviálisnak hangzik, de lehet eltérő portokon is megnyitni belső portokat a külvilág felé, ami azért lehet hasznos, mivel, ha két különböző forrásból származó konténert használok, de mindkettő ugyan azon a porton működik, akkor a gazda gépen összeakadnának ezek, így legalább az egyiket másik porton kell megnyitni. Ezek után beleteszem az image-et egy hálózatba (network), amit a compose fájlban már definiáltam, ennek segítségével teljes izolációban futhatnak a konténereim. A `depends_on` tag alá azokat a service-eket sorolom fel, amiknek futniuk kell ahhoz, hogy a konténerünk teljes funkcionalitását elérhessük. A példában bemutatott esetben ahhoz, hogy a user-service futhasson, kell a naming-server, hiszen oda fogja magát beregisztrálni a szolgáltatás, az api-gateway, mivel azon keresztül fogják a felhasználók elérni ennek a szolgáltatásnak a funkcióit, majd a Kafka is, mivel ennek segítségével küld értesítést a felhasználóknak a regisztrációjukról és ha a Kafka újraindul valami miatt, akkor a user-service-nek is újra kell indulnia, hogy újra beregisztrálja magát, mint producer.

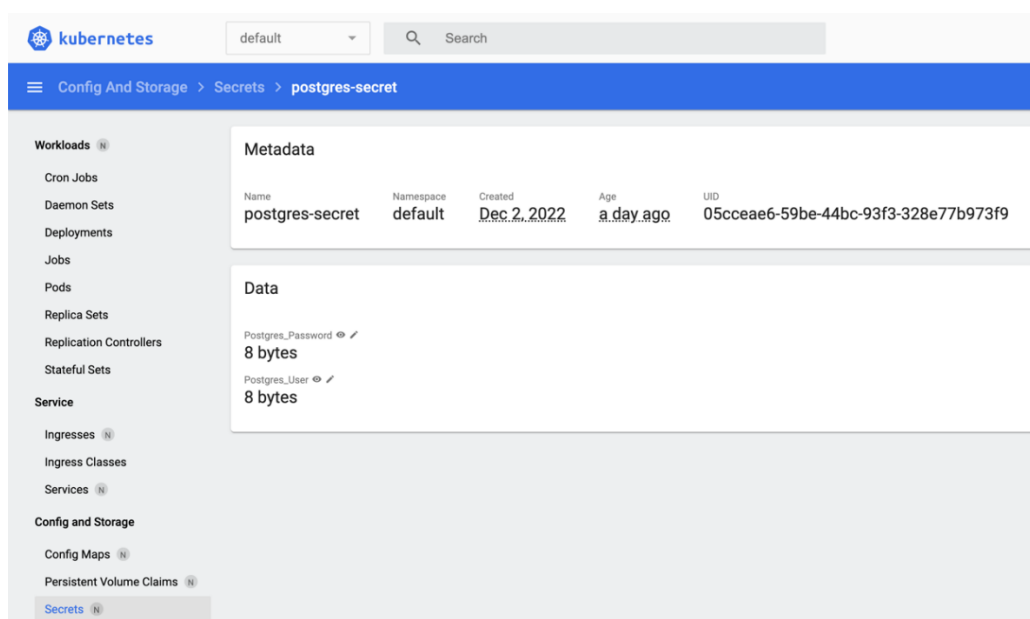
Az environment tag alatti értékek főként arra szolgálnak, hogy az eredetileg az application.properties-ben definiált beállításokat felülírjam, a Docker környezetben is működő értékekkel, így ha lokálisan futtatom, akkor nincs akadályozva az alkalmazás működése. Ez abban az esetben fontos például, ha nem a saját számítógépemnek, hanem a Dockernek a lokális tárhelyén futó adatbázisát akarom használni, vagy egy másikat, ami egy fizikailag teljesen másik helyen helyezkedik el. Ha például az ElephantSQL szerverén futtatott PostgreSQL adatbázisban akarom tárolni az adatokat, akkor így írnám át a konfigurációt:

```
SPRING.DATASOURCE.URL:  
jdbc:postgresql://lucky.db.elephantsql.com:5432/<username>?user=<username>  
&password=<password>&sslmode=require
```

Ennek köszönhetően a Dockert futtatva máris egy egészen más adatbázishoz kötöttem hozzá az alkalmazást, egy sor egyszerű megváltoztatásával. Ahhoz, hogy Kubernetes-en is könnyen át tudjam állítani a konfigurációt, a következő módon hoztam létre a konfigurációt az application.properties fájlokban, így a Kubernetes-ben lévő környezeti változókkal tudtam beállítani az adatbázis kapcsolatokat.

```
spring.datasource.url=\  
jdbc:postgresql://\  
${PG_NAME:localhost}:${PG_PORT:5432}/${DB_NAME:user-service}  
spring.datasource.username=${POSTGRES_USER:postgres}  
spring.datasource.password=${POSTGRES_PASSWORD:postgres}
```

Ahhoz, hogy a lehető legbiztonságosabb módon kezeljem az adatbázisok felhasználóit és jelszavait, Kubernetes Secret-ekbe tettem azokat base64 encode-olva.



### 3.11:1 Kubernetes Secret kezelés

## 4 Megvalósítás és üzemeltetés

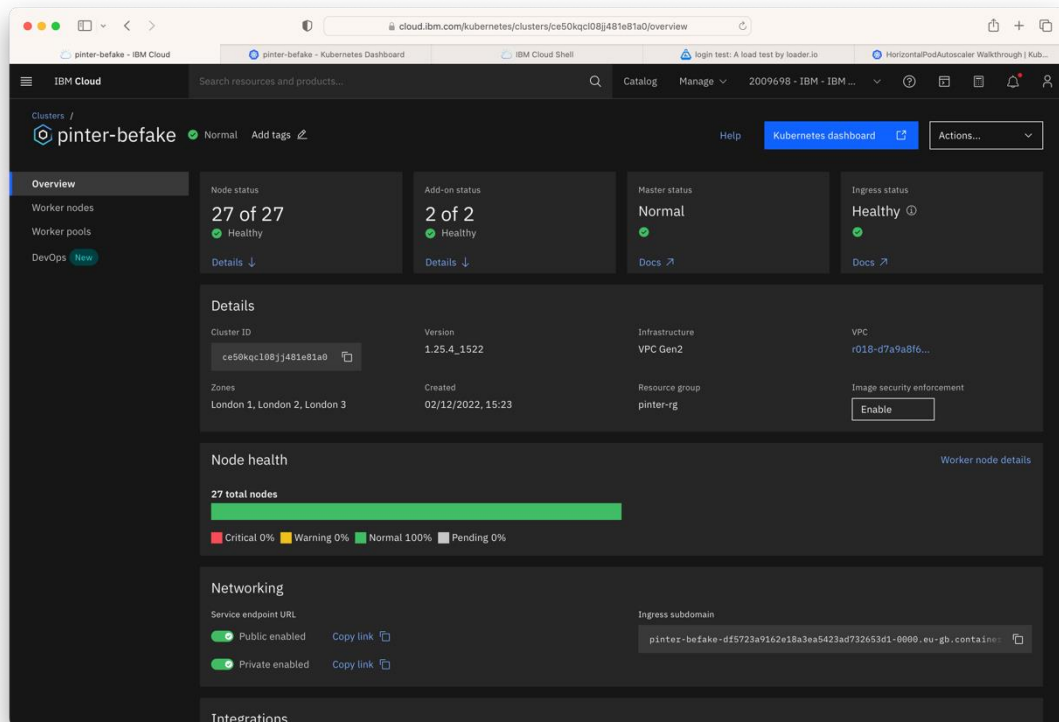
A következőkben bemutatom azokat az érdekesebb részleteket, amik segítségével meg tudtam valósítani és üzemeltetni ezt a projektet úgy, hogy az a lehető leginkább hasonlít egy éles környezetben is helyét álló alkalmazáshoz.

### 4.1 Hostolás az IBM Cloud-ra

Az elkészült alkalmazást először lokális környezetben, minikube [17] segítségével futtattam Kubernetes-en, viszont hamar megláttam a hátrányait ennek, ahogy arról már korábban is írtam, ilyen esetben saját módon kell megvalósítani a terhelés elosztást a szolgáltatások között, ami nem egy kis falat lett volna számomra, ezért szerettem volna elérhetővé tenni valamelyik felhőszolgáltatáson is, hogy használhassam a választott cloud provider load balancer-ét és kipróbálhassam, hogyan működik éles környezetben egy projekt megjelenítése.

A probléma ezzel az, hogy ennek a kivitelezése közel sem olcsó és minden felhőszolgáltató más funkciókat kínál az ingyenes csomagjain belül, nekem viszont ezek nem lettek volna elegek ahhoz, hogy értelmes keretek között futtathassam a projektemet. Volt, amelyik csak pár pod-ot engedélyezett ingyen létrehozni és kezelni, volt amelyiknek nem is tartozott az ingyenes szolgáltatásai közé a K8s cluster használata, valamelyik pedig csak akkor engedte használni a load balancer-t, ha csomagot frissíték. Szerencsére az IBM-en keresztül, tanulási célokra kaptam egy cluster-t az IBM Cloudban, amin létre lett hozva a projektem számára jó pár worker node, amiken futtatni tudtam az alkalmazást, miközben tanultam a Kubernetes működéséről és használatáról. A szolgáltatás kezelői felületét a 4.1:1 ábra mutatja be. Látható rajta, hogy a node-jaim milyen státuszban vannak és a korábban már említett helth check eredménye is, azaz hogy „egészségesek”-e a node-ok. Ha egy ennél jóval nagyobb projektet kellene kezelnem, akkor nagyon jól jönne az a funkciója a cloud-nak, hogy különféle Add-on-okat lehet a projekthez adni, mint például olyan kiegészítőket, amik tesztelik és részletes diagnózist adnak a cluster különböző komponenseiről, vagy az általam is használt Cluster Autoscaler, ami a cluster automatikus skálázásáért, azaz node-ok létrehozásáért és törléséért felel. Az Integrations fül alatt érhetőek el olyan funkciók, amik sokkal részletesebb monitorozó és logoló lehetőségeket biztosítanak a felhasználók számára.





#### 4.1:1 Az IBM Cloud kezelői felülete

A következőkben meg fogom mutatni a dashboard-ot és a cloud shell-t használva, hogy a clusterem milyen elemekből áll és hogyan néz ki az a konfiguráció, aminek a felépítéséről korábban már írtam.

Alapvetően a cluster konfigurálásához yaml fájlokat kell használni, amikben meghatározom, hogy adott komponensek milyen beállításokkal jöjjenek létre. Ez azért előnyös, mert ha később másik felhő környezetben akarnám futtatni az alkalmazást, akkor a már meglévő konfigurációs fájlok segítségével azonnal meg tudnám ezt tenni, hiszen a parancsok mindenhol ugyan azok és a fájlok értelmezése sem tér el, így a beállítások is ugyan úgy jönnek létre platformtól függetlenül.

A lentebb látható konfigurációban a „kind” tag-et használva azt határozom meg, hogy egy Deployment típusú komponenst hozzon létre. Be kell továbbá állítani a namespace-t, ami különböző Deploymenteket összefogó névtér, ilyen módon is több részre lehet bontani a cluster-einket, valamint, hogy milyen néven jöjjön létre, így lehet majd később rá hivatkozni, ha például expose-olni szeretném Service-ként. Az első „spec” beállítástól kezdve a pod-ok létrehozásának módját lehet beállítani, jelen esetben azt adtam meg, hogy a pod-ok legalább 25%-a legyen minden esetben elérhető, így, ha újraindítom a Deployment-et, akkor is meghagyja az addig használt pod-ok negyedét az

előző beállításokkal, ezzel megtartva a szolgáltatás elérhetőségét. A „template” tag-et követően a pod-okban futó konténerek beállításait adhatom meg olyan módon, hogy melyik image fusson bennük, milyen környezeti változókat adjon át nekik és egy nagyon fontos beállítás is itt adható meg, az imagePullPolicy, ami azt határozza meg, hogy ha egy Deployment-et újraindítunk, akkor milyen szabály szerint használja az image-et. Az alapbeállítás az, hogy „ifNotPresent”, ami azt jelenti, hogy a node csak akkor fogja letölteni az adott image-et, ha az még nem létezik rajta, ha viszont már létezik, akkor azt hasznosítja újra. Mivel én általában akkor indítottam újra a Deployment-et, amikor a kódban is változtattam valamit, ezért az „Always” beállításnál maradtam, ami miatt minden esetben letölti az image adott verzióját, ezzel biztosítva, hogy nem egy elavult verzió fog a konténerben futni. Ahogy ez a beállítás is, a restart policy is ismerős lehet a Docker miatt, ugyanis ahogy ott is, itt is arra vonatkozik, hogy az image futásának leállása után hogyan viselkedjen a konténer. Újraindítsa-e azt, vagy hagyja álló állapotban.

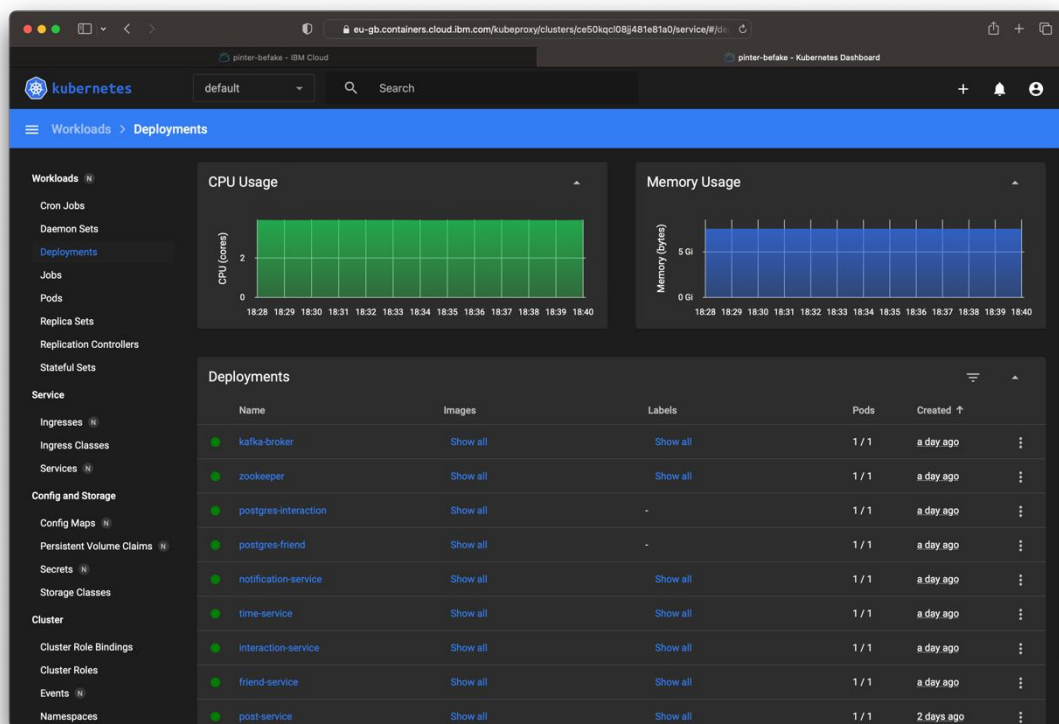
```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: api-gateway
    name: api-gateway
    namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api-gateway
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: api-gateway
    spec:
      containers:
        - env:
            - name: EUREKA_CLIENT_ENABLED
              value: "false"
            - name: SPRING_ZIPKIN_ENABLED
              value: "false"
            - name: DISCOVERY_CLIENT_ENABLED
              value: "true"
          image: pintertamas/api-gateway:SNAPSHOT-0.0.1
          imagePullPolicy: Always
          name: api-gateway
          resources: {}
      restartPolicy: Always
```

Mivel az IBM Cloud is egy menedzselte felhőszolgáltatás, így ehhez a konfigurációhoz még hozzászert generált értékeket, amik a belső működéséhez szükségesek.

A konfigurációt a következő módon hoztam létre és expose-oltam 4.1:2. Lehet fájlból is létrehozni, vagy a dashboard-on egy form-ot kitöltve készíteni, de én inkább létrehoztam egy alapbeállítást és utólag szerkesztettem a dashboard felületén.

```
Tamas_Pinter1@cloudshell:~$ kubectl create deployment load-test-validation-service --image=pintertamas/load-test-validation-service:latest
deployment.apps/load-test-validation-service created
Tamas_Pinter1@cloudshell:~$ kubectl expose deployment load-test-validation-service --port=8888
service/load-test-validation-service exposed
Tamas_Pinter1@cloudshell:~$
```

#### 4.1:2 Deployment létrehozása és expose-olása

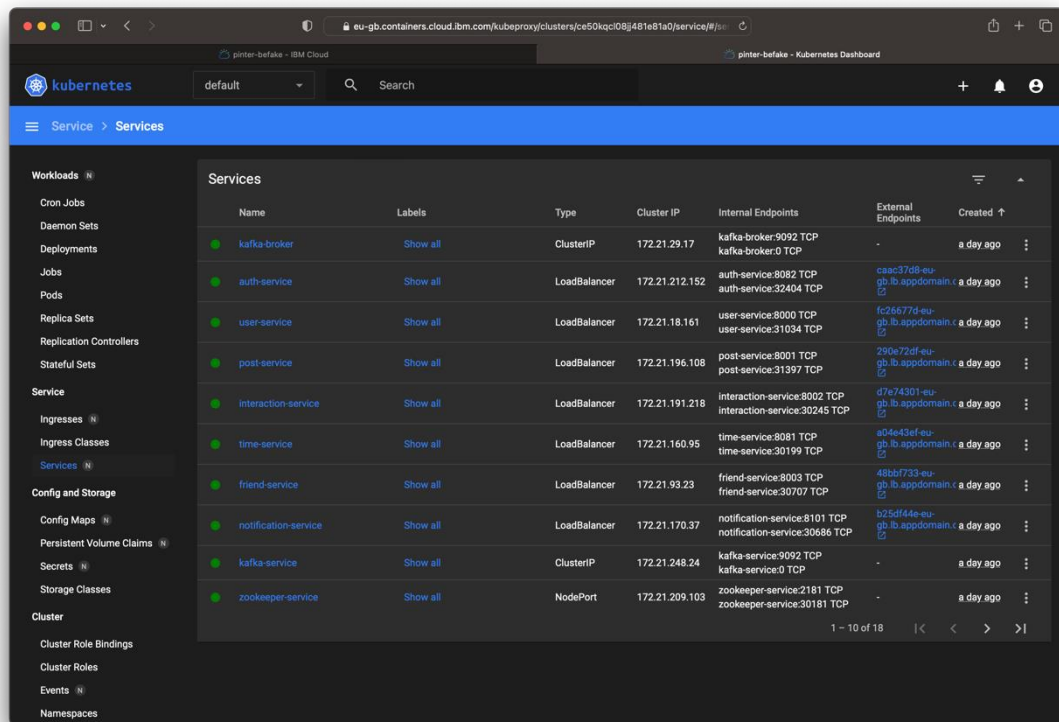


#### 4.1:3 Kubernetes deployment-ek futás közben

A Deployment-ekből Service-eket kellett csinálni ahhoz, hogy kapjanak egy saját IP címet a clusteren belül (ennek a típusa a korábban már említett ClusterIP, NodePort és LoadBalancer közül az egyik volt) és elérjék egymást ezen keresztül. Ahhoz, hogy ezt megtegyem, a cloud shell-ben kiadott

```
kubectl expose deployment <deployment neve> --port=<port> --type=<service típusa>
```

paranccsal tettem meg. Az itt megadott <deployment neve> paraméter lett az, amit hivatkozva a FeignClientből kommunikálhattak egymással.



#### 4.1:4 A futó service-ek egy része

A dashboardon látható minden service neve, a típusukkal. Az API gateway-en kívül nem volt más service, aminek LoadBalancer típusúnak kellett volna lennie, viszont a könnyen tesztelhetőség érdekében mégis ezzel a típussal hoztam őket létre, így mindegyik microservice kapott saját External Endpointot. Ez azt eredményezte, hogy kívülről elértem őket a megjelölt URL-en keresztül, ha a Postman segítségével tesztelni szerettem volna valamit, ezen felül pedig az API gateway-ről jövő kérések között meg tudták oldani a terhelés elosztást. Látható még az is, hogy milyen belső IP címet kaptak a clustertől a kommunikációra, az Internal Endpoint pedig azt mutatja, hogy az adott DNS-en elérhető service adott portját melyik külső port-on érhetjük el. Itt fontos megjegyezni, hogy a ClusterIP-nál a 0 port van megjelölve, mivel az ahogy már írtam, egy belső IP, kívülről nem érjük el, így azt ugyan látjuk, hogy belül melyik port van nyitva, de kívülről erre nem tudunk rácsatlakozni. Az is látszik, amiről szintén írtam korábban, hogy a LoadBalancer típusú service-ek valójában NodePort-tal vannak megvalósítva, ugyanis a port, amin kívülről elérhetőek 30000-32767 közé esik.

Ahhoz, hogy az API hívásokat ne egy bonyolult és könnyen elteveszthető URL-en kelljen hívni, a saját, tamaspinter.com domain-emen beállítottam egy CNAME értéket,

aminek köszönhetően, ha a saját domain-emben végzem az API hívásokat, akkor az elfedi a felhő URL-jét és ugyan azt az eredményt kapom, mintha a hívást a 3c1157f7-eu-gb.lb.appdomain.cloud címen tettem volna ezt meg.

## 4.2 Load test

A load test végrehajtásához a loader.io weboldalt használtam. Ahhoz, hogy kiküszöböljék a termékük rosszra való felhasználását, fel kellett tölteni egy txt fájlt az elérni kívánt oldalra úgy, hogy a domain cím után írva a fájl nevét azt letölthesse, ezzel ellenőrizve, hogy saját weboldalt akarok tesztelni. Ahhoz, hogy ezt megcsináljam, létrehoztam egy új service-t, load-test-validation-service néven és az api-gateway-ben a routing-ot frissítettem, valamint a kért endpointot elérhetővé tettem autentikáció nélküli elérésre is. Ez a service annyit csinál, hogy ha meghívom a

`www.tamaspinter.com/ loaderio-6d8749c36ee8505f56ab515308006e16`

URL-t, akkor visszatér a fájlban található szöveggel, így a loader.io már engedi is használni a szolgáltatását.

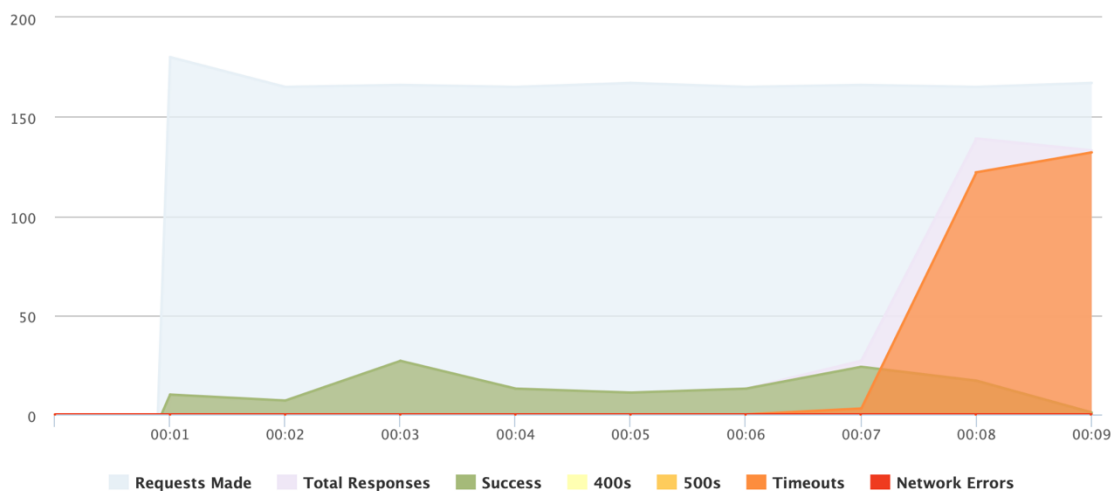
A terhelés teszteléshez a bejelentkezést használtam. Azért ezt választottam, mivel itt összesen 3 általam írt mikroszolgáltatást és egy Kubernetes service-ben lévő postgres adatbázist használok. A kérés először elmegy az api-gateway-hez, ami továbbirányítja az auth-service-nek azt, ami pedig egy belső hívás során kommunikál a user-service-szel, ami az adatbázisból lekéri a felhasználó adatait, hogy összevesse a kapott értékekkel.

Futtattam egy tesztet úgy, hogy még nem voltak skálázva a service-ek, hogy látható legyen, egy pod mennyi kérést képes kiszolgálni magától. Első próbálkozásra 5000 felhasználót küldtem rá az alkalmazásra 30 másodperc alatt, a grafikonok pedig nagyon szépen mutatják, hogy hogy nőtt meg a válaszidő ezekre a kérésekre. Mivel nem tudták a K8s service-ek átadni a feladatot másik pod-oknak, így az egyetlen működő pod egyre jobban le lett terhelve, emiatt a felhasználók feltorlódtak, a válaszidő pedig lineárisan (azaz a felhasználók érkezésének függvényében) nőtt. Ezt a 4.2:1 ábrán lehet látni.



#### 4.2:1 A kérések és a választidő grafikonja

Egy másik ábráról 4.2:2 azt olvashatjuk le, hogy a teszt során érkező felhasználókat a szolgáltatás egy idő után nem tudta kiszolgálni bizonyos időn belül, így azok timeout-oltak a 7. másodperctől kezdve, ami azt eredményezte, hogy a teszt automatikusan leállt, ugyanis azt állítottam be, hogy 50%-nál nagyobb hibaaráta esetén fejezze be a futását.



#### 4.2:2 A teszt részletei

Ahhoz, hogy az automatikus skálázás működjön, ki kellett egészítenem a Deployment fájlokat az erőforrásokra vonatkozó limitációkkal, hogy a horizontális autoskálázója a felhőszolgáltatásnak működni tudjon. Ehhez a „resources” tag-hez a következő kódrészletben látható sorokat írtam, így a cloud már tudja, hogy mikor kell a pod-ok számát növelni.

```
resources:
  limits:
    cpu: 500m
    memory: 512Mi
  requests:
    cpu: 500m
    memory: 512Mi
```

Ezeknek a változtatásoknak a végrehajtása után, a cloud shell-ben kiadott

```
kubectl autoscale deployment <deployment neve> --cpu-percent=50
--min=<minimum replikák száma> --max=<maximum replikák száma>
```

paranccsal vittem végbe a pod-ok automatikus skálázódását a Kubernetes HorizontalPodAutoscaler-ét használva [18], így, ha nagyobb terhelés érte a szervert, akkor a K8s létrehozott újabb pod-okat, amikre utána a service el tudta osztani a bejövő kéréseket. Mivel elsőre nem tudtam, hogy milyen értékekkel lenne érdemes dolgoznom, ezért néha változtatni kellett a metrikán, amit a következő paranccsal oldottam meg.

```
kubectl -n default patch hpa/<hpa neve>
--patch '{"spec":{"<változtatni kívánt kulcs>":<az új érték>}}'
```

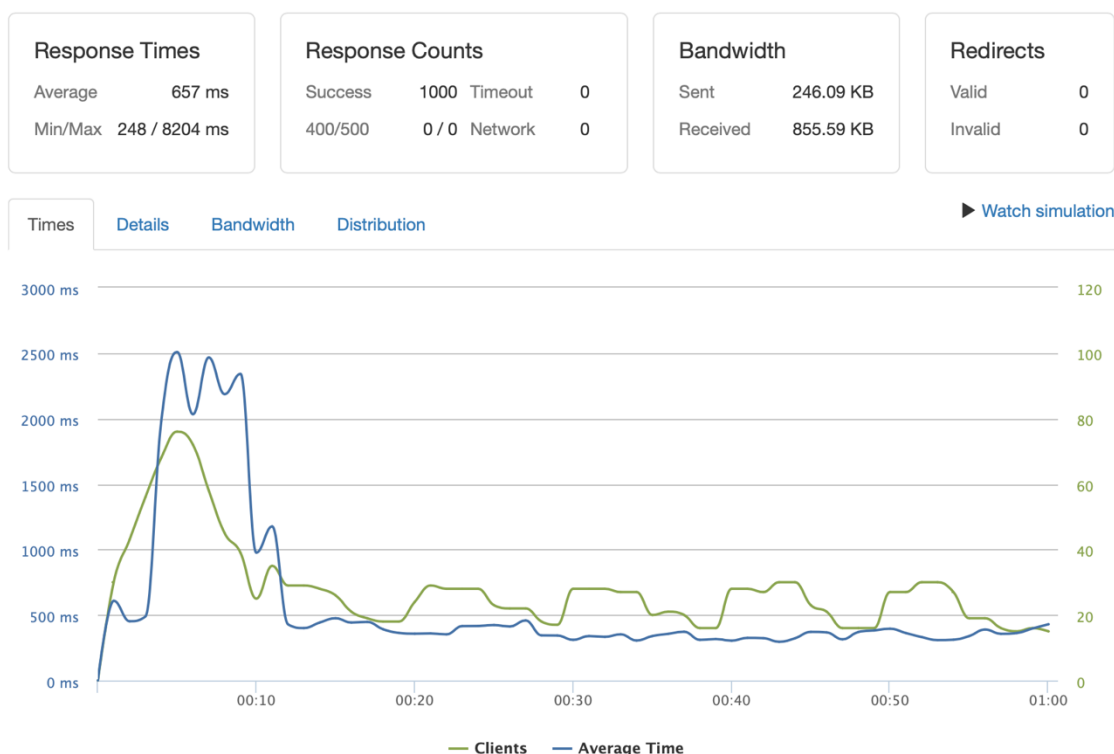
A célom a terhelés tesztel az volt, hogy a loader.io ingyenesen elérhető legnagyobb terhelését gond mentesen rá tudjam küldeni az alkalmazásra, ami egy perc alatt 10000 felhasználót jelent, így a pod-ok skálázásán kívül a node-ok számát is növelnem kellett, mivel nagyjából a percenkénti 2500 kérésnél már nagyon sok hiba keletkezett, ugyanis elfogytak a munkaképes node-jaim. Ezt egészen egyszerűen a node-jaim növelésével oldottam meg, majd megpróbáltam ismét a tesztek futtatását. A probléma ezzel az, hogy a pod-ok száma ugyan elkezd nőni, ezzel a service-ek is elkezdik az újak között szétosztani a terhelést, azonban a pod-okban futó mikroszolgáltatásoknak idő kell az elinduláshoz, emiatt a kérések 500-as hibára futnak. Ahhoz, hogy ezt a problémát elhárítsam, a Spring Boot Actuator-t használtam, ami a service-eken végzett health check alapján egy endpoint-on tér vissza a szerver aktuális állapotával. Ezt először engedélyezni kell a mikroszolgáltatások konfigurációs fájljában, majd a Deployment-ben beállítani, hogy a pod-ok állapotát az Actuator által visszaadott értékek alapján állapítsa meg a Kubernetes. Ezen felül be kell még azt is állítani, hogy mennyi időt adjon a mikroszolgáltatásnak a K8s, mire először ellenőrzi az állapotát, valamint olyan további beállításokat, hogy mennyi időnként ellenőrizze az endpoint válaszait. Ezekre a beállításokra az alábbi kódrészleten lehet látni egy példát.

```

livenessProbe:
  httpGet:
    path: /actuator/health/liveness
    port: 8000
    scheme: HTTP
  initialDelaySeconds: 20
  timeoutSeconds: 10
  periodSeconds: 3
  successThreshold: 1
  failureThreshold: 5
readinessProbe:
  httpGet:
    path: /actuator/health/readiness
    port: 8000
    scheme: HTTP
  initialDelaySeconds: 20
  timeoutSeconds: 10
  periodSeconds: 3
  successThreshold: 1
  failureThreshold: 5

```

Miután ezeket a beállításokat megettem, futtattam egy load testet 1000 felhasználóval, hogy lássam, működik a skálázódás és elindulnak az újonnan létrehozott pod-ok.



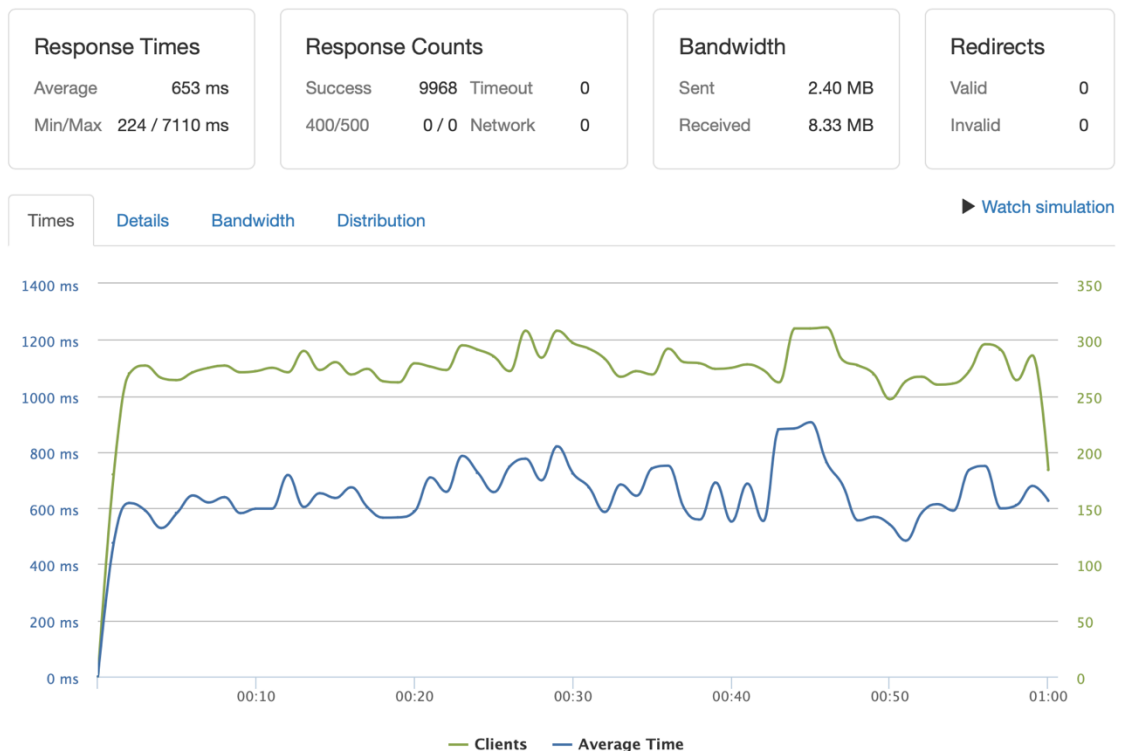
#### 4.2:3 A szerver skálázása 1000 felhasználóval

Az előbbi ábrán nagyon jól látszik, hogy mikor elindítottam a tesztet, az első 10 másodpercen nagyon nagy volt a válaszidő (körülbelül 2.5 másodperc), ami érthető is, mivel ekkor kezdte meg a szerver a skálázást, így várniuk kellett a felhasználóknak a



belépéssel, mire elindultak a szolgáltatások, amik ki tudták őket szolgálni, utána viszont mivel megnövekedett a rendelkezésre álló pod-ok száma, vissza is esett rohamosan a válaszidő átlagosan fél másodpercre.

Miután ezzel végeztem és volt már pár elindított pod, növeltem a teszt méretét és megpróbáltam 10000 felhasználót kihasználni 1 perc alatt. Nagy öröömre ez sikerült is, viszont az alábbi grafikonon már az látszik, hogy a szerver ekkorra már „be volt melegedve”, hiszen nincsen rajta olyan nagy kiugrás, mint előző alkalommal.

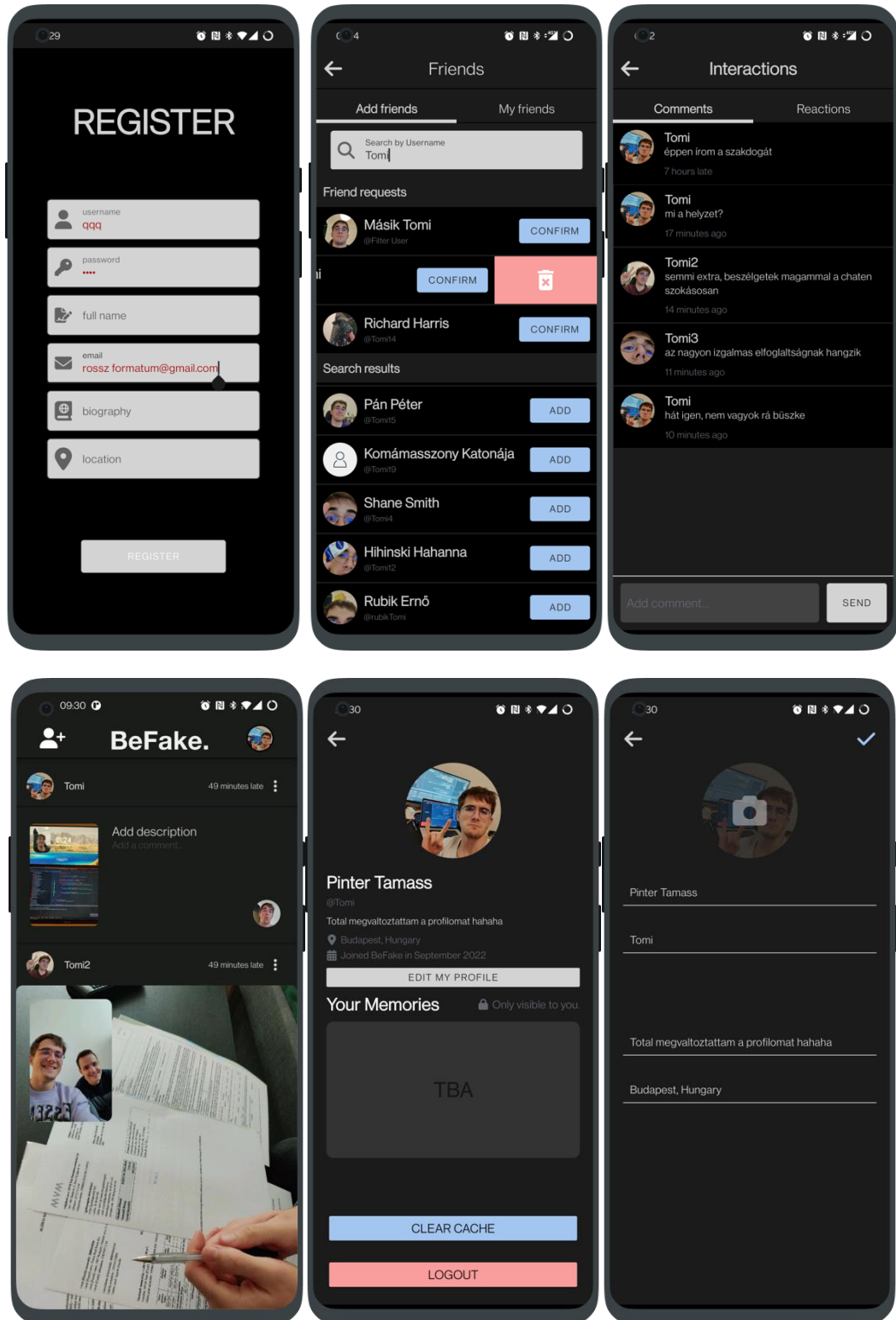


#### 4.2:4 Terhelés teszt 10000 felhasználóval

Ugyan nem mindegyik teszt ment ilyen sikeresen, de megtanultam belőlük azt, hogy akkor a legjobb a terhelése a szervereimnek, ha a felhasználók száma egyenletes ütemben növekszik, ezzel folyamatosan, kisebb lépésekben beindítva a szervert. Ha hirtelen küldöm rá felhasználók tömegét, akkor eleinte sok lesz a timeout, mivel ebben az esetben azonnal túlterhelődik a rendszer, ahelyett, hogy a limit feletti tartalék erőforrás kiszolgálná azokat a felhasználókat, akiket már másik pod-on kellene, amikor meg jön a többi felhasználó, addigra az újonnan elindult pod-ok már működésre készek ezzel elhárítva a túlterhelést.

## 4.3 Képek az elkészült alkalmazásról

Az alkalmazásom mobil kliensének kinézetéről készítettem mockup-okat, hogy könnyebben elképzelhető legyen a használata.



## 5 Az eredmények értékelése

### 5.1 Az elkészült munka értékelése

A témám kidolgozása közben úgy gondolom, hogy nagyon hasznos tudáshoz jutottam, hiszen az iparban is nagyon keresett azoknak a fejlesztőknek a tapasztalata, akik hasonló architektúrák tervezésén és karbantartásán dolgoznak. Nagyon jó volt látni a végére, hogy már képes vagyok olyan alkalmazások fejlesztésére, amik terhelhetősége csupán a rendelkezésre álló hardvertől függ. Remélem még sok helyen lesz lehetőségem alkalmazni fogom ezt a tudásomat és hasznomra válik, hogy ennél a témánál döntöttem félévvel ezelőtt.

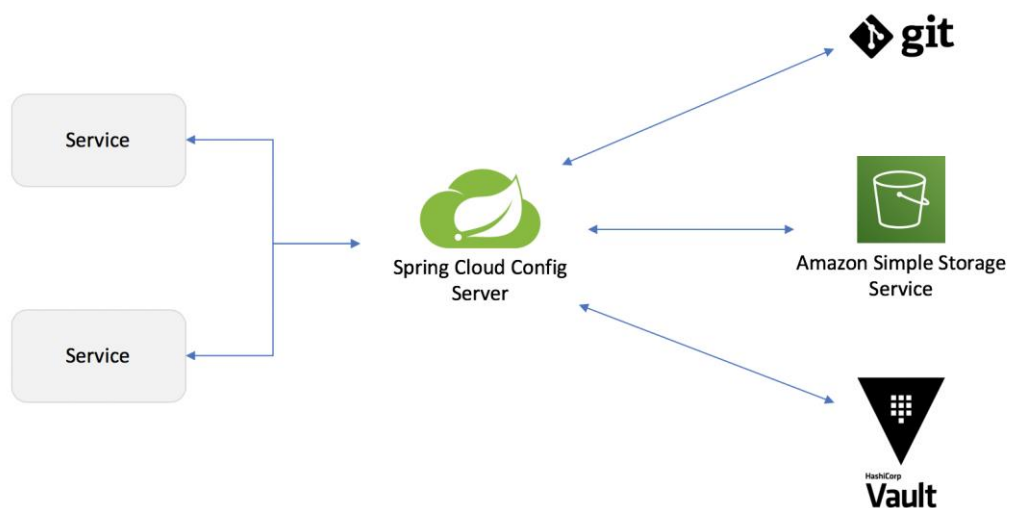
A kedvenc részem az volt, mikor a Kubernetes konfigurációk értékeit végre eltaláltam jól, ennek köszönhetően a terhelés tesztek futtatása sikeres lett és láttam, ahogy fut fel a grafikonon a kiszolgált felhasználók száma anélkül, hogy bármennyi timeout, vagy hiba keletkezett volna. Most már jobban átlátom, hogy milyen mennyiségű munka van azok mögött az alkalmazások mögött, amiknek hatalmas mennyiségű felhasználót kell kiszolgálniuk, de örülök, hogy nekem is sikerült egy hasonlót csinálni.

Csak hogy legyen mihez hasonlítani, a BeReal jelenleg több, mint 70 millió felhasználóval rendelkezik [19], amikből 20 millióan használják napi szinten, aminek az alkalmazás lényegéből kiindulva jelentős része a nap egy adott pontján aktív leginkább. Ez azt jelenti, hogy ahhoz, hogy megpróbáljam szimulálni a BeReal-t érő terhelést, 1-2000 ugyan ilyen méretű load test-et kellett volna futtatnom egy időben, valamint a Kubernetes-t futtató node-ok számát is jócskán megnövelni.

### 5.2 Továbbifejlesztési lehetőségek

Ahogy azt már írtam, van egy nagyobb biztonsági jellegű megoldandó kérdése az alkalmazásomnak, ami a tanúsítványok, secret-ek és kulcsok kezelése. Az alkalmazás élesbe állása esetén ezt a következőképpen gondolom megoldani. Jelenleg a fentieket a fájlok között bárki megnézheti, aki jogosult a GitHub repository-m megtekintéséhez, vagy bármiféle módon hozzáfér a kódbázisomhoz. Egy másik probléma az, hogy rengeteg beállítás minden mikroszolgáltatásnál ugyanaz, emiatt nagyon repetitív kódokat lehet a szolgáltatások között találni. Erre a két problémára nyújt megoldást a Spring Cloud Config Server és a Spring Vault. A Config Server segítségével egy közös szerveren lehet

kezelni az összes microservice konfigurációit, ezzel elkerülve a repetitivitást. Ahhoz pedig, hogy ezek között a beállítások között ne legyenek olyan információk, amiket rosszindulatú emberek felhasználhatnak általunk nem kívánt célra, a Spring Vault-ot használhatjuk, aminek segítségével biztonságos módon tárolhatunk és érhetünk el secret-eket, ezzel megelőzve az esetleges infrastrukturális veszélyeket. A Spring Cloud Config Server és a Spring Vault, a git verziókezelő és az AWS S3 kapcsolatát az 5.2:1 ábra mutatja be. Ezeknek a megoldásoknak főleg a lokális, vagy konténerizált környezetben futáskor van haszna, mivel a Kubernetes ahogy rengeteg más problémára is, erre is ad saját megoldásokat. A Deployment fájlokban meg tudtam adni olyan környezeti változókat, amiket a cluster-ben lévő Secret-ekben tároltam, a K8s pedig biztosítja, hogy ezek biztonságosan lesznek tárolva, ezzel kikerülve a Spring Vault és Config Server szükségét.



**5.2:1 Dinamikus szerver konfiguráció a Spring Cloud Config Server segítségével**

### **5.2.1 Milyen további funkciókkal egészíteném még ki a rendszert?**

Az alkalmazás prototípusom megvalósít minden olyan funkciót, amit szerettem volna beletenni, azonban van pár olyan képesség, amivel szívesen kiegészíteném. Az első ilyen, az az, hogy a posztoláskor a képeket az adatbázisba a felhasználó lokációjával mentse el. Ez két dolog miatt lenne jó, az egyik, hogy láthatnák a barátok, hogy hol készült a kép, a másik, ami nem sokban tér el ettől az az, hogy több barátnál lehetne egy olyan térkép nézetet is beletenni az applikációba, amin látni lehetne a barátok elhelyezkedését, ezzel azt is mutatva, hogy kik vannak éppen egy helyen, de még akár

egy értesítést is lehetne küldeni a közelben tartózkodó ismerősöknek, hogy ha van kedvük, csatlakozzanak a társasághoz.

Ha már értesítésekről van szó, akkor az is egy fontos funkció lenne, hogy a notification-service küldjön értesítéseket a felhasználóknak a Firebase Messaging segítségével, ezzel jelezve nekik, ha az ismerősük posztolt, reagált, kommentelt, vagy éppen ideje posztolni. Ennek a szerver oldali része már nagyjából le lett fejlesztve, annyi hiányzik, hogy a Firebase szolgáltatás megkapja az üzeneteket, a mobilok pedig reagálhassanak rá.

### **5.2.2 Milyen komponenst lenne érdemes cserélni?**

Mivel a frontendemet Kotlinban írtam, ezért a felhasználók által elérhető platformok terén kötve van a kezem, ugyanis jelenlegi állapotában csak az Androidot támogatja az applikációm.

Ezt ki lehetne egészíteni azzal, hogy iOS-re is lefejlesztsem, amihez a Swift programozási nyelvet és a SwiftUI használatát kellene elsajátítanom, viszont a Kotlin lehetőséget nyújt cross-platform alkalmazásfejlesztésre is, amit használva megoszthatjuk a kód alapvető részeit, mint például az üzleti logikát, amit pedig meg kell írni külön, az az alkalmazás kinézete.

Egy másik komponense az alkalmazásom mobil kliensének, aminek a kicserélésével sokat javíthatnák a működésén az az, amiről már részben írtam, a cache-elés. Mivel jelenleg a Pisacco könyvtárat használom a képek ideiglenes elmentéséhez, az elmentett képeknek meg egy Room adatbázisban tárolom a neveit a hozzájuk tartozó URL-lel, kellett hozni egy döntést, ami miatt az S3 előre aláírt URL-ei egy napig érhetőek el, ideális esetben viszont pár másodpercnek, esetleg egy percnak kellene ennek lennie. Ahhoz, hogy ezt megtehessem és az alkalmazás ezen része effektíven működhessen, arra gondoltam, hogy egy saját cache megoldást kellene létrehoznom, ezzel teljes mértékben lecserélve a Picasso által kínált funkciókat.

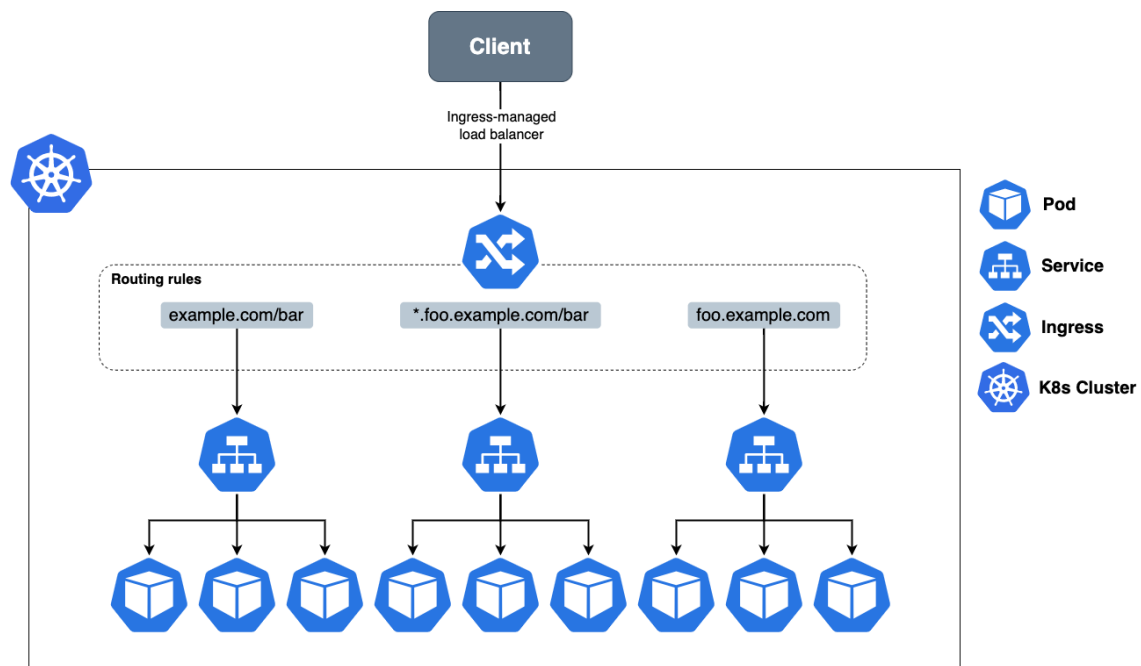
Ez a saját megoldás úgy nézne ki, hogy mikor egy képet letöltök, akkor azt az applikáció saját cache tárhelyére mentem el [12], majd a Room adatbázisba nem a kép ideiglenes URL-jét, hanem a lementett kép helyét tárolnám el. Azért ide menteném le, mivel itt nem érik el más alkalmazások és gyorsan lehet ide írni és innen olvasni is a fájlokat, ami a cache szempontjából optimális. Mivel a képeim méretét már feltöltéskor, a szerver és kliens oldalon is limitálom 1MB-ra (amit nem érnek el általában a képek),

egy felhasználónak meg nincsen sok ismerőse, így nem fognak sok helyet foglalni ezek a képek, viszont, hogy nagyjából fix tárhelyet foglaljon el maga az alkalmazás a telefonon, minden nap a képek posztolása előtt kitörölném a poszt specifikus cache-eket, így ami fixen lenne elmentve, az a felhasználónak és a barátainak a profilképei, így a reakciók és posztok naponta cserélődnek. Arra figyelnem kell majd a megvalósításkor, hogy legyen egy fallback megoldásom arra, ha a felhasználó kitörli az alkalmazás cache-ét, hiszen olyankor elvesznének az imént említett képek, emiatt pedig újra le kell azokat tölteni a szerverről.

Az utolsó dolog, amit máshogy csinálnék, ha most kezdeném előlről a projektet az az, hogy a mobil oldalon a Fragmentek közti váltogatást nem tranzakciókkal, hanem Navigation Component-tel valósítanám meg, mivel ez egy sokkal megbízhatóbb megoldást nyújt a problémára, hiszen átláthatóbb és egyszerűbb. Sok időm ment el a fejlesztés során azzal, hogy a mobilon bizonyos dolgokat megcsináljak, amikről később tudtam csak meg, hogy lehet sokkal egyszerűbben, mint például, amit az előbb említettem, vagy a szerverrel való kommunikációhoz használt függvényeim, amiket lecserélnék coroutine-okra, vagy használnék EventBus-okat a könnyebb eseménykezeléshez.

A backendemben egy olyan változtatási ötlet merült fel, amit máshogy lehetne csinálni, ez pedig a Kubernetes-sel való ismerkedés során jutott eszembe. Mivel a Kubernetes nyújt olyan szolgáltatásokat, amiket jelenleg az API gateway segítségével oldok meg, ezért, ha úgy akarnám, akár ki is törölhetném az egész gatewayt a megoldásomból és a Kubernetes-re bízhatnám a feladatai jelentős részének az elvégzését. Ahhoz, hogy ezt megcsináljam, csupán annyira lenne szükség, hogy az autentikációt áttegyem az API gateway-ből a mikroszolgáltatásokba, tehát minden szolgáltatás maga döntse el, hogy azt milyen feltételek mellett lehet elérni, ne legyen egyáltalán központosítva. Ez így utólag okosabb megoldásnak tűnik, viszont arról, hogy miért döntöttem a jelenlegi megvalósításnál, korábban már írtam. Ahhoz, hogy a változtatás után is működjön az alkalmazás routing-ja, az API gateway helyett Ingress-t kell használnom, ami gyakorlatilag a K8s alternatívája az API gateway-re. Ebben megadhatom, hogy a rá beérkező címetek hova továbbítsa, akár több, különböző host esetén is, valamint könnyedén összeköthetem a saját DNS-emmel, így, ha például a foo.baz.com-ra érkezik egy kérés, azt ki tudja szolgálni úgy, hogy belül a felhő IP címével

dolgozik. Erre a megoldásra készítettem is egy tervet, hogy érthetőbb legyen a változtatás 5.2:2.



5.2:2 Ingress által kezelt load balancer

### 5.2.3 A téma más alkalmazási területei

A képmegosztó alkalmazások tárháza kifogyhatatlan, hiszen az internet aköré épül, hogy a felhasználói meg akarnak osztani egymással bármilyen tartalmat, aminek nagyon nagy része multimédia típusú. Vehetjük példának a Twitter-t, vagy a Facebookot is, amiknél nem az volt a fő célpont, hogy képeket oszthassanak meg a felhasználók, mégis kulcsfontosságú funkcióikká nőtte ki magát a rajtuk való multimédia megosztás, aztán vannak a kimondottan erre a célra kitalált platformok is, mint az Instagram, vagy a Pinterest, de még sokáig lehetne sorolni, annyi létezik.

A közösségi médiáknak is még nagy jövője van, erre példának tudom hozni a BeReal-t is, ami alig több mint 2 éve létezik és mégis jelenleg az egyik legnagyobb közösségi platform, főleg a fiatalok között.

Talán technológia tekintetében mondanám a projektem témáját a leginkább jövőállónak, mivel egyre több alkalmazás esetében próbálnak a fejlesztők a mikroszolgáltatás alapú megoldások felé haladni, ami nem is meglepő, hiszen a régen írt szoftverek közül sok esetben jobb választás lett volna ezt az architektúrát alkalmazni, viszont az akkori technológia korlátai és a mikroszolgáltatásokról való tudás hiánya miatt nem ilyen módon tervezték meg az alkalmazásokat, mostanra viszont már akkora

méretűek azok, hogy elképzelhetetlen lenne alapjaikban lecserélni őket. A tudás hiánya teljesen érthető, hiszen a mikroszolgáltatások ötlete viszonylag újnak mondható, nagyjából 10 éve hívják így az addig hasonlóan felépített architektúrákat és azóta terjedt el nagy mértékben a fejlesztők között. Jelenleg az alkalmazások nagyjából harmada használ mikroszolgáltatásokra épülő architektúrát és egyes források szerint évi, nagyjából 20%-os emelkedés várható ebben [15].



## Köszönetnyilvánítások

Szeretném megköszönni a segítségét néhányaknak, akik segítsége nélkül nem tudtam volna elkészíteni a dolgozatomat.

Elsősorban köszönöm Forstner Bertalannak, a konzulensemnek a segítségét és irányítását a félév során, hogy értékes tanácsokkal látott el a konzultációkon és azokon kívül is, Nanys Patricknak és Hársádi Máténak, hogy elmagyarázták a Kubernetes alapjait, ezzel megspórolva rengeteg dokumentáció tanulmányozását és megkönnyítve a dolgozattal való haladásomat. Legvégül szeretném megköszönni az IBM Watson Media infrastruktúra csapatának, hogy lehetőséget adtak a saját demo célokra fenntartott Cluster-üknek a használatára mindenféle korlátozás nélkül és legfőképpen Hidasi Zsoltnak, hogy segített a cluster beállításában és fordulhattam hozzá kérdésekkel, ha elakadtam valamivel.

# Irodalomjegyzék

- [1] Spring Boot dokumentáció <https://spring.io/projects/spring-boot>
- [2] Spring Boot auto-configurations: <https://docs.spring.io/spring-boot/docs/1.4.x/reference/html/using-boot-auto-configuration.html>
- [3] Spring Cloud dokumentáció <https://spring.io/projects/spring-cloud>
- [4] Spring Security dokumentáció <https://spring.io/projects/spring-security>
- [5] Spring Security JWT with an RSA keypair  
<https://www.danvega.dev/blog/2022/09/06/spring-security-jwt/>
- [6] JSON Web Token <https://jwt.io/>
- [7] Android platform <https://developer.android.com/guide/platform>
- [8] Monolitikus vs mikroszolgáltatás alapú alkalmazások  
<https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>
- [9] Apache Kafka <https://kafka.apache.org/>
- [10] Kafka publish-subscribe messaging pattern  
<https://www.redhat.com/en/blog/apache-kafka-10-essential-terms-and-concepts-explained>
- [11] Kafka replication <https://kafka.apache.org/documentation/-replication>
- [12] Android app specific storage <https://developer.android.com/training/data-storage/app-specific>
- [13] Spring Cloud Config with Vault <https://blog.naamdesigns.com/dynamic-server-configuration-management>
- [14] DiscoveryClient for Kubernetes <https://docs.spring.io/spring-cloud-kubernetes/docs/current/reference/html/-discoveryclient-for-kubernetes>
- [15] History of Microservices <https://www.dataversity.net/a-brief-history-of-microservices/>
- [16] Retrofit könyvtár Kotlin-ra <https://square.github.io/retrofit/>
- [17] Minikube lokális Kubernetes cluster <https://minikube.sigs.k8s.io/docs/>
- [18] Kubernetes HorizontalPodAutoscaler <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>
- [19] BeReal statistics <https://www.businessofapps.com/data/bereal-statistics/>

## Függelék

A projekt elérhető a GitHubon keresztül az alábbi linkeken:

- backend: <https://github.com/pintertamas/befake-backend>
- mobile: <https://github.com/pintertamas/befake-mobile>