

GÁBOR DÉNES FŐISKOLA

MÉRNÖKINFORMATIKUS ALAPKÉPZÉS

---

**Helpdesk rendszer megvalósítása  
mikroszerviz alapú elosztott  
alkalmazással**

---

**Bőle Balázs**

Konzulens:

Dr. Nagy Elemér Károly

Szoftverfejlesztés szakirány



2020. december

FM008/01



## SZAKDOLGOZATTERV

GÁBOR DÉNES FŐISKOLA

hallgató neve: **Bőle Balázs**

születési ideje: 1993.07.31

Neptun-kód: DXQRPJ

értesítési címe: 1073 Bp., Erzsébet krt 19 3/34

lakástelefon: –

munkahelyi telefon:

mobil: +36 70 708 5003

e-mail: **bolebalazs@gmail.com**

szak: Mérnök informatikus

szakirány/specializáció:  
szoftverfejlesztés

A szakdolgozat területe: **Szoftverfejlesztés**

A szakdolgozat tervezett címe: **Helpdesk rendszer megvalósítása Microservices alapú elosztott alkalmazással**

A szakdolgozat készítésének helye (intézet): Gábor Dénes Főiskola

**Intézeti konzulens kijelölése szükséges: igen**

konzulens neve:

iskolai végzettsége:

munkahelye:

munkahelyi címe:

beosztása:

értesítési címe:

telefon:

e-mail:

**A szakdolgozatterv elbírálása a beadástól számított kb. három hét. A terv elfogadását/elutasítását a Neptunban**

**(Tanulmányok\Hallgató szakdolgozatai) tekintheti meg.**

A szakdolgozat célja, rövid tartalma és vázlata (tervezett tartalomjegyzéke):

A szakdolgozat célja:

Hogy bemutassam egy microservice alapú elosztott alkalmazás felépítését és működését.

A fejlesztés során megismert és használt technológiák átfogó összefoglalása (úgy mint hexagonális architektúra, MVC, docker, Angular, Spring Boot, kafka, load balancer, TDD, SOLID, etc.).

Az alkalmazás átfogó dokumentálása (például felhasználói-, üzemeltetési kézikönyv, komponens- és message flow diagram létrehozása).

A szakdolgozat rövid tartalma:

Az alkalmazás üzleti leírása:

A bestpractical által fejlesztett, open source "Request tracker" alkalmazáshoz hasonló funkciókkal bíró webes program, ami lehetőséget ad különböző csoportokhoz tartozó regisztrált ügyfélszolgálati felhasználók különböző e-mail címre érkező problémák vagy feladatok feldolgozására. A példaalkalmazás elérhető a [www.bestpractical.com/rt](http://www.bestpractical.com/rt) címen.

Egy új beérkező feladat egy előre meghatározott problémásorba kerül, ahonnan a sorhoz hozzárendelt csoport valamelyik ilyen jogokkal felruházott tagja felelőst vagy felelősöket rendelhet az adott kérdéshez, illetve a kérést más problémásorba is helyezheti. A felelős további levelezésbe bonyolódhat a probléma bejelentőjével a webes felületen keresztül. A probléma egy előre meghatározott állapotsoron megy keresztül, az állapotváltásról minden érintett értesítést kap.

Az alkalmazás technikai leírása:

Angular felhasználói felülettel, spring boot frameworkot használó java backenddel, és PostgreSQL adatbázissal működő dockerizált hexagonális alkalmazás. A buildhez használt programok: maven, nodeJs, npm, angular-cli.

Az autentikációért és autorizációért dedikált keycloak szerver felel. A frontend és a backend között REST alapú, a backend és az adatbázis között jpa alapú, a különböző microservicek között REST és kafka alapú kommunikáció valósul meg.

A servicek metrikái prometheusba integrálva érhetőek el.

A fejlesztés TDDben, a SOLID és a clean code elvek mentén történik. A kódminőségért eslint és sonarqube felel. A verziókövetésre github áll rendelkezésre.

A szakdolgozat vázlata (tervezett tartalomjegyzéke):

- 1) Abstract, bevezetés, a projekt átfogó leírása és célja (1 oldal).
- 2) Felhasznált technológiák irodalmi áttekintése (25 oldal)
- 3) A rendszer átfogó dokumentációja, a felmerült problémák leírása. Felhasználói, üzemeltetési kézikönyv (35 oldal)
- 4) Összefoglalás, A kitűzött célokkal az elért eredmények összevetése (1 oldal)
- 5) A továbbfejlesztés lehetséges irányai (1 oldal)

**A szakdolgozatterv elbírálása a beadástól számított kb. három hét. A terv elfogadását/elutasítását a Neptunban**

**(Tanulmányok\Hallgató szakdolgozatai) tekintheti meg.**

**Vállalom, hogy szakdolgozatomat az Egységes tájékoztatóban megtalálható „Szakdolgozatokkal szemben támasztott követelmények”-nek megfelelően készítettem el.**

(A követelmények megtalálhatóak a főiskola ILIAS felületén: Taneszköztároló\Záróvizsgáztatás)

Budapest....., 2020. év szeptember..... hó 17..... nap

.....  
hallgató

....., 20..... év ..... hó ..... nap

.....  
konzulens

**A szakdolgozatterv elbírálása a beadástól számított kb. három hét. A terv elfogadását/elutasítását a Neptunban  
(Tanulmányok\Hallgató szakdolgozatai) tekintheti meg.**

# Helpdesk rendszer megvalósítása mikroszerviz alapú elosztott alkalmazással

készítette

**Bőle Balázs**

Neptun kód: DXQRPJ

Elérhetőség: [bolebalazs@gmail.com](mailto:bolebalazs@gmail.com)

Konzulens: Dr. Nagy Elemér Károly

A dolgozat elektronikus változata elérhető a <https://github.com/balazsBole/> címen.



Budapest, 2020. december.

## Kivonat

Dolgozatomban bemutatom a létrehozott helpdesk alkalmazás célját, felépítését és működését.

Ismertetem az alkalmazással szemben fennálló üzleti igényeket, a tervezés során fellépő általános problémákat, és az igények kielégítése céljából felhasznált technológiákat.

Áttekintést adok a megvalósítás során megoldott feladatokról, a meghozott döntések szempontjairól és a lehetséges alternatívákról.

Működés közben, egy valódi példán keresztül szemléltetem az alkalmazás struktúráját, a különböző feladatokért felelős komponensek felépítését és működését.

Végül terheléses tesztel megvizsgálom hogy a megvalósított helpdesk alkalmazás megfelel-e az üzleti elvárásoknak, és az üzemeltetéséhez létrehozott eszközök segítségével elemzem az alkalmazás teljesítményét.

# Tartalomjegyzék

<b>Tartalomjegyzék</b>	<b>vi</b>
<b>Bevezetés</b>	<b>1</b>
<b>1. Üzleti igények</b>	<b>2</b>
1.1. Funkcionális igények . . . . .	2
1.1.1. E-mail fogadása és küldése . . . . .	2
1.1.2. E-mail szálak kezelése . . . . .	2
1.1.3. Több felhasználó . . . . .	4
1.2. Nem funkcionális igények . . . . .	5
1.2.1. Skálázhatóság . . . . .	5
1.2.2. Granuláris felosztottság . . . . .	5
1.2.3. Mérhető indikátorok . . . . .	5
1.2.4. L10N . . . . .	6
<b>2. Technológiai áttekintés</b>	<b>7</b>
2.1. Mikroszerviz architektúra . . . . .	7
2.2. Hexagonális architektúra . . . . .	8
2.3. Rétegek szeparálása . . . . .	9
2.4. Konkurencia kezelése . . . . .	9
2.5. Alkalmazások szeparálása . . . . .	10
2.6. Apache Kafka . . . . .	11
2.7. Angular . . . . .	11
2.8. Spring Boot . . . . .	12
<b>3. Az alkalmazás felépítése</b>	<b>13</b>
3.1. Legfontosabb komponensek . . . . .	13
3.2. Adatbázis UML diagram . . . . .	14
3.3. E-mail fogadásának és küldésének folyamata . . . . .	15

<b>4. Technikai tervezés</b>	<b>18</b>
4.1. Webszerver . . . . .	18
4.2. Adatbázis . . . . .	19
4.2.1. Teljesítményvizsgálat . . . . .	19
4.3. JPA implementáció . . . . .	20
4.4. Adatbázis migrációs eszköz . . . . .	20
4.4.1. Liquibase . . . . .	21
4.4.2. Flyway . . . . .	21
4.4.3. Felhasznált eszköz . . . . .	21
4.5. Monitorozás . . . . .	21
4.5.1. Prometheus . . . . .	21
4.5.2. Grafana . . . . .	22
4.5.3. Graphite . . . . .	22
4.5.4. Megvalósítás . . . . .	23
<b>5. Implementáció</b>	<b>24</b>
5.1. Mikroszerviz infrastruktúra . . . . .	24
5.1.1. Nginx . . . . .	24
5.1.2. Docker konténerizáció . . . . .	24
5.1.3. Metrikák . . . . .	25
5.2. E-mail kliens . . . . .	25
5.2.1. E-mail szabvány . . . . .	25
5.2.2. Üzleti funkciók megvalósítása . . . . .	26
5.3. Helpdesk backend . . . . .	27
5.3.1. Spring Boot . . . . .	28
5.3.2. Adatbázis . . . . .	28
5.3.3. Pessimista konkurenciakezelés . . . . .	28
5.3.4. Optimista konkurenciakezelés . . . . .	28
5.3.5. Üzleti funkciók megvalósítása . . . . .	29
5.3.6. Egyéb eszközök . . . . .	30
5.4. Helpdesk frontend . . . . .	30
5.4.1. Kommunikáció a backenddel . . . . .	30
5.4.2. Komponensek . . . . .	30
5.4.3. Üzleti funkciók megvalósítása . . . . .	31
5.4.4. Futtatási környezet . . . . .	32
5.5. Keycloak . . . . .	34
5.5.1. Jogosultságkezelés . . . . .	34
5.5.2. JSON Web Token . . . . .	34



5.5.3. OAuth 2.0 keretrendszer . . . . .	34
5.6. Kafka . . . . .	35
5.7. Helpdesk backend és a Keycloak elkülönítése . . . . .	36
<b>6. Alkalmazás bemutatása</b>	<b>37</b>
6.1. Alkalmazás elindítása . . . . .	37
6.2. Virtuális gép . . . . .	37
6.3. Deployment . . . . .	39
6.4. Több példány . . . . .	40
6.5. E-mail fogadásának és küldésének folyamata . . . . .	41
6.6. Adatbázistáblák . . . . .	43
6.6.1. Liquibase . . . . .	43
6.6.2. Hibernate Envers . . . . .	44
6.7. Apache Kafka . . . . .	45
6.8. Eureka . . . . .	46
<b>7. Terheléses tesztelés</b>	<b>47</b>
7.1. Terheléses teszt . . . . .	47
7.2. Apache JMeter . . . . .	47
7.3. Átlagos teljesítmény vizsgálata . . . . .	48
7.4. Csúcsteljesítmény vizsgálata . . . . .	49
7.5. Szűk keresztmetszet meghatározása . . . . .	51
7.5.1. Nginx . . . . .	52
7.5.2. HikariCP . . . . .	54
7.5.3. Megnövekedett processzorigény . . . . .	55
7.5.4. Szűk keresztmetszet meghatározása . . . . .	56
7.6. Összevetés a követelményekkel . . . . .	57
7.6.1. Átlagos teljesítmény vizsgálata . . . . .	57
7.6.2. Csúcsteljesítmény vizsgálata . . . . .	57
<b>8. Továbbfejlesztési lehetőségek</b>	<b>58</b>
8.1. A deploymentről . . . . .	58
8.2. A kódról . . . . .	58
<b>Irodalomjegyzék</b>	<b>62</b>
<b>Ábrák jegyzéke</b>	<b>64</b>
<b>A. OpenApi dokumentáció</b>	<b>65</b>

# Bevezetés

A mikroszerviz alapú alkalmazások egyre nagyobb népszerűségnek örvendenek, derül ki az O'Really által készített felmérésből [1]. Egyre több cég szeretné lecserélni meglévő monolit rendszerét, vagy a szükséges új funkciókat a régebbi rendszertől függetlenül, hibrid rendszerben valósítaná meg.

A wiredelta a témában készített cikkében [2] összegyűjtötte a mikroszerviz alapú architektúra előnyeit. Míg a nagyvállalati környezetben sokszor a folyamatos szállítási igény, vagy az egymástól függetlenül fejleszthető alrendszerek miatt döntenek emellett a technológia mellett, az én esetemben a legfontosabb szerepet a skálázhatóság, az újrafelhasználhatóság, és az alacsony fenntartási költség játszotta.

Úgy gondolom, hogy nincs olyan technológia, ami minden problémára megoldást nyújtana. De úgy érzem hogy a mikroszerviz elvei mentén kialakított alkalmazások, természetükből adódóan időtállóbbak maradnak. Ha el tudjuk érni, hogy egy alkalmazás valóban csak egy funkcióért kell hogy felelős legyen, azzal a problémamegoldás analitikus oldalát emeljük rendszerszintre. Az én meglátásom szerint pont ebben, a feladatok és felelősségek rendszerezésében rejlik a mikroszerviz architektúra valódi előnye.

# 1. fejezet

## Üzleti igények

Ebben a fejezetben bemutatom a Helpdesk alkalmazás felé megfogalmazott üzleti igényeket.

### 1.1. Funkcionális igények

Az alkalmazásnak az alábbi funkcionális igényeknek kell megfelelnie.

#### 1.1.1. E-mail fogadása és küldése

Az ügyfelektől érkező e-maileket az alkalmazás képes fogadni, hosszú távra megőrizni. Az ügyfelek számára formázott válasz e-mail küldhető.

A rendszernek képesnek kell lennie több e-mail cím kezelésére. A beérkező új üzeneteket – a kapcsolódó üzenetszálon keresztül – a címzettnek megfelelő előre definiált sorhoz kell hozzárendelnie.

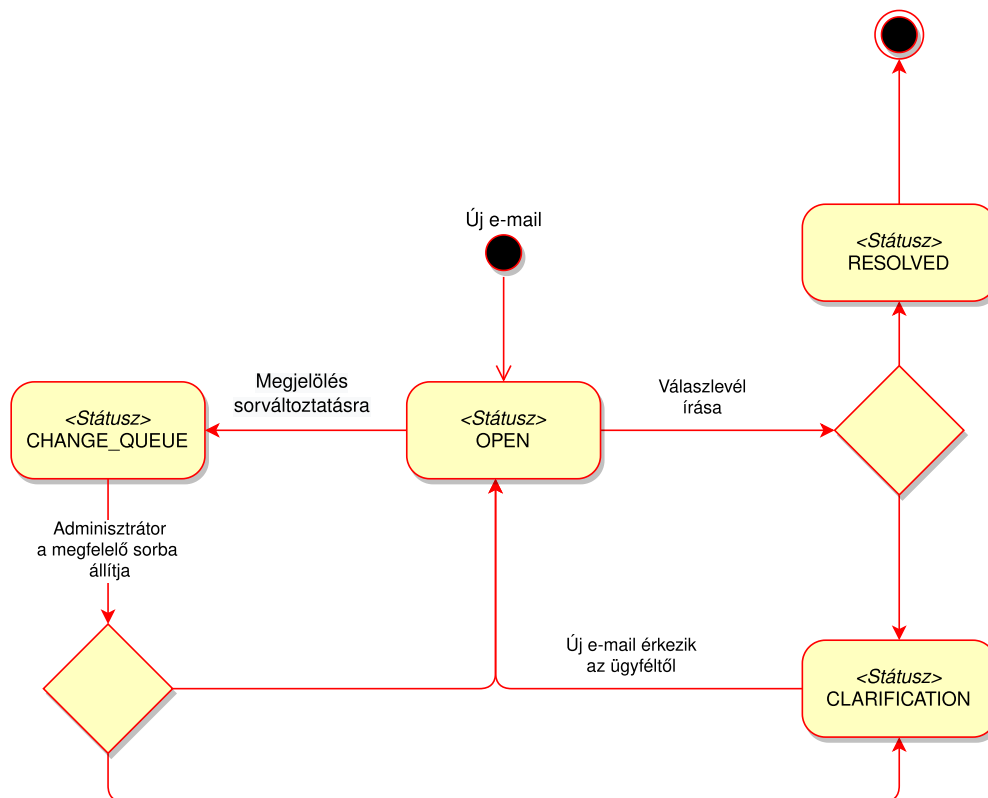
#### 1.1.2. E-mail szálak kezelése

A rendszer által kezelt üzenetek szálakba rendezve érhetőek el. Egy szál az ügyfél és a felhasználó közötti üzenetváltásokból épül fel.

Az üzenetszálakra vonatkozó összes adat visszamenőleg lekérdezhető, státuszuk az [1.1](#) ábrán definiált útvonalaknak megfelelően változtatható.

**OPEN** vagy nyitott állapotba kerül az üzenetszál, ha új e-mail érkezik az ügyféltől, vagy ha a felhasználó átállítja az e-mail szál státuszát OPEN-re. A felhasználók csak OPEN állapotú e-mail szálra képesek válaszüzenet küldeni.

**RESOLVED** vagy lezárt állapotú egy e-mail szál, ha a felhasználó a válasz üzenet küldése során a szálát RESOLVED státusszal jelöli meg. A felhasználó ezzel az állapottal



1.1. ábra. Az e-mailszálak státuszváltozásai

Forrás: saját ábra

jelezheti, ha szerinte az üzenetszál lezárható, egy e-mail szál lezárt státuszban marad mindaddig, míg az ügyféltől nem érkezik kapcsolódó üzenet, ekkor az üzenetszál visszakerül OPEN státuszba.

**CLARIFICATION** vagy tisztázandó állapotba kerül az üzenetszál, ha a felhasználó a válasz üzenet küldése során a szálát CLARIFICATION státusszal jelöli meg. A felhasználó ezzel az állapottal jelezheti, ha szerinte az üzenetszál még nem zárható le, a továbblépéshez több információ, vagy az ügyfél általi további tisztázás szükséges.

A CLARIFICATION státuszú üzenetszál állapota a felhasználó által, kézzel is állítható. Abban az esetben, ha az ügyféltől válaszüzenet érkezik, akkor a szál státusza automatikusan OPEN állapotúra változik.

**CHANGE\_QUEUE** vagy sor váltására várakozó állapottal jelezheti a felhasználó, ha úgy érzi, hogy az üzenetszál másik sorba, másik felhasználóhoz tartozik.

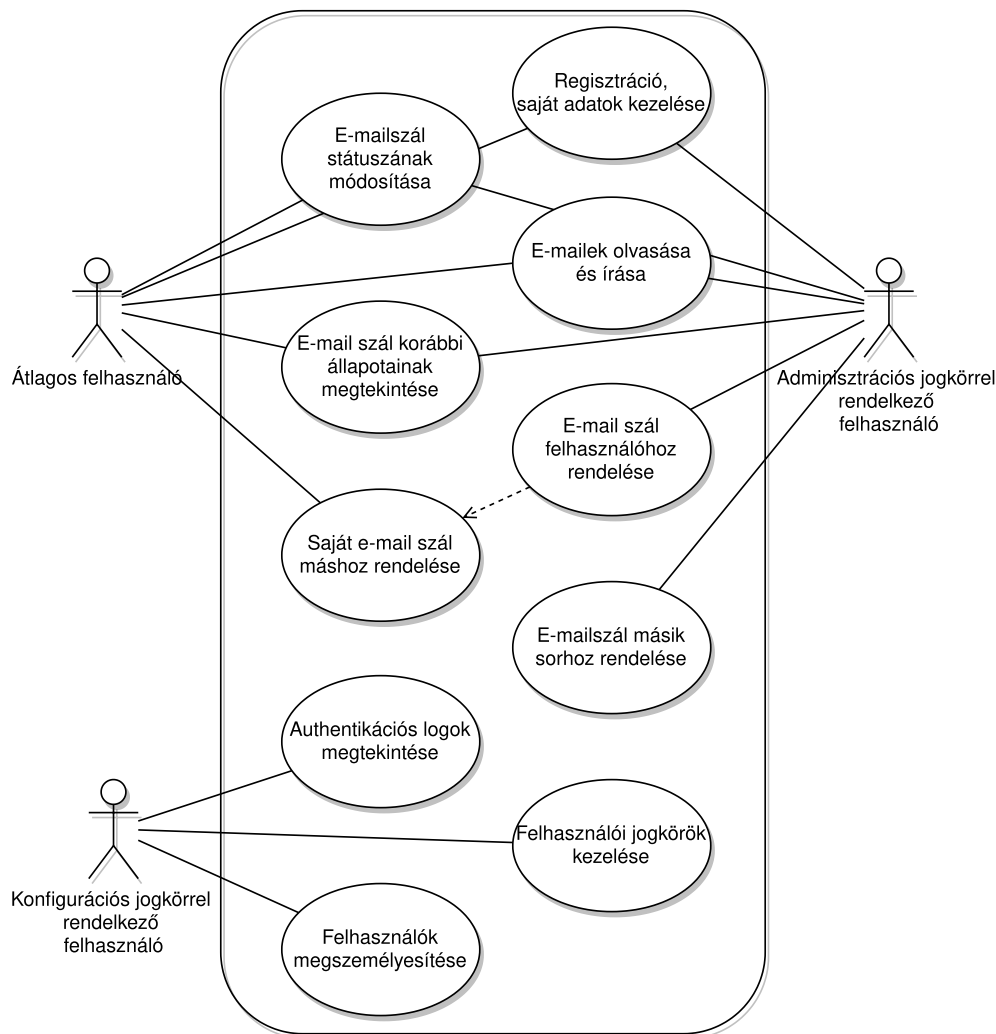
A CHANGE\_QUEUE állapotú üzenetszálakat az adminisztrációs jogkörrel rendelkező felhasználók megvizsgálják, a sor váltására vonatkozó igényt elbírálják, és

a megfelelő sor, felhasználó és státusz kiválasztásával, az üzenetszál állapotát véglegesítik.

### 1.1.3. Több felhasználó

A rendszert egyszerre több felhasználó használhatja. Regisztrációjukat az alkalmazásba, és saját adataik kezelését egyénileg végzik.

A felhasználói felületen elérhető funkciókat az 1.2 ábrán foglaltam össze.



1.2. ábra. Elérhető funkciók jogosultság szerint csoportosítva

Forrás: saját ábra

Minden felhasználó csak a saját üzenetszállait kezelheti, csak azokra válaszolhat. Pontosabban egy, az 1.1.1 pontban említett sorhoz tartozik, és csak az ugyanabba a sorba tartozó e-mail szál rendelhető hozzá. A számára kijelölt szálat képes – a saját során belül – más felhasználóhoz rendelni.

A felhasználók eltérő jogkörökkel rendelkezhetnek. Az adminisztratív jogkörrel rendelkező felhasználó végzi az új e-mailszál felhasználóhoz rendelését, valamint a `CHANGE_QUEUE` státuszban (1.1 ábra) lévő üzenetszálak új sorba irányítását.

A konfigurációs jogkörrel rendelkező felhasználó feladata az alkalmazásban használt jogkörök (*role*-ok) kezelése, felhasználók adminisztrációs jogkörhöz rendelése. Lehetősége van továbbá az autentikációs események megtekintésére, jelszó visszaállítására és más felhasználók kizárására vagy megszemélyesítésére (*impersonate*).

## 1.2. Nem funkcionális igények

Az alkalmazásnak az alábbi nem funkcionális igényeknek kell megfelelnie.

### 1.2.1. Horizontális skálázhatóság

A kiszorgálandó kliensek száma napi és havi szinten is eltérő. Az év egyes időszakában nagyobb volumenű ügyfél-interakció prognosztizálható. A hibatűrés javítása, és a megnövekedett forgalom érdekében – ezekben az előre meghatározott időszakokban – horizontális skálázódás szükséges.

### 1.2.2. Granuláris felosztottság

A helpdesk alkalmazást használó ügyfélszolgálat munkaórákban a legaktívabb, míg az e-maileket küldő ügyfelek hétvégente és hétköznapi munkaórákon kívül a legaktívabbak.

A hosszútávú tervekben szerepel a helpdesk alkalmazás és a belső céges levelezés integrálása.

A fenti két szempont miatt célszerű a megvalósítandó funkciók minél nagyobb mértékű szeparálására törekedni.

### 1.2.3. Mérhető indikátorok

A rendszernek átlagosan 100 felhasználót kell kiszorgálnia másodpercenként. A várható csúcsteljesítmény 5 000–10 000 lekérdezés másodpercenként. A felhasználók száma elfogadható legnagyobb válaszidő 3 másodperc/lekérés.

**1.2.4. L10N<sup>1</sup>**

A felhasználói, adminisztratív és karbantartói felületek angol nyelven érhetőek el. A magyar, vagy attól eltérő egyéb nyelv kezelése nem szükséges.

A tervezés során a honosításhoz, vagy többnyelvűsítéshez szükséges szempontokat nem kell figyelembe venni.

---

<sup>1</sup>A nyelvi lokalizációt, vagy kulturális beágyazást szokás L10N-ként rövidíteni.

## 2. fejezet

# Felhasznált technológiák

Az alkalmazás rendszer szinten mikroszerviz (2.1), a modulok szintjén hexagonális architektúrába (2.2) rendezve készült el. A frontend Angulart (2.7), a backend és az e-mail kliens Spring Boot-ot (2.8) használ. A alkalmazáson belüli események kezelésére és tárolására Apache Kafkát (2.6) használok.

### 2.1. Mikroszerviz architektúra

Bár a kifejezés már régóta ismert, nincs egy központilag elfogadott, egységes definíció arra nézve, miket nevezünk mikroszervizeknek. A legtöbb szerző jobb híján a visszatérő karakterisztikus tulajdonságuk alapján sorolja be az alkalmazásokat ebbe a kategóriába [3]. Egy tipikus mikroszerviz a következő tulajdonságoknak felel meg:

- pontosan egy üzleti funkció köré szerveződik,
- más szolgáltatásokkal laza, általában hálózaton keresztül megvalósuló kapcsolatban áll,
- ha szüksége van adatbázisra, akkor sajáttal rendelkezik, más rendszer ezt az adatbázist nem éri el,
- önmagában is működőképes,
- decentralizált, tehát nincs egy a munkáját befolyásoló központi irányítórendszer.

A hasonló felépítésükből adódóan, számos olyan eszköz van, ami – nem kötelezően, de legtöbbször – együtt fordul elő a mikroszerviz architektúrával. A legfontosabb ilyen fogalmak a:



**skálázhatóság** a rendszer képessége az áteresztőképességének növelésére. Létezik vertikális<sup>1</sup> és horizontális skálázhatóság<sup>2</sup>.

**konténerizálás** az adott szolgáltatás futtatása saját elszeparált környezetében hardveres virtualizáció segítségével nélkül.

**szolgáltatás felderítés** a rendszer által nyújtott szolgáltatások, szervizek automatikus felfedezhetősége<sup>3</sup>.

**loadbalancer** az a folyamat, ami a bejövő feladatokat erőforrásokhoz rendeli. Legegyszerűbb megvalósítása a *round robin* algoritmus, célja a terhelés egyforma elosztása.

**monitorozás** az önálló szolgáltatások állapotának felügyelése. A monitorozás során nyújtott metrikák kiterjedhetnek a felhasznált memória mennyiségére, processzor-igényére, vagy processzeire is.

## 2.2. Hexagonális architektúra

A hexagonális architektúra – vagy más néven portok és adapterek architektúrája – egy Alistair Cockburn által létrehozott [4] szoftvertervezési minta. Nevét a cikkben felrajzolt hatszögletű rendszerábrázolásról kapta (2.1 ábra), ami szembeötlően a korábban elterjedt réteges elrendezéssel.

Az eredeti szándék mögöttese az alkalmazás függetlenítése mindennemű külső függőségtől<sup>4</sup>, így lehetővé téve az üzleti és a technikai igények nagy mértékű szeparálását. Egy absztrakt port feladata kell legyen a külvilággal való kapcsolat, így az üzleti logika csak az üzenet tartalmáért felelős, az üzenetküldés módjáért már nem.

Ahogy Robert C. Martin a *The Clean Architecture* című cikkében [5] összeszedte, a port-adapter és a hasonló architektúrával készülő alkalmazások mind:

- Könnyen, és önmagukban is tesztelhetők, mivel az üzleti szabályoknak nincs külső függőségük.
- Függetlenek a külső tényezőktől. Így az alkalmazás által használt felület vagy adatbázis könnyen cserélhető.
- Keretrendszerrel függetlenül is megvalósíthatóak. A megvalósítás nem függ semmilyen könyvtártól vagy egyéb tulajdonságtól.

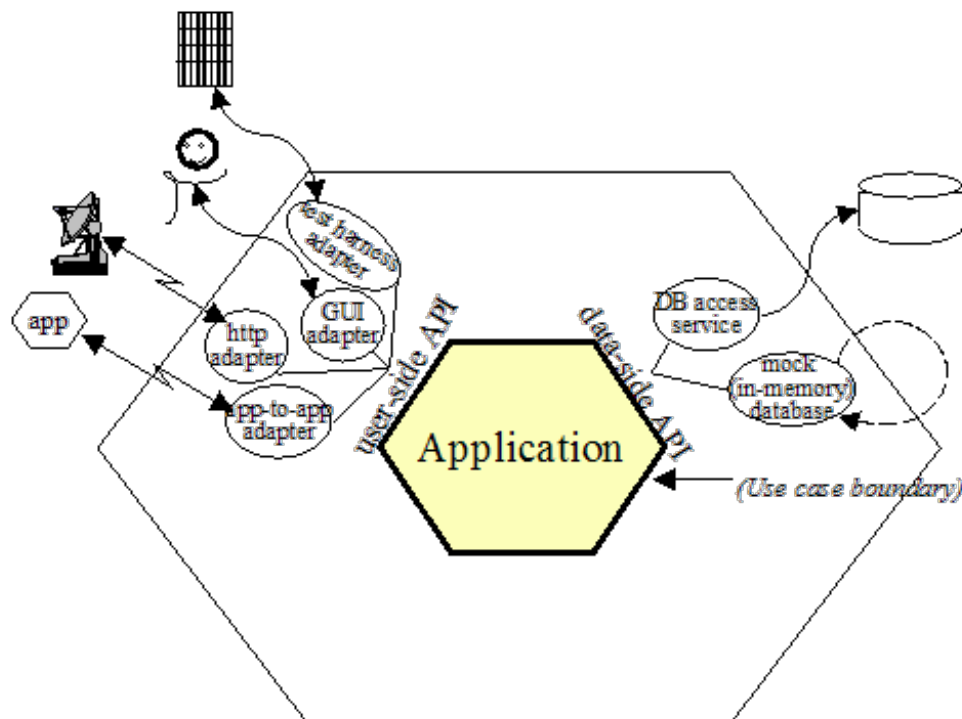
---

<sup>1</sup>több processzor vagy memória bevonása

<sup>2</sup>újabb példányok futtatása

<sup>3</sup>angolul *service discovery*-nek hívják

<sup>4</sup>például adatbázis, felhasználók, automatizált tesztek



2.1. ábra. A hexagonális alkalmazás külső függőségeinek elszeparálása

Forrás: Alistair Cockburn [4]

## 2.3. Rétegek szeparálása

A hexagonális architektúra (2.2 pont) és a hasonló *clean code* [6] elvek sokszor a különböző szoftver rétegek elkülönítésén alapszanak.

Annak érdekében, hogy a feladatok elkülönítése ne vonzza magával az ismétlődő program részletek megnövekedését, célszerű generálni a visszatérő, üzleti funkciót nem hordozó sorokat. Ilyen – a fordítási időben – kódot generáló eszköz a Mapstruct és a Lombok.

## 2.4. Konkurencia kezelése

Ha az alkalmazásnak egyszerre több felhasználót kell kiszolgálnia, vagy bármilyen oknál fogva ugyanazt az adatot egy időben több program módosítaná, akkor az inkonzisztens állapot elkerülése érdekében célszerű valamilyen konkurenciakezelési stratégiát alkalmazni. Alapvetően két fajta konkurenciakezelő megoldás létezik:

**optimista** konkurenciakezelést akkor érdemes használni, mikor számíthatunk arra, hogy az esetek többségében nincs párhuzamos módosítás. Ütközés esetén – ha

egyszerre kellene ugyanazt az adatot módosítani – a tranzakciót elvetjük és értesítjük a módosítást kezdeményező felet, hogy időközben az adat megváltozott.

Ez a megoldás tehát nagy mennyiségű adat, és hozzá képest relatív kis számú felhasználó esetén ideális.

**pessimista** konkurenciakezelés esetén, a módosítani kívánt adatot olvasáskor zároljuk, az csak a módosítás befejezése után lesz újra hozzáférhető a többi fél számára.

Az adatok a teljes tranzakció ideje alatt zárolva vannak, ezért ez a megoldás gyakran jár együtt teljesítménycsökkenéssel. A kölcsönös zárolás pedig, – mikor két vagy több tranzakció egymás befejezésére vár – könnyen vezethet *deadlock*hoz.

## 2.5. Alkalmazások szeparálása

Ahogy azt a 2.1. pontban is írtam, hogy megvalósítható legyen a szolgáltatások laza kapcsolata és egymástól független működése, a mikroszerviz csak a saját adatbázisához férhet hozzá. Ez lehetővé teszi a feladatnak megfelelő adatbázis választását is.

A mikroszervizeken átnyúló üzleti funkciók megvalósítására több megoldás is létezik:

**API kompozíció** A legegyszerűbben megvalósítható az API kompozíció. Ebben az esetben az applikáció maga végzi el, saját memóriájában az adatok egymáshoz rendelését.

Kis számú adatnál használható, és célszerű elkerülni hogy az adat kettő vagy annál több számú mikroszervizen keresztül érkezzen meg.

**CQRS** A CQRS<sup>5</sup> az olvasás és írás műveletének elszeparálásán alapuló megoldás [7].

Lényege hogy a CRUD műveletekről minden esetben egy esemény keletkezik. Ezekre az eseményekre bármelyik mikroszerviz feliratkozhat.

Ha más rendszernek szüksége van az aktuális állapotra, az az események újrátjátszásával bármikor megkapható.

Az Apache Kafkát (2.6) gyakran használják az események kezelésére, mert natívan támogatja az események csoportosítását egyedi azonosító alapján. Beállítható, hogy UUID alapján mindig csak a legfrissebb állapot legyen elérhető, ezzel lecsökkentve a kezdeti olvasáshoz szükséges időt.

---

<sup>5</sup>Command Query Responsibility Segregation

**Elosztott tranzakciók és Saga** Ha nem csak más szolgáltatások adatainak olvasásáról van szó, hanem több szolgáltatáson átívelő, visszagörgethető tranzakciókat kell megvalósítani, arra az esetre találták ki a *Saga*-t.

A *Saga* egy hosszú életű elosztott tranzakció [8]. A folyamat lépései sorban hajtnak végre, minden lépés tartalmaz egy utasítást arra az esetre ha vissza kellene görgetni a teljes folyamatot. Ha a folyamat bármelyik lépésnél megghiúsul, onnantól fogva visszafelé minden rendszer egyesével visszaáll a tranzakció előtti állapotra.

## 2.6. Apache Kafka

Az Apache Kafka egy üzenet tárolásra és továbbításra kifejlesztett hibatűrő, magas áteresztő képességű, nyílt forráskódú alkalmazás [9].

A feladó az üzenetet nem közvetlenül a fogadónak küldi, hanem egy üzenetbrókeren keresztül egy (*topic*)-ba teszi közzé. A fogadó fél dönti el, hogy melyik téma üzeneteit szeretné megkapni. Annak érdekében, hogy megkaphassa az üzenetet, feliratkozik az üzenet témájára.

Redundancia és skálázhatóság miatt egy *topic* több partícióra van elosztva, és ezen felül minden partíció replikálva is van [10]. A partíciók eltérő szerveren lehetnek, ezáltal egy *topic* horizontálisan skálázható. Egy szerver esetleges kiesése esetén a többi szerver át tudja venni a kiesett szerver szerepét.

Az üzenetbrókerek összehangolását a Zookeeper szolgáltatás végzi. Mivel minden kafka bróker beregisztrálja magát a szolgáltatásba, így a Zookeeper mindig naprakész információval rendelkezik az üzenetbrókerekről.

Az üzeneteket Apache Avroval szerializálom. Az Avro lehetővé teszi a kompakt bináris tárolást, de natívan támogatja a JSON reprezentációt is. Az Avrohoz szükséges séma nyilvántartásért és az eltérő verziók kezelésért a Schemaregistry szerver felelős. A kafka kliensek a Schemaregistry szerveren keresztül tudják az üzeneteket olvasni és írni.

## 2.7. Angular

Az Angular egy a Google által fejlesztett TypeScript alapú platform és keretrendszer [11]. A segítségével létrehozott kód erősen modularizált, így könnyű vele újra felhasználható és a modell-nézet-vezérlő elvet követő alkalmazást létrehozni.

Az Angularral készített honlap teljes mértékben a kliens oldalon fut, így a szervertől elegendő egy egyszerű, statikus HTML-oldalt visszaadó alkalmazáserver használata.

## 2.8. Spring Boot

A Spring Boot, egy a Springre épülő keretrendszer. Mindkét rendszer alapja a függőség befecskendezése<sup>6</sup>, ami egy a 2.3 pontban említett tiszta kód [6] eszköze.

A Spring Boot [12] célja, hogy gyorsan és egyszerűen lehessen önálló, magas minőségű alkalmazásokat fejleszteni:

- az alapbeállítástól való eltérést kell meghatározni<sup>7</sup> ezzel lecsökkentve a konfigurációval töltött időt,
- valamint sok gyakran visszatérő problémára<sup>8</sup> nyújt könnyen elérhető megoldást.

---

<sup>6</sup>Angolul *Dependency Injection*

<sup>7</sup>A Spring Boot dokumentációban ezt röviden *convention over configuration*-nek hívják

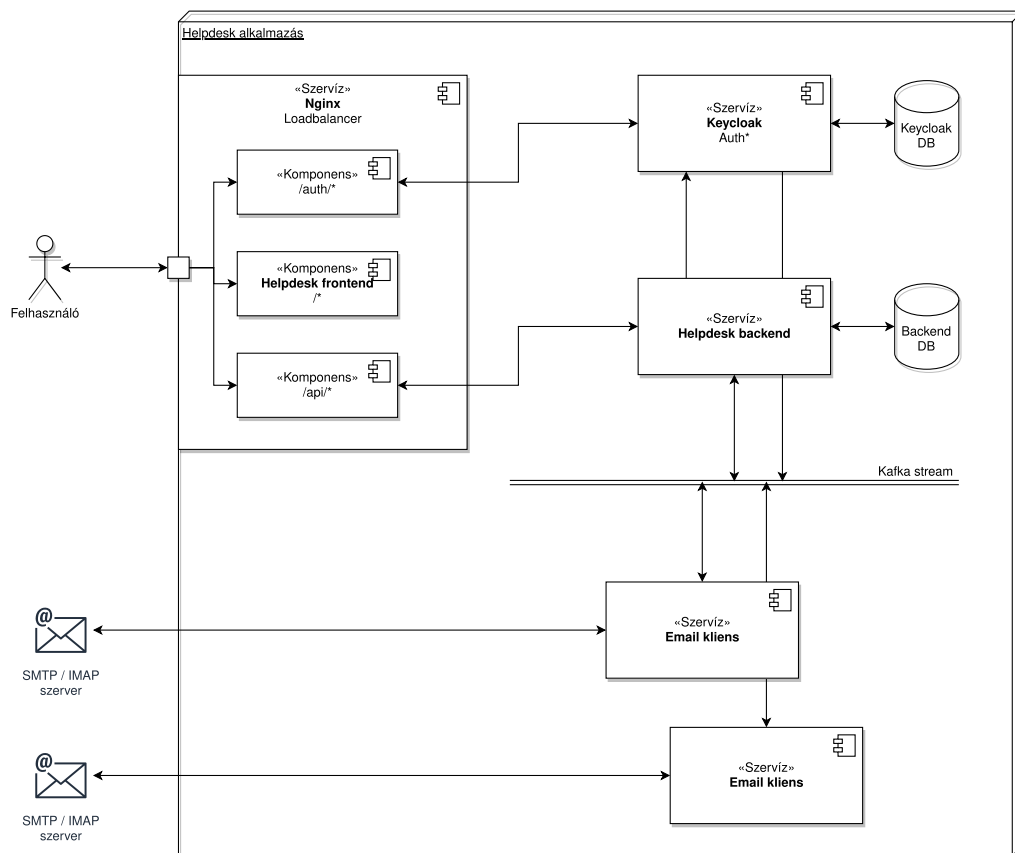
<sup>8</sup>Például: metrikák, biztonság, adattárolás

## 3. fejezet

# Az alkalmazás felépítése

Ebben a fejezetben átfogó képet adok az általam létrehozott helpdesk alkalmazásról. Az egyes komponensek részletes leírása az 5. fejezetben található.

### 3.1. Legfontosabb komponensek



3.1. ábra. A legfontosabb komponensek

Forrás: saját ábra

A 3.1. ábrán a legfontosabb szolgáltatásokat gyűjtöttem össze. Az üzleti funkcionális megvalósulása az itt bemutatott komponensek összehangolt munkáján keresztül valósul meg.

- A felhasználó az nginx-en (5.1.1 pont) keresztül éri el a helpdesk alkalmazást. Az ő perspektívájából nem látszik az alkalmazás réteges, széttagolt felépítése.
- Az nginx az üzenet URL-je alapján dönti el, hogy melyik kérést, melyik szolgáltatás szolgáljon ki.
- Az e-mail kliens és a helpdesk backend kafka streamen keresztül éri el egymást. Közvetlenül nem kommunikálnak.
- Az e-mail kliensek kezelik az e-mail szerverekkel való adatcserét. Az e-mailek fogadásáért és küldéséért felelősek.
- A Keycloak szolgáltatás adatot küld a kafka streamen keresztül, amit a helpdesk backend fogad.

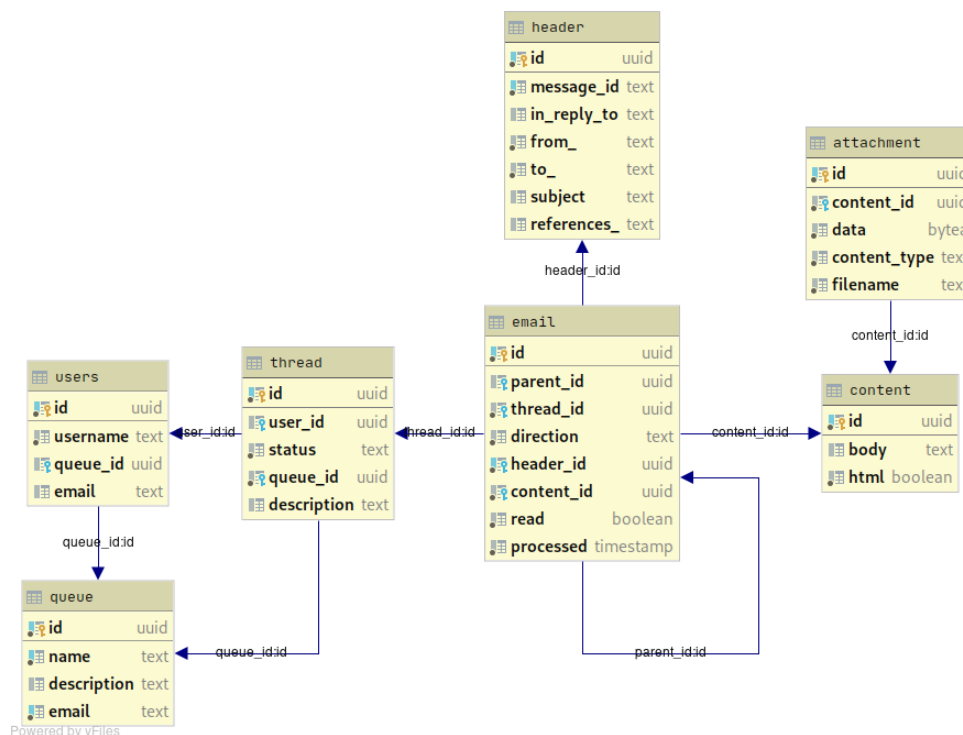
## 3.2. Adatbázis UML diagram

A helpdesk backend adatbázis legfontosabb tábláit a 3.2. ábra tartalmazza. Az ábrán nem szerepelnek az audit és a Liquibase által használt táblák (5.3.2 pont), azokat – az összes többi táblával együtt – a 6. fejezetben található 6.3. ábra tartalmazza.

A 3.2. ábrán megjelenített adatszerkezetről elmondható, hogy:

- egy felhasználó (`users` tábla) pontosan egy sorhoz (`queue` tábla) tartozik, és több e-mail szállal rendelkezhet;
- egy e-mail szál (`threads` tábla) pontosan egy sorhoz tartozik, és csupán egy felhasználóhoz tartozhat. Egy e-mail szálhoz legalább egy e-mail tartozik;
- egy e-mailnek (`email` tábla) pontosan egy fejléce (`header` tábla) és egy tartalma (`content` tábla) van, valamint pontosan egy e-mail szálhoz tartozik. Ezen kívül, minden e-mail legfeljebb egy szülő e-mail alá tartozhat;
- egy e-mail tartalomnak (`content` tábla) nulla vagy több csatolmánya (`attachment` tábla) lehet.

Az adatbázis hármas normálformában van, mert az entitások minden tulajdonsága csak az entitás teljes kulcsától függ, és a kulcson kívül semmilyen más tulajdonságától nem függ.



3.2. ábra. A backend legfontosabb adatbázistáblái

Forrás: saját ábra

### 3.3. E-mail fogadásának és küldésének folyamata

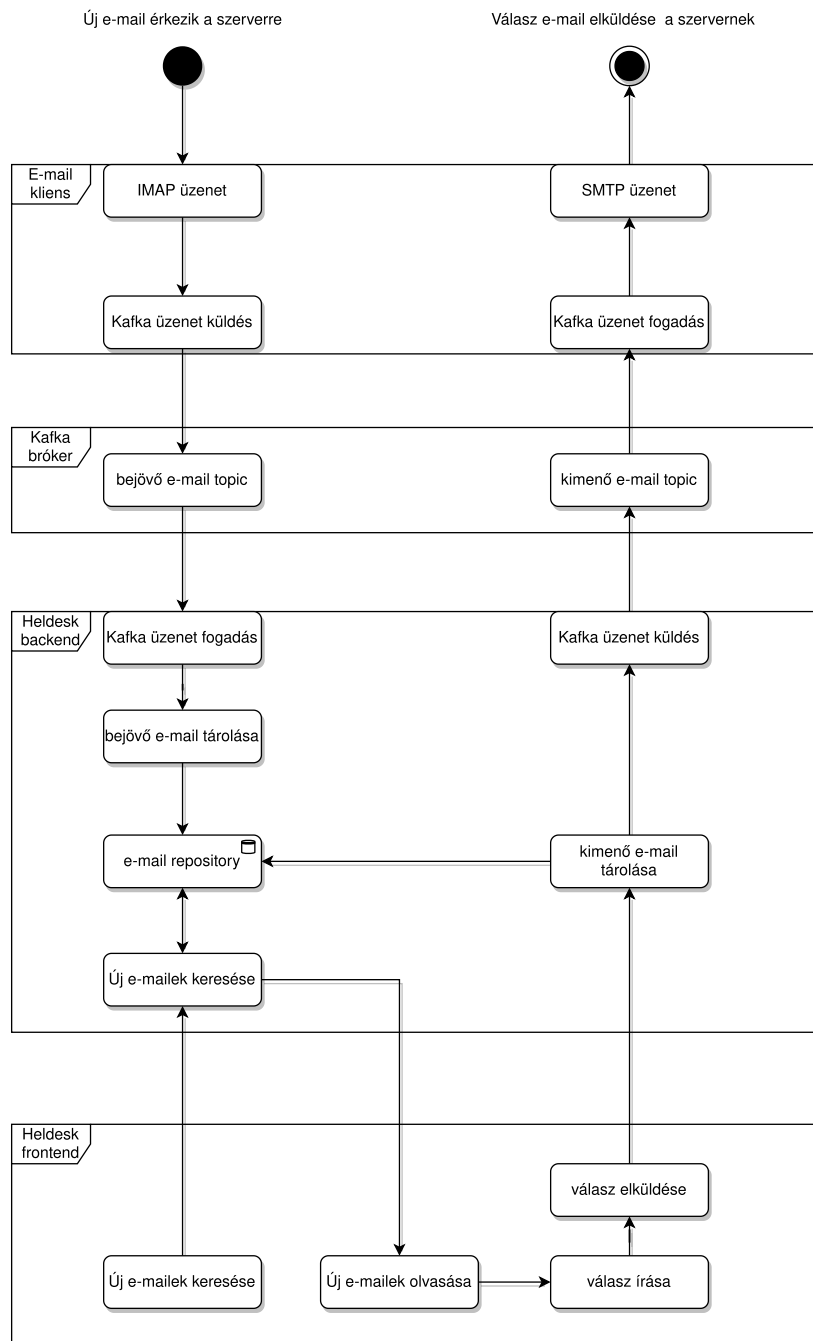
A könnyebb átláthatóság érdekében, a folyamatokat egy e-mail szemszögéből mutatom be a 3.3. ábrán. Az e-mail fogadása az alábbi felsorolás által leírt folyamat szerint történik.

1. Az e-mail kliens IMAP protokollon keresztül megkapja az új e-mailt.
2. Az e-mail kliens a bejövő e-mailt egy kafka üzenetként teszi közzé a bejövő e-mailek kafka *topic*-ban.
3. A bejövő e-mailek *topic*-ra feliratkozott helpdesk backend megkapja a kafka üzenetet.
4. A helpdesk backend eltárolja az új üzenetet az adatbázisban
5. A felhasználó a frontend segítségével lekérdezi az újonnan beérkezett e-maileket.
6. A helpdesk backend a kérésre elküldi az újonnan fogadott e-mailt.

Az e-mail küldése pedig az alábbi felsorolás szerint hajtodik végre.



1. A felhasználó az új e-mail elolvasása után a frontend segítségével megírja a választ.
2. A felhasználó elküldi a választ a helpdesk backendnek.
3. A helpdesk backend eltárolja az adatbázisba az új e-mailt, majd az e-mail szálnak megfelelő kimenő e-mail *topic*ba közzéteszi az új üzenetet.
4. Az e-mail cím specifikus kimenő e-mailek *topic*ra feliratkozott e-mail kliens megkapja a kafka üzenetet.
5. Az e-mail kliens SMTP protokollon keresztül elküldi az új e-mailt.



3.3. ábra. A bejövő és kimenő e-mail útja

Forrás: saját ábra

## 4. fejezet

# A tervezés eldöntendő kérdései

Ebben a fejezetben összegyűjtöttem a tervezés során felmerült kérdéseket, az azokra adott válaszaimat, és az egy-egy eszköz kiválasztása során figyelembe vett alternatívákat.

### 4.1. Webszerver

A megfelelő webszerver kiválasztása során a három legelterjedtebb[13] webszerveret – Apache, Nginx és az Internet Information Services – vettem számításba. A főbb szempontjaim a következők voltak:

- legyen nyílt forráskódú, vagy ingyenesen használható, ezzel minimalizálva a költségeket,
- lehetőleg minél kevesebb erőforrást használjon fel,
- úgy, hogy közben elegendő csak statikus tartalmat kiszolgálnia.

Az IIS<sup>1</sup> – a többi nyílt forráskódú alternatívával ellentétben – nem felel meg az első pontnak, mivel Windows NT licenc alá tartozik.

Az Nginx alapvetően jobb az Apache-nál a statikus oldalak, a párhuzamos lekérések kiszolgálásában [14], emiatt szívesen használják a két alkalmazást együtt. Ilyenkor az Nginx egy fordított proxyként kezeli az ügyfeleket és a statikus tartalmak kiszolgálását, a dinamikus tartalmakat pedig továbbítja az Apache szerver felé.

Mivel a webszervernek elegendő statikus tartalmat kiszolgáltatni – hiszen a helpdesk frontend egy egyszerű statikus HTML-oldallá fordul (lásd 2.7 pont) – így adja magát, hogy webszerverként az Nginx-et használjam.

---

<sup>1</sup>Internet Information Services

## 4.2. Adatbázis

A megfelelő relációs adatbázis kiválasztása során, a következő három alternatívát vettem számításba:

- Oracle DBMS,
- MySQL,
- és PostgreSQL.

Az Oracle adatbázist a drága licenc miatt nem tartom jó választásnak, helyette inkább egy nyílt forráskódú megoldást választanék.

Az alkalmazás működése szempontjából lényeges funkciók a MySQL és a PostgreSQL adatbázisban is elérhetőek.

A backend alkalmazásban az adatokat JPA-n keresztül kezelem, így – a megfelelő adatbázis kiválasztásában – a legfontosabb szempontnak a két adatbázis Hibernate-en keresztül elért teljesítményét tartom.

### 4.2.1. Teljesítményvizsgálat

Az ObjectDB által készített teljesítményvizsgálat[15] a saját – memóriában futtatott – adatbázisuk teljesítményéhez hasonlítja többi alkalmazás teljesítményét (4.1 egyenlet). Így a normalizált mérőszámok egymással összevethetőek.

$$\text{vizsgált rendszer mérőszáma} = 100 \cdot \frac{\text{vizsgált rendszer teljesítménye} [\text{művelet}/s]}{\text{ObjectDB teljesítménye} [\text{művelet}/s]} \quad (4.1)$$

A 4.1 táblázatban – a dokumentált eredményekből[15] – összegyűjtöttem a vizsgálat szempontjából mérvadó adatokat. Mindegyik mérés külön példányon, egyesével, egyenként 100 000 véletlenszerű entitás létrehozásával készült. A 4.1 egyenletből következik, hogy egy mérőszám minél magasabb értékű, a rendszer annál jobb teljesítményűnek számít.

A 4.1 táblázatban látszódik, hogy a PostgreSQL adatbázis a Hibernate implementációval átlagosan háromszor olyan jól teljesít, mint a MySQL adatbázis. Ám ha figyelembe vesszük, hogy a két leggyakrabban előforduló funkció az elsődleges kulcs, valamint szóeleji egyezés alapján keresés, akkor egyértelmű, hogy a PostgreSQL adatbázist érdemes választani.

Tesztesetek	MySQL adatbázis	PostgreSQL adatbázis
Új adat létrehozás	2,8	7,7
Keresés elsődleges kulcs alapján	5,2	7,0
Keresés szóeleji egyezés alapján	2,9	20,7
Meglévő adat módosítása	1,2	5,4
Meglévő törlése	1,2	6,6
Tesztesetek átlaga	2,7	9,1

4.1. táblázat. JPA teljesítményvizsgálata Hibernate implementációval és MySQL illetve PostgreSQL adatbázissal

### 4.3. JPA implementáció

A helpdesk backend által használt adatok kezelésére objektum-relációs leképzést használok, mert az ORM nagy mértékben leegyszerűsíti az adatbázis és az entitások egymásnak való megfeleltetését.

A számtalan – EclipseLink, Hibernate, Spring Data JPA, ... – ORM-et megvalósító JPA implementációk közül a Hibernate mellett döntöttem, mert

- PostgreSQL adatbázissal hatékonyan működik együtt
- és a JPA implementáción felül, az adatok auditálására szeretném használni a projekten.

Így a Hibernate használatával jelentős mennyiségű forráskód elkészítését és karbantartását lehet kiváltani, hiszen az adatok tárolásáért, módosításáért, verziókezeléséért mind a Hibernate a felelős.

A használatával kiváltott munka mennyisége bőven meghaladja a megismerésével és beállításával eltöltött időt.

### 4.4. Adatbázis migrációs eszköz

Az eszköz célja az adatbázis verziókövetésének javítása. A megvalósítás alapja, hogy ha az adatbázis változásait egy szöveges dokumentumban tároljuk az alkalmazás forráskódjával együtt, akkor a verziókövető rendszerrel nyomon lehet követni a kód aktuális állapotához tartozó adatbázis-struktúrát is. Összesen két eszközt találtam és vizsgáltam meg: a Liquibase-t és a Flywayt.

### 4.4.1. Liquibase

A Liquibase egy – adatbázis sémaváltozások nyilvántartására és alkalmazására létrehozott – nyílt forráskódú adatbázis-független könyvtár.

A sémaváltozásokat tudja kezelni SQL, XML, JSON és YAML formátumban is. A nem SQL formátumban tárolt változásokból adatbázis-specifikus SQL-t tud generálni, és szükség esetén akár automatikusan visszagörgetni is képes azokat.

A *changelog*ban a változások könnyedén csoportosíthatóak, sorrendjük módosítható végrehajtásuk feltételéhez köthető.

### 4.4.2. Flyway

A Flyway egy nyílt forráskódú, SQL-alapú, egyszerű, adatbázis nyilvántartó könyvtár.

A változások csak SQL formátumban hozhatóak létre, és sorrendjük az őket tartalmazó állomány sorszámától függ. A migráció során figyelembe vett állományok neveinek meg kell felelniük a Flyway által előírt szigorú szabályoknak.

### 4.4.3. Felhasznált eszköz

A Liquibase – a Flyway funkcióinak megvalósítása mellett – sokkal több, szélesebb körű felhasználást tesz lehetővé. A visszagörgethetőség és a változások egyszerűbb csoportosíthatósága miatt célszerű a Liquibase-t választani.

## 4.5. Monitorozás

Hogy megtaláljam a legmegfelelőbb monitorozó eszközt, összevetettem a három leginkább használt alternatívát, a Prometheus, Grafanát és a Graphite-ot.

### 4.5.1. Prometheus

Prometheus egy nyílt forráskódú monitorozó eszköz. Kifejezetten alkalmas a metrikák idősoros tárolására, gyűjtésére és megjelenítésére.

A Prometheus által létrehozott és használt PromQL<sup>2</sup> lekérdező nyelv segítségével könnyen lehet az adatokból táblázatot vagy grafikont készíteni. A PromQL-ben létrehozott kifejezések alkalmasak riasztások létrehozására is, ilyenkor a hibásnak tekintett eseményről – a Prometheus *Alertmanager*-én keresztül – képes a riasztást kiváltó helyzet elhárításában érintetteket értesíteni.

---

<sup>2</sup>Prometheus Query Language

A megjelenítésre használt *Console template* meglehetősen sok és szerteágazó funkcióval rendelkezik. A rendelkezésre álló átfogó dokumentáció ellenére is jelentős időt vesz igénybe a legalapvetőbb grafikonok elkészítése.

A Prometheus HTTP-n keresztül tudja a megfigyelt rendszerek állapotát szabályos időintervallumokban lekérdezni. Az integrációt nagyban segíti, hogy összeköthető *service discovery* szerverekkel.

### 4.5.2. Grafana

Grafana egy nyílt forráskódú adatelemző, megjelenítő és monitorozó eszköz.

Nem tárol vagy gyűjt adatot, de támogat többféle – Graphite, Prometheus, Influx DB, Elasticsearch, MySQL, PostgreSQL – adatforrással való kapcsolatot. *Plugin*okon keresztül továbbá támogatja a felhő alapú AWS Cloudwatch és OpenStack Gnocchi adatforrást is.

A Grafana legnagyobb előnye a *dashboard*-jaiban rejlik. A felületen – mint egy műszerfalon – egymás mellett rendszerezve helyezkednek el a különböző paneleken ábrázolt metrikák. Számtalan előre elkészített típusú panelből lehet válogatni, többek között elérhetőek hisztogramok, grafikonok, hőtérképek, táblázatok, riasztások, RSS- és naplóbejegyzés-olvasó felületek.

A Grafana oldalán ezen kívül számos előre elkészített és szabadon felhasználható *dashboard* érhető el. A leggyakrabban használt alkalmazások metrikái újra használhatóan és szerkeszthetően hozzáférhetőek. Így a fejlesztést jelentősen leegyszerűsítve, elegendő csupán az alkalmazás specifikus paneleket letölteni és testre szabni.

### 4.5.3. Graphite

A Graphite idősoros tárolására és megjelenítésére alkalmas nyílt forráskódú monitorozó eszköz. Összesen két feladatot lát el, a Whisper könyvtárral tárolja a neki küldött adatokat, valamint megjeleníti a tárolt adatot a webes felületén.

Az adatok tárolása nem olyan kifinomult, mint a Prometheuson: passzívan képes csak fogadni az adatokat és nem rendelkezik a PromQL-hez hasonló lekérdező nyelvel sem.

A beépített felülete – körülbelül a Prometheus-éhoz hasonlóan – messze nem olyan részletes, mint a Grafanáé. Egyszerű grafikonok és táblázatok ugyanúgy elkészíthetőek vele, de ezen kívül támogatja még *dashboard*ok létrehozását.

#### 4.5.4. Megvalósítás

A fenti ismeretek fényében egy hibrid megoldást választottam (lásd az [5.1.3](#) pontban). A Grafana könnyen integrálható a Prometheussal, így egyszerre tudom kihasználni a Grafana kifinomult megjelenítését és szerkeszthető *dashboard*jait, valamint a Prometheus PromSQL-jét, kiforrott adatgyűjtési és tárolási módszereit.



## 5. fejezet

# Implementáció

Ebben fejezetben külön-külön bemutatom a helpdesk alkalmazást felépítő komponenseket. Kiemelem a komponensek által megvalósított funkciókat és a megvalósítás szempontjából fontos részleteket.

### 5.1. Mikroszerviz infrastruktúra

A mikroszerviz specifikus funkciók megvalósításáért az itt bemutatott eszközök felelősek.

#### 5.1.1. Nginx

Az Nginx-nek három különböző szerepe van:

- a helpdesk frontend alkalmazásszervereként működik (2.7 pont),
- routingt valósít meg, rajta keresztül érhető el a helpdesk backend és a Keycloak szerviz,
- HTTP cache-ként működik a frontend és a backend között.

A loadbalancer funkcionalitás – mivel a docker azt natívan támogatja, így – a docker round-robin DNS-en (5.1.2) keresztül valósul meg.

#### 5.1.2. Docker konténerizáció

Az alkalmazás összes szolgáltatása saját docker konténerben fut. A docker konfigurációs leírása a `docker-compose.yml` állományban van. A `docker-compose` parancs ez alapján indítja el az alkalmazást, tölti le a szükséges *image*-eket, hozza létre a saját alhálózatát, valósítja meg a hálózaton belüli DNS-funkciót.

A konténerek skálázása is a dockeren keresztül (`docker-compose --scale`) valósul meg.

### 5.1.3. Metrikák

A 4.5 pontnak megfelelően a springes alkalmazásaim egy-egy HTTP endpointon keresztül érhetőek el a Prometheus számára (`/actuator/prometheus`) és induláskor beregisztrálják magukat az Eureka<sup>1</sup> szerverbe.

A Prometheus az Eureka-n keresztül találja meg az instance-eket, és 15 másodpercenként összegyűjti a metrikákat. Az alkalmazások információt küldenek a Kafka konnektoraikról, REST interfészeikről és az adatbázis kapcsolataikról<sup>2</sup>.

A Prometheus által összegyűjtött adatokat Grafanában létrehozott – Spring Boot és JVM metrikákat tartalmazó – *dashboard*okon ábrázolom.

## 5.2. E-mail kliens

Az e-mail kliens szerepe az üzenetek küldése és fogadása egy meghatározott e-mail címről. Feladata a külső protokollok leválasztása az alkalmazásról. Irányítja és karbantartja az IMAP és SMTP szerverrel való kapcsolatot.

Az 5.1. ábrán látható a két irányú kommunikáció megvalósulása:

- az IMAP-on keresztül fogadott e-mailt az `email.in.v1.pub` kafka topicba írja,
- a saját – e-mail cím specifikus – topic-jából kiolvassa az üzenetet és továbbítja az SMTP szerver felé.

### 5.2.1. E-mail szabvány

Az elküldött üzenetek megfelelnek az *rfc5322* szabványnak, különös tekintettel a 3.6.4. pontban [16] meghatározott mezőkre:

**Message-ID** egy globálisan egyedi azonosító ami egyértelműen azonosítja az üzenetet,

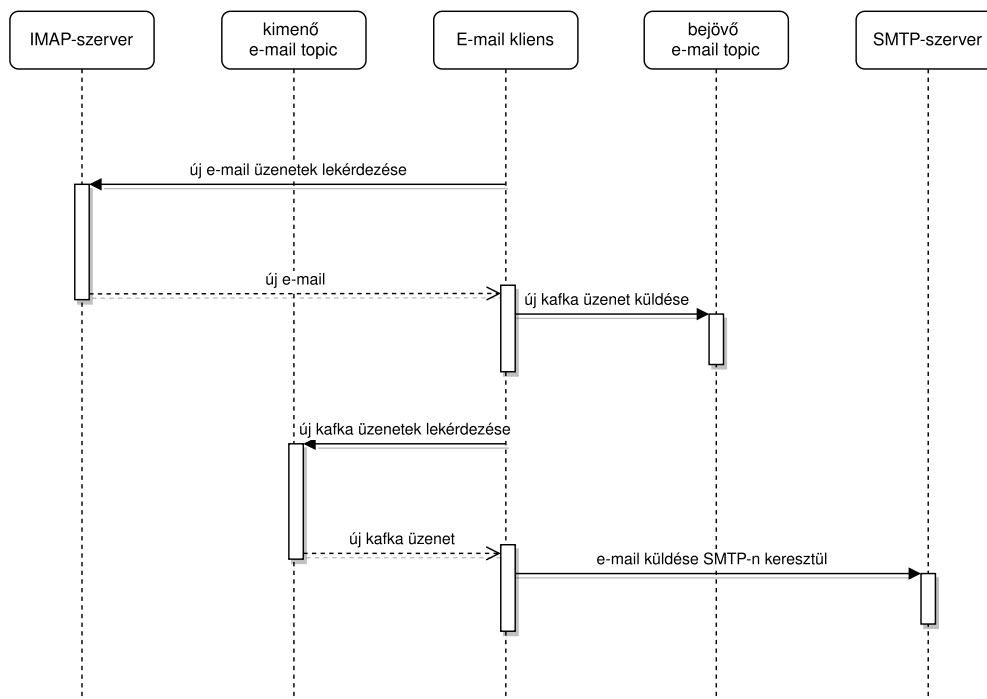
**In-Reply-To** válasz esetén értéke eredeti üzenet **Message-ID**-ja,

**References** azonosítja az üzenet szálat, értéke az eredeti üzenetek **Message-ID**-jai vesszővel elválasztva.

---

<sup>1</sup>Az Eureka a Netflix által fejlesztett *discovery server*. Feladata az összes kliens port és ip adatának nyilkvántartása.

<sup>2</sup>HikariCP-t használok JDBC kapcsolathoz



5.1. ábra. E-mail kliens szekvencia diagramja

Forrás: saját ábra

### 5.2.2. Üzleti funkciók megvalósítása

Az e-mail kliens fontosabb Java csomagjain keresztül bemutatom a forráskód felépítését, így betekintést nyújtva a megvalósított funkciókba.

**imap** csomag tartalmazza az IMAP protokoll megvalósításához szükséges kódrészleteket. A kapcsolat felépítéséért az `ImapMailReceiver`, míg az e-mailek fogadásáért az `EmailReceiver` osztály felelős.

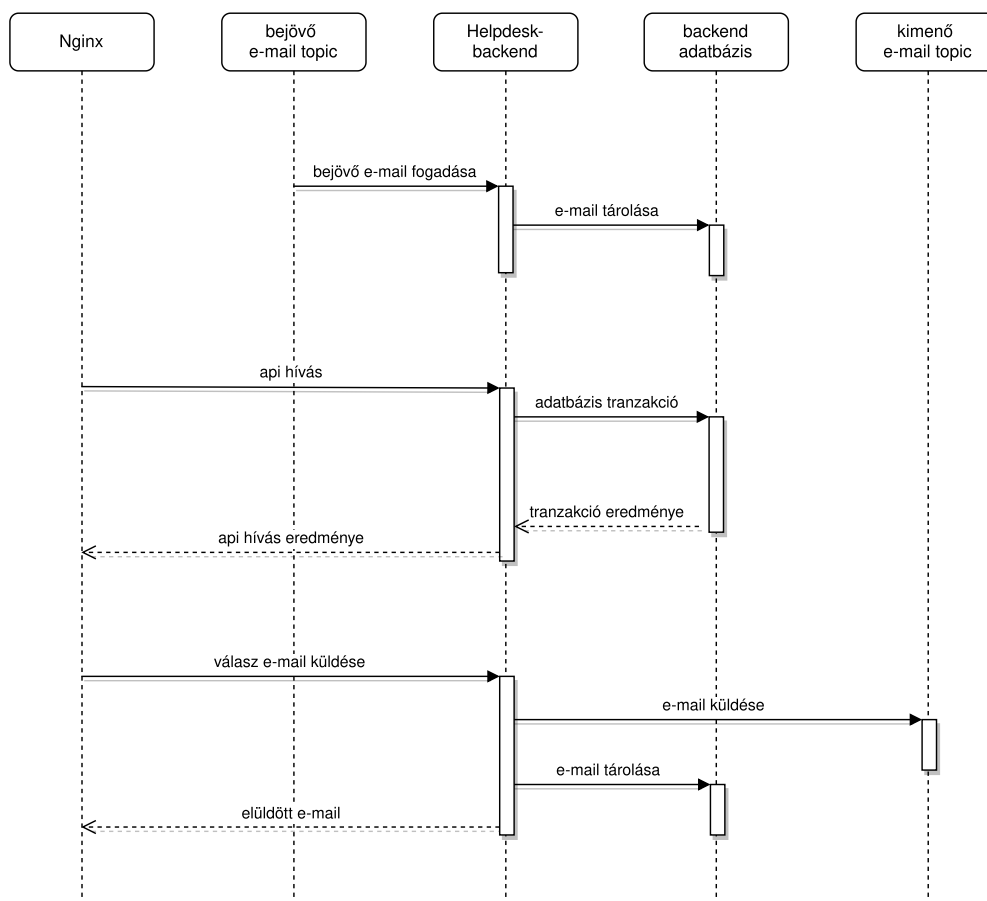
**kafka** csomagba tartozó osztályok a `springframework.kafka` csomagja segítségével Kafka üzenetet küldenek (`EmailKafkaProducer` osztály) és fogadnak (`OutgoingEmailObserver` osztály).

**smtp** csomagba tartozó `EmailSender` osztály a `springframework.mail` segítségével üzenetet küld az SMTP-szerveren keresztül.

## 5.3. Helpdesk backend

A backend felelős az e-mail szálakkal kapcsolatos üzleti feladatok ellátásáért. Az 5.2. ábrán láthatóak a helpdesk backend funkciói:

- fogadja és adatbázisban tárolja az `email.in.v1.pub` kafka topic-ból érkező e-maileket,
- kiszolgálja a bejelentkezett felhasználók Nginx-en keresztül érkező kéréseit,
- válasz e-mail küldése esetén, adatbázisban tárolja, és a megfelelő kafka topic-ba írja az elküldendő üzeneteket,
- tárolja az e-mail szálakkal kapcsolatos adatokat.



5.2. ábra. Helpdesk backend szekvencia diagramja

Forrás: saját ábra

### 5.3.1. Spring Boot

A forráskód a Spring Boot (2.8 pont) keretrendszerrel készült. Az elérhető modulok közül a *data-jpa*-t az adatbázis **repository**-jaihoz, a *security*-t a keycloak integrációhoz, a *web*-et a **rest** **controllerek**hez, a *micrometer*-t és az *actuator*-t a metrikák elkészítéséhez használtam.

### 5.3.2. Adatbázis

A Spring kezeli – HikariCP-n keresztül – a PostgreSQL adatbázishoz való kapcsolódást. Az adatok kezelését Hibernate-en (4.3 pont) keresztül, az adatbázis verziókövetését Liquibase-en (6.6.1 pont) keresztül valósítom meg.

Az e-mail szálak audit információinak és verzióinak követésére a Hibernate Envers (6.6.2 pont) eszközt használom. Az Envers a neki létrehozott táblában automatikusan követi az annotációval megjelölt entitások állapotát.

### 5.3.3. Pessimista konkurenciakézelés

Pessimista konkurenciakézelésre jó példával szolgál a Liquibase (6.6.1 pont) működése.

Minden indítás alkalmával a Liquibase – az adatbázis módosításának befejezéséig – zárolja a *databasechangelock* táblát. Így – a várakozás miatt – egyszerre mindig maximum egy Liquibase példány tud elindulni és módosításokat végrehajtani.

### 5.3.4. Optimista konkurenciakézelés

A 2.4 pontban ismertetett optimista konkurenciakézelést az e-mail szálak módosítása során valósítja meg a backend.

A frontend kérésére egy verziószámmal ellátott e-mail szálát küld a backend. Ezt a HTTP-protokollnak megfelelő **eTag**-et a frontend megőrzi, majd a módosítások elvégzését követően – mint **if-match** paraméter – visszaküldi a módosítási kérésével együtt.

A backend összehasonlítja a módosítani kívánt erőforrás verziószámát a kérésben érkezett **if-match** verziószámmal. Ha a két szám egyezik, akkor végrehajtja a változásokat, és az erőforrás új állapota új verziószámot kap.

Ha a két verzió nem egyezik – ami csak úgy történhet meg, ha valaki más időközben módosította a kérdéses adatot – akkor a tranzakció nem hajtódik végre, és a kliens egy HTTP **Conflict** hibaüzenettel értesül a történekről. A felhasználó ilyenkor az oldal frissítése után megvizsgálja az aktuális állapotot és – amennyiben a módosításaira még mindig szükség van – újból kezdi a folyamatot.

### 5.3.5. Üzleti funkciók megvalósítása

A backend által elvégzett feladatok bemutatása érdekében – a legfontosabb csomagok ismertetésével – áttekintést adok a forráskód felépítéséről.

**kafka** csomagba a `springframework.kafka` csomagot használó, Kafka üzenetek kezelését végző osztályok tartoznak. Külön osztály foglalkozik a felhasználók (`UserObserver`) és az e-mailek (`IncomingEmailObserver`) fogadásával, és külön az e-mailek küldésével (`EmailKafkaProducer`).

**security** a csomag feladata az alkalmazás HTTP protokollal kapcsolatos biztonsági paramétereinek beállítása. Itt lehet többek között a *referrerPolicy*-t, *CORS*-t és a jogosultság kezeléshez használt *JSON Web Token* (5.5.2 pont) adatait megadni.

**service** a csomagban az üzleti feladatok ellátásért felelős **service** interfészek és az azokat megvalósító osztályok találhatók. Feladatuk az adatok üzleti szempontból lényeges tulajdonságainak ellenőrzése, valamint az üzleti logika végrehajtásához szükséges technikai feladatok delegálása.

**web.rest** csomagban találhatóak a `RestController`-k. A Spring ezeknek az osztályoknak továbbítja a REST kéréseket. Az osztályok fő célja a HTTP specifikus feladatok leválasztása az alkalmazásról, továbbá az üzleti igények továbbítása a megfelelő **service**-ek számára.

Az **email-kliens**, a **keycloak-plugin** (lásd az 5.7 pont), és a **helpdesk-backend** által közösen használt kódot – a moduláris felépítés érdekében – külön, a **helpdesk-domain** maven projektben helyeztem el. A közösen használt modul legfontosabb csomagjait szintén itt mutatom be.

**dto** csomag nevét a *Data Transfer Object* elnevezésről kapta. A csomag olyan érték osztályokat tartalmaz, amiknek nincs önálló viselkedésük, egyetlen feladatuk az adatok reprezentálása. Ezeknek a **dto**-k segítségével küldenek és fogadnak adatot a `RestController`-k.

**entity** csomagban találhatóak a JPA entitások. Minden entitás osztály egy adatbázis-tábla Java oldali leképezése, és minden entitás objektum a tábla egy sorának a megfelelője. Az entitások állapotainak kezeléséért a Hibernate felel.

**repository** csomagban a Spring `JpaRepository` interfészből öröklődő **repository** interfészek találhatók. Feladatuk az adatbázisnak küldött utasítások reprezentálása. A szükséges SQL-t az interfészt implementáló Spring osztály hozza létre a metódus annotációjából vagy elnevezéséből [17].

**mapper** csomagba tartozó osztályok feladata az adatok különböző reprezentációk – **dto**, **entity**, **avro** – közötti leképezés megvalósítása. A **@Mapper** annotációival ellátott interfészek implementációját a Mapstruct (2.3 pont) generálja fordítási időben.

**avro** a Kafka üzenetekhez használt avro (lásd 2.6 pont) üzenetek Java reprezentációit a `hu.gdf.balazsbole.kafka` csomag tartalmazza. A csomagban található osztályokat fordítási időben az adatstruktúrát leíró `avdl` fájlból hozza létre az `avro-maven-plugin`.

### 5.3.6. Egyéb eszközök

A **DTO**-kban és az **entity**-kben használt getter és setter metódusok generálását a Lombok (2.3 pont) segítségével végzem. A REST **endpoint**ok dokumentációját Swagger segítségével generálom. A Swagger a felannotált **dto** osztályokból és **RestController** metódusokból szabványos OpenApi dokumentációt készít. A dokumentációt az A függelékben csatoltam a dolgozatomhoz.

## 5.4. Helpdesk frontend

A frontend az e-mailek és e-mail szálakkal összefüggő üzleti feladatok megjelenítéséért felelős. A felhasználók jogosultság ellenőrzését végzi el, a bejelentkeztetésüket átirányítja a Keycloak szervernek.

### 5.4.1. Kommunikáció a backenddel

A backenddel való kommunikáció HTTP protokollon keresztül zajlik, a szükséges **service** osztályokat az OpenApi dokumentációból (5.3.6) a `swagger angular generator` hozza létre.

Az aszinkron HTTP hívásokat az NgRx könyvtár alakítja adatfolyamokká. Az így *Observable*-ként kezelt események már támogatják a stream műveleteket, megkönnyítik a filterezhetőséget és az egységes hibakezelést.

Az NgRx használatával továbbá elkerülhetőek az aszinkron hívások mellékhatásai, és egy globális, alkalmazás szintű belső állapot hozható létre.

### 5.4.2. Komponensek

Az egységes megjelenés és az ismerős kinézet miatt a komponenseim alapján az Angular Material UI könyvtárat választottam. A könyvtár népszerű az Angular fej-

lesztők körében, mert a leggyakrabban előforduló felhasználói igényekre elérhető benne kész, könnyen használható megoldás.

A válasz e-mail létrehozására a nyílt forráskódú Quill szövegszerkesztőt használtam, mert egyszerűen beilleszthető az Angular környezetbe, és a felhasználó számára intuitív kezelőfelülettel rendelkezik.

### 5.4.3. Üzleti funkciók megvalósítása

Az üzleti funkciók megvalósításának bemutatása érdekében ismertetem a helpdesk frontend menüjét és a menüpontok által megvalósított üzleti funkciókat.

**Reply** a menüpont minden felhasználónak elérhető. Segítségével az OPEN státuszú üzenetszálak elolvashatóak – az üzenetek olvasottnak és olvasatlannak jelölhetőek – és megválaszolhatóak. A válasz e-mail küldése során az üzenetszál új állapotra állítható.

**Edit threads** a menüpont minden felhasználónak elérhető. Segítségével a felhasználóhoz tartozó OPEN, RESOLVED és CLARIFICATION státuszú e-mail szálak szerkeszthetőek. Az üzenetszálak tulajdonos, státusz, valamint – adminisztrátor jogkörrel rendelkező felhasználó esetén – sor tulajdonsága változtatható meg.

**History** szintén minden felhasználó számára elérhető menüpont. Segítségével – táblázatba rendezve – lekérdezhetőek és visszakereshetőek a bejelentkezett felhasználóval kapcsolatba került e-mail szálak tulajdonságainak korábbi változásai.

**Change queue** csak az adminisztrátor felhasználónak elérhető menüpont. A felhasználó aktuális sorához tartozó CHANGE\_QUEUE státuszú e-mail szálak érhetőek el és módosíthatóak vele. A kiválasztott e-mail szál az **Edit threads** menüponthoz hasonlóan szerkeszthető.

**User account** minden felhasználónak elérhető. A menüpont a felhasználó Keycloak profil oldalára navigál. Az oldalon megváltoztathatóak a bejelentkezett felhasználó személyes adatai és jelszava, megtekinthetőek a jogosultságai és azonosítással kapcsolatos eseményei, valamint érvényteleníthetőek a munkamenetei.

**Choose queue** a menüpont adminisztrátor felhasználónak többször is, minden más felhasználónak csak egyszer érhető el. Segítségével – az aktív felhasználót – a rendelkezésre álló sorok valamelyikéhez lehet rendelni.



**Login service** minden felhasználónak megjelenő, de csak az konfigurációs jogkörrel rendelkező felhasználónak elérhető menüpont. Az elérhető funkciók leírását az 5.5 pont tartalmazza.

**Spring metrics** az 5.1.3 pontban leírt Spring Boot metrikákat ábrázoló Grafana oldalra navigáló menüpont.

**Backend metrics** hasonlóan a Grafan oldalra navigáló – a JVM metrikákat tartalmazó – menüpont.

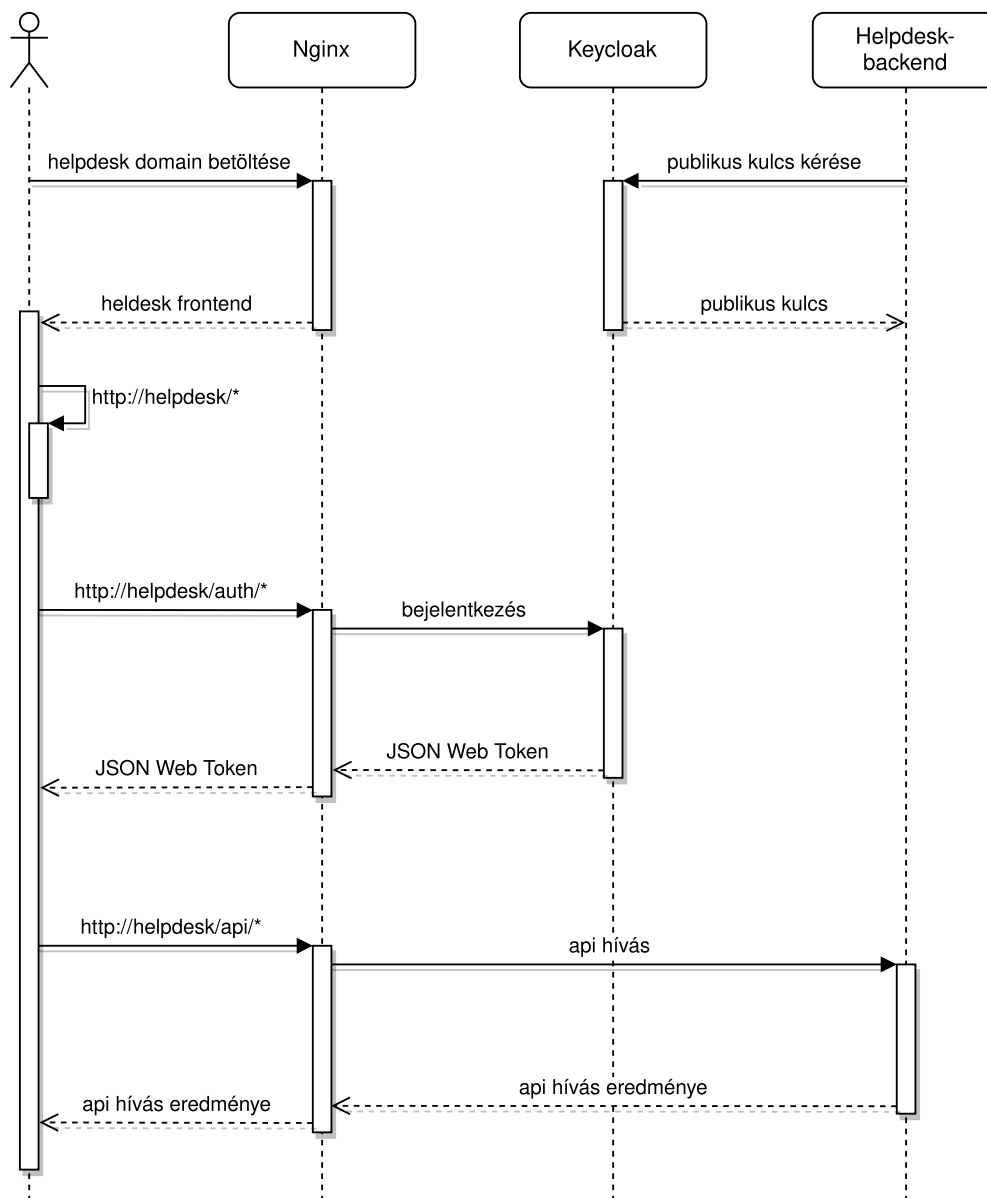
**Eureka service discovery** a 6.8 pontban részletesen bemutatott Eureka szolgáltatás oldalára navigáló menüpont.

**Kafka messages** mindenkinek megjelenő, a Kafka Topics UI oldalára navigáló menüpont. Részletes bemutatása a 6.7 pontban olvasható.

#### 5.4.4. Futtatási környezet

A kész program egy egyszerű HTML, CSS és JavaScript állománnyá fordul. A körülbelül 1,5 MB-nyi forráskódot elegendő a böngészőbe egyszer letölteni, onnantól a program a kliens oldalon fut (lásd 5.3 ábra). A backend felé induló REST kéréseket a webservert (5.1.1) osztja szét a rendelkezésre álló példányok között.

A frontend működését, és függőségeit az 5.3. ábra tartalmazza.



5.3. ábra. Helpdesk frontend szekvencia diagramja

Forrás: saját ábra

## 5.5. Keycloak

A Keycloak egy nyílt forráskódú jogosultság- és hozzáférés-kezelő. Támogatja az LDAP-ot, SSO-t és a kétlépcsős azonosítást [18].

A helpdesk alkalmazásban feladata a felhasználók azonosítása, és adataiknak nyilvántartása. Különálló mikroszervizként, saját adatbázissal rendelkezik.

Adminisztrátor felülete segítségével nyomon követhető a különböző autentikációhoz köthető események, szerkeszthetők az aktuálisan érvényes szerepkörök, és – hibakezelési céllal – megszemélyesíthetők a felhasználók.

### 5.5.1. Jogosultságkezelés

A jogosultságokat két eltérő területre osztottam fel. A `master realm` a regisztrációért és a jogkörök kiosztásáért, míg a `helpdesk realm` az alkalmazás funkcionális (1.1.3) feladatáért felelős.

A `helpdesk realm`on belül további két jogkört különböztetek meg. Az `admin_user` szerepbe tartozó felhasználók képesek más e-mail szárait is kezelni, míg a csupán `regular_user` jogkörbe tartozóak csak a saját e-mail szálaikhoz férhetnek hozzá.

### 5.5.2. JSON Web Token

A jogosultságkezelés technikai alapját az *rfc7519*-es szabványban [19] leírt JSON Web Token (JWT) adja.

A Keycloak szervere által digitálisan aláírt *access\_token* (lásd 5.5.3 pont) tartalmazza a felhasználó jogosultságait. A frontend minden HTTP lekérdezéshez csatolja ezt a Keycloaktól kapott azonosítót. A backend hitelesíti a token a Keycloak publikus kulcsával (lásd 5.3 ábra), és a megfelelő jogosultság megléte esetén engedélyezi a hozzáférést az erőforráshoz.

### 5.5.3. OAuth 2.0 keretrendszer

Mivel a Keycloak az *emphrfc7519*-es szabványban [20] leírtaknak megfelelően állítja ki a jogosultságkezeléshez használt JSON Web Token, ezért Helpdesk Frontendnek ismernie kell, és a szabványban meghatározott módon szükséges az *access\_token* kérényeznie.

A Helpdesk Frontend az itt ismertetett módon kapja meg az *access\_token*. A folyamat, az OAuth 2.0 jogosultságkezelő keretrendszernek megfelelően valósul meg.

1. A Helpdesk Frontend a *Login* gombra kattintva átirányítja a felhasználót a Keycloak regisztrációt és bejelentkezést kezelő oldalára. Az alkalmazás a kéréshez hozzáfűzi a Keycloak rendszerben tárolt kliensre jellemző azonosítót, belső állapotát és a – harmadik lépésben – átirányításra használt webcímet.
2. A Keycloak szerver megvizsgálja az egyes pontban kapott adatokat. Amennyiben a kérés a kliensazonosítónak megfelelő webcímről érkezett, az adatok pedig a klienshez regisztrált lehetséges értékekkel megegyeznek, akkor elkéri a felhasználó nevét és jelszavát. A szolgáltatás eldönti, hogy a megadott felhasználónév és jelszó alapján engedélyezhető-e a hozzáférés.
3. Amennyiben a hozzáférés engedélyezhető, a szolgáltatás átirányítja a felhasználót az első pontban megadott webcímre, a címhez hozzáfűzi a négyes pontban használt hozzáférési kódot.  
  
A *Login* gomb a felhasználót a Frontend üdvözlő oldalára irányítja vissza.
4. A Frontend a Keycloak szervertől HTTP POST módszerrel kérvényezi az *access\_token*. A lekérdezéssel elküldi az URL paraméterből kiolvasott hozzáférési kódot, a kliens azonosítót, és az egyes pontban használt átirányítási címet.
5. A Keycloak ellenőrzi a hozzáférési kódot és megbizonyosodik arról, hogy az átirányítási cím megegyezik-e a hármas pontban használt címmel. Sikeres ellenőrzés után visszaküldi az *access\_token* és a *refresh\_token*
6. Az *access\_token* csupán 10 percig használható. Az idő lejártá után a Frontendnek új *access\_token* kell igényelnie az előző pontban kapott *refresh\_token* segítségével.

A fent ismertetett folyamattal elérhető, hogy

- az azonosításra használt érzékeny adatok – az *access\_token* és a *refresh\_token* – védetten, csupán a POST módszer paramétereiként elérhetőek,
- a Helpdesk Frontendnek nem szükséges a felhasználó jelszavát kezelnie
- és bejelentkezés illetve a regisztráció folyamata aszinkron módon történhet.

## 5.6. Kafka

Annak érdekében, hogy teljesen elválasszam egymástól az e-mail klienst és a helpdesk backendet, a bejövő és kimenő e-mailek Kafka *topic*-okon (2.6 pont) mennek ke-

resztül. A szeparációval függetlenné teszem egymástól a két rendszer működését, ami lehetővé teszi az eltérő igénybevételnek (1.2.2 pont) megfelelő skálázhatóságot.

Ugyanígy, a funkciók szeparálása (5.7) miatt a felhasználók adatai egy külön kafka topicban érhetőek el. Bármelyik mikroszerviznek szüksége lenne valamilyen felhasználóval kapcsolatos információra, azokat a topic végigolvasásával megkaphatja.

## 5.7. Helpdesk backend és a Keycloak elkülönítése

A felhasználók adataiért a Keycloak (5.5), az e-mail szálakért pedig a backend (5.3) felelős. Az üzleti igény megköveteli hogy a felhasználók e-mail sorokhoz, és az e-mail szálak felhasználókhoz legyenek rendelve. A helpdesk backendnek éppen ezért tárolnia kell a fennálló kapcsolatokat.

A felhasználók a Keycloak felületén keresztül tudnak regisztrálni, és a személyes adataikat kezelni. A Keycloak által generált JSON Web Token (5.5.2) tartalmazza a felhasználók egyedi azonosítóját, a backendnek ezen az azonosítón keresztül kell a felhasználókat nyilvántartania és kiszolgálnia.

A felhasználók regisztrációja és adatainak változása – a 2.5 pontban megismert CQRS útnak megfelelően – a `user.v1.pub` kafka (5.6) topicban követhetőek nyomon.

A Keycloak Kafka integrációjának céljából hoztam létre a `keycloak-plugin` (6.1 ábra) maven modult. A Keycloak eseményfigyelőként működő plugin, a megfigyelt eseményekről kafka üzenetet küld a kijelölt topicba. A helpdesk backend – a topic üzeneteit olvasva – tartja karban a `users` táblát (3.2, 6.3 ábra). Így a helpdesk backend a felhasználókról mindig aktuális információval rendelkezik.

## 6. fejezet

# Alkalmazás bemutatása

A helpdesk alkalmazás az 1. fejezetben leírtaknak megfelelően szolgál ki három különböző e-mail címet:

- a `generic` sorhoz tarozó [helpdesk.gdf@yandex.com](mailto:helpdesk.gdf@yandex.com)-ot,
- a `travel` sorhoz tarozó [helpdesk.gdf.travel@yandex.com](mailto:helpdesk.gdf.travel@yandex.com)-ot,
- és a `theater` sorhoz tarozó [h.gdf.theater@gmx.com](mailto:h.gdf.theater@gmx.com)-ot.

### 6.1. Alkalmazás elindítása

Az alkalmazás a `start.sh` bash scripttel indítható el. A script két dolgot csinál:

1. a `docker-compose` paranccsal elindítja a docker *containereket* (5.1.2 pont),
2. `helpdesk` domain névvel hozzáadja az Nginx (5.1.1 pont) IP címét a `/etc/hosts` állományhoz.

A script indítása után a helpdesk alkalmazás elérhető a <http://helpdesk> domain alatt.

### 6.2. Virtuális gép

Annak érdekében, hogy a létrehozott alkalmazást könnyedén ki lehessen próbálni, egy virtuális számítógépre letöltöttem az alkalmazás 1.0.0 verzióját. A virtuális számítógép elérhető a DVD mellékleten, a [github release](#)-ek között, vagy közvetlenül is letölthető a:

<https://www.dropbox.com/sh/i6ldjowdgo0gupi/AAD9okdo-xehshXTdVg9FoPba> cím-ről.

Annak érdekében, hogy az elkészült ova<sup>1</sup> állomány minél kevesebb helyet használjon, egy minimális rendszerigényű *Ubuntu server*t telepítettem. A virtuális gépre ezen kívül még a *xubuntu-core*, *docker* és a *Firefox* csomagokat telepítettem. A helpdesk alkalmazás forráskódját *git* helyett böngészőn keresztül töltöttem le.

A virtuális gépbe a *dxqrpj* felhasználóval és jelszóval lehet bejelentkezni. Bejelentkezés után a háttérben automatikusan elindul a helpdesk alkalmazás, így a 6.1 pontban említett *start.sh* bash script kézi futtatása nem szükséges.

Az alkalmazás elindítása, a virtuális számítógép szűk erőforrásai miatt, a natív indításhoz képest akár hatszor annyi időt is igénybe vehet:

**natívan** futtatva, az én számítógépemen az alkalmazás indítása – a konténerek közötti függőségek miatt – 1 perc 10 másodpercet vesz igénybe;

**virtualizálva, az első indításkor** a *start.sh* bash script letölti az összes szükséges (5,3 GB-nyi) *docker image*-et, ez a folyamat az én gépemen 5 perc 20 másodperccel növelte az alkalmazás első indításának idejét;

**virtualizálva, második indításkor**, az én gépemen az alkalmazás 6 perc 50 másodperc alatt elindul.

A konténerek állapotáról, és az indítás óta eltelt időről, *docker ps* és a *docker ps -a* parancs nyújt információt.

A helpdesk alkalmazás elindulása után, – a <http://helpdesk> címen – a Firefox web-böngészővel elérhető. A böngésző három felhasználót tárol:

**admin** felhasználót, aki az *admin\_user* jogkörrel (5.5.1 pont) rendelkezik, és a *generic* sorhoz tartozik;

**test** felhasználót, aki a *regular\_user* jogkörrel (5.5.1 pont) rendelkezik, és a *generic* sorhoz tartozik;

**testTheater** felhasználót, aki szintén a *regular\_user* jogkörrel rendelkezik, viszont a *theater* sorhoz tartozik.

A felhasználók bejelentkezése után, az 5.4.3 pontban összegyűjtött funkciók elérhetővé válnak. Az alkalmazás a már említett három – *generic*, *travel* és *theater* – sorhoz tartozó e-mail címről fogad és küld üzeneteket. Az indítás után nem sokkal, az alkalmazásban megjelennek az e-mail címekre – korábban, tesztelési és bemutatási céllal – küldött teszt üzenetek.

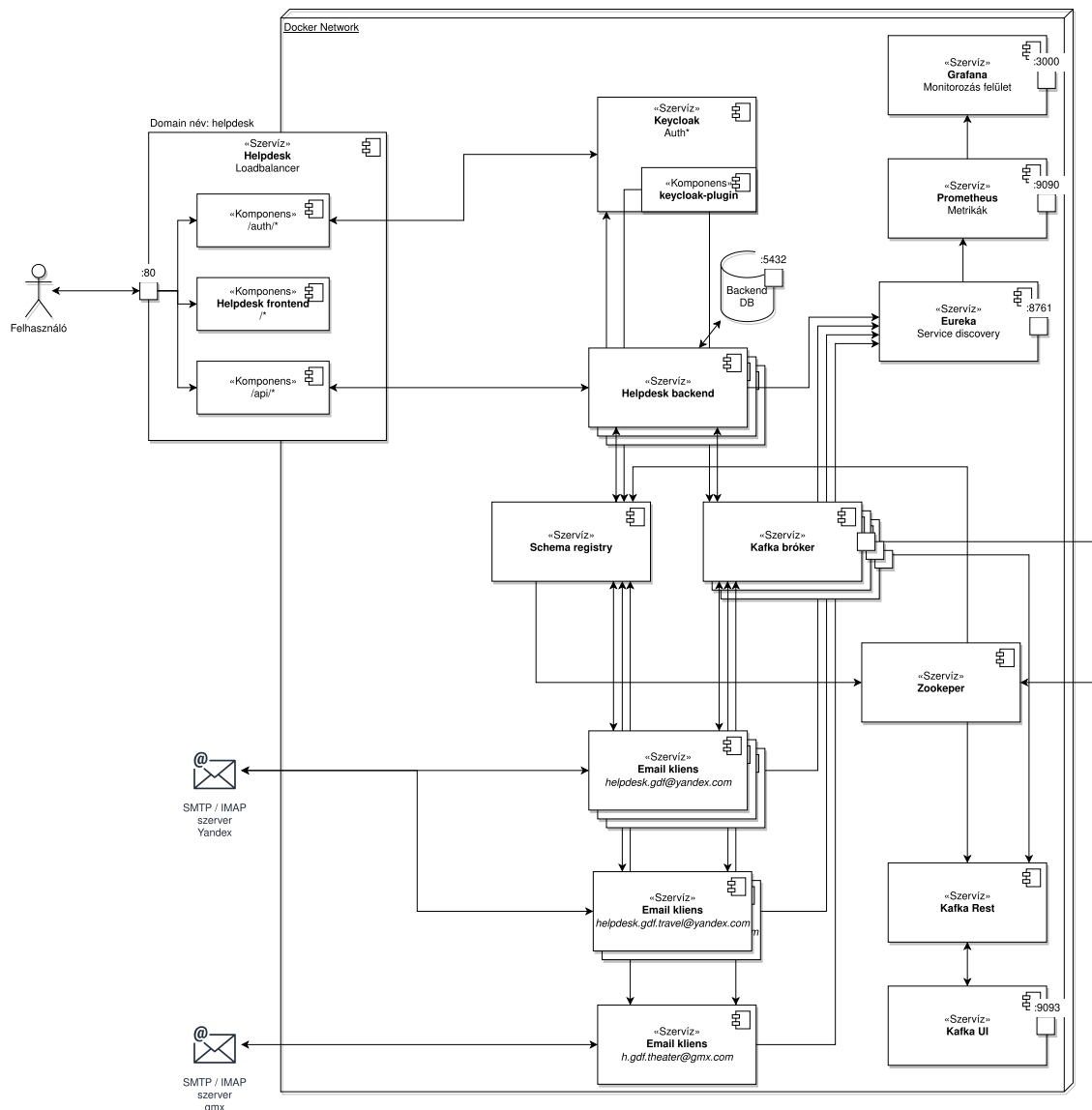
A Grafana és a Keycloak konfigurációs felülete az *admin* felhasználónév-jelszó párossal érhető el.

---

<sup>1</sup>Az Open Virtual Appliance által használt állományok kiterjesztése.

## 6.3. Deployment

A könnyebb bemutathatóság érdekében a szemléletesebb szolgáltatásokat – hogy ne a docker daemon által kiosztott IP címen keresztül kelljen elérni – a docker hálózaton kívül is elérhetővé tettem:



6.1. ábra. Deployment diagram

Forrás: saját ábra

`infrastructure_helpdesk_1` konténerben fut az nginx szolgáltatás. Elérhető a helpdesk és a localhost cím 80-as portján.

`infrastructure_db_1` konténerben fut a helpdesk backend Postgres adatbázisa, a localhost 5432-es portján érhető el.



**infrastructure\_eureka-service\_1** konténerben fut, és a 8761-es porton keresztül érhető el az Eureka discovery szolgáltatás.

**infrastructure\_prometheus\_1** konténerben fut a Prometheus monitorozó szolgáltatás. A dockert futtató felhasználó számára, a Prometheus a `localhost` 9090-es portján érhető el.

**infrastructure\_grafana\_1** konténerben fut a Grafana adatelemző és megjelenítő eszköz. A Grafanában létrehozott *dashboardok* a 3000-es porton keresztül érhetőek el.

**infrastructure\_kafka-ui\_1** konténerben fut a Kafka *topicok*at monitorozó Kafka Topics UI eszköz. Az alkalmazás felülete a 9093-as porton érhető el.

A `docker-compose` (6.1. pont) által elindított konténereket a 6.1. ábrán foglaltam össze. Minden, a helpdesk alkalmazás alá tartozó konténer az **infrastructure** előtagot kapta. Az ábrán feltüntettem, hogy az adott konténer a `localhost` melyik portján érhető el.

## 6.4. Több példány

A különböző szolgáltatásokból a terhelésnek megfelelően eltérő számú példány indul el:

- a helpdesk backendből három,
- a `theater` sort kezelő e-mail kliensből egy,
- a `travel` sort kezelő e-mail kliensből kettő,
- a `generic` sort kezelő e-mail kliensből három,
- és a Kafka brókerből (6.7) szintén három darab.

A példányok metrikáit (5.1.3 pont) nyomon lehet követni az erre a célra létrehozott Grafana oldalon (6.2c ábra). Az oldal elérhető a **Spring metrics** és a **Backend metrics** menüpontok alatt. A bejelentkezéshez, az `admin` felhasználónév és jelszó szükséges.

A 6.2c ábrán csak a Grafana oldal legfelső néhány panele látható, az instance-okra lebontott legfontosabb mérőszámokkal:

- a legfelső sorban a Java Virtual Machine, által aktuálisan felhasznált Heap space,

- alatta a feldolgozott Kafka üzenetek száma,
- a harmadik sorban a Trace log bejegyzések száma,
- míg az utolsó sorban az aktuális REST lekérések száma látható.

## 6.5. E-mail fogadásának és küldésének folyamata

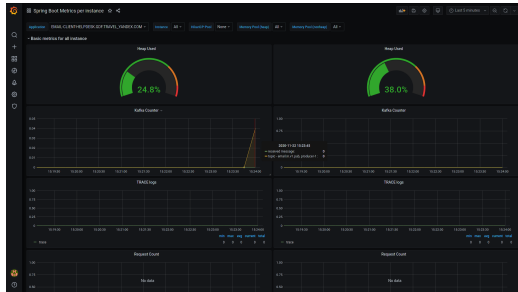
A 3. fejezetben a 3.3. ábrán bemutattam egy e-mail fogadásának elméleti útját. Most a 6.2. ábrán bemutatom hogyan követhető végig a rendszerben – a rendelkezésre álló monitorozó eszközök segítségével – egy e-mail valódi útja.

E-mail fogadása:

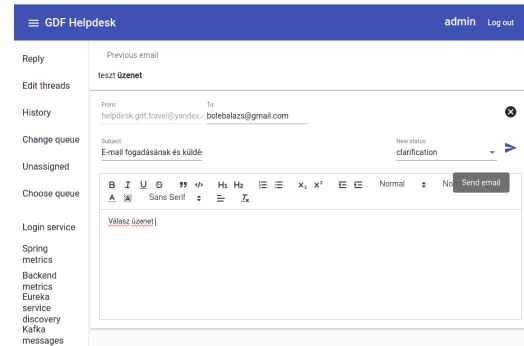
1. Egy teszt üzenet érkezik a `helpdesk.gdf.travel@yandex.com` címre *E-mail fogadásának és küldésének folyamata* tárggyal.
2. Az e-mail kliens kafka üzenetként publikálja az üzenetet az `email.in.v1.pub` topicba (6.2a. ábra). Az esemény megjelenik az e-mail klienshez tartozó Grafana monitorozó *dashboardon*.
3. A backend megkapja a kafka üzenetet (6.2c. ábra). A kafka üzenet olvasáshoz köthető esemény megjelenik a helpdesk backendhez tartozó Grafana monitorozó *dashboardon*.
4. A backend elmenti az új üzenet az adatbázisba (6.2e. ábra).
5. A felhasználói felületen (6.2g. ábra) elérhető az új üzenet.

E-mail küldése:

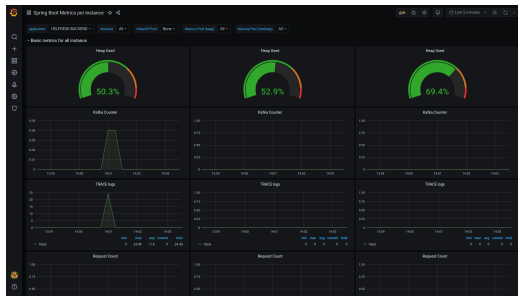
1. A felhasználó elküldi a válaszát a felhasználói felületen (6.2b. ábra).
2. A backend megkapja az üzenetet és eltárolja az adatbázisba (6.2d. ábra).
3. A `h.gdf.theater_gmx.com.v1.pub` topicban – a Kafka Topics UI felületen – megjelenik (6.2f. ábra) a backend által publikált kafka üzenet.
4. A topicra feliratkozott e-mail kliens fogadja és továbbítja az üzenetet. Az esemény megjelenik az e-mail klienshez tartozó Grafana monitorozó *dashboardon* (6.2h. ábra).



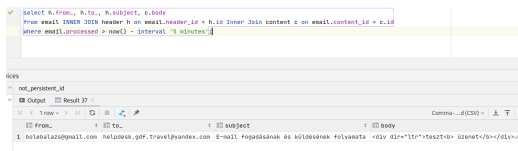
(a) Az egyes instance kafka üzenet küld



(b) A felületen válasz e-mailt küld a felhasználó



(c) Az egyes instance kafka üzenet fogad



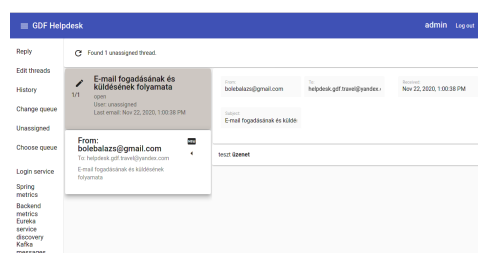
(e) Az új e-mail az adatbázisban



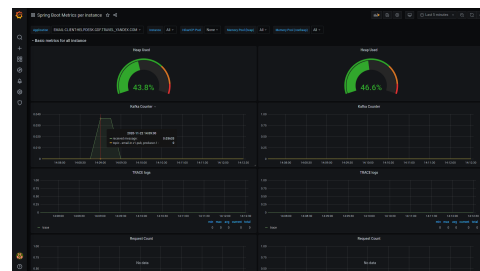
(d) A válasz e-mail az adatbázisban



(f) A h.gdf.theater\_gmx.com.v1.pub topic új üzenete



(g) A felületen elérhető az új e-mail

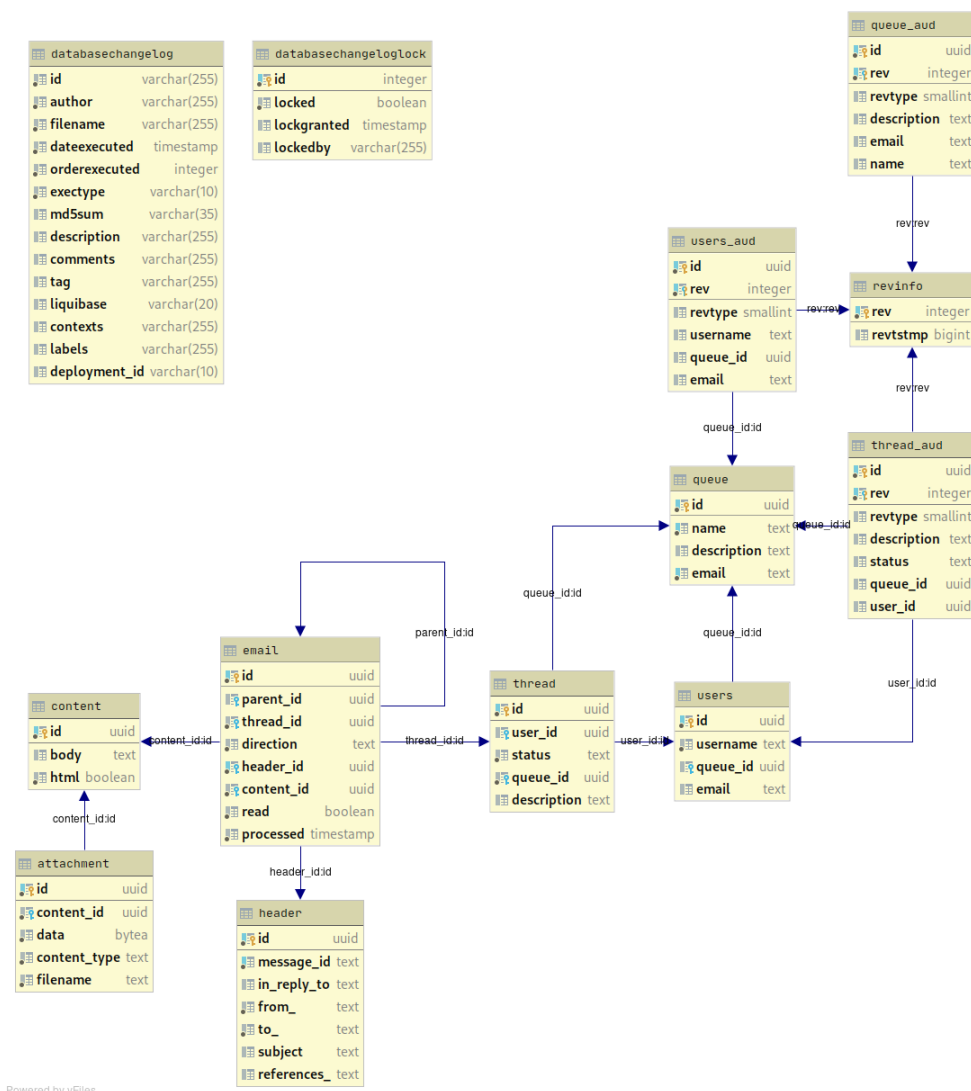


(h) Az egyes instance kafka üzenet fogad

6.2. ábra. E-mail fogadásának és küldésének folyamata során követhető lépések

## 6.6. Adatbázistáblák

A helpdesk backend adatbázistábláit a 6.3. ábra tartalmazza. A 3.2. pontban ismertetett táblákon kívül ezen az ábrán már szerepel a Liquibase és a Hibernate Envers által használt táblák.



6.3. ábra. A backend összes adatbázistáblája

Forrás: saját ábra

### 6.6.1. Liquibase

A databasechangelog és databasechangeloglock táblákat a Liquibase (5.3.2. pont) az adatbázis séma verziójának karbantartására használja:

**databasechangelog** tábla tartalmazza a resources/db.changelog könyvtárban ta-

lálható, `db.changelog-master.yaml` állományban tárolt utasítások futási eredményeit.

**databasechangeloglockot** minden végrehajtásnál a Liquibase példánya zárolja, ezzel biztosítva hogy mindig maximum egy példány hajtsa végre módosításokat az adatbázison (lásd 2.4 pont, pesszimista konkurenciakezelési stratégia).

A helpdesk backend szolgáltatás minden induláskor elindítja a Liquibase-t. A Liquibase csatlakozik az adatbázishoz a **helpdesk** felhasználóval, és lefuttatja a `db.changelog-master.yaml` állományban tárolt utasításokat.

A `db.changelog-master.yaml` állományba fel van véve az összes olyan DDL-utasítás és más SQL-parancs, ami az adatbázis kezdő állapotának létrehozásához szükséges. Mivel a Liquibase ezeket a parancsokat a **helpdesk** felhasználóval hajtja végre, a létrejött táblák is **helpdesk** felhasználóhoz fognak tartozni.

A helpdesk backend alkalmazás rendes működése során – a Liquibase futása után – a **helpdesk\_app** felhasználón keresztül kapcsolódik az adatbázishoz, így csak a Liquibase utasításokban meghatározott táblákhoz fér hozzá, és csak olyan típusú – CRUD – utasítást tud végrehajtani, ami külön engedélyezve van neki.

Így biztosítható, hogy a helpdesk alkalmazás csak a feladatának ellátásához szükséges szinten férjen hozzá az alkalmazáshoz, és hogy futása során ne módosíthassa a táblákat.

### 6.6.2. Hibernate Envers

A **revinfo** és az összes **\_aud** végződésű táblát – **users\_aud**, **queue\_aud** és **thread\_aud** – a Hibernate Envers használja az e-mail szál és a kapcsolódó entitások állapotának követésére.

**revinfo** tábla tartalmazza a módosítás időpontját és sorszámát.

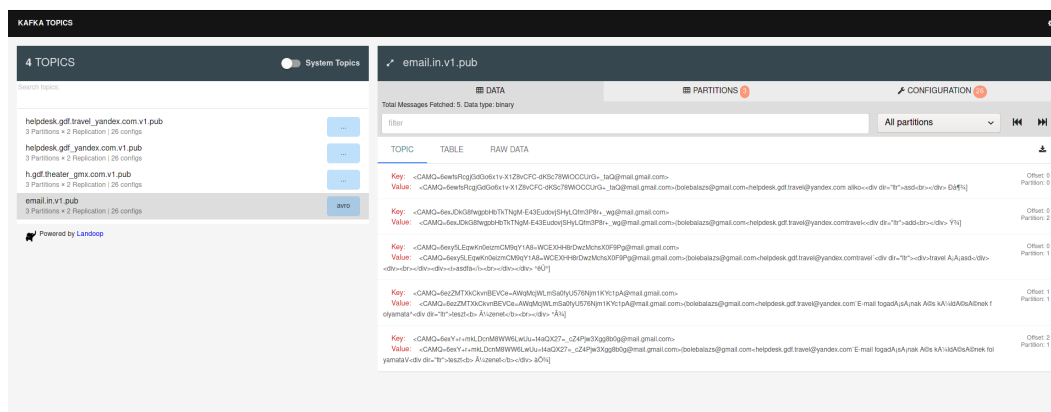
**\_aud** tábla tartalmazza a módosítás típusát, sorszámát és az entitás új értékeit.

Az Envers – a *Hibernate Event systemen* keresztül – figyeli, és feltartóztatja az e-mail szál állapotváltozásait. Hozzáadja a saját verziókövetéshez szükséges utasításait, és csak akkor engedi sikeresen lezárni a tranzakciót, ha az **\_aud** tábla is sikeresen módosul.

Az Envers minden állapothoz eltárolja a módosítás típusát – **insert**, **update**, vagy **delete** –, sorszámát és dátumát. Így mindig visszakereshető hogy melyik időpillanatban mi volt az entitás értéke.

## 6.7. Apache Kafka

A kafka *topic*ok és üzenetek elérhetőek és követhetőek a Kafka messages menü pontja alatti Kafka Topics UI (6.4 és 6.2f ábra) eszközzel.



6.4. ábra. A Kafka Topics UI eszközzel követhetőek a kafka *topic*ok üzenetei, partíciói és beállításai

Forrás: saját ábra

A helpdesk alkalmazás összesen hat *topic*-ot használ:

**user.v1.pub** a Keycloak-ban regisztrált és karbantartott felhasználókat tartalmazza,

**email.in.v1.pub** az összes beérkező e-mailt tartalmazza,

**helpdesk.gdf\_yandex.com.v1.pub** a [helpdesk.gdf@yandex.com](mailto:helpdesk.gdf@yandex.com) címre küldött e-maileket tartalmazza,

**helpdesk.gdf.travel\_yandex.com.v1.pub** a [helpdesk.gdf.travel@yandex.com](mailto:helpdesk.gdf.travel@yandex.com) címre küldött e-maileket tartalmazza,

**h.gdf.theater\_gmx.com.v1.pub** a [h.gdf.theater@gmx.com](mailto:h.gdf.theater@gmx.com) címre küldött e-maileket tartalmazza,

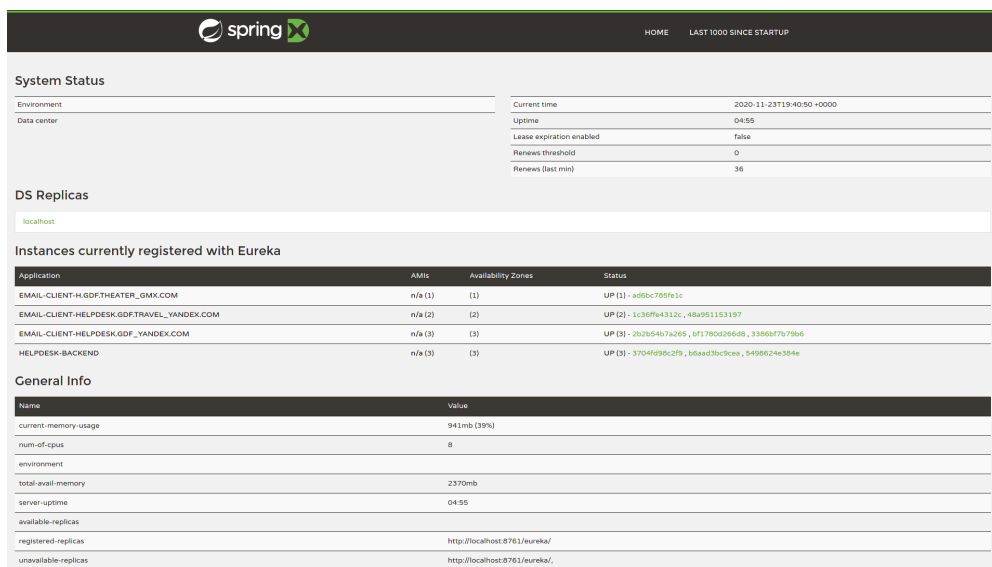
**\_schemas** a Schemaregistry ebben a *topic*ban tárolja az alkalmazásban használt Avro schemákat.

Az alkalmazás három kafka brókert futtat egy clusterben, avagy számítógépfürtben. Továbbá minden üzleti funkcionalitást hordozó *topic* – a **\_schemas**-on kívül mindegyik – három partícióval és kettes replikációs faktorral lett létrehozva. Így a kafka fürt egy bróker kiesése, vagy egy partíció sérülése esetén is működőképes marad.

## 6.8. Eureka

Az 5.1.3. pontban említett szolgáltatások felderítésre az Eureka szervert használom. A helpdesk backend és az e-mail kliens az elindításuk után közvetlenül beregisztrálják magukat az Eureka szolgáltatásba.

Az Eureka szerveren keresztül megtekinthető és más szolgáltatások számára elérhető a példányok aktuális állapota és neve. Az oldal elérhető a *Eureka service discovery* menüpont alatt (6.5. ábra).



The screenshot displays the Spring Eureka service discovery dashboard. At the top, there's a navigation bar with the Spring logo and links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content is divided into several sections:

- System Status:** A table showing system metrics.

Environment	Current time
Data center	2020-11-23T19:40:50 +0000
	Uptime
	04:55
	Lease expiration enabled
	false
	Renews threshold
	0
	Renews (last min)
	36
- DS Replicas:** A section showing the local host as a replica.

localhost
- Instances currently registered with Eureka:** A table listing registered instances.

Application	AMIs	Availability Zones	Status
EMAIL-CLIENT-H.GDF.THEATER_GMX.COM	n/a (1)	(1)	UP (1) - a95bc7959a1c
EMAIL-CLIENT-HELPDESK.GDF.TRAVEL_YANDEX.COM	n/a (2)	(2)	UP (2) - 1c36ff4312c, 48a951153197
EMAIL-CLIENT-HELPDESK.GDF_YANDEX.COM	n/a (3)	(3)	UP (3) - 2b2b54b7a265, bf1780d266d8, 3386bf7b78b6
HELPDESK-BACKEND	n/a (3)	(3)	UP (3) - 3704f930c2f9, b6aad3bc3cea, 5498624e304e
- General Info:** A table showing general system information.

Name	Value
current-memory-usage	941mb (39%)
num-of-cpus	8
environment	
total-avail-memory	2370mb
server-up-time	04:55
available-replicas	
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/

6.5. ábra. Az Eureka service discovery-n látható a mikroszerviz példányainak állapota és egyedi azonosítója

Forrás: saját ábra

## 7. fejezet

# Terheléses tesztelés

Annak érdekében, hogy információt szerezzek arról, hogyan viselkedik az alkalmazás nagyobb terhelés esetén, és hogy megvizsgáljam a rendszerrel szemben állított nem funkcionális igények (1.2.3 pont) teljesülését, terheléses teljesítményvizsgálatnak vetettem alá a létrehozott helpdesk programot.

### 7.1. Terheléses teszt

A terheléses teszt egy – a teljesítménytesztelés alá tartozó – nem funkcionális vizsgálat. Célja a rendszer működésének vizsgálata, viselkedésének megértése bizonyos előre meghatározott terhelés esetén.

A vizsgálat párhuzamos felhasználók modellezésén alapszik. A vizsgált alkalmazás folyamatos monitorozás alatt áll, miközben a felhasználók – azonos időben – végre próbálják hajtani az előre meghatározott üzleti igényeiket.

A tesztelés során rögzített válaszidőket ezután célszerű összevetni a rendszer metrikáival, hogy feltárhatóak legyenek az alkalmazás összteljesítményét kritikusan érintő szűk keresztmetszetek.

### 7.2. Apache JMeter

Az Apache Jmeter egy nyílt forrású terheléses teszt végrehajtására alkalmas eszköz. Számos protokollt képes kezelni, a helpdesk alkalmazás vizsgálatánál HTTP üzenetek küldésével modelleztem a párhuzamos felhasználókat.

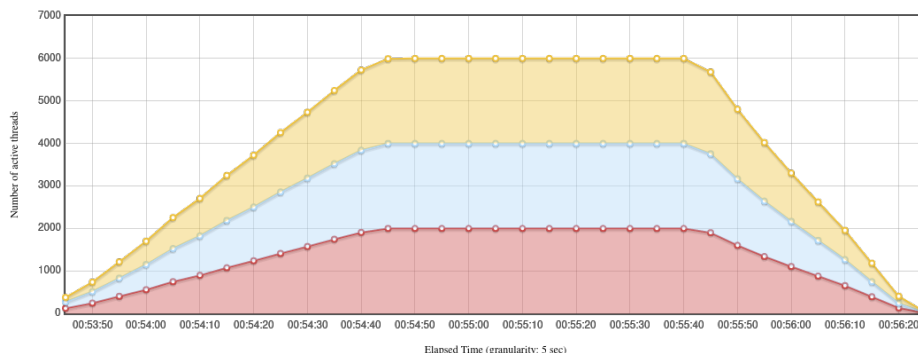
Mivel a helpdesk frontend a kliens oldalon (2.7 pont) – a felhasználó számítógépén – fut, így elegendő csak a frontend-backend közötti kommunikációt imitálni.

Minden teszt egy rövid felfutási idővel kezdődik, ami során a JMeter elindítja a párhuzamos teszteléshez használt szálakat (7.1 ábra). A szálak – a teszt teljes ideje



alatt – folyamatosan hajtják végre a számukra kijelölt feladatot. A JMeter méri és rögzíti a vizsgált rendszer válaszait és válaszidejét. A rögzített adatok további elemzésével meghatározható a vizsgált rendszer viselkedése, a Grafana-ban megjelenített metrikák összevetésével pedig azonosíthatóak a teljesítmény szempontjából kritikus rendszerek.

A létrehozott JMeter tesztjeim a leggyakrabban és a legtöbb erőforrást igénylő funkciókat tesztelik 120 másodpercig.



7.1. ábra. A JMeter tesztelés során használt számai. A különböző üzleti funkciókat hívó szálak eltérő színnel szerepelnek.

Forrás: saját ábra

## 7.3. Átlagos teljesítmény vizsgálata

A helpdesk backend egy önálló példányát vizsgáltam 100 párhuzamos felhasználó modellezésével. A teszt során használt felfutási idő 30 másodperc volt.

A teszt eredményeit a 7.1 táblázat tartalmazza.

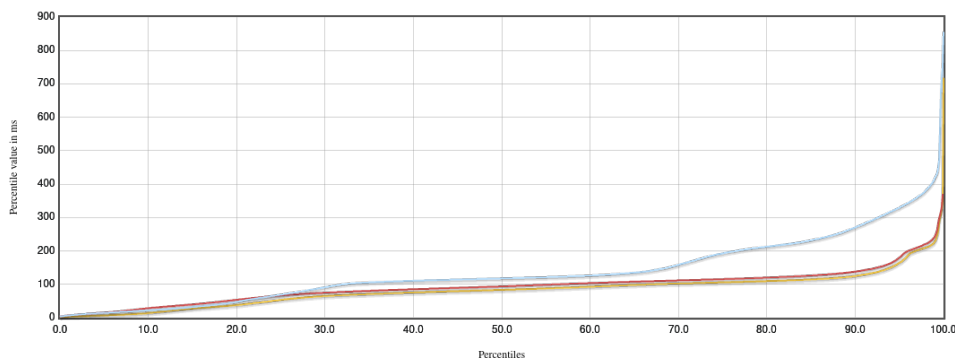
Átlag	Válaszidő [ <i>ms</i> ]				Áteresztőképesség [Tranzakció/ <i>s</i> ]	Válasz [%]	
	Max.	Medián	P90	P95		< 3 <i>s</i>	< 6 <i>s</i>
99,32	856	88,00	223,00	340,00	900,01	100	100

7.1. táblázat. Egy példányban futó helpdesk backend terheléssel teszt eredménye 100 felhasználóval

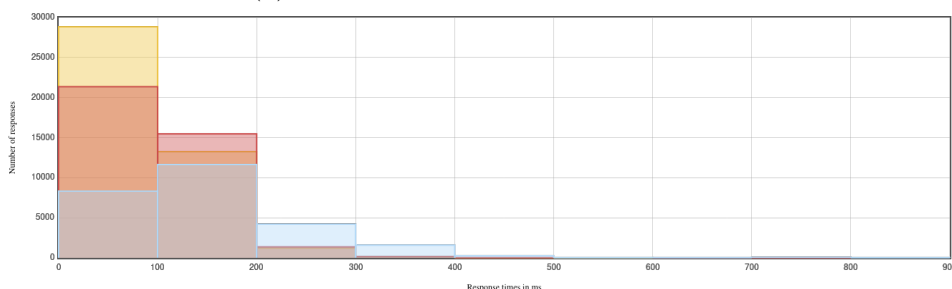
A 7.2 ábrán és a 7.1 táblázat adataiból látszódik, hogy:

- a legtöbb válasz 88 *ms* alatt megérkezik,
- a backend legrosszabb esetben is 1 másodperc alatt válaszol,
- a rendszer áteresztőképessége 900 tranzakció másodpercenként.

Az alkalmazás tehát megfelel a vele szemben állított követelményeknek.



(a) A válaszidők percentilis eloszlása



(b) A válaszidők eloszlása

7.2. ábra. Egy példányban futó helpdesk backend terheléssel teszt eredménye 100 felhasználóval

Forrás: saját ábra

## 7.4. Csúcsteljesítmény vizsgálata

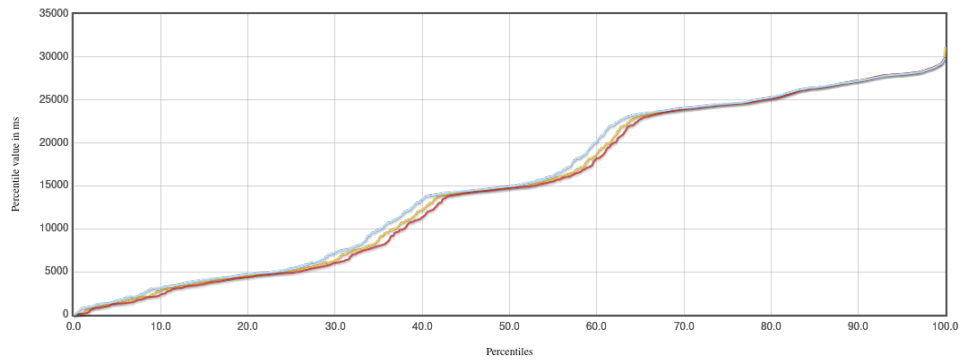
A helpdesk backend egy önálló példányát vizsgáltam 6 000 párhuzamos felhasználó modellezésével. Mint ahogy a teszt során használt párhuzamos szálakról készített a 7.1 ábrán is látszódik, a felfutási idő 60 másodperc volt. Az ábrán látható ahogy a JMeter a felfutási idő után egyszerre 6 000 szálon futtatja a teszteket.

A teszt futtatása során mért eredményeket a 7.2 táblázatban foglaltam össze. Mint láthatjuk, az alkalmazás messze elmarad a vele szemben állított követelményektől (1.2.3 pont). Három másodperc alatt csak a kérések 10,19%-át szolgálja ki. A legtöbb kérésre 25 másodpercet vesz igénybe a válasz adása.

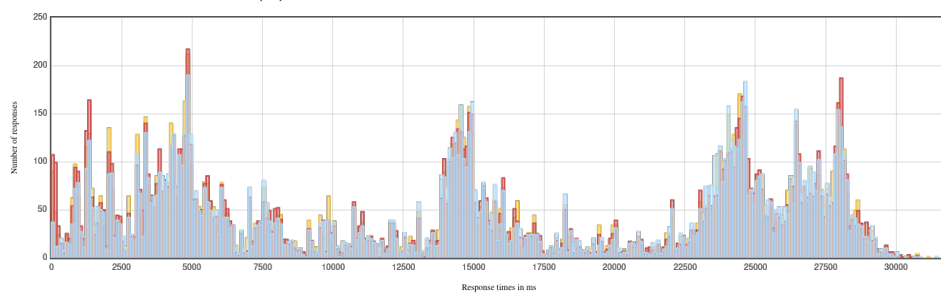
Átlag	Válaszidő [ms]				Áteresztőképesség [Tranzakció/s]	Válasz [%]	
	Max.	Medián	P90	P95		< 3s	< 6s
15 133,81	31 732	24 553,00	28 011	28 407	281,72	10,19	27,78

7.2. táblázat. Egy példányban futó helpdesk backend terheléssel teszt eredménye 6 000 felhasználóval

Az eltérés okainak azonosítására célszerű a szűk keresztmetszeteket meghatározni és feloldani (7.5 pont).



(a) A válaszidők percentilis eloszlása



(b) A válaszidők eloszlása

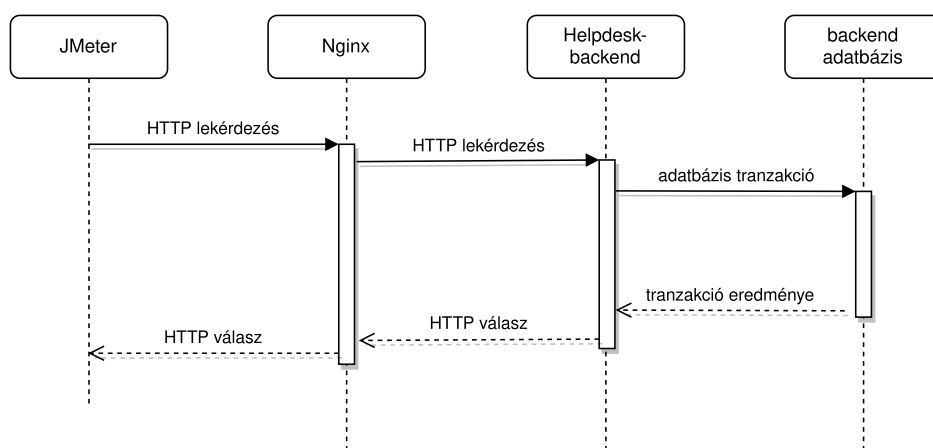
7.3. ábra. Egy példányban futó helpdesk backend terheléses teszt eredménye 6 000 felhasználóval

Forrás: saját ábra

## 7.5. Szűk keresztmetszet meghatározása

A JMeter által indított kérés az alábbi rendszereken megy keresztül (7.4 ábra):

1. Az Nginx fogadja és továbbítja a HTTP kérést a Helpdesk backendnek.
2. A backend végrehajtja az üzleti funkciót, és amennyiben szükséges
3. lekérdezi és visszaadja az eltárolt adatokat az adatbázisból.

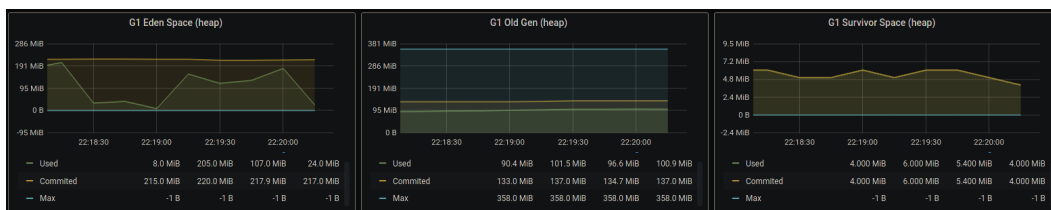


7.4. ábra. A terheléses tesztben résztvevő alkalmazások

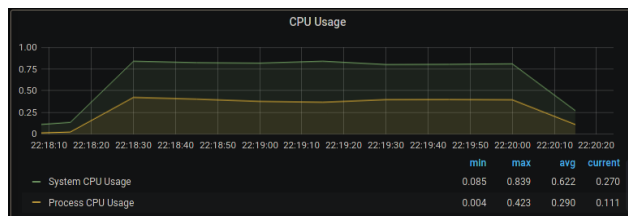
Forrás: saját ábra

Így a szűk keresztmetszet is csak ebben a három rendszerben fordulhat elő. A backend metrikáinak vizsgálatából látszódik, hogy:

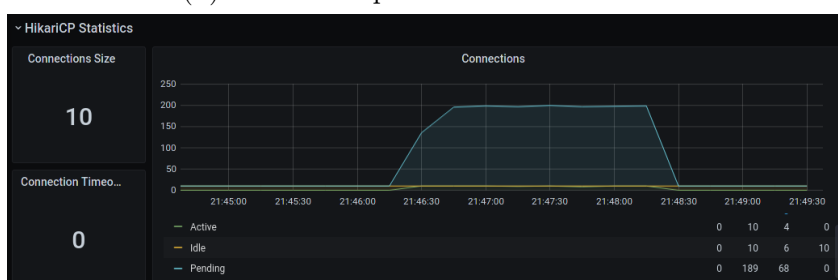
- A backend memóriaigénye nem nőtt meg jelentősen (7.5a ábra).
- A processzorigény szignifikánsan megemelkedett (7.5b ábra). A rendszer CPU felhasználása 80%-ra a backendé 40%-ra nőtt.
- Az adatbázis kapcsolatok fenntartására használt HikariCP-ben (5.3.2 pont) lényegesen feltorlódtak a lekérdezések (7.5c ábra). A függőben lévő kapcsolatok markánsan megugrottak.



(a) A backend memóriafelhasználása



(b) A backend processzor használata



(c) HikariCP adatbázis-kapcsolatai

7.5. ábra. A csúcsteljesítmény vizsgálata során vizsgált legfontosabb metrikák a Grafana monitorozó eszköz felületén

Forrás: saját ábra

### 7.5.1. Nginx

Ahhoz, hogy az Nginx párhuzamosan kiszolgáljon legalább 6 000 felhasználót<sup>1</sup>, a beállításában felül kell írni a `worker_connections` paramétert.

Ha az alapbeállításoktól való eltérés okozza a kérések – még backend előtti – feltorlódását, vagy a megnövekedett processzorigényt, akkor az Nginx kihagyásával jelentős javulás lenne elérhető.

Ezért a megismételt a JMeter tesztben a backendet közvetlenül az IP-címén keresztül értem el. Az új mérés – 7.3 táblázat – adataiból látszódik, hogy

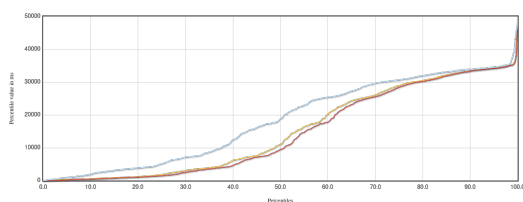
- az átlagos és a medián válaszidő, valamint áteresztőképesség nem változott,
- a három másodperc alatt megérkező válaszok aránya – az első esethez (7.4) képest – meg két és félszereződött, 26,41%-ra nőtt.

<sup>1</sup>Az alapbeállítás 1 024 kapcsolat

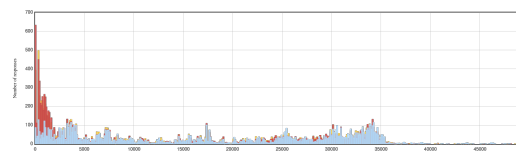
Átlag	Válaszidő [ms]					Áteresztőképesség [Tranzakció/s]	Válasz [%]	
	Max.	Medián	P90	P95	P95		< 3s	< 6s
15 772,71	49 367	29 450,00	34 256	34 898	34 898	265,72	26,41	36,78

7.3. táblázat. Nginx nélkül futó helpdesk backend terheléssel teszt eredménye 6 000 felhasználóval

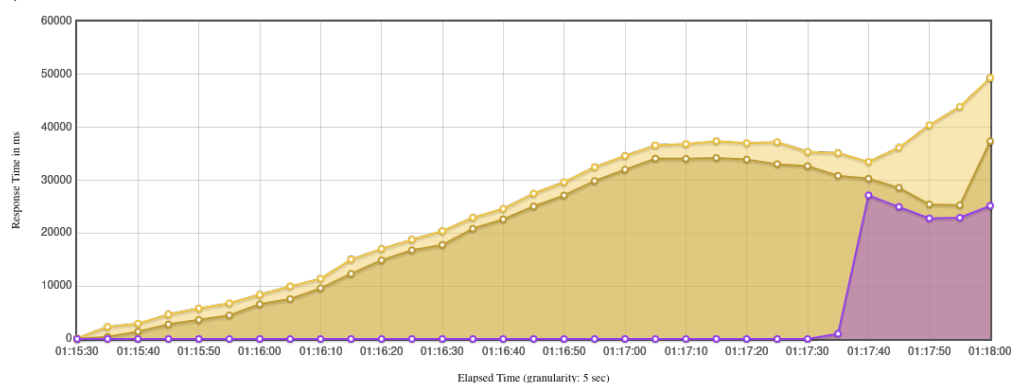
Ugyanez jelenik meg a 7.6c ábrán is, a teszt első háromnegyedében a backend néhány kérésre azonnal válaszol, míg a többségre – lásd medián – csak sokkal később.



(a) A válaszidők percentilis eloszlása



(b) A válaszidők eloszlása



(c) A minimum, medián és maximum válaszidők időbeli eloszlása

7.6. ábra. Nginx nélkül futó helpdesk backend terheléssel teszt eredménye 6 000 felhasználóval

Forrás: saját ábra

Ha ezt a tényt összevetjük azzal, hogy

- a különböző tesztesetekre adott válaszidő különbözik (7.6a ábra),
- az esetek 0.19%-ában `Connection reset` hibát kap a JMeter,
- valamint hogy a processzorigény nem tér el az Nginx-el együtt futtatott tesztől,

akkor arra a következtetésre juthatunk, hogy az Nginx használata nem hogy rontotta volna a válaszidőt, hanem sokkal inkább kiszámíthatóbbá jobban tervezhetővé tette azt.

### 7.5.2. HikariCP

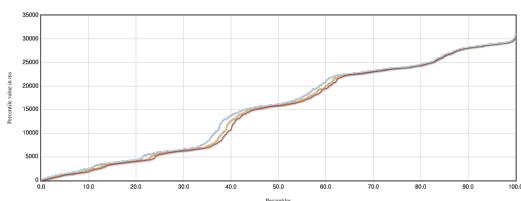
A 7.5c ábrán látszódik hogy a HikariCP-ben beállított 10 kapcsolat hamar elfogy, már mérés elején 189-re nő, és tartósan ott is marad a poolra várakozók száma.

Ha az adatbázissal való kapcsolat a szűk keresztmetszet, akkor a HikariCP pooljának megnövelésével jelentős javulás lenne elérhető.

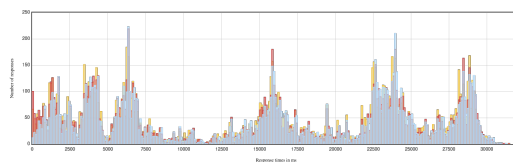
Ezért a megismételt a JMeter tesztben a backendben megháromszoroztam a HikariCP pooljának a méretét. A mérési eredmények – 7.4 táblázat – nem mutatnak eltérést az első, kiinduló állapothoz képest.

Válaszidő [ms]					Áteresztőképesség	Válasz [%]	
Átlag	Max.	Medián	P90	P95	[Tranzakció/s]	< 3s	< 6s
15 220,66	32 136	23 836,00	28 876	29 177	278,42	12,51	26,41

7.4. táblázat. 30 adatbázis-kapcsolattal futó helpdesk backend terheléssel teszt eredménye 6 000 felhasználóval



(a) A válaszidők percentilis eloszlása



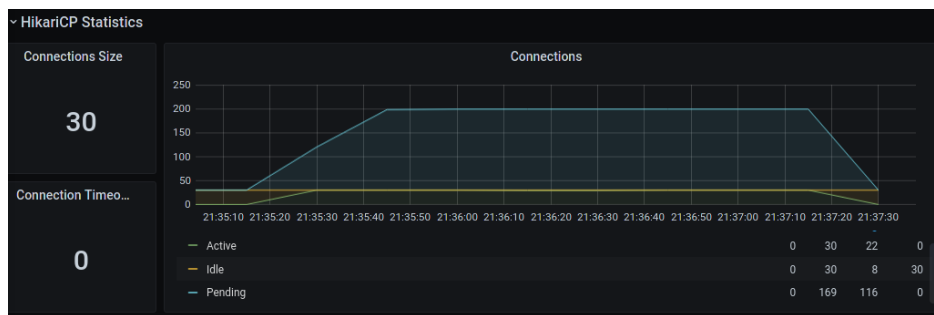
(b) A válaszidők eloszlása

7.7. ábra. 30 adatbázis-kapcsolattal futó helpdesk backend terheléssel teszt eredménye 6 000 felhasználóval

Forrás: saját ábra

A 7.8 ábrán látszódik, hogy mind a harminc pool részt vesz az adatbáziskapcsolatokban. A kapcsolatra várakozók száma 169 lett, csak azzal a 20-szal lett kevesebb, amit hozzáadtunk a poolhoz.

Mivel a teszteredményekben nincs jelentős változás, nem a HikariCP a szűk keresztmetszet.



7.8. ábra. HikariCP adatbázis-kapcsolatai a Grafana felületén

Forrás: saját ábra

### 7.5.3. Megnövekedett processzorigény

A 7.5b ábrán látszódik hogy még van szabad processzoridő. Ha a helpdesk backend képes lenne magának még processzoridőt allokalni, azzal javulhatna a rendszer áteresztőképessége.

Ha a processzoridő a szűk keresztmetszet, akkor új backend példányok indításával – és így újabb erőforrások bevonásával – jelentős javulást lehetne elérni.

Ezért a megismételt JMeter tesztben a backendből három példányt indítottam el. Az eredmények így összemérhetőek maradtak a 7.5.2. ponttal, hiszen összességében ugyanannyi kapcsolat van a backend és az adatbázis között.

Átlag	Max.	Válaszidő [ms]				Áteresztőképesség [Tranzakció/s]	Válasz [%]	
		Medián	P90	P95			< 3s	< 6s
5748,49	16 577	7452,00	11 482,00	12 011,95		755,62	22,23	49,01

7.5. táblázat. Három példányban futó helpdesk backend terheléses teszt eredménye 6 000 felhasználóval

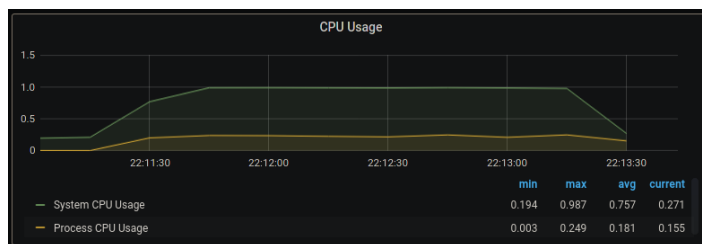
A 7.5 táblázat adatai alapján kijelenthető,

- hogy az átlagos válaszidő 6 másodperc alá esett,
- az áteresztőképesség megháromszorozódott,
- és a három másodpercen belül érkező válaszok aránya is megduplázódott.

A 7.10 ábrán megjelenített mérési eredményeken látszódik hogy a válaszidők szórása nagy mértékben csökkent, a várható bizonytalanság így már sokkal inkább tolerálható.

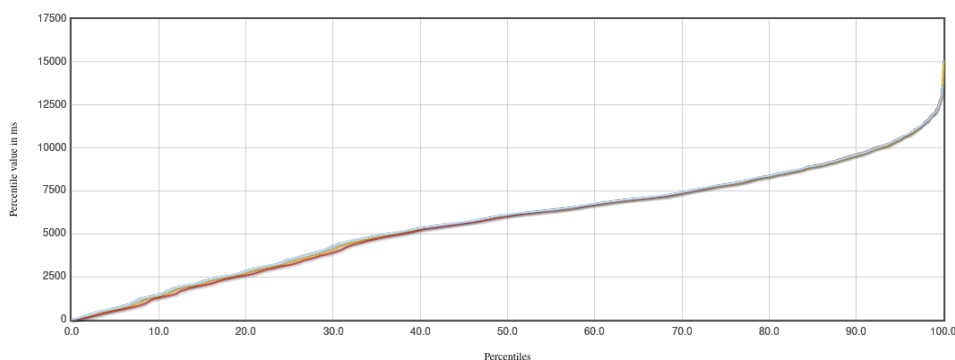
Ha megvizsgáljuk a teszt során mérhető processzor használatot (7.9 ábra), akkor látható, hogy nem maradt már allokalatlan processzoridő. Valószínűleg a két újabb backend példány használta fel a maradék erőforrást.



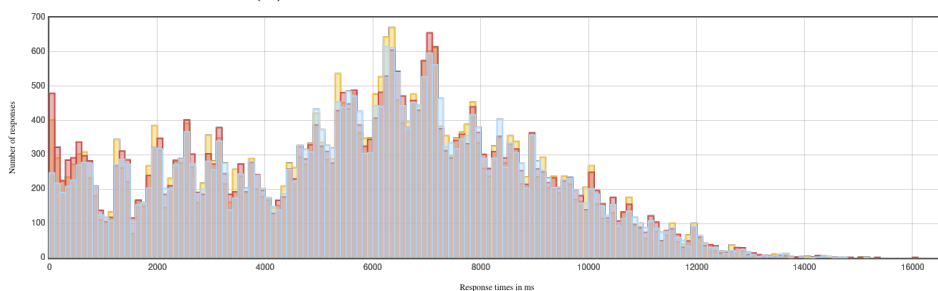


7.9. ábra. A backend egy példányának a processzor használata a Grafana felületén

Forrás: saját ábra



(a) A válaszidők percentilis eloszlása



(b) A válaszidők eloszlása

7.10. ábra. Három példányban futó helpdesk backend terheléses teszt eredménye 6 000 felhasználóval

Forrás: saját ábra

#### 7.5.4. Szűk keresztmetszet meghatározása

A mérések alapján világosan látszik hogy a szűk keresztmetszetet a backend kevés processzorideje jelentette. Újabb példányok indításával jelentősen megnövekedett a rendszer áteresztőképessége, és lecsökkent a válaszideje.

A 7.9 ábrán az is látszódik, hogy a futtatásra használt számítógépnek nincs annyi számítási kapacitása, mint amennyire a backendnek szüksége lenne. Így a valódi szűk keresztmetszetet a rendelkezésre álló erőforrások jelentik.

## 7.6. Összevetés a követelményekkel

Az elvégzett tesztekben látszódik, hogy horizontális skálázhatósággal jelentősen megnövelhető a helpdesk alkalmazás áteresztőképessége, és csökkenthető a válaszideje.

### 7.6.1. Átlagos teljesítmény vizsgálata

Az alkalmazás minden probléma nélkül teljesíti az átlagos terhelés esetére meghatározott követelményeket. Legrosszabb esetben is 1 másodpercen belül válaszol.

### 7.6.2. Csúcsteljesítmény vizsgálata

A rendszerrel szembeni elvárások teljesíthetőségére nem lehet egyértelmű választ adni. A tesztelt számítógépen – erőforráshiány miatt (7.5.4) – az alkalmazás nem teljesíti a három másodperces válaszidőre vonatkozó megkötést.

A mérésekből azonban arra lehet következtetni, hogy a valós rendszer – akár több különálló számítógépen – bírni fogja a terhelést, nem jelent majd gondot neki a csúcsterhelés során sem feladatainak ellátása.

## 8. fejezet

# Továbbfejlesztési lehetőségek

### 8.1. A deploymentről

A helpdesk alkalmazás szolgáltatásai úgy lettek kialakítva, hogy képesek legyenek egymástól függetlenül, akár több példányban is működni. Ezáltal költséghatékonyá téve a működést, leegyszerűsítve a hibatűrést, és lehetővé téve a változó terhelés miatti skálázhatóságot.

Azonban amíg a docker konténerek egy host gépen futnak és egy erőforráson osztoznak, soha nem lehet gazdaságosan megoldani a skálázást, és nem tud a rendszer felkészülni a számítógép kiesésére.

A következő logikus lépés tehát az alkalmazás számítógépfürtre migrálása. A docker natívan támogatja a Microsoft Azure és az Amazon [21] szolgáltatókat. Így tehát a kód és a beállítások módosítása nélkül lehetséges az alkalmazás fürtösítése docker swarmmal.

### 8.2. A kódról

A helpdesk alkalmazásba – architektúrája miatt – könnyű új funkciót fejleszteni. A most működő modulok mind lazán kapcsolódnak egymáshoz, így könnyű egy teljesen különböző, akár eltérő programnyelven íródott új funkció integrálása.

Mivel az összes technikai megkötés csupán a protokollok megvalósítása, nyugodtan lehet az új funkció tervezésénél a feladathoz választani a programnyelvet vagy a programozási módszertant is.

Ha az új funkciónak szüksége lenne az alkalmazás más rendszeréhez tartozó adatra, akkor a kérdéses rendszer – ahogy a Keycloak plugin példáján megmutattam – ugyanúgy bővíthető, a komponensek közötti laza kapcsolat megőrizhető.

Hasonlóképpen, a laza kapcsolatok és jól definiált határok miatt, egyszerű egy-egy modult teljesen lecserélni, vagy más nyelven, más technológiával újraírni.

Mivel egy szolgáltatás egy feladattal foglalkozik, ha például le kell cserélni a front-endet, akkor az új felhasználói felületen csak a megjelenítéssel kell foglalkozni, az üzleti funkciók megvalósítása a backend feladata, így azok továbbra is változatlanok maradnak.

Ugyanez nem csak a szolgáltatások, hanem a kód szintjén is igaz. A hexagonális architektúra miatt, az adatbázis – mint külső függőség – könnyen cserélhető.

# Összefoglalás

Dolgozatom célja egy több e-mail címet és szálát kezelő teljes értékű elosztott alkalmazás létrehozása volt.

## Egyszerűen integrálható

Az alkalmazás teljes funkcionalitását több, egymástól függetlenül működő, pontosan egy körülhatárolt részfeladatért felelős mikroszerviz látja el. A szolgáltatások egymáshoz lazán, HTTP-n és káfkán keresztül kapcsolódnak.

A moduláris felépítés, és a szolgáltatások laza kapcsolata nagy mértékben leegyszerűsíti a helpdesk alkalmazás akár mások által fejlesztett programokkal való együttműködését is. A rendelkezésre álló számtalan, szabadon használható, nyílt forrású program használata pedig nagy mértékben megkönnyíti az új alkalmazás fejlesztését.

Jó példa erre a Keycloak szolgáltatás esete, ahol egy biztonságos és megannyi funkcióval rendelkező jogosultság- és hozzáférés-kezelőt sikerült a helpdesk alkalmazásba integrálni. Így a jelszókezelésnél vagy más hozzáféréssel és jogosultsággal kapcsolatos részletnél hagyatkozhattam a Keycloak funkcionalitására, nekem elegendő volt a felhasználók üzletileg is releváns, e-mailekhez köthető kapcsolatával foglalkoznom.

## Célorientált megvalósítás

Három fő részfeladatra osztottam fel az alkalmazás működését. Az e-mail szerverekkel való kapcsolattartásért az **e-mail kliens**, az e-mailek, e-mail szálak logikai rendszerezésért, tárolásáért a **helpdesk backend**, és az e-mail szálak megjelenítéséért, a felhasználókkal való interakcióért a **helpdesk frontend** lett a felelős.

A közöttük fennálló laza kapcsolat miatt az adott feladatnak megfelelően választhattam meg a programnyelvet és technológiát. Így az **e-mail klienst** és a **helpdesk backendet** Java alapon Spring Boottal, a **helpdesk frontendet** pedig TypeScript alapon Angularral készítettem.

## **Változó igényeknek megfelelő**

Terheléssel tesztel megvizsgáltam a helpdesk alkalmazás komoly igénybevétel során mérhető teljesítményét. Az alkalmazás monitorozásához használt eszközök segítségével elemeztem a vizsgálat eredményét, És az alkalmazás horizontális skálázásával sikerült jelentős teljesítményjavulást elérni.

Össességében elmondható, hogy sikerült egy modern, időtálló, mikroszerviz alapú elosztott alkalmazást létrehoznom. Bemutattam a megvalósítás során felmerült problémákat, azok megoldását, a felhasznált technológiákat és módszertanokat.

# Irodalomjegyzék

- [1] Mike Loukides and Steve Swoyer. Microservices adoption in 2020, Jul. 15 2020. URL: <https://www.oreilly.com/radar/microservices-adoption-in-2020/>.
- [2] Andzhela Angelova. 10 reasons why microservices are the future, Jun. 20 2020. URL: <https://wiredelta.com/10-reasons-why-microservices-are-the-future/>.
- [3] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*, chapter 1st, The Microservices Way. O'Reilly, 1st edition, 2016.
- [4] Alistair Cockburn. Hexagonal architecture, Apr. 1 2005. URL: <https://web.archive.org/web/20180822100852/http://alistair.cockburn.us/Hexagonal+architecture>.
- [5] Robert C. Martin. The clean architecture, Aug. 13 2012. URL: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>.
- [6] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*, chapter 11th, Systems. Prentice Hall, 1st edition, 2008.
- [7] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*, chapter 5th, CQRS. O'Reilly, 1st edition, 2016.
- [8] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*, chapter 5th, Distributed Transactions and Sagas. O'Reilly, 1st edition, 2016.
- [9] Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale, and Culture*, chapter 1st, Meet Kafka. O'Reilly, 1st edition, 2016.

- [10] Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale, and Culture*, chapter 5th, Kafka Internals. O'Reilly, 1st edition, 2016.
- [11] Introduction to angular concepts. Letöltve: 2020. Nov. 16. URL: <https://angular.io/guide/architecture>.
- [12] Introducing spring boot. Letöltve: 2020. Nov. 16. URL: <https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started-introducing-spring-boot>.
- [13] Usage statistics of web servers. Letöltve: 2020. Dec. 2. URL: [https://w3techs.com/technologies/overview/web\\_server](https://w3techs.com/technologies/overview/web_server).
- [14] Amir Rawdat. Testing the performance of nginx and nginx plus web servers, Aug. 24 2017. URL: <https://www.nginx.com/blog/testing-the-performance-of-nginx-and-nginx-plus-web-servers/>.
- [15] Comparison of hibernate with mysql server vs hibernate with postgresql server. Letöltve: 2020. Dec. 3. URL: <https://www.jpab.org/Hibernate/MySQL/server/Hibernate/PostgreSQL/server.html>.
- [16] Identification fields. Letöltve: 2020. Nov. 16. URL: <https://tools.ietf.org/html/rfc5322#section-3.6.4>.
- [17] Query creation. Letöltve: 2020. Dec. 09. URL: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>.
- [18] Keycloak. Letöltve: 2020. Nov. 18. URL: <https://www.keycloak.org/>.
- [19] Json web token (jwt). Letöltve: 2020. Nov. 18. URL: <https://tools.ietf.org/html/rfc7519>.
- [20] The oauth 2.0 authorization framework. Letöltve: 2020. Dec. 9. URL: <https://tools.ietf.org/html/rfc6749>.
- [21] Deploying docker containers on ecs. Letöltve: 2020. Nov. 22. URL: <https://docs.docker.com/engine/context/ecs-integration/>.



# Ábrák jegyzéke

1.1. Az e-mailszálak státuszváltozásai . . . . .	3
1.2. Elérhető funkciók . . . . .	4
2.1. Hexagonális alkalmazások felépítése . . . . .	9
3.1. A legfontosabb komponensek . . . . .	13
3.2. A backend legfontosabb adatbázistáblái . . . . .	15
3.3. A bejövő és kimenő e-mail útja . . . . .	17
5.1. E-mail kliens szekvencia diagramja . . . . .	26
5.2. Helpdesk backend szekvencia diagramja . . . . .	27
5.3. Helpdesk frontend szekvencia diagramja . . . . .	33
6.1. Deployment diagram . . . . .	39
6.2. E-mail fogadásának és küldésének folyamata . . . . .	42
6.3. A backend összes adatbázistáblája . . . . .	43
6.4. A Kafka Topics UI felülete . . . . .	45
6.5. Az Eureka service discovery felülete . . . . .	46
7.1. A JMeter tesztelés során használt számai . . . . .	48
7.2. Helpdesk backend terheléses teszt 100 felhasználóval . . . . .	49
7.3. Helpdesk backend terheléses teszt egy példánnyal . . . . .	50
7.4. A terheléses tesztben résztvevő alkalmazások . . . . .	51
7.5. A csúcsteljesítmény vizsgálata során vizsgált legfontosabb metrikák a Grafana monitorozó eszköz felületén . . . . .	52
7.6. Helpdesk backend terheléses teszt Nginx nélkül . . . . .	53
7.7. Helpdesk backend terheléses teszt módosított HikariCp beállításokkal . . . . .	54
7.8. HikariCP adatbázis-kapcsolatai a Grafana felületén . . . . .	55
7.9. A backend egy példányának a processzor használata a Grafana felületén . . . . .	56
7.10. Helpdesk backend terheléses teszt három példánnyal . . . . .	56

A. függelék

OpenApi dokumentáció

# helpdesk-backend

## Overview

### Version information

Version : 1.0.0

### URI scheme

Schemes : HTTP

### Tags

- Audit
- Email
- EmailThread
- Queue
- User

## Paths

### Get the email threads that is related to the user.

```
GET /api/audit/email-thread/
```

### Responses

HTTP Code	Description	Schema
200	Returns the current state of the email threads, that are related to the user.	< <a href="#">EmailThread</a> > array
403	User not authorized.	No Content

### Produces

- `application/json`

## Tags

- Audit

## Get history of an email thread by UUID.

```
GET /api/audit/email-thread/{emailThreadId}
```

### Parameters

Type	Name	Schema
Path	<b>emailThreadId</b> <i>required</i>	string (uuid)

### Responses

HTTP Code	Description	Schema
200	Returns history of the email thread.	< <a href="#">EmailThreadAudit</a> > array
403	User not authorized.	No Content

### Produces

- `application/json`

## Tags

- Audit

## Get the emailThreads of the authenticated user with a specific status.

```
GET /api/email-thread/assigned-to-me
```

### Parameters

Type	Name	Description	Schema	Default
Header	<b>status</b> <i>optional</i>	EmailThread status	string	"OPEN"

## Responses

HTTP Code	Description	Schema
200	Return found emailThreads.	< <a href="#">EmailThread</a> > array
403	User not authorized.	No Content

## Produces

- `application/json`

## Tags

- EmailThread

# Get the emailThreads of the authenticated user's queue with a specific status.

```
GET /api/email-thread/status
```

## Parameters

Type	Name	Description	Schema	Default
Header	<b>status</b> <i>optional</i>	EmailThread status	string	"CHANGE_QUEUE"

## Responses

HTTP Code	Description	Schema
200	Return found emailThreads.	< <a href="#">EmailThread</a> > array
403	User not authorized.	No Content

## Produces

- `application/json`

## Tags

- EmailThread

# Get all the unassigned emailThreads from the users queue.

```
GET /api/email-thread/unassigned
```

## Responses

HTTP Code	Description	Schema
200	Return unassigned emailThreads.	< <a href="#">EmailThread</a> > array
403	User not authorized.	No Content

## Produces

- `application/json`

## Tags

- EmailThread

# Get EmailThread by UUID.

```
GET /api/email-thread/{emailThreadId}
```

## Parameters

Type	Name	Schema
Path	<b>emailThreadId</b> <i>required</i>	string (uuid)

## Responses

HTTP Code	Description	Schema
200	Returns email. <b>Headers :</b> <b>ETag</b> (string) : The version of the EmailThread.	<a href="#">EmailThreadVersion</a>
403	User not authorized.	No Content
404	EmailThread with the given ID does not exists.	No Content

## Produces

- `application/json`

## Tags

- EmailThread

# Change the owner, or the status of the emailThread.

```
PATCH /api/email-thread/{emailThreadId}
```

## Parameters

Type	Name	Description	Schema
Path	<b>emailThreadId</b> <i>required</i>	Id of the emailThread	string (uuid)
Query	<b>ifMatch</b> <i>required</i>	If-Match header	string
Body	<b>body</b> <i>required</i>	EmailThread containing the new properties	<a href="#">EmailThread</a>

## Responses

HTTP Code	Description	Schema
200	New fileds of the emailThread has been set	No Content

HTTP Code	Description	Schema
403	User not authorized.	No Content
404	EmailThread with the given ID does not exists.	No Content
409	EmailThread concurrently changed	No Content

## Produces

- `application/json`

## Tags

- EmailThread

# Send an Email.

POST /api/email/send

## Parameters

Type	Name	Description	Schema
Body	<b>body</b> <i>required</i>	Email to send	<a href="#">Email</a>

## Responses

HTTP Code	Description	Schema
200	Returns email.	<a href="#">Email</a>
403	User not authorized.	No Content

## Produces

- `application/json`

## Tags

- Email



# Get Email by UUID.

```
GET /api/email/{emailId}
```

## Parameters

Type	Name	Schema
Path	<b>emailId</b> <i>required</i>	string (uuid)

## Responses

HTTP Code	Description	Schema
200	Returns email.	<a href="#">Email</a>
403	User not authorized.	No Content
404	Email with the given ID does not exists.	No Content

## Produces

- `application/json`

## Tags

- Email

# Change ths status of the email's read property'.

```
PATCH /api/email/{emailId}
```

## Parameters

Type	Name	Description	Schema
Path	<b>emailId</b> <i>required</i>		string (uuid)
Body	<b>body</b> <i>required</i>	Email has been read	< string, boolean > map

## Responses

HTTP Code	Description	Schema
200	New status of the email has been set	No Content
403	User not authorized.	No Content
404	Email with the given ID does not exists.	No Content

## Produces

- `application/json`

## Tags

- Email

## Get all queue.

```
GET /api/queue/all
```

## Responses

HTTP Code	Description	Schema
200	Returns queues.	< <a href="#">Queue</a> > array
403	User not authorized.	No Content

## Produces

- `application/json`

## Tags

- Queue

## Get details of the authenticated user.

```
GET /api/user/details
```

## Responses

HTTP Code	Description	Schema
200	Return authenticated user details.	<a href="#">User</a>
403	User not authorized.	No Content
404	User with the given ID does not exists.	No Content

## Produces

- `application/json`

## Tags

- User

## Change ths active user's queue'.

PATCH /api/user/queue

## Parameters

Type	Name	Description	Schema
Body	<b>body</b> <i>required</i>	New property	< string, string (uuid) > map

## Responses

HTTP Code	Description	Schema
200	New queue of the user has been set	No Content
403	User not authorized.	No Content
404	Queue with the given ID does not exists.	No Content

## Produces

- `application/json`

## Tags

- User

## AutoComplete search for User. Searches for username with like.

```
GET /api/user/search/autocomplete
```

### Parameters

Type	Name	Description	Schema
Query	<b>queueId</b> <i>required</i>	Queue id	string (uuid)
Query	<b>username</b> <i>optional</i>	Username, min length 1	string

### Responses

HTTP Code	Description	Schema
200	First (size) count values about BIC field.	< <a href="#">User</a> > array
403	User not authorized.	No Content
422	Operation not permitted.	No Content

### Produces

- `application/json`

## Tags

- User

# Definitions

## Attachment

Full DTO for attachment

Name	Description	Schema
<b>content</b> <i>required</i>	The attachment is part of this email content.	<a href="#">Content</a>
<b>contentType</b> <i>required</i>	The content type of the data.	string
<b>data</b> <i>required</i>	The binary bytearray of the data.	< string (byte) > array
<b>dataAsStream</b> <i>optional</i>		<a href="#">ByteArrayOutputStream</a>
<b>filename</b> <i>required</i>	The name of the data.	string

## ByteArrayOutputStream

Type : object

## Content

Full DTO for content

Name	Description	Schema
<b>attachments</b> <i>optional</i>	All the attachment the content has.	< <a href="#">Attachment</a> > array
<b>body</b> <i>required</i>	The body of the email.	string
<b>html</b> <i>required</i>	The body should be read as an html document.	boolean

## Email

Full DTO for email

Name	Description	Schema
<b>content</b> <i>required</i>	The content of the email.	<a href="#">Content</a>

Name	Description	Schema
<b>direction</b> <i>optional</i>	The direction of the email.	enum (IN, OUT)
<b>emailThread</b> <i>optional</i>	The emailThread which contains this email.	<a href="#">EmailThread</a>
<b>header</b> <i>required</i>	The header of the email.	<a href="#">Header</a>
<b>id</b> <i>optional</i>	Unique internal identifier <b>Example</b> : "06484c9f-6f59-4b9f-ad5e-aaaa0ec332cc"	string (uuid)
<b>parentId</b> <i>optional</i>	Reply to this email.	string (uuid)
<b>processed</b> <i>optional</i>	Processed at.	string (date-time)
<b>read</b> <i>optional</i>	The email has been read.	boolean

## EmailThread

Full DTO for thread

Name	Description	Schema
<b>description</b> <i>optional</i>	The description of the emailThread.	string
<b>emails</b> <i>required</i>	The emails related to the emailThread.	< <a href="#">Email</a> > array
<b>id</b> <i>required</i>	Unique internal identifier <b>Example</b> : "06484c9f-6f59-4b9f-ad5e-aaaa0ec332cc"	string (uuid)
<b>queue</b> <i>required</i>	The queue of the emailThread.	<a href="#">Queue</a>
<b>status</b> <i>required</i>	The status of the emailThread.	enum (OPEN, RESOLVED, CLARIFICATION, CHANGE_QUEUE)

Name	Description	Schema
<b>user</b> <i>optional</i>	The user who is working on the emailThread.	<a href="#">User</a>

## EmailThreadAudit

Full DTO for thread audit

Name	Description	Schema
<b>description</b> <i>optional</i>	Description.	string
<b>id</b> <i>optional</i>	Unique internal increasing index <b>Example</b> : 5	integer (int32)
<b>queue</b> <i>optional</i>	Queue name.	string
<b>status</b> <i>required</i>	Status.	enum (OPEN, RESOLVED, CLARIFICATION, CHANGE_QUEUE)
<b>type</b> <i>optional</i>	Type of the change.	enum (insert, update, delete)
<b>user</b> <i>optional</i>	Username.	string

## EmailThreadVersion

EmailThread with version number

Name	Description	Schema
<b>emailThread</b> <i>required</i>	The emailThread.	<a href="#">EmailThread</a>
<b>version</b> <i>required</i>	Version number of the EmailThread <b>Example</b> : 35	integer (int32)

# Header

Full DTO for header

Name	Description	Schema
<b>from</b> <i>required</i>	The email received from this address.	string
<b>inReplyTo</b> <i>optional</i>	The messageID of the previous email. See rfc5322.	string
<b>messageId</b> <i>optional</i>	The globally unique identifier (messageID) of the corresponding email. See rfc5322.	string
<b>references</b> <i>optional</i>	The References identifier. It contains the messageIDs of the previous emails. See rfc5322.	string
<b>subject</b> <i>optional</i>	The subject of the mail.	string
<b>to</b> <i>required</i>	The email sent to this address.	string

# Queue

Full DTO for queue

Name	Description	Schema
<b>description</b> <i>optional</i>	The description of the queue.	string
<b>email</b> <i>required</i>	The queue belongs to this address.	string
<b>id</b> <i>required</i>	Unique internal identifier <b>Example</b> : "06484c9f-6f59-4b9f-ad5e-aaaa0ec332cc"	string (uuid)
<b>name</b> <i>required</i>	The unique name of the queue.	string

# User

Full DTO for users



Name	Description	Schema
<b>email</b> <i>required</i>	The email address of the user.	string
<b>id</b> <i>optional</i>	Unique internal identifier <b>Example :</b> "06484c9f-6f59-4b9f-ad5e-aaaa0ec332cc"	string (uuid)
<b>queue</b> <i>required</i>	The user can operate on this queue.	<a href="#">Queue</a>
<b>username</b> <i>required</i>	Username.	string