

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 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

Konzulens: Dr. Nagy Elemér Károly

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



Budapest, 2020 december.

Kivonat

Dolgozatomban ismertetem egy mikroszerviz alapú elosztott alkalmazás felépítését, a tervezés során fellépő általános problémákat, valamint ezekre a problémákra adható megoldásokat.

Nagy vonalakban és feladatspecifikusan áttekintem a felhasznált technológiákat és módszertanokat.

Ezek tükrében bemutatom a létrehozott szoftvert, infrastruktúrát és az üzemeltetéséhez szükséges eszközöket.

Tartalomjegyzék

Tartalomjegyzék	vi
Ábrák jegyzéke	viii
Bevezetés	1
1. Üzleti igények	2
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	2
1.2. Nem funkcionális igények	3
1.2.1. Skálázhatóság	3
1.2.2. Granuláris felosztottság	4
1.2.3. Mérhető indikátorok	4
1.2.4. I18N	5
2. Technológiai áttekintés	6
2.1. Mikroszerviz architektúra	6
2.2. Hexagonális architektúra	7
2.3. Rétegek szeparálása	8
2.4. Konkurencia kezelése	8
2.5. Alkalmazások szeparálása	9
2.6. Apache Kafka	10
2.7. Angular	10
2.8. Spring Boot	11
3. Az alkalmazás felépítése	12
3.1. Legfontosabb komponensek	12
3.2. Adatbázis UML diagram	13

3.3. E-mail fogadásának és küldésének folyamata	13
4. Implementáció	16
4.1. Mikroszerviz infrastruktúra	16
4.1.1. Nginx	16
4.1.2. Docker konténerizáció	16
4.1.3. Metrikák	17
4.2. E-mail kliens	17
4.2.1. E-mail szabvány	17
4.3. Helpdesk backend	18
4.3.1. Spring Boot	18
4.3.2. Adatbázis	19
4.3.3. Pesszimista konkurenciakezelés	19
4.3.4. Optimista konkurenciakezelés	20
4.3.5. Egyéb eszközök	20
4.4. Helpdesk frontend	20
4.4.1. Kommunikáció a backenddel	20
4.4.2. Komponensek	21
4.4.3. Futtatási környezet	21
4.5. Keycloak	21
4.5.1. Jogosultságkezelés	21
4.5.2. JSON Web Token	23
4.6. Kafka	23
4.7. Helpdesk backend és a Keycloak elkülönítése	23
5. Alkalmazás bemutatása	25
5.1. Alkalmazás elindítása	25
5.2. Több példány	25
5.3. Deployment	26
5.4. E-mail fogadásának és küldésének folyamata	26
5.5. Adatbázistáblák	27
5.5.1. Liquibase	29
5.5.2. Hibernate Envers	30
5.6. Apache Kafka	30
5.7. Eureka	31
6. Terheléses tesztelés	33
6.1. Terheléses teszt	33

6.2.	Apache JMeter	33
6.3.	Átlagos teljesítmény vizsgálata	34
6.4.	Csúcs teljesítmény vizsgálata	35
6.5.	Szűk keresztmetszet meghatározása	36
6.5.1.	Nginx	37
6.5.2.	HikariCP	39
6.5.3.	Megnövekedett processzor igény	41
6.5.4.	Szűk keresztmetszet meghatározása	42
6.6.	Összevetés a követelményekkel	42
6.6.1.	Átlagos teljesítmény vizsgálata	43
6.6.2.	Csúcs teljesítmény vizsgálata	43
7.	Továbbfejlesztési lehetőségek	44
7.1.	A deploymentről	44
7.2.	A kódról	44
	Irodalomjegyzék	47
A.	OpenApi dokumentáció	49

Ábrák jegyzéke

1.1.	Az e-mailszálak státuszváltozásai	3
1.2.	Elérhető funkciók	4
2.1.	Hexagonális alkalmazások felépítése	8
3.1.	A legfontosabb komponensek	12
3.2.	A backend legfontosabb adatbázistáblái	13
3.3.	A bejövő és kimenő e-mail útja	15
4.1.	E-mail kliens szekvencia diagramja	18
4.2.	Helpdesk backend szekvencia diagramja	19
4.3.	Helpdesk frontend szekvencia diagramja	22
5.1.	Deployment diagram	27

5.2. E-mail fogadásának és küldésének folyamata	28
5.3. A backend összes adatbázistáblája	29
5.4. A Kafka Topics UI felülete	31
5.5. Az Eureka service discovery felülete	32
6.1. A JMeter –tesztelés során használt– számai	34
6.2. Helpdesk backend terheléses teszt 100 felhasználóval	35
6.3. Helpdesk backend terheléses teszt egy példánnyal	36
6.4. A terheléses tesztben résztvevő alkalmazások	37
6.5. A csúcsteljesítmény vizsgálata során vizsgált legfontosabb metrikák	38
6.6. Helpdesk backend terheléses teszt Nginx nélkül	39
6.7. Helpdesk backend terheléses teszt módosított HikariCp beállításokkal	40
6.8. HikariCP adatbázis-kapcsolatai	40
6.9. A backend egy példányának a processzor használata	41
6.10. Helpdesk backend terheléses teszt három példánnyal	42

Bevezetés

Ahogy az O'Really által az év elején készített felmérésből [1] is látszik, a mikroszerviz alapú alkalmazások egyre nagyobb népszerűségnek örvendenek. Egyre több cég szeretné lecserélni meglévő monolit rendszerét, vagy a szükséges új funkciókat a régebbi rendszertől függetlenül, hibrid rendszerben valósítana meg.

Mint az a wiredelta cikkéből [2] is látszik, a mikroszerviz architektúrának számtalan előnye van. 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 az ilyen elvek mentén kialakított alkalmazások, természetükből adódóan időtállóbbak lesznek. Ha el tudjuk érni, hogy egy alkalmazás valóban csak egy funkcióért kell hogy felelős legyen, azzal a problémamegoldás analitikus oldalát emeljük rendszerszintre.

Éppen ezért, a mikroszerviz architektúra legnagyobb előnye szerintem a rendszerezésből következik.

1. fejezet

Üzleti igények

Ebben a fejezetben szeretném bemutatni a Helpdesk alkalmazás felé megfogalmazott üzleti igényeket.

1.1. Funkcionális igények

1.1.1. E-mail fogadása és küldése

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

A rendszernek képesnek kell lennie több e-mail cím kezelésére. A beérkező új üzeneteket a címzettnek megfelelő előre definiált sorhoz kell hozzárendelni.

1.1.2. E-mail szálak kezelése

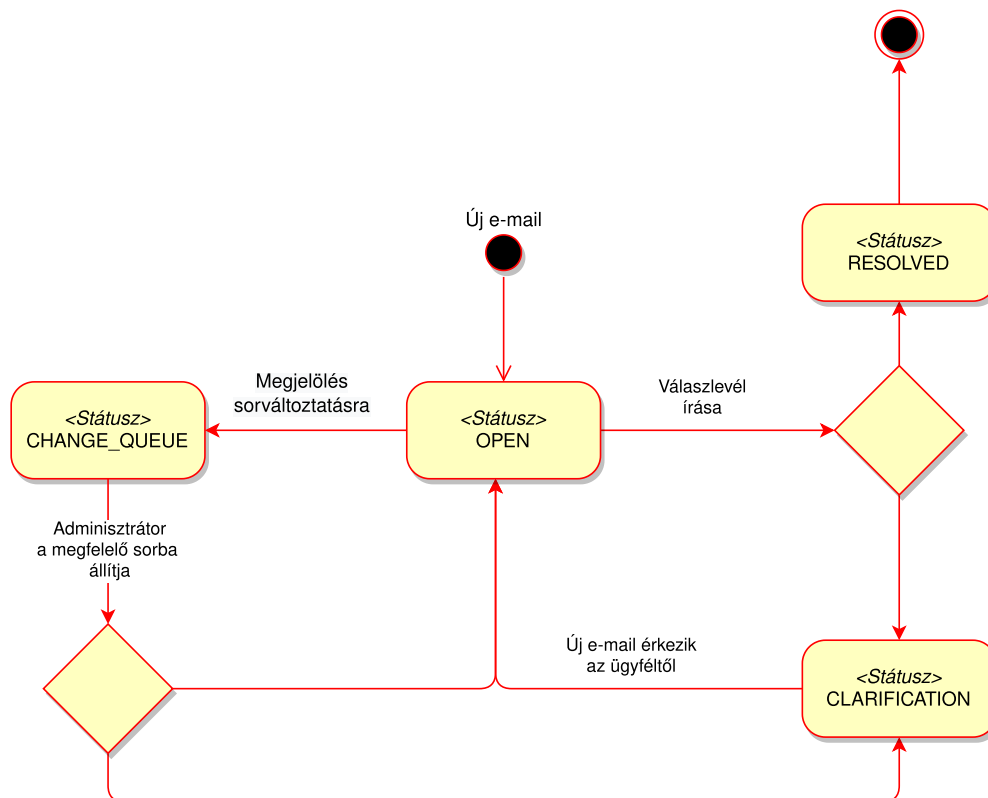
A rendszer által kezelt üzenetek szálakba rendezve érhetőek el. Egy szál az ügyfél és a felhasználó közötti üzenetváltásokból épül fel.

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

1.1.3. Több felhasználó

A rendszert egyszerre több felhasználó használhatja. Minden felhasználó csak a saját emailszárait kezelheti, csak azokra válaszolhat.

Minden felhasználó pontosan egy az [1.1.1](#) fejezetben említett sorhoz tartozik. Csak az ugyanabba a sorba tartozó e-mail szál rendelhető hozzá. A számára kijelölt szálakat képes –a saját során belül– más felhasználóhoz rendelni.



1.1. ábra. Az e-mailszálak státuszváltozásai

Forrás: saját ábra

A felhasználók eltérő jogkörökkel rendelkezhetnek. Az adminisztratív jogkörrel rendelkező felhasználó végzi az új emailszál felhasználóhoz rendelését, valamint a *change queue* státuszban (1.1 ábra) lévő üzenetszálak új sorba irányítását.

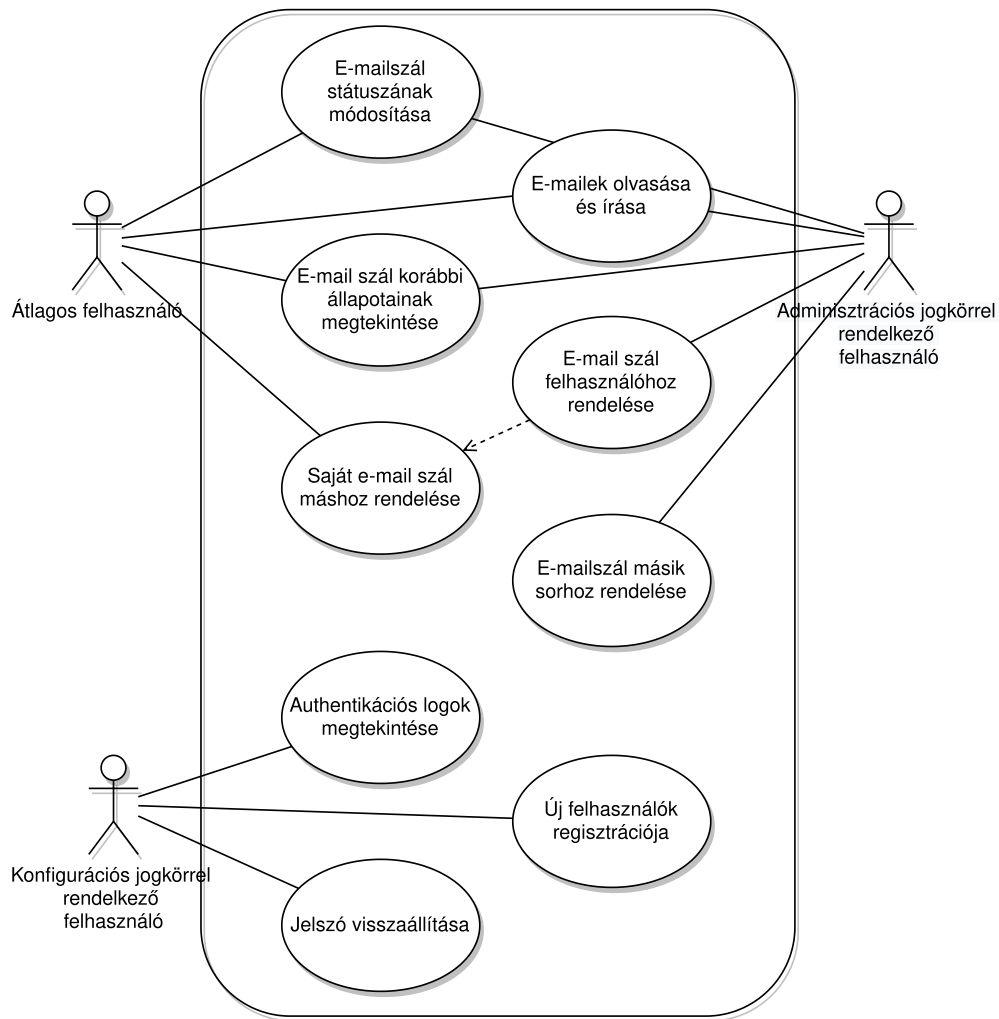
A konfigurációs jogkörrel rendelkező felhasználó feladata más felhasználók regisztrálása, valamint az alkalmazásban használt jogkörök (*role-ok*) kezelése. Lehetősége van továbbá autentikációs logok megtekintésére, jelszó visszaállítására és más felhasználók megszemélyesítésére (*impersonate*).

A felhasználói felületen elérhető funkciókat az 1.2 ábra foglalja össze.

1.2. Nem funkcionális igények

1.2.1. Horizontális skálázhatóság

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



1.2. ábra. Elérhető funkciók jogosultság szerint csoportosítva

Forrás: saját ábra

1.2.2. Granuláris felosztottság

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

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 kiszolgálnia másodpercenként. A várható csúcsteljesítmény 5 000–10 000 lekérdezés másodpercenként. A felhasználók szá-

méra elfogadható legnagyobb válaszidő 3 másodperc/lekérés.

1.2.4. I18N

A felhasználói, adminisztratív és karbantartói felületek angol nyelven érhetőek el. Több nyelv kezelése nem szükséges.

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 szervizekkel 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 vertikális¹ és horizontális skálázhatóság².

¹több processzor vagy memória bevonása

²újabb példányok futtatása

konténerizálás a szerviz futtatása saját elszeparált környezetében hardveres virtualizáció segítségével nélkül.

szerviz felderítés a rendszer által nyújtott szolgáltatások, szervizek automatikus felfedezhetősége³.

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

monitorozás az önálló szervizek állapotának felügyelése. A monitorozás során nyújtott metrikák kiterjedhetnek a felhasznált memória mennyiségére, processzorigényére, 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 szembenegy a korábban elterjedt réteges elrendezéssel.

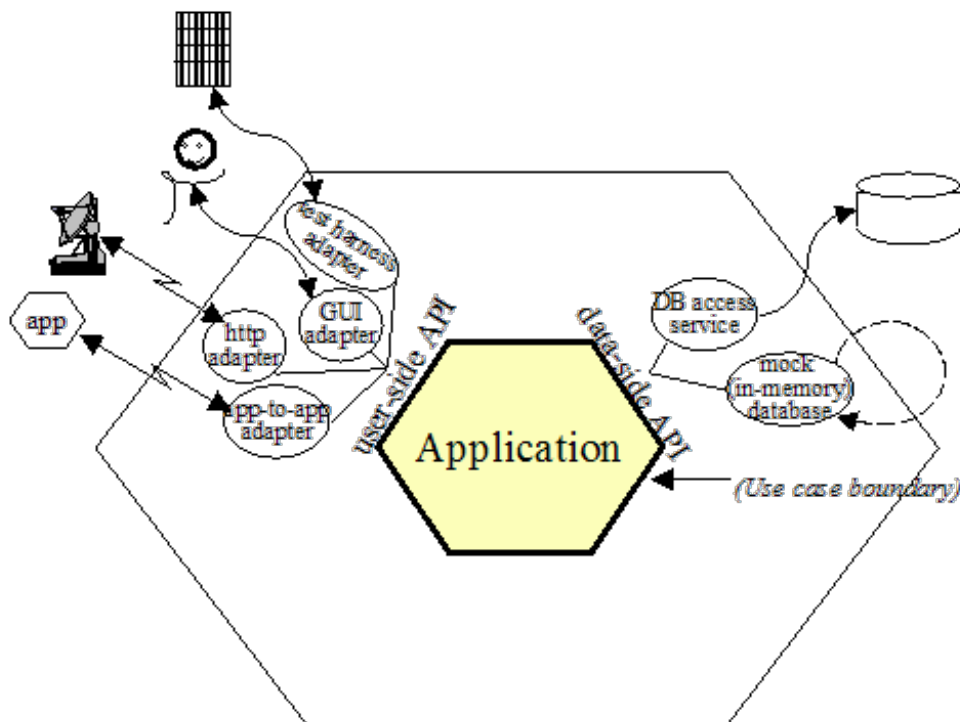
Az eredeti szándék mögöttese az alkalmazás függetlenítése mindennemű külső függőségtől⁴, így lehetővé téve az üzleti és a technikai igények nagy mértékű szeparálását. Egy absztrakt port feladata kell legyen a külvilággal való kapcsolat, így az üzleti logika csak az üzenet tartalmáért felelős, az üzenetküldés módjáért már nem.

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

- Könnyen, és önmagukban is tesztelhetőek. 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.

³angolul *service discovery*-nek hívják

⁴például 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.

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 szervizek 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 CQRS⁵ 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 újrajátzásával 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.

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

⁵Command Query Responsibility Segregation

A *Saga* egy hosszú életű elosztott tranzakció [8]. A folyamat lépései sorban hajódnak 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, open source alkalmazás [9].

A feladó az üzenetet nem közvetlenül a fogadónak küldi, hanem egy üzenetbrókeren keresztül egy (*topic*)-ba teszi közzé. A fogadó fél hogy megkaphassa az üzenetet, feliratkozik az adott témára.

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 *node* esetleges kiesése esetén a többi *node* át tudja venni a kiesett *node* szerepét.

A üzenetbörkerek összehaangolását a Zookeeper szervíz végzi. Mivel minden kafka bróker beregisztálja magát a szervízbe, 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 az MVC-elveit követő alkalmazást létrehozni.

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

2.8. Spring Boot

A Spring Boot egy a Springre épülő keretrendszer. Mindkét rendszer alapja a függőség befecskendezése⁶, 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 meghatározni⁷ ezzel lecsökkentve a konfigurációval töltött időt,
- valamint sok gyakran visszatérő problémára⁸ nyújt könnyen elérhető megoldást.

⁶Angolul *Dependency Injection*

⁷A Spring Boot dokumentációban ezt röviden *convention over configuration*-nek hívják

⁸Például: metrikák, biztonság, adattárolás

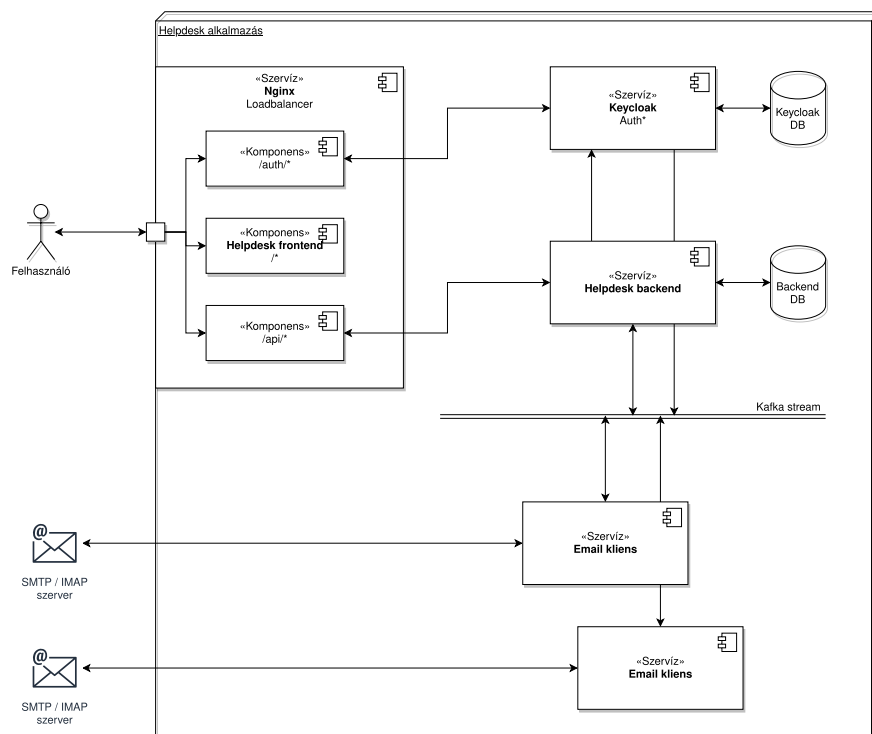
3. fejezet

Az alkalmazás felépítése

Ebben a fejezetben szeretnék egy átfogó képet adni a helpdesk alkalmazásról. Az egyes komponensek részletes leírása a 4. fejezetben található.

3.1. Legfontosabb komponensek

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



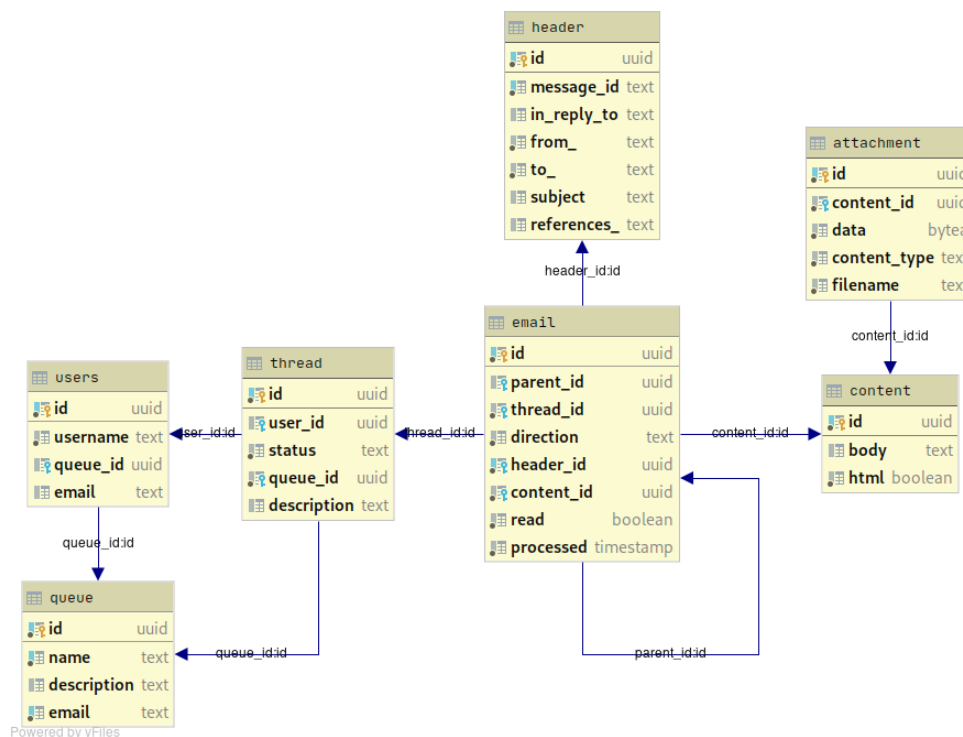
3.1. ábra. A legfontosabb komponensek

Forrás: saját ábra

- A felhasználó az nginx-en (4.1.1 pont) keresztül éri el a helpdesk alkalmazást.
- Az nginx dönti el, hogy melyik URL-t melyik szerviz szolgálja ki.
- Az email kliens és a helpdesk backend kafka streamen keresztül éri el egymást.
- Az email kliensek kezelik az e-mail szerverekkel való adatcserét.

3.2. Adatbázis UML diagram

A helpdesk backend adatbázis legfontosabb tábláit a 3.2. ábra tartalmazza. Az ábrán nem szerepelnek az audithoz, és a liquibase által használt táblák (4.3.2 pont). Az 5. fejezetben található 5.3. ábra tartalmazza az adatbázis összes tábláját.



3.2. ábra. A backend legfontosabb adatbázistáblái

Forrás: saját ábra

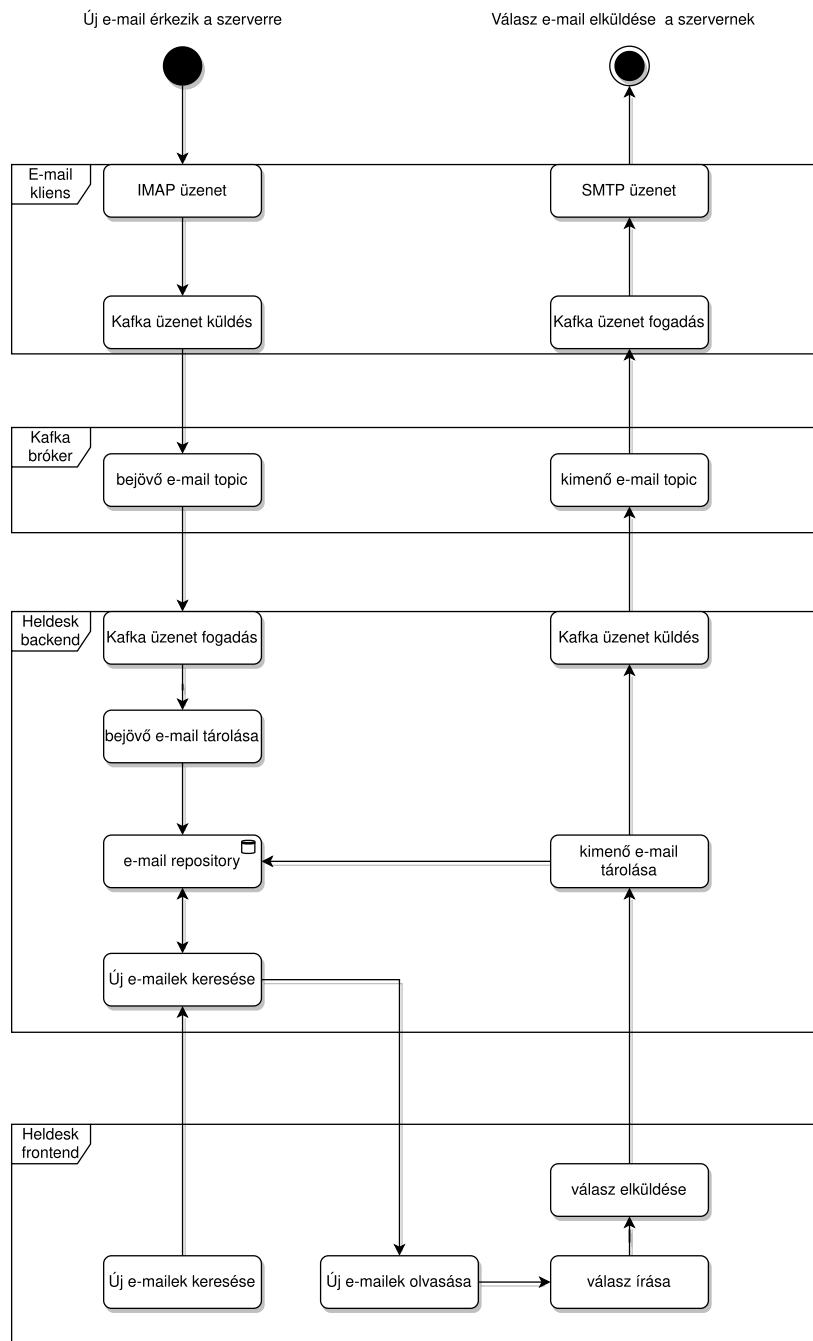
3.3. E-mail fogadásának és küldésének folyamata

A könnyebb átláthatóság érdekében, a folyamatokat egy e-mail szemszögéből mutatom be a 3.3. ábrán. Az e-mail fogadása során:

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 *topicban*.
3. A bejövő e-mailek *topicra* 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 során:

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 *topicba* közzéteszi az új üzenetet.
4. Az e-mail cím specifikus kimenő e-mailek *topicra* 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

Implementáció

Szeretném külön-külön bemutatni az egyes komponenseket. Kiemelni a komponensek által megvalósított funkciókat, és a megvalósítás szempontjából fontos részleteket.

4.1. Mikroszerviz infrastruktúra

4.1.1. Nginx

Az nginxnek három elkülönülő szerepe van:

- A helpdesk frontend alkalmazásszervereként működik (lásd 2.7 pont)
- Routingot valósít meg, rajta keresztül érhető el a helpdesk backend és a keycloak szerviz
- HTTP cache-ként működik a frontend és a backend között, illetve a frontend és a keycloak között

A loadbalancer funkcionalitás a docker round-robin DNS-en (4.1.2) keresztül valósul meg.

4.1.2. Docker konténerizáció

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

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

4.1.3. Metrikák

A springes alkalmazásaim egy-egy HTTP endpointon keresztül érhetőek el a prometheus számára (*/actuator/prometheus*) és induláskor beregisztrálják magukat az eureka¹ szerverbe.

A prometheus² az eurekán keresztül találja meg az instanceokat, és gyűjti össze a metrikákat. Az alkalmazások információt küldenek a Kafka konnektorukról, REST interfészükről és az adatbázis kapcsolatukról³.

A Prometheus által összegyűjtött adatokat grafanában⁴ ábrázolom.

4.2. E-mail kliens

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

A 4.1. 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.

4.2.1. E-mail szabvány

Az elküldött üzenetek megfelelnek az *rfc5322* szabványnak, különös tekintettel a 3.6.4. pontban [13] meghatározott mezőkre:

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

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

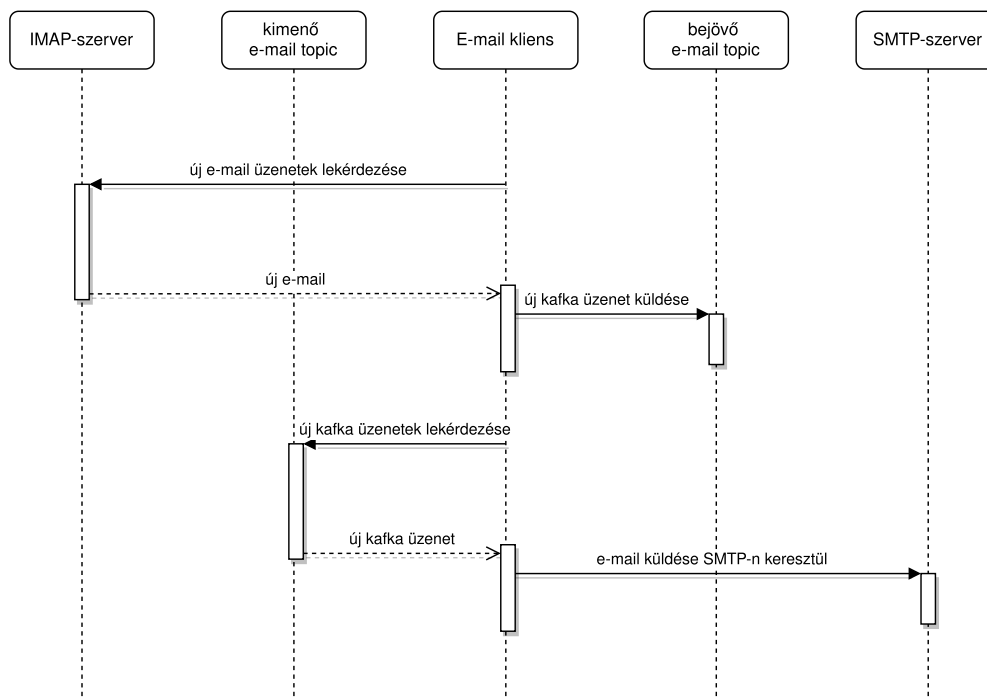
References azonosítja az üzenet szálat, értéke az eredeti üzenetek *Message-ID*-jai vesszővel elválasztva.

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

²A Prometheus egy open source monitorozó eszköz. 15 másodpercenként lekérdezi a szervizek állapotát.

³HikariCP-t használok JDBC kapcsolathoz

⁴A Grafana egy open source elemző és megjelenítő web alkalmazás



4.1. ábra. E-mail kliens szekvencia diagramja

Forrás: saját ábra

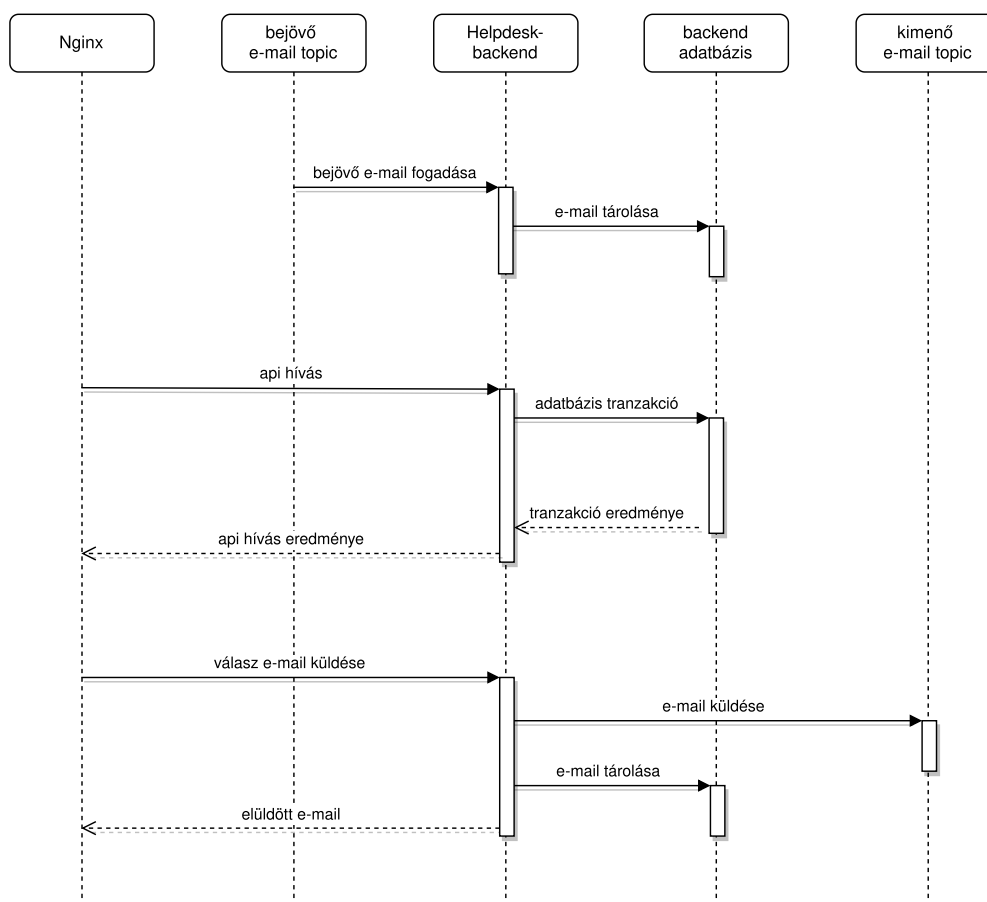
4.3. Helpdesk backend

A backend felelős az e-mail szálakkal kapcsolatos üzleti feladatok ellátásáért. A 4.2. ábrán láthatóak a helpdesk backend funkciói:

- fogadja az *email.in.v1.pub* kafka topic-ból érkező e-maileket,
- kiszolgálja a frontend Nginx-en keresztül érkező kéréseit,
- a megfelelő kafka topic-ba írja az elküldendő üzeneteket,
- tárolja az e-mail szálakkal kapcsolatos adatokat.

4.3.1. Spring Boot

A forráskód 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 webet a *rest controllerek*hez, a prometheus-t és az actuatort a metrikák elkészítéséhez használtam.



4.2. ábra. Helpdesk backend szekvencia diagramja

Forrás: saját ábra

4.3.2. Adatbázis

PostgreSQL adatbázishoz kapcsolódást a HikariCP-n keresztül a Spring kezeli. Az adatok kezelését Hibernate⁵-en keresztül, az adatbázis verziókövetését Liquibase-en (5.5.1 pont) keresztül valósítom meg.

Az e-mail szálak audit információinak és verzióinak követésére a Hibernate Envers (5.5.2 pont) eszközt használom. Az Envers a neki létrehozott táblában automatikusan követi a megjelölt Hibernate objektumok állapotát.

4.3.3. Pesszimista konkurenciakezelés

Pesszimista konkurenciakezelésre jó példával szolgál a Liquibase (5.5.1 pont) működése.

Minden indítás során a Liquibase –az adatbázis módosításának befejezéséig– zárolja a *databasechangeloglock* táblát. Így –a várakozás miatt– egyszerre mindig maximum egy

⁵A Hibernate egy JPA implementáció, ami objektum relációs leképztést valósít meg

Liquibase példány tud elindulni, és módosításokat végrehajtani.

4.3.4. Optimista konkurenciakezelés

A 2.4 pontban ismertetett optimista konkurenciakezelést 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 történetekről. A felhasználó ilyenkor az oldal frissítése után, megvizsgálja az aktuális állapotot, és –amennyiben a módosításaira még mindig szükség van– újból kezdi a folyamatot.

4.3.5. Egyéb eszközök

A *DTO*-k és az *entity*k közötti leképezést a Mapstruct (2.3) segítségével végzem. A REST *endpoint*ok dokumentációját Swagger segítségével generálom. A Swagger a felannotált osztályokból és metódusokból szabványos OpenApi dokumentációt készít. A dokumentációt az A függelékben csatoltam a dolgozatomhoz.

4.4. 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.

4.4.1. Kommunikáció a backenddel

A backenddel való kommunikáció REST protokollon keresztül zajlik, a szükséges *service*-eket az OpenApi dokumentációból (4.3.5) 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.

4.4.2. Komponensek

Az egységes megjelenés és az ismerős kinézet miatt, a komponenseim alapján az Angular Material UI könyvtárat választottam. A könyvtár népszerű az Angular fejlesztők körében, mert a leggyakrabban előforduló felhasználói igényekre elérhető benne kész, könnyen használható megoldás.

A válasz e-mail létrehozására az open source Quill szövegszerkesztőt használtam. Egyszerűen beilleszthető az Angular környezetbe, és a felhasználó számára intuitív kezelőfelülettel rendelkezik.

4.4.3. 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 4.3 ábra). A backend felé induló REST kéréseket a loadbalancer (4.1.1) osztja szét a rendelkezésre álló példányok között.

A frontend működését, és függőségeit a 4.3. ábra tartalmazza.

4.5. Keycloak

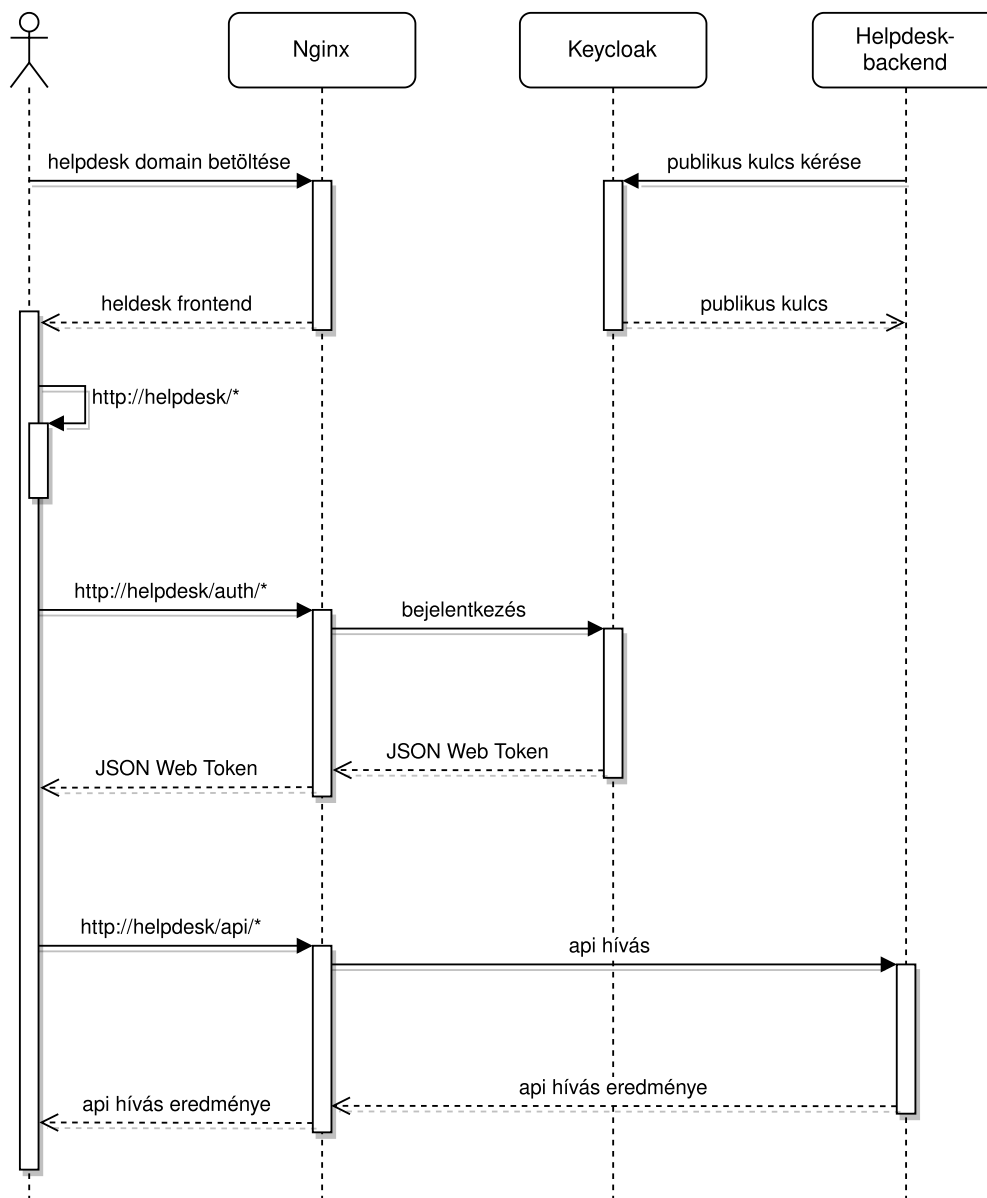
A Keycloak egy open source jogosultság- és hozzáférés-kezelő. Támogatja az LDAP-ot, SSO-t és a kétlépcsős azonosítást [14].

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, szerkeszthetőek az aktuálisan érvényes szerepkörök, és –hibakezelési céllal– megismerésíthetőek a felhasználók.

4.5.1. 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) feladatiért felelős.



4.3. ábra. Helpdesk frontend szekvencia diagramja

Forrás: saját ábra

A *helpdesk realmon* belül további két jogkört különböztetnek meg. Az *admin_user* szerepbe tartozó felhasználók képesek más e-mail szálaikat is kezelni, míg a csupán *regular_user* jogkörbe tartozóak csak a saját e-mail szálaikhoz férhetnek hozzá.

4.5.2. JSON Web Token

A jogosultságkezelés technikai alapját az *rfc7519*-es szabványban [15] leírt JSON Web Token (JWT) adja.

A keycloak szervere által digitálisan aláírt token tartalmazza a felhasználó jogosultságait. A frontend minden HTTP lekérdezéshez csatolja a keycloaktól kapott azonosítót. A backend hitelesíti a token a keycloak publikus kulcsával (lásd 4.3 ábra), és a megfelelő jogosultság megléte esetén engedélyezi a hozzáférést az erőforráshoz.

4.6. Kafka

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 keresztül. 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 (4.7) miatt a felhasználók adatai egy külön kafka *topic*-ban érhetőek el. Bármelyik mikroszerviznek szüksége lenne valamilyen felhasználóval kapcsolatos információra, azokat a *topic* végigolvasásával megkaphatja.

4.7. Helpdesk backend és a Keycloak elkülönítése

A felhasználók adataiért a Keycloak (4.5), az e-mail szálaikért pedig a backend (4.3) felelős. Az üzleti igény megköveteli hogy a felhasználók e-mail sorokhoz, és az e-mail szálaik 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 (4.5.2) 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 (4.6) *topic*-ban követhetőek nyomon.

A Keycloak Kafka integrációjának céljából hoztam létre a *keycloak-plugin* (5.1 ábra) maven modult. A Keycloak esemény figyelőként működő plugin, a megfigyelt eseményekről kafka üzenetet küld a kijelölt *topic*-ba. A helpdesk backend –a *topic* üzeneteit

olvasva– tartja karban a *users* táblát (3.2, 5.3 ábra). Így a helpdesk backend a felhasználókról mindig aktuális információval rendelkezik.

5. 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-ot,
- a *travel* sorhoz tarozó helpdesk.gdf.travel@yandex.com-ot,
- és a *theater* sorhoz tarozó h.gdf.theater@gmx.com-ot.

5.1. Alkalmazás elindítása

Az alkalmazás a *start.sh* bash *script*tal indítható el. A *script* két dolgot csinál:

1. a *docker-compose* paranccsal elindítja a docker *containereket* (4.1.2 pont),
2. „helpdesk” domain névvel hozzáadja az Nginx (4.1.1 pont) IP címét a */etc/hosts* állományhoz.

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

5.2. Több példány

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

- a helpdesk-backendből három,
- a *theater* sort kezelő email-kliensből egy,
- a *travel* sort kezelő email-kliensből kettő,

- a *generic* sort kezelő email-kliensből három,
- és a Kafka brókerből (5.6) szintén három darab.

A példányok metrikáit (4.1.3 pont) nyomon lehet követni az erre a célra létrehozott Grafana oldalon (?? ábra). Az oldal elérhető a *Spring metrics* menüpont alatt.

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

- a legfelső sorban a Java Virtual Machine, által akutálisan felhasznált Heap space,
- alatta az aktuális REST lekérések száma,
- a harmadik sorban a feldolgozott Kafka üzenetek száma,
- míg az utolsó sorban a Trace log bejegyzések száma látható.

5.3. Deployment

A könnyebb bemutathatóság érdekében a szemléletesebb szervizeket –hogya ne a docker daemon által kiosztott IP címen keresztül kelljen elérni– a docker hálózaton kívül is elérhetővé tettem.

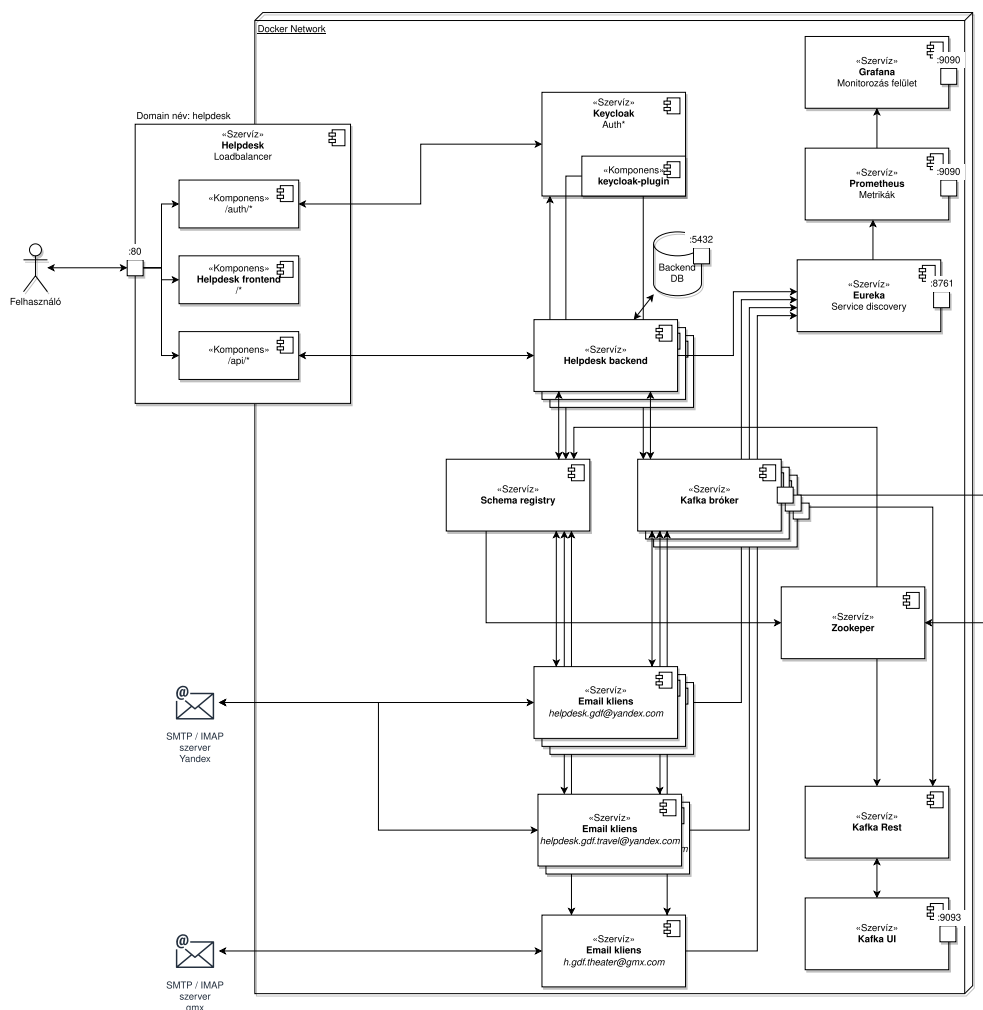
A *docker-compose* (5.1) által elindított *containereket* az 5.1. ábrán foglaltam össze. Az ábrán feltüntettem, hogy az adott *containert* a *localhost* melyik portján lehet elérni.

5.4. E-mail fogadásának és küldésének folyamata

A 3. fejezetben a 3.3. ábrán bemutattam egy e-mail fogadásának elméleti útját. Most az 5.2. ábrán szeretném bemutatni hogyan követhető végig a rendszerben 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 (5.2a. ábra).
3. A backend megkapja a kafka üzenetet (5.2c. ábra)
4. A backend elmenti az új üzenet az adatbázisba (5.2e. ábra)



5.1. ábra. Deployment diagram

Forrás: saját ábra

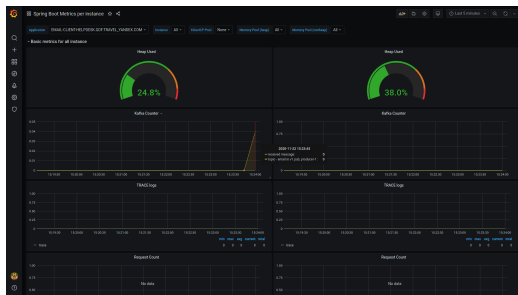
5. A felhasználói felületen (5.2g. ábra) elérhető az új üzenet.

E-mail küldése:

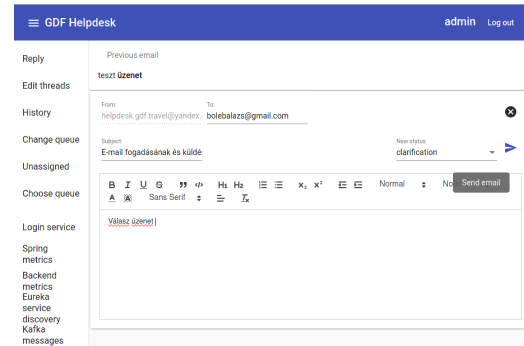
1. A felhasználó elküldi a válaszát a felhasználói felületen (5.2b. ábra).
2. A backend megkapja az üzenetet és eltárolja az adatbázisba (5.2d. ábra).
3. A *h.gdf.theater_gmx.com.v1.pub* topicban megjelenik (5.2f. ábra) a backend által publikált kafka üzenet
4. A topicra feliratkozott e-mail kliens fogadja és továbbítja az üzenetet (5.2h. ábra).

5.5. Adatbázistáblák

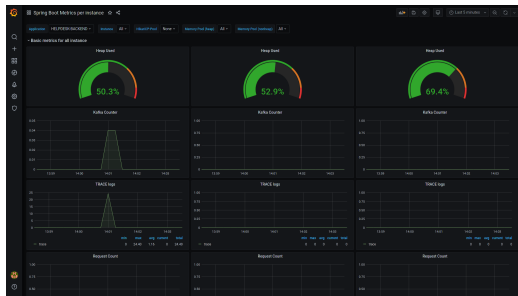
A helpdesk backend adatbázistábláit az 5.3. ábra tartalmazza.



(a) Az egyes instance kafka üzenet küld



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



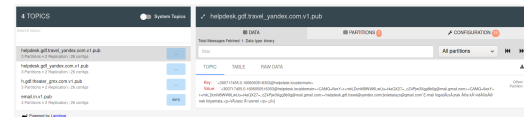
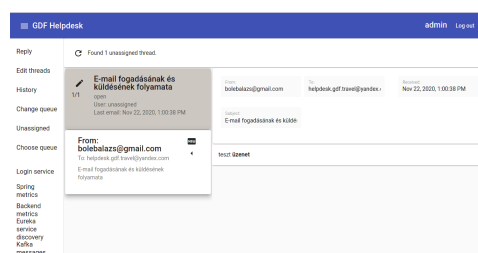
(c) Az egyes instance kafka üzenet fogad



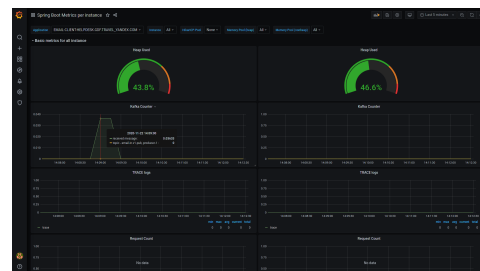
(e) Az új e-mail az adatbázisban



(d) A válasz e-mail az adatbázisban

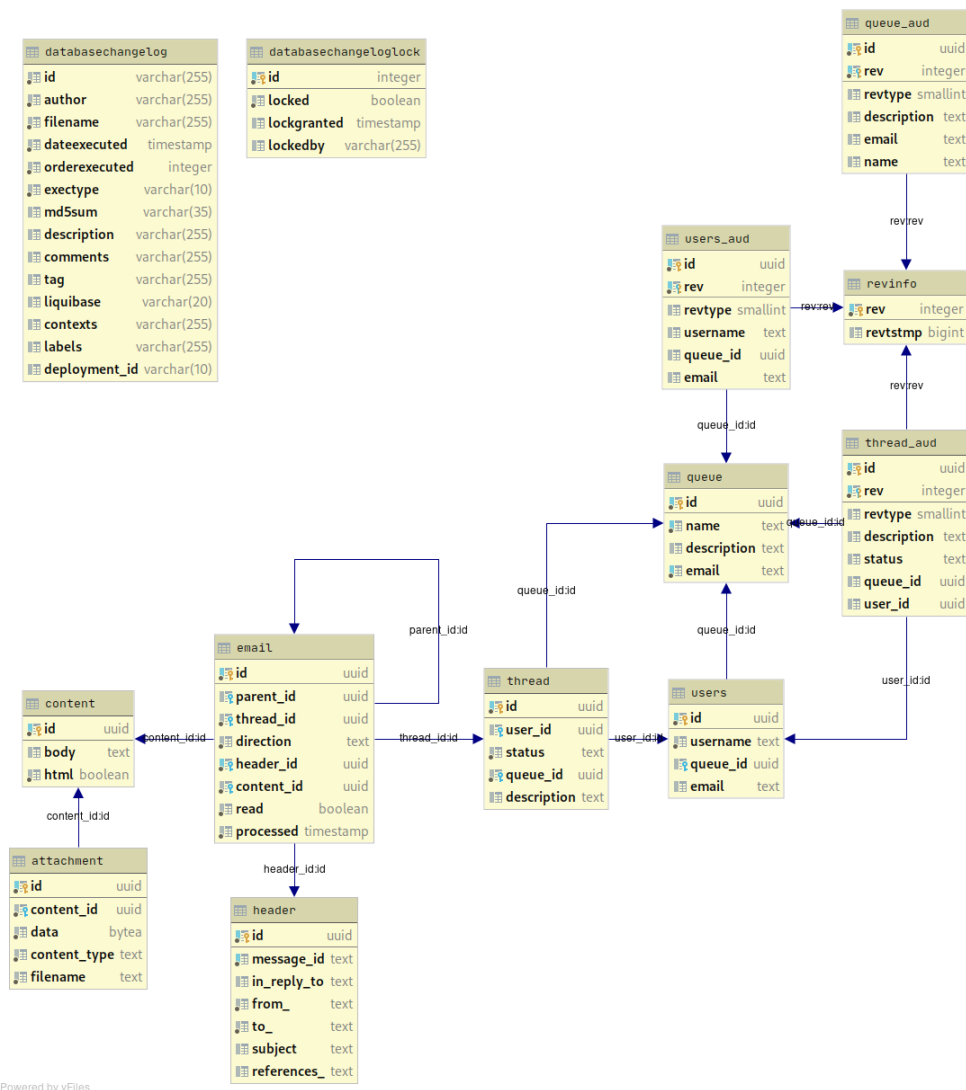
(f) A *h.gdf.theater_gmx.com.v1.pub* topic új üzenete

(g) A felületen elérhető az új e-mail



(h) Az egyes instance kafka üzenet fogad

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



5.3. ábra. A backend összes adatbázistáblája

Forrás: saját ábra

5.5.1. Liquibase

A *databasechangelog* és *databasechangeloglock* táblákat a Liquibase (4.3.2. pont) az adatbázis séma verziójának karbantartására használja:

databasechangelog tábla tartalmazza a *resources/db.changelog* könyvtárban található, *db.changelog-master.yaml* állományban tárolt utasítások futási eredményeit.

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

A helpdesk backend szervíz minden induláskor elindítja a Liquibase-t. A Li-

quibase 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 feladatnak ellátásához szükséges szinten férjen hozzá az alkalmazáshoz, és hogy futása során ne módosíthassa a táblákat.

5.5.2. 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 kódját, 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.

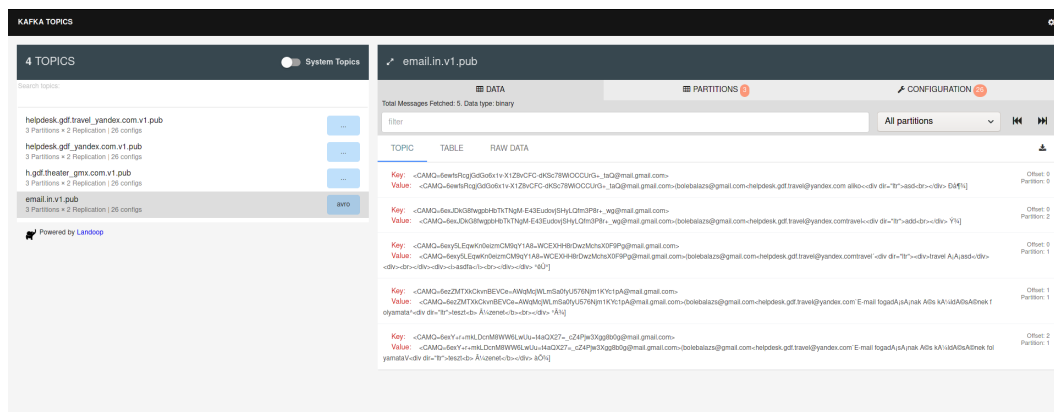
5.6. Apache Kafka

A kafka *topicok* és üzenetek elérhetőek és követhetőek a *Kafka messages* menü pontja alatti Kafka Topics UI (5.4. ábra) eszközzel.

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.



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

Forrás: saját ábra

helpdesk.gdf_yandex.com.v1.pub A helpdesk.gdf@yandex.com címre küldött e-maileket tartalmazza.

helpdesk.gdf.travel_yandex.com.v1.pub A helpdesk.gdf.travel@yandex.com címre küldött e-maileket tartalmazza.

h.gdf.theater_gmx.com.v1.pub A h.gdf.theater@gmx.com címre küldött e-maileket tartalmazza.

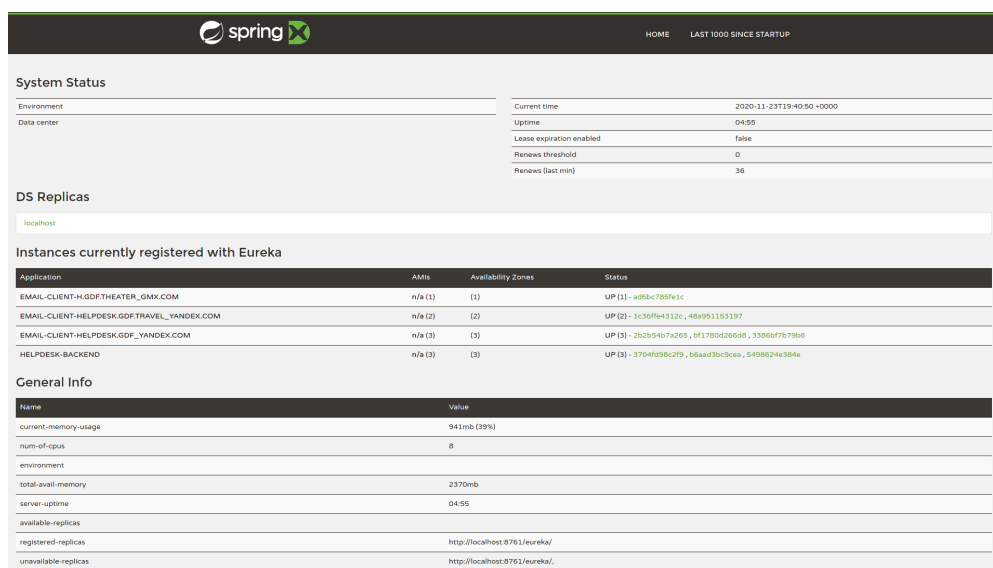
_schemas A Schemaregistry ebben a *topic*ban tárolja az alkalmazásban használt Avro schemákat.

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

5.7. Eureka

A 4.1.3. pontban megemlített szerviz felderítésre az Eureka szervert használom. A helpdesk backend és az e-mail kliens induláskor beregisztálják magukat az Eureka szervizbe.

Az Eureka serveren keresztül megtekinthető, és más szervizek számára elérhető a példányok aktuális állapota és neve. Az oldal elérhető a *Eureka service discovery* menüpont alatt (5.5. ábra).



The screenshot displays the Spring Cloud Eureka Admin interface. At the top, the Spring logo and navigation links 'HOME' and 'LAST 1000 SINCE STARTUP' are visible. The main content is divided into three sections: 'System Status', 'DS Replicas', and 'Instances currently registered with Eureka'.

System Status

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

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EMAIL-CLIENT-H.GDFTHEATER_GMX.COM	n/a (1)	(1)	UP (1) - a95bc785fe1c
EMAIL-CLIENT-HELPPDESK.GDFTTRAVEL_YANDEX.COM	n/a (2)	(2)	UP (2) - 1c36ffe4312c, 45a951153197
EMAIL-CLIENT-HELPPDESK.GDF_YANDEX.COM	n/a (3)	(3)	UP (3) - 2b2b54b7a265, bf170cd266d9, 33966f7b79b6
HELPPDESK-BACKEND	n/a (3)	(3)	UP (3) - 3704f998c2f9, b6aad3bc9caa, 5498624e384e

General Info

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

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

Forrás: saját ábra

6. fejezet

Terheléses tesztelés

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.

6.1. 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 összevetik a rendszer metrikáival, ezzel elősegítve az alkalmazás összteljesítményét kritikusan érintő szűk keresztmetszetek megtalálását.

6.2. Apache JMeter

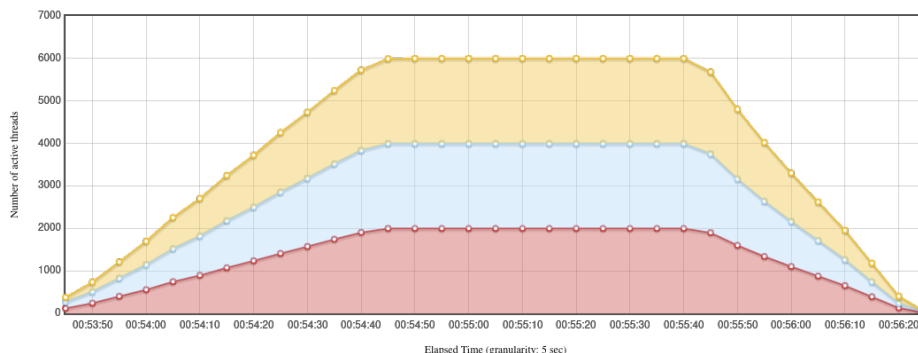
Az Apache Jmeter egy open source terheléses teszt végrehajtására alkalmas eszköz. Számos protokollt képes kezelni, a helpdesk alkalmazás vizsgálatánál HTTP üzeneteket küldésével modelleztem a párhuzamos felhasználókat.

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 (6.1 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 azonosíthatók a teljesítmény szempontjából kritikus rendszerek.

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



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

Forrás: saját ábra

6.3. 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 6.1 táblázat tartalmazza.

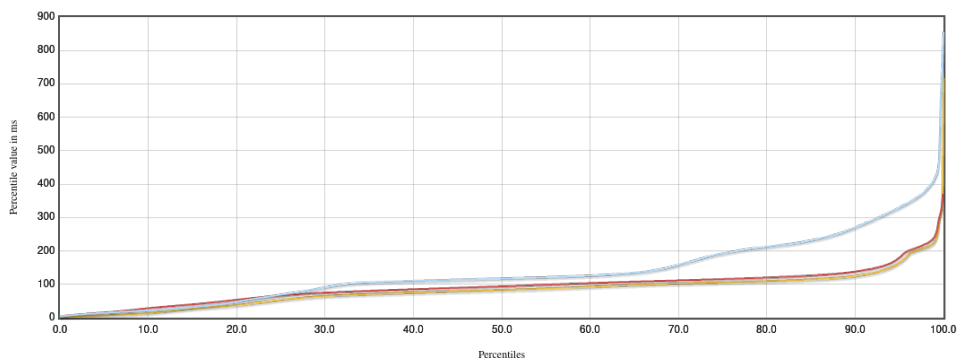
Átlag	Válaszidő [<i>ms</i>]				Áteresztőképesség [Tranzakció/ <i>s</i>]	Válasz [%]	
	Max.	Medián	P90	P95		< 3 <i>s</i>	< 6 <i>s</i>
99,32	856	88,00	223,00	340,00	900,01	100	100

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

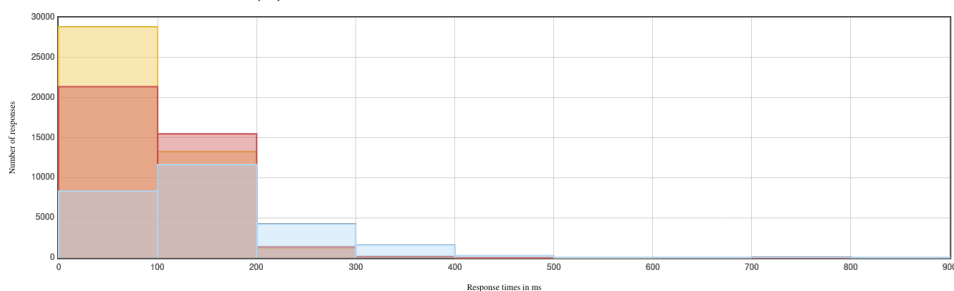
A 6.2 ábrán és a 6.1 táblázat adataiból látszódik, hogy:

- 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

6.2. ábra. Egy példányban futó helpdesk backend terheléses teszt eredménye 100 felhasználóval

Forrás: saját ábra

6.4. Csúcsteljesítmény vizsgálata

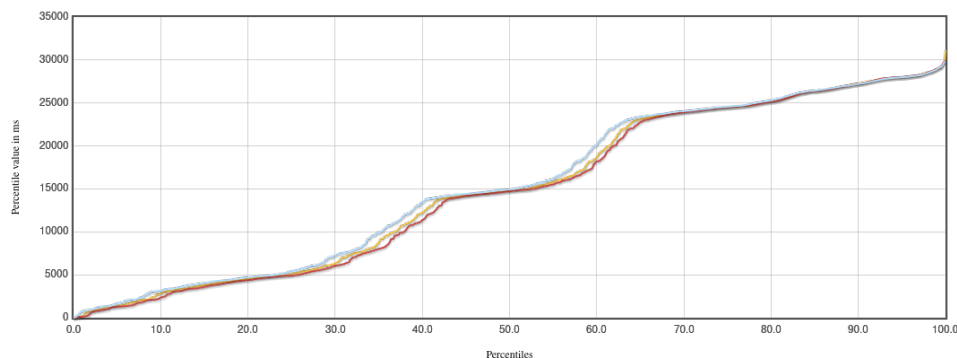
A helpdesk backend egy önálló példányát vizsgáltam 6 000 párhuzamos felhasználó modellezésével. A teszt során használt párhuzamos szálakat a 6.1 ábrán ábrázoltam, a felfutási idő 60 másodperc volt. Az ábrán látható ahogy a JMeter a felfutási idő után egyszerre 6 000 szálon futtatja a tesztek.

A teszt futtatása során mért eredményeket a 6.2 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.

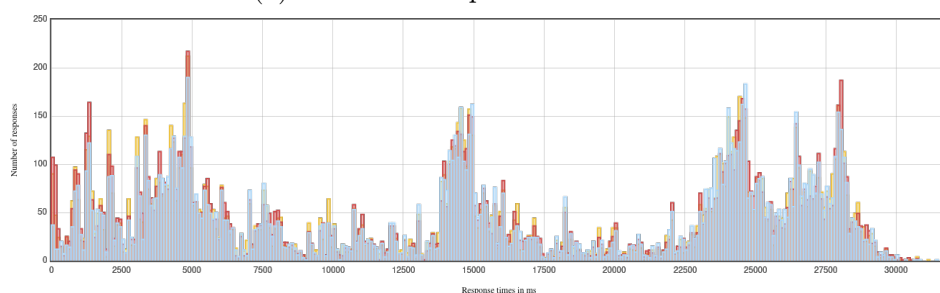
Átlag	Válaszidő [ms]				Áteresztőképesség [Tranzakció/s]	Válasz [%]	
	Max.	Medián	P90	P95		< 3s	< 6s
15 133,81	31 732	24 553,00	28 011	28 407	281,72	10,19	27,78

6.2. táblázat. Egy példányban futó helpdesk backend terheléses teszt eredménye 6 000 felhasználóval

Az eltérés okainak azonosítására célszerű a szűk keresztmetszeteket meghatározni és feloldani (6.5 pont).



(a) A válaszidők percentilis eloszlása



(b) A válaszidők eloszlása

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

Forrás: saját ábra

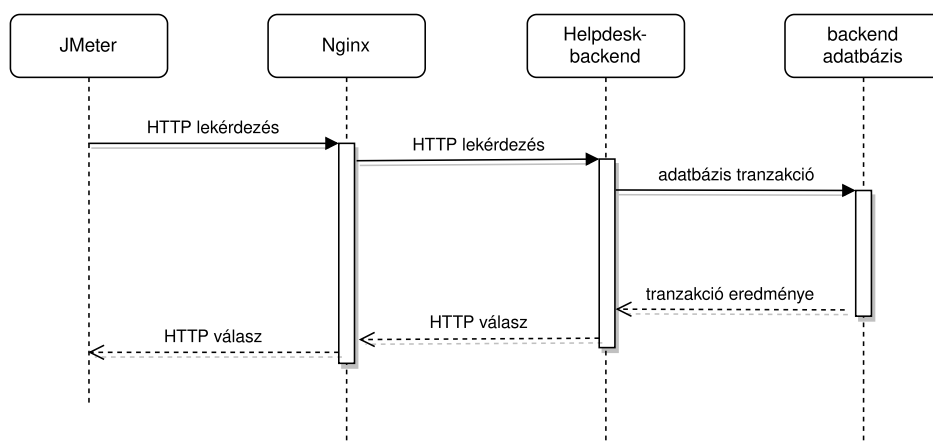
6.5. Szűk keresztmetszet meghatározása

A JMeter által indított kérés az alábbi rendszereken megy keresztül (6.4 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.

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 (6.5a ábra).
- A processzorigény szignifikánsan megemelkedett (6.5b ábra). A rendszer CPU felhasználása 80%-ra a backendé 40%-ra nőtt.



6.4. ábra. A terhelésselés tesztben résztvevő alkalmazások

Forrás: saját ábra

- Az adatbázis kapcsolatok fenntartására használt HikariCP-ben (4.3.2 pont) lényegesen feltorlódtak a lekérdezések (6.5c ábra). A függőben lévő kapcsolatok markánsan megugrottak.

6.5.1. Nginx

Hogy az Nginx párhuzamosan kiszolgáljon legalább 6 000 felhasználót¹, a beállításokban 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.

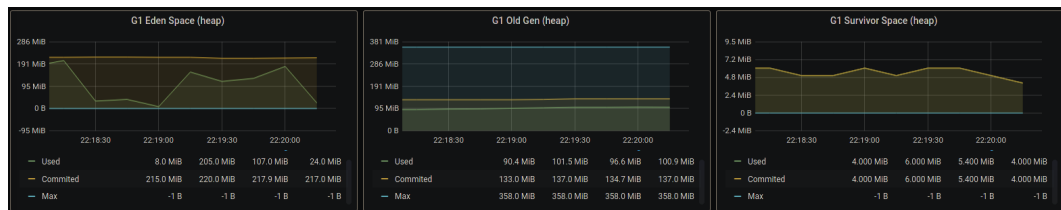
Ezért a megismételt a JMeter tesztben a backendet közvetlenül az IP-címén keresztül értem el. Az új mérés –6.3 táblázat– adataiból látszódik,

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

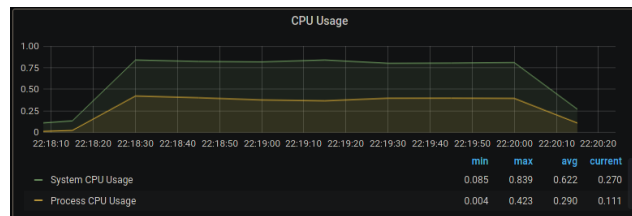
6.3. táblázat. Nginx nélkül futó helpdesk backend terhelésselés teszt eredménye 6 000 felhasználóval

- hogy az átlagos és a medián válaszidő, valamint áteresztőképesség nem változott,
- a három másodperc alatt megérkező válaszok aránya –az első esethez (6.4) képest– meg két és félszereződött, 26,41%-ra nőtt.

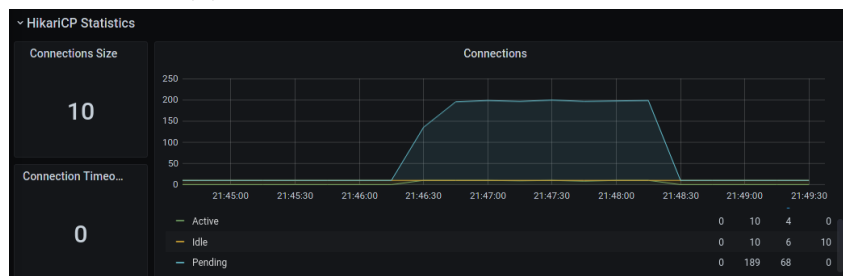
¹Az alapbeállítás 1 024 kapcsolat



(a) A backend memóriafelhasználása



(b) A backend processzor használata

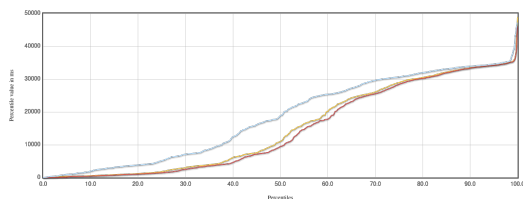


(c) HikariCP adatbázis-kapcsolatai

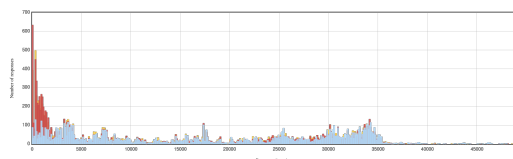
6.5. ábra. A csúcsteljesítmény vizsgálata során vizsgált legfontosabb metrikák

Forrás: saját ábra

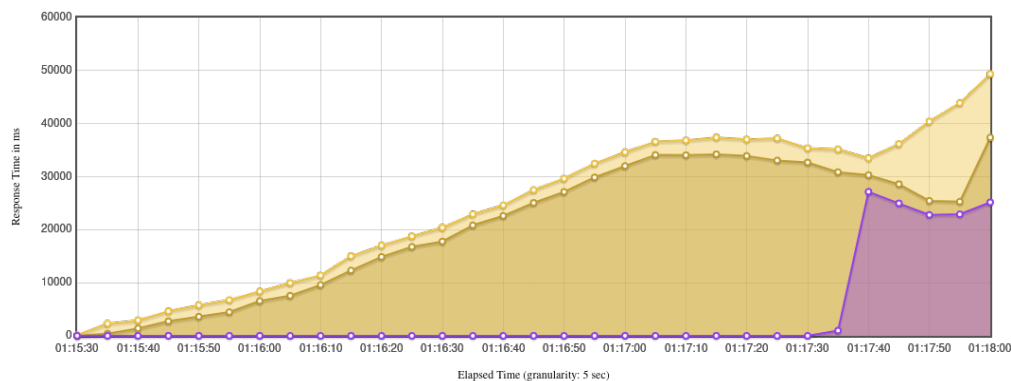
Ugyanez látszódik a 6.6c ábrán is, a teszt első háromnegyedében a backend néhány kérésre azonnal válaszol, míg a többségre –medián– csak sokkal később.



(a) A válaszidők percentilis eloszlása



(b) A válaszidők eloszlása



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

6.6. 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 (6.6a 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.

6.5.2. HikariCP

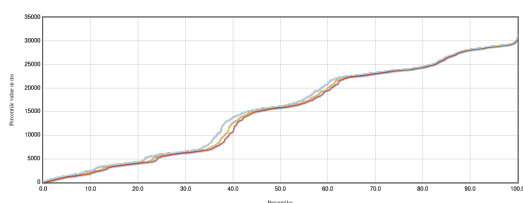
A 6.5c ábrán látszódik hogy a HikariCP-ben beállított 10 kapcsolat hamar elfogy, már mérés elején 189-re nő, és tartósan ott is marad a pool-ra várakozók száma.

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

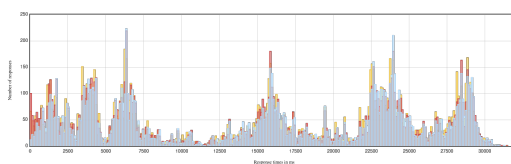
Ezért a megismételt a JMeter tesztben a backendben megháromszoroztam a HikariCP pooljának a méretét. A mérési eredmények –6.4 táblázat– nem mutatnak eltérést az első esethez (6.4) képest.

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

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



(a) A válaszidők percentilis eloszlása

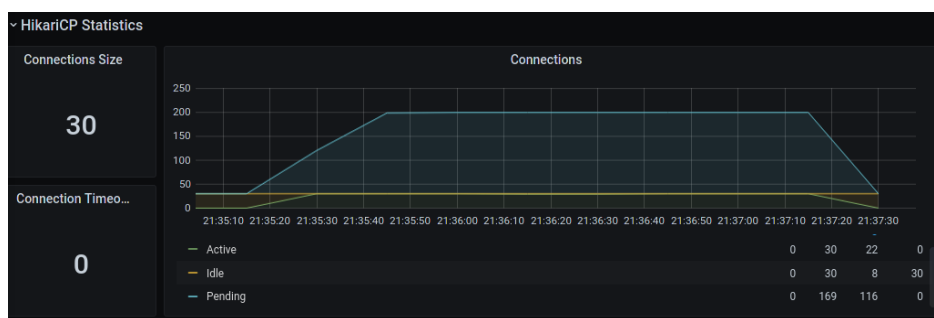


(b) A válaszidők eloszlása

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

Forrás: saját ábra

A 6.8 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.



6.8. ábra. HikariCP adatbázis-kapcsolatai

Forrás: saját ábra

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

6.5.3. Megnövekedett processzorigény

A 6.5b ábrán látszódik hogy még van szabad processzoridő. Ha a helpdesk backend képes lenne magának még processzoridőtallokálni, 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 a JMeter tesztben a backendből három példányt indítottam el. Az eredmények így összemérhetőek maradtak a 6.5.2. ponttal, hiszen összességében ugyanannyi kapcsolat van a backend és az adatbázis között.

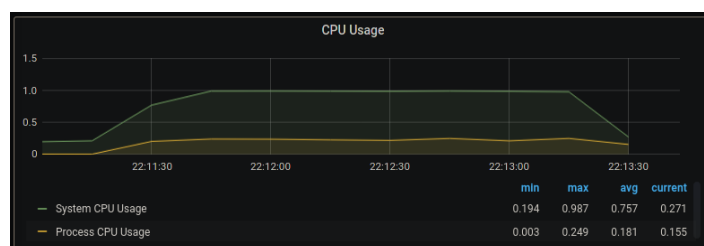
Átlag	Max.	Válaszidő [ms]				Áteresztőképesség [Tranzakció/s]	Válasz [%]	
		Medián	P90	P95	P95		< 3s	< 6s
5748,49	16 577	7452,00	11 482,00	12 011,95		755,62	22,23	49,01

6.5. táblázat. Három példányban futó helpdesk backend terheléssel teszt eredménye 6 000 felhasználóval

A 6.5 táblázat adatai alapján kijelenthető,

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

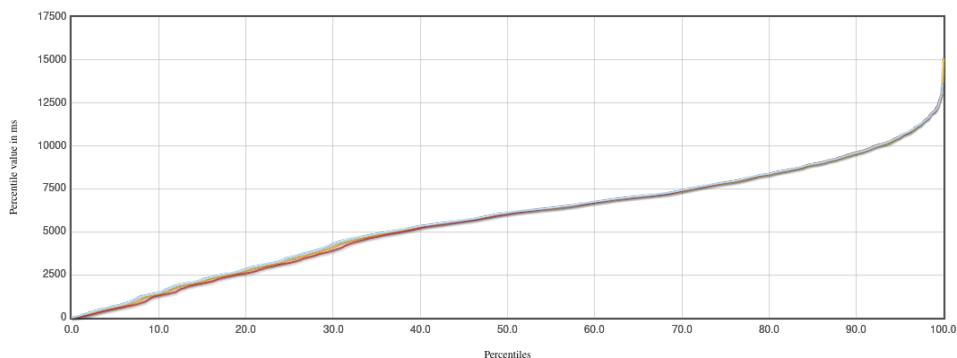
A 6.10 ábrán megjelenített mérési eredményeken látszódik hogy a válaszidők szórása nagy mértékben csökkent, a várható bizonytalanság így már sokkal inkább tolerálható.



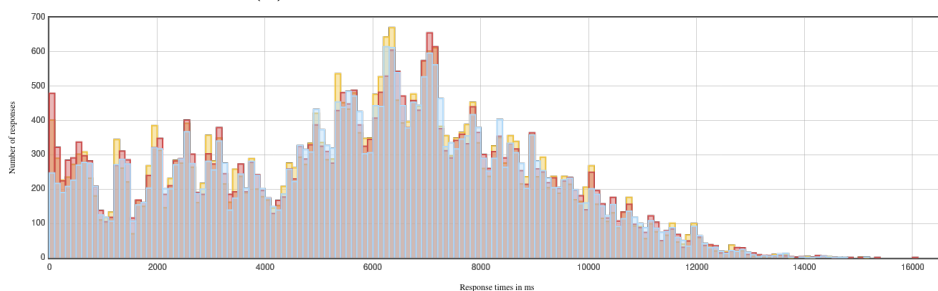
6.9. ábra. A backend egy példányának a processzor használata

Forrás: saját ábra

Ha megvizsgáljuk a teszt során mérhető processzor használatot (6.9 ábra), akkor látható, hogy nem maradt már allokalatlan processzoridő. Valószínűleg a két újabb backend példány használta fel a maradék erőforrást.



(a) A válaszidők percentilis eloszlása



(b) A válaszidők eloszlása

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

Forrás: saját ábra

6.5.4. 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 6.9 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.

6.6. Összevetés a követelményekkel

Horizontális skálázhatósággal jelentősen megnövelhető a rendszer áteresztőképessége, és csökkenthető a válaszideje.

6.6.1. Átlagos teljesítmény vizsgálata

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

6.6.2. Csúcsteljesítmény vizsgálata

A rendszerrel szembeni elvárások teljesíthetőségére nem lehet egyértelmű választ adni. A tesztelt számítógépen –erőforráshiány miatt (6.5.4)– 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 feladatának ellátása.

7. fejezet

Továbbfejlesztési lehetőségek

7.1. A deploymentről

A helpdesk alkalmazás szervizei 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.

Ám 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 *clusterre* migrálása. A docker natívan támogatja a Microsoft Azure és az Amazon [16] szolgáltatókat. Így tehát a kód és a beállítások módosítása nélkül lehetséges az alkalmazás *clusteresítése* docker swarmmal.

7.2. 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.

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

Mivel egy szerviz egy feladattal foglalkozik, ha például le kell cserélni a frontendet, akkor az új felhasználó felületen csak a megjelenítéssel kell foglalkozni, az üzleti funkciók megvalósítása a backend feladta, így azok továbbra is változatlanok maradnak.

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

Összefoglalás

Sikerült egy mikroszerviz alapú elosztott alkalmazás létrehozása. Bemutattam a megvalósítás során felmerült problémákat és azok megoldását, felhasznált technológiákat és módszertanokat.

Számtalan gyakran visszatérő problémára létezik már szabadon használható open source megoldás. Az új alkalmazások fejlesztését nagyban megkönnyíti, ha nem kell minden problémát újból megoldani. Éppen ezért kiemelten fontos, hogy a létrejött helpdesk alkalmazás modularitása miatt könnyen integrálható más alkalmazásokkal.

Egyik felhasznált program sem licenc köteles. És mivel a létrejött funkciók a modulok között erősen szeparálva helyezkednek el, könnyű egy-egy modul lecserélése. Így létrejött szoftver fenntartási költsége alacsony. Amennyiben megszűnik egy-egy technológia támogatása, vagy bármilyen más okból le kell cserélni egy modult, akkor sem szükséges a teljes kódbázis cseréje, így értékes emberórákat spórolva meg a fejlesztőknek.

Bízom benne hogy sikerült egy modern, időtálló, és az ügyfél igényeit maradéktalanul kielégítő szoftvert létrehoznom.

Irodalomjegyzék

- [1] Mike Loukides és Steve Swoyer. Microservices adoption in 2020, Júl. 15 2020. URL: <https://www.oreilly.com/radar/microservices-adoption-in-2020/>.
- [2] Andzhela Angelova. 10 reasons why microservices are the future, Jún. 20 2020. URL: <https://wiredelta.com/10-reasons-why-microservices-are-the-future/>.
- [3] Matt McLarty Mike Amundsen Irakli Nadareishvili, Ronnie Mitra. *Microservice Architecture: Aligning Principles, Practices, and Culture*, chapter 1, The Microservices Way. O'Reilly, 1 edition, 2016.
- [4] Dr. Alistair Cockburn. Hexagonal architecture, Ápr. 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 11, Systems. Prentice Hall, 1 edition, 2008.
- [7] Matt McLarty Mike Amundsen Irakli Nadareishvili, Ronnie Mitra. *Microservice Architecture: Aligning Principles, Practices, and Culture*, chapter 5, CQRS. O'Reilly, 1 edition, 2016.
- [8] Matt McLarty Mike Amundsen Irakli Nadareishvili, Ronnie Mitra. *Microservice Architecture: Aligning Principles, Practices, and Culture*, chapter 5, Distributed Transactions and Sagas. O'Reilly, 1 edition, 2016.
- [9] Todd Palino Neha Narkhede, Gwen Shapira. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale, and Culture*, chapter 1, Meet Kafka. O'Reilly, 1 edition, 2016.

- [10] Todd Palino Neha Narkhede, Gwen Shapira. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale, and Culture*, chapter 5, Kafka Internals. O'Reilly, 1 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] Identification fields. Letöltve: 2020. Nov. 16. URL: <https://tools.ietf.org/html/rfc5322#section-3.6.4>.
- [14] Keycloak. Letöltve: 2020. Nov. 18. URL: <https://www.keycloak.org/>.
- [15] Json web token (jwt). Letöltve: 2020. Nov. 18. URL: <https://tools.ietf.org/html/rfc7519>.
- [16] Deploying docker containers on ecs. Letöltve: 2020. Nov. 22. URL: <https://docs.docker.com/engine/context/ecs-integration/>.

A. függelék

OpenApi dokumentáció

helpdesk-backend

Overview

Version information

Version : 0.0.1

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.	< EmailThread > 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	emailThreadId <i>required</i>	string (uuid)

Responses

HTTP Code	Description	Schema
200	Returns history of the email thread.	< EmailThreadAudit > 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	status <i>optional</i>	EmailThread status	string	"OPEN"

Responses

HTTP Code	Description	Schema
200	Return found emailThreads.	< EmailThread > 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	status <i>optional</i>	EmailThread status	string	"CHANGE_QUEUE"

Responses

HTTP Code	Description	Schema
200	Return found emailThreads.	< EmailThread > 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
```

Parameters

Type	Name	Description	Schema	Default
Query	page <i>optional</i>	Pagination page	integer (int32)	<code>0</code>
Query	size <i>optional</i>	Pagination size	integer (int32)	<code>1</code>

Responses

HTTP Code	Description	Schema
200	Return unassigned emailThreads.	< EmailThread > 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	emailThreadId <i>required</i>	string (uuid)

Responses

HTTP Code	Description	Schema
200	Returns email.	EmailThread
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	emailThreadId <i>required</i>	Id of the emailThread	string (uuid)
Body	body <i>required</i>	New properties	< string, string > map

Responses

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

Produces

- `application/json`

Tags

- EmailThread

Send an Email.

POST /api/email/send

Parameters

Type	Name	Description	Schema
Body	body <i>required</i>	Email to send	Email

Responses

HTTP Code	Description	Schema
200	Returns email.	Email
403	User not authorized.	No Content

Produces

- `application/json`

Tags

- Email

Get Email by UUID.

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

Parameters

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

Responses

HTTP Code	Description	Schema
200	Returns email.	Email
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	emailId <i>required</i>		string (uuid)
Body	body <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.	< Queue > 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.	User
403	User not authorized.	No Content

Produces

- `application/json`

Tags

- User

Change ths active user's queue'.

```
PATCH /api/user/queue
```

Parameters

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

Responses

HTTP Code	Description	Schema
200	First (size) count values about BIC field.	< User > 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
content <i>required</i>	The attachment is part of this email content.	Content

Name	Description	Schema
contentType <i>required</i>	The content type of the data.	string
data <i>required</i>	The binary bytearray of the data.	< string (byte) > array
dataAsStream <i>optional</i>		ByteArrayOutputStream
filename <i>required</i>	The name of the data.	string

ByteArrayOutputStream

Type : object

Content

Full DTO for content

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

Email

Full DTO for email

Name	Description	Schema
content <i>required</i>	The content of the email.	Content
direction <i>optional</i>	The direction of the email.	enum (IN, OUT)

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

EmailThread

Full DTO for thread

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

EmailThreadAudit

Full DTO for thread audit

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

Header

Full DTO for header

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

Name	Description	Schema
subject <i>optional</i>	The subject of the mail.	string
to <i>required</i>	The email sent to this address.	string

Queue

Full DTO for queue

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

User

Full DTO for users

Name	Description	Schema
id <i>optional</i>	Unique internal identifier Example : "06484c9f-6f59-4b9f-ad5e-aaaa0ec332cc"	string (uuid)
keycloakID <i>required</i>	Unique keycloak id.	string (uuid)
queue <i>required</i>	The user can operate on this queue.	Queue
username <i>required</i>	Username.	string