

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 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

Tartalomjegyzék	vi
Ábrák jegyzéke	ix
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	4
1.2.1. Skálázhatóság	4
1.2.2. Granuláris felosztottság	4
1.2.3. Mérhető indikátorok	5
1.2.4. L10N	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	14
4. Technikai tervezés	16
4.1. Webszerver	16
4.2. Adatbázis	17
4.2.1. Teljesítményvizsgálat	17
4.3. JPA implementáció	18
4.4. Adatbázis migrációs eszköz	18
4.4.1. Liquibase	19
4.4.2. Flyway	19
4.4.3. Felhasznált eszköz	19
4.5. Monitorozás	19
4.5.1. Prometheus	19
4.5.2. Grafana	20
4.5.3. Graphite	20
4.5.4. Megvalósítás	21
5. Implementáció	22
5.1. Mikroszerviz infrastruktúra	22
5.2. Nginx	22
5.2.1. Docker konténerizáció	22
5.2.2. Metrikák	23
5.3. E-mail kliens	23
5.3.1. E-mail szabvány	23
5.4. Helpdesk backend	24
5.4.1. Spring Boot	24
5.4.2. Adatbázis	25
5.4.3. Pesszimista konkurenciakezelés	25
5.4.4. Optimista konkurenciakezelés	26
5.4.5. Egyéb eszközök	26
5.5. Helpdesk frontend	26
5.5.1. Kommunikáció a backenddel	26
5.5.2. Komponensek	27
5.5.3. Futtatási környezet	27
5.6. Keycloak	27
5.6.1. Jogosultságkezelés	27
5.6.2. JSON Web Token	29
5.7. Kafka	29

5.8. Helpdesk backend és a Keycloak elkülönítése	29
6. Alkalmazás bemutatása	30
6.1. Alkalmazás elindítása	30
6.2. Deployment	30
6.3. Több példány	31
6.4. E-mail fogadásának és küldésének folyamata	32
6.5. Adatbázistáblák	34
6.5.1. Liquibase	34
6.5.2. Hibernate Envers	35
6.6. Apache Kafka	35
6.7. Eureka	36
7. Terheléses tesztelés	38
7.1. Terheléses teszt	38
7.2. Apache JMeter	38
7.3. Átlagos teljesítmény vizsgálata	39
7.4. Csúcsteljesítmény vizsgálata	40
7.5. Szűk keresztmetszet meghatározása	42
7.5.1. Nginx	43
7.5.2. HikariCP	45
7.5.3. Megnövekedett processzorigény	46
7.5.4. Szűk keresztmetszet meghatározása	47
7.6. Összevetés a követelményekkel	48
7.6.1. Átlagos teljesítmény vizsgálata	48
7.6.2. Csúcsteljesítmény vizsgálata	48
8. Továbbfejlesztési lehetőségek	49
8.1. A deploymentről	49
8.2. A kódról	49
Irodalomjegyzék	53
A. OpenApi dokumentáció	55

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
5.1. E-mail kliens szekvencia diagramja	24
5.2. Helpdesk backend szekvencia diagramja	25
5.3. Helpdesk frontend szekvencia diagramja	28
6.1. Deployment diagram	31
6.2. E-mail fogadásának és küldésének folyamata	33
6.3. A backend összes adatbázistáblája	34
6.4. A Kafka Topics UI felülete	36
6.5. Az Eureka service discovery felülete	37
7.1. A JMeter tesztelés során használt – számai	39
7.2. Helpdesk backend terheléses teszt 100 felhasználóval	40
7.3. Helpdesk backend terheléses teszt egy példánnyal	41
7.4. A terheléses tesztben résztvevő alkalmazások	42
7.5. A csúcsteljesítmény vizsgálata során vizsgált legfontosabb metrikák a Grafana monitorozó eszköz felületén	43
7.6. Helpdesk backend terheléses teszt Nginx nélkül	44
7.7. Helpdesk backend terheléses teszt módosított HikariCp beállításokkal	45
7.8. HikariCP adatbázis-kapcsolatai a Grafana felületén	46
7.9. A backend egy példányának a processzor használata a Grafana felületén	47
7.10. Helpdesk backend terheléses teszt három példánnyal	47

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

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

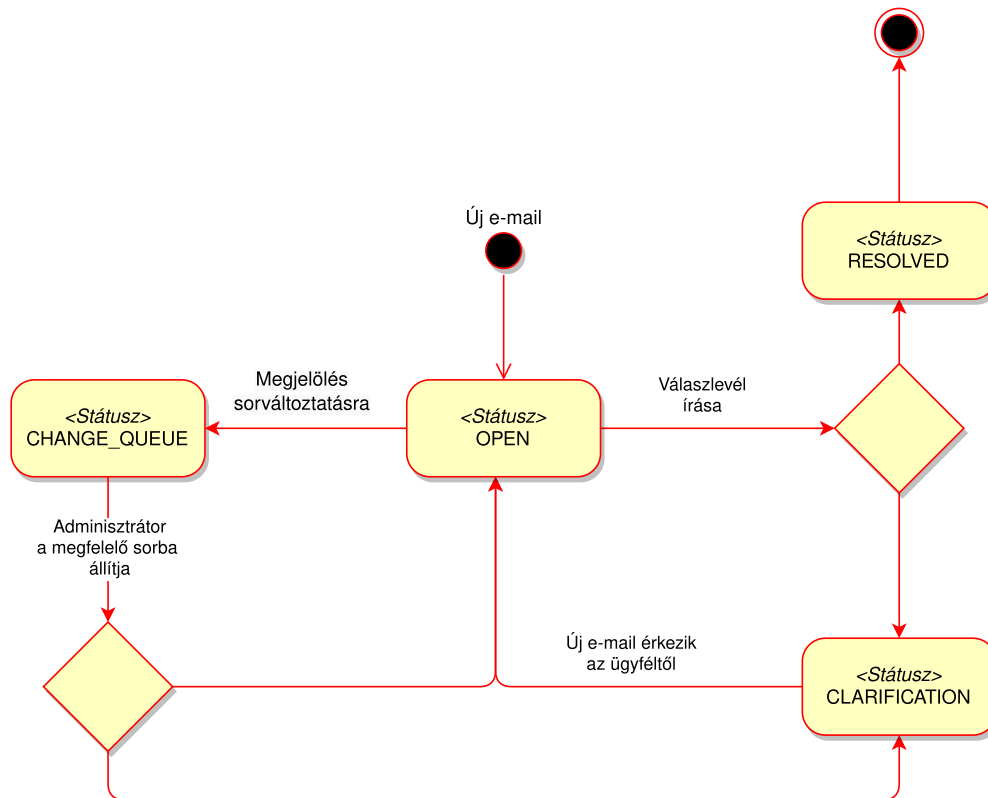
A rendszer által kezelt üzenetek szálakba rendezve érhetőek el. Egy szál az ügyfél és a felhasználó közötti üzenetváltásokból épül fel.

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.



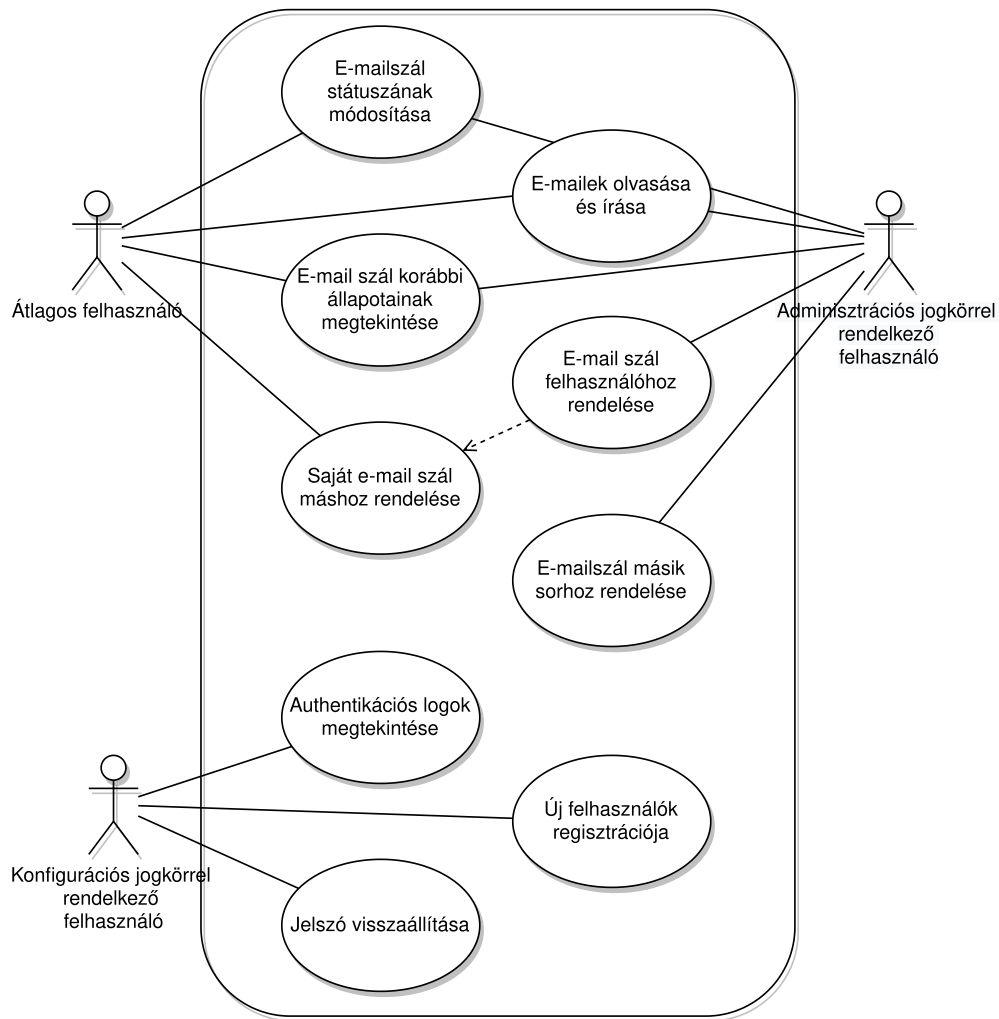
1.1. ábra. Az e-mailszálak státuszváltozásai

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 események megtekintésére, jelszó visszaállítására és más felhasználók megszemélyesítésére (*impersonate*).

A felhasználói felületen elérhető funkciókat az 1.2 ábra foglalja össze.



1.2. ábra. Elérhető funkciók jogosultság szerint csoportosítva

Forrás: saját ábra

1.2. Nem funkcionális igények

1.2.1. Horizontális skálázhatóság

A kiszolgálandó 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 hétköznap munkaórákon kívül a legaktívab-

bak.

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 számára elfogadható legnagyobb válaszidő 3 másodperc/lekérés.

1.2.4. L10N¹

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

¹A 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 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.

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

szerviz felderítés a rendszer által nyújtott szolgáltatások, szervizek automatikus fedezhetősége³.

loadbalancer az a folyamat, ami a bejövő feladatokat erőforrásokhoz rendeli. Leg-egyszerűbb 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 szembeötlő 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* című cikkében [5] összeszedte, a port-adapter és a hasonló architektúrával készülő alkalmazások mind:

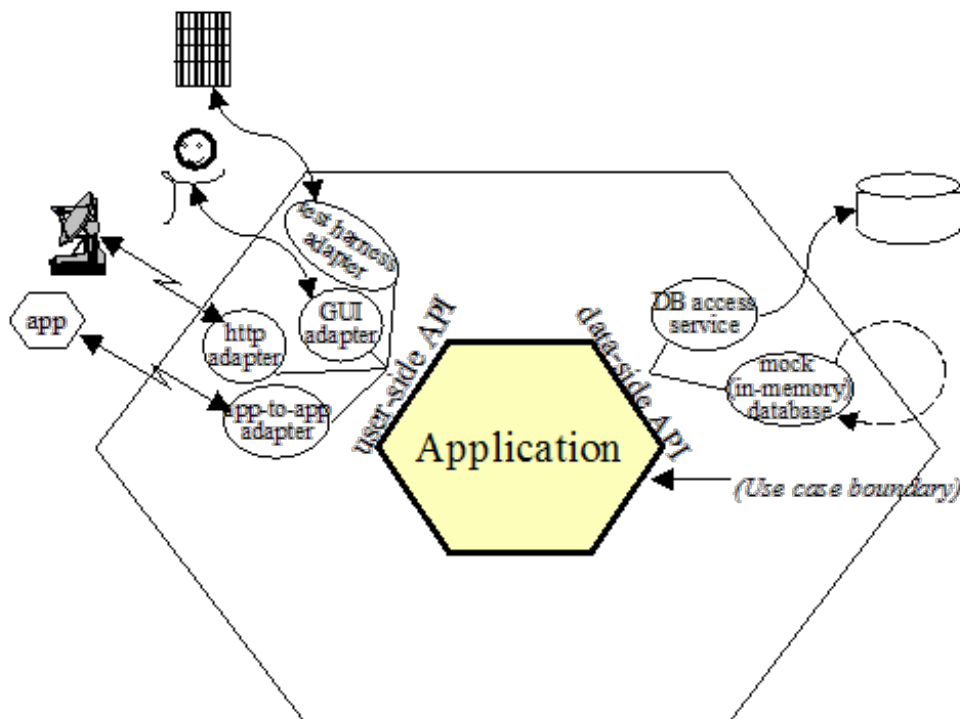
- Könnyen, és önmagukban is tesztelhetők, mivel az üzleti szabályoknak nincs külső függőségük.
- Függetlenek a külső tényezőktől. Így az alkalmazás által használt felület vagy adatbázis könnyen cserélhető.
- Keretrendszerrel függetlenül is megvalósíthatóak. A megvalósítás nem függ semmilyen könyvtártól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

¹több processzor vagy memória bevonása

²újabb példányok futtatása

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.

pesszimista 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 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.

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 szerver esetleges kiesése esetén a többi szerver át tudja venni a kiesett szerver szerepét.

Az üzenetbrókerek összehangolását a Zookeeper szerviz végzi. Mivel minden kafka bróker beregisztrálja magát a szervizbe, 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

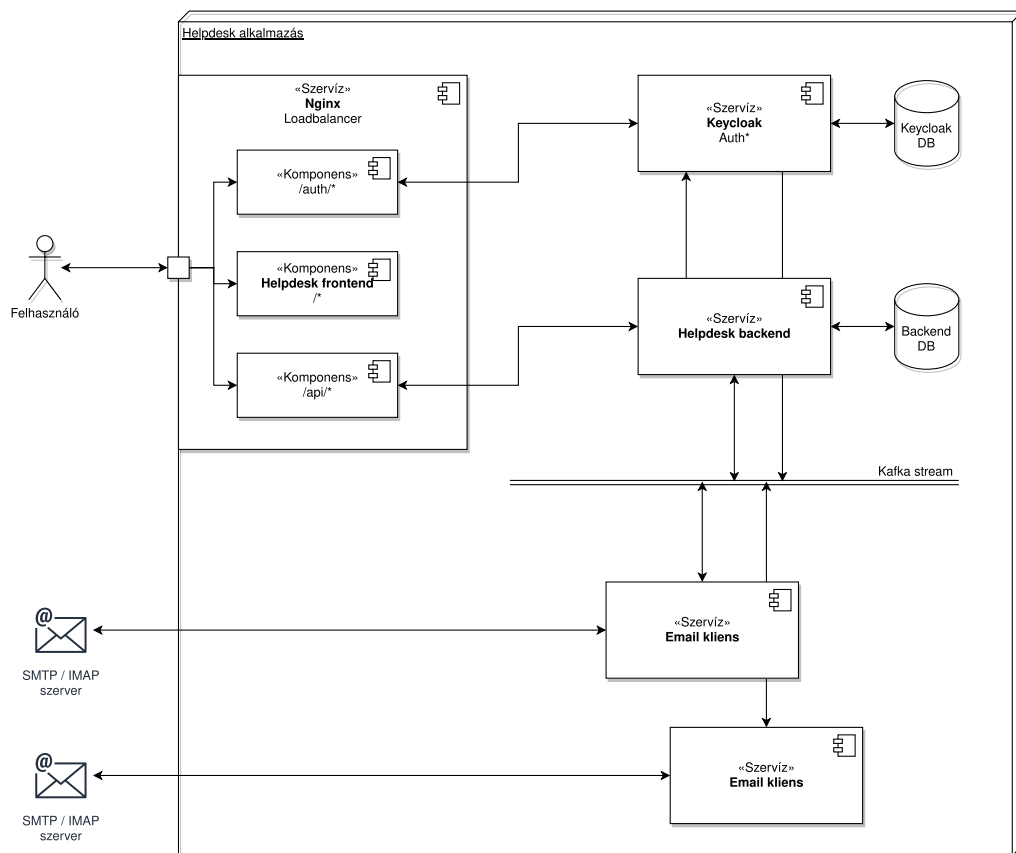
⁸Például: metrikák, biztonság, adattárolás

3. fejezet

Az alkalmazás felépítése

Ebben a fejezetben 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

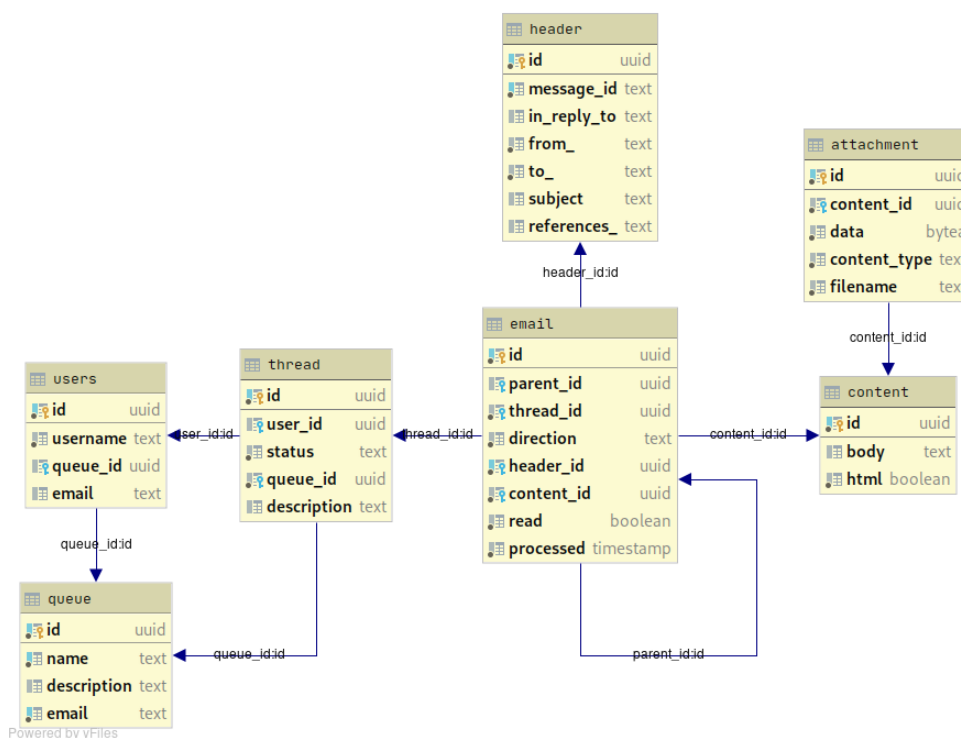
Forrás: saját ábra

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

- A felhasználó az nginx-en (5.2 pont) keresztül éri el a heldesk alkalmazást.
- Az nginx dönti el, hogy melyik URL-t melyik szerviz szolgálja ki.
- Az email kliens és a heldesk backend kafka streamen keresztül éri el egymást.
- Az email kliensek kezelik az e-mail szerverekkel való adatcserét.

3.2. Adatbázis UML diagram

A heldesk backend adatbázis legfontosabb tábláit a 3.2. ábra tartalmazza. Az ábrán nem szerepelnek az audit és a Liquibase által használt táblák (5.4.2 pont). A 6. fejezetben található 6.3. ábra tartalmazza az adatbázis összes tábláját.



3.2. ábra. A backend legfontosabb adatbázistáblái

Forrás: saját ábra

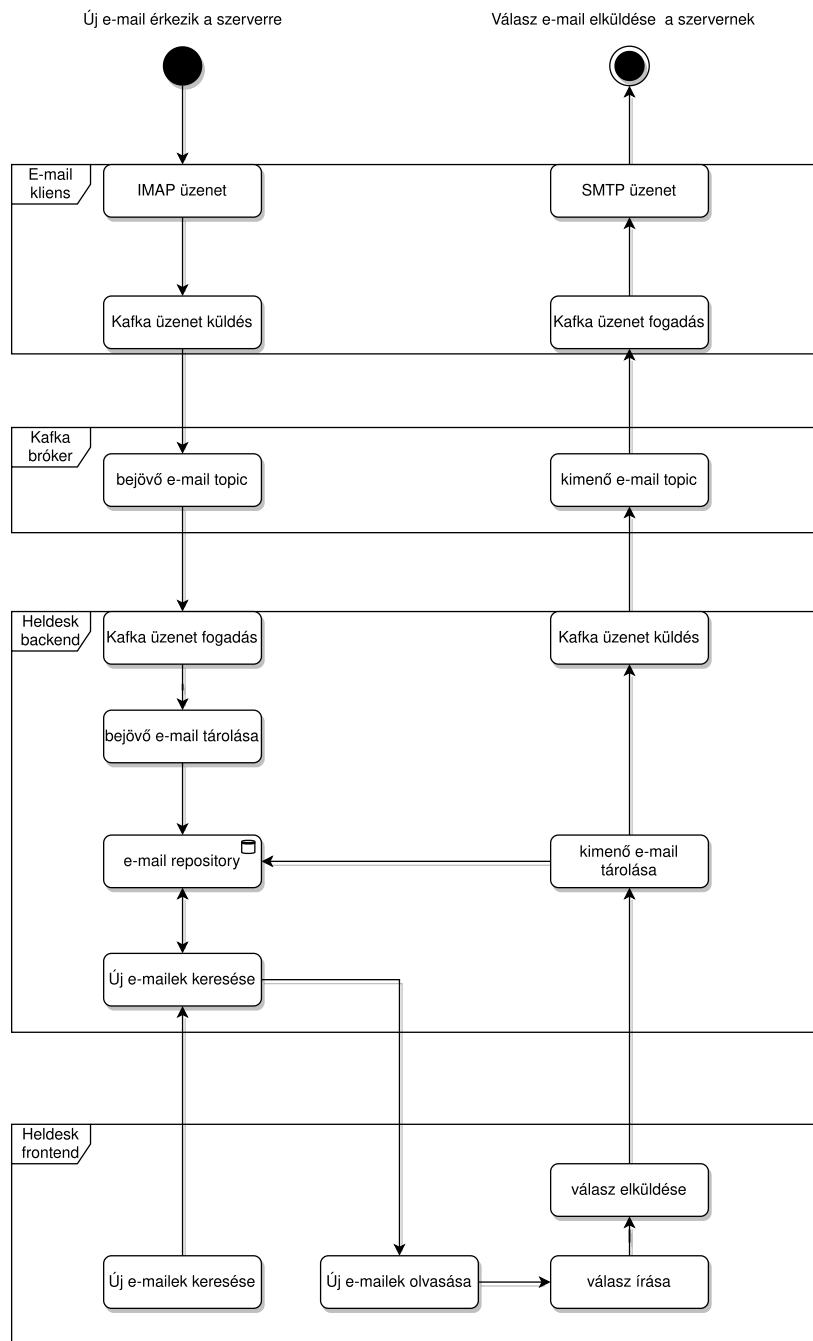
3.3. E-mail fogadásának és küldésének folyamata

A könnyebb átláthatóság érdekében, a folyamatokat egy e-mail szemszögéből mutatom be a 3.3. ábrán. Az e-mail fogadása 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 email küldése pedig az alábbi felsorolás szerint hajtódik 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 open source, vagy ingyenesen használható, ezzel minimalizálva a költségeket,
- lehetőleg minél kevesebb erőforrást használjon fel,
- úgy hogy közben elegendő csak statikus tartalmat kiszolgálnia.

Az IIS¹ – a többi open source 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.

¹Internet 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 open source 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 leképzt 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 open source monitorozó eszköz. Kifejezetten alkalmas a metrikák idősoros tárolására, gyűjtésére és megjelenítésére.

A Prometheus által létrehozott és használt PromQL² 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.

²Prometheus 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 open source 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, hő térképek, 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 open source 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 [5.2.2](#) 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

5.2. Nginx

Az Nginx-nek három különböző szerepe van:

- a helpdesk frontend alkalmazásszervereként működik (2.7 pont),
- routingszert 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-én (5.2.1) keresztül valósul meg.

5.2.1. 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.

5.2.2. Metrikák

A 4.5 pontnak megfelelően a springes alkalmazásaim egy-egy HTTP endpointon keresztül érhetőek el a Prometheus számára (*/actuator/prometheus*) és induláskor be-regisztrálják magukat az Eureka¹ szerverbe.

A Prometheus az Eurekán keresztül találja meg az instance-eket, és 15 másodpercenként összegyűjti a metrikákat. Az alkalmazások információt küldenek a Kafka konnektoraikról, REST interfészeikről és az adatbázis kapcsolataikról².

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

5.3. 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.

5.3.1. E-mail szabvány

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

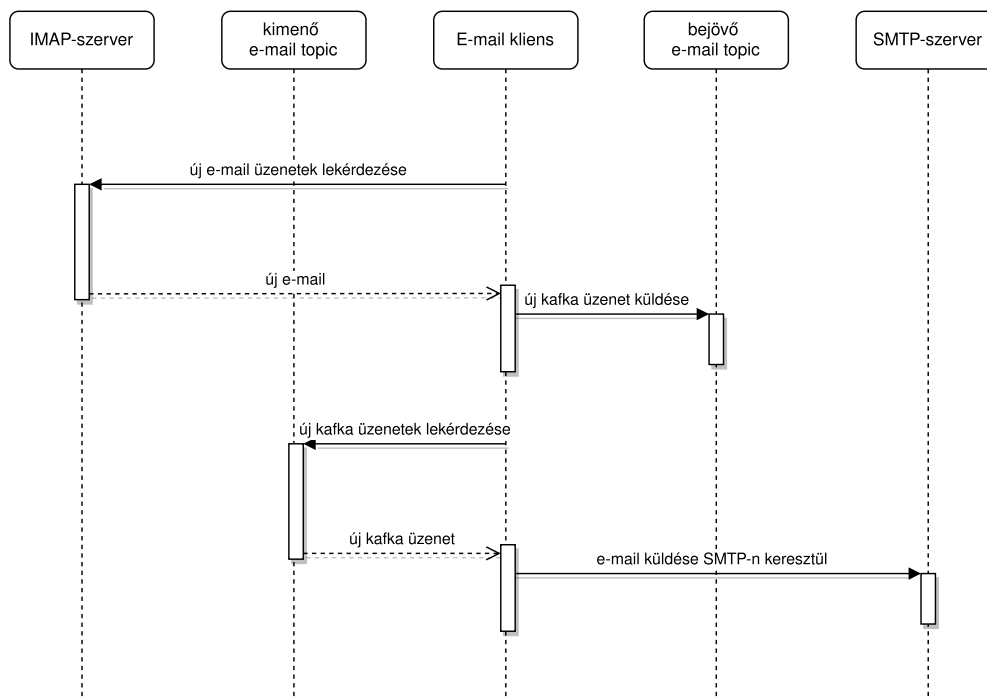
Message-ID egy globálisan egyedi azonosító ami egyértelműen azonosítja az üzenetet,

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.

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

²HikariCP-t használok JDBC kapcsolathoz



5.1. ábra. E-mail kliens szekvencia diagramja

Forrás: saját ábra

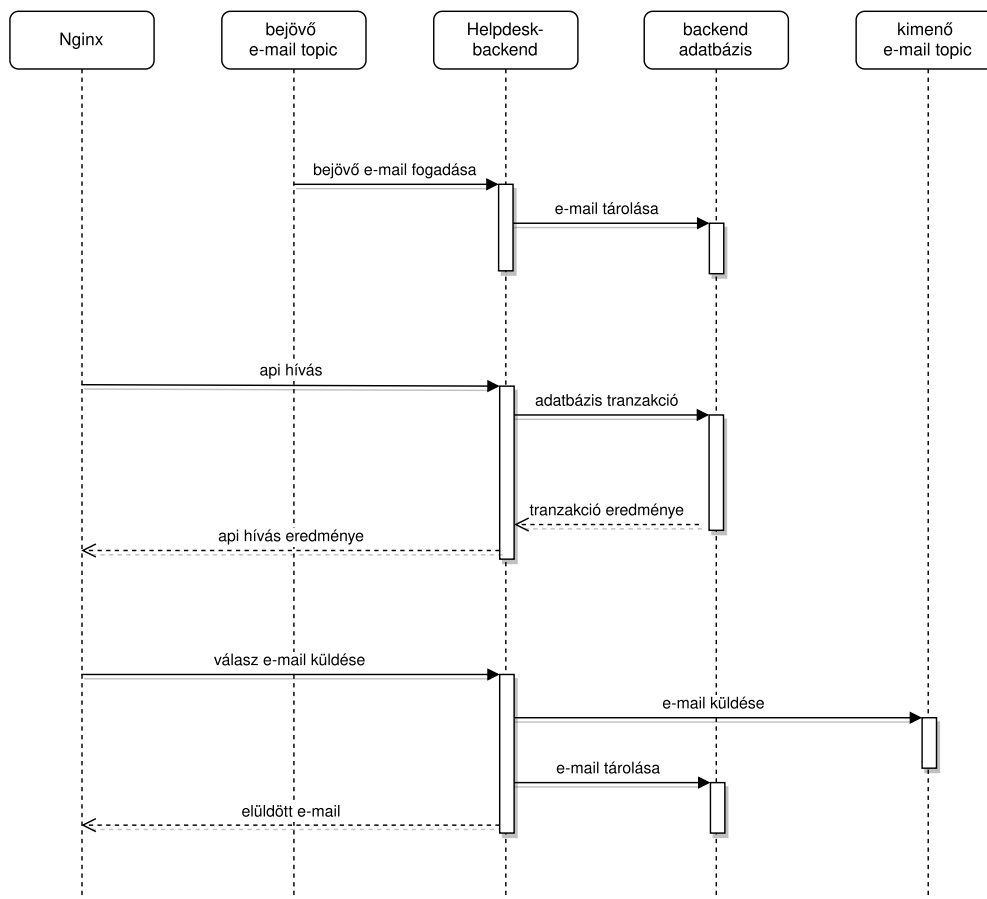
5.4. Helpdesk backend

A backend felelős az e-mail szálakkal kapcsolatos üzleti feladatok ellátásáért. Az 5.2. 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.

5.4.1. 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 micrometert és az actuatort a metrikák elkészítéséhez használtam.



5.2. ábra. Helpdesk backend szekvencia diagramja

Forrás: saját ábra

5.4.2. Adatbázis

A Spring kezeli – HikariCP-n keresztül – a PostgreSQL adatbázishoz való kapcsolódást. Az adatok kezelését Hibernate³-en keresztül, az adatbázis verziókövetését Liquibase-en (6.5.1 pont) keresztül valósítom meg.

Az e-mail szálak audit információinak és verzióinak követésére a Hibernate saját – Envers (6.5.2 pont) eszközét használom. Az Envers a neki létrehozott táblában automatikusan követi az annotációval megjelölt entitások állapotát.

5.4.3. Pesszimista konkurenciakezelés

Pesszimista konkurenciakezelésre jó példával szolgál a Liquibase (6.5.1 pont) működése.

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

³A Hibernate egy JPA implementáció, ami objektum relációs leképztést valósít meg

egy Liquibase példány tud elindulni és módosításokat végrehajtani.

5.4.4. 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 történekről. A felhasználó ilyenkor az oldal frissítése után megvizsgálja az aktuális állapotot és – amennyiben a módosításaira még mindig szükség van – újból kezdi a folyamatot.

5.4.5. 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.

5.5. 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.

5.5.1. Kommunikáció a backenddel

A backenddel való kommunikáció HTTP protokollon keresztül zajlik, a szükséges *service*-eket az OpenApi dokumentációból (5.4.5) 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.

5.5.2. 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, mert egyszerűen beilleszthető az Angular környezetbe, és a felhasználó számára intuitív kezelőfelülettel rendelkezik.

5.5.3. 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 5.3 ábra). A backend felé induló REST kéréseket a webszerver (5.2) osztja szét a rendelkezésre álló példányok között.

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

5.6. Keycloak

A Keycloak egy open source jogosultság- és hozzáférés-kezelő. Támogatja az LDAP-ot, SSO-t és a kétlépcsős azonosítást [17].

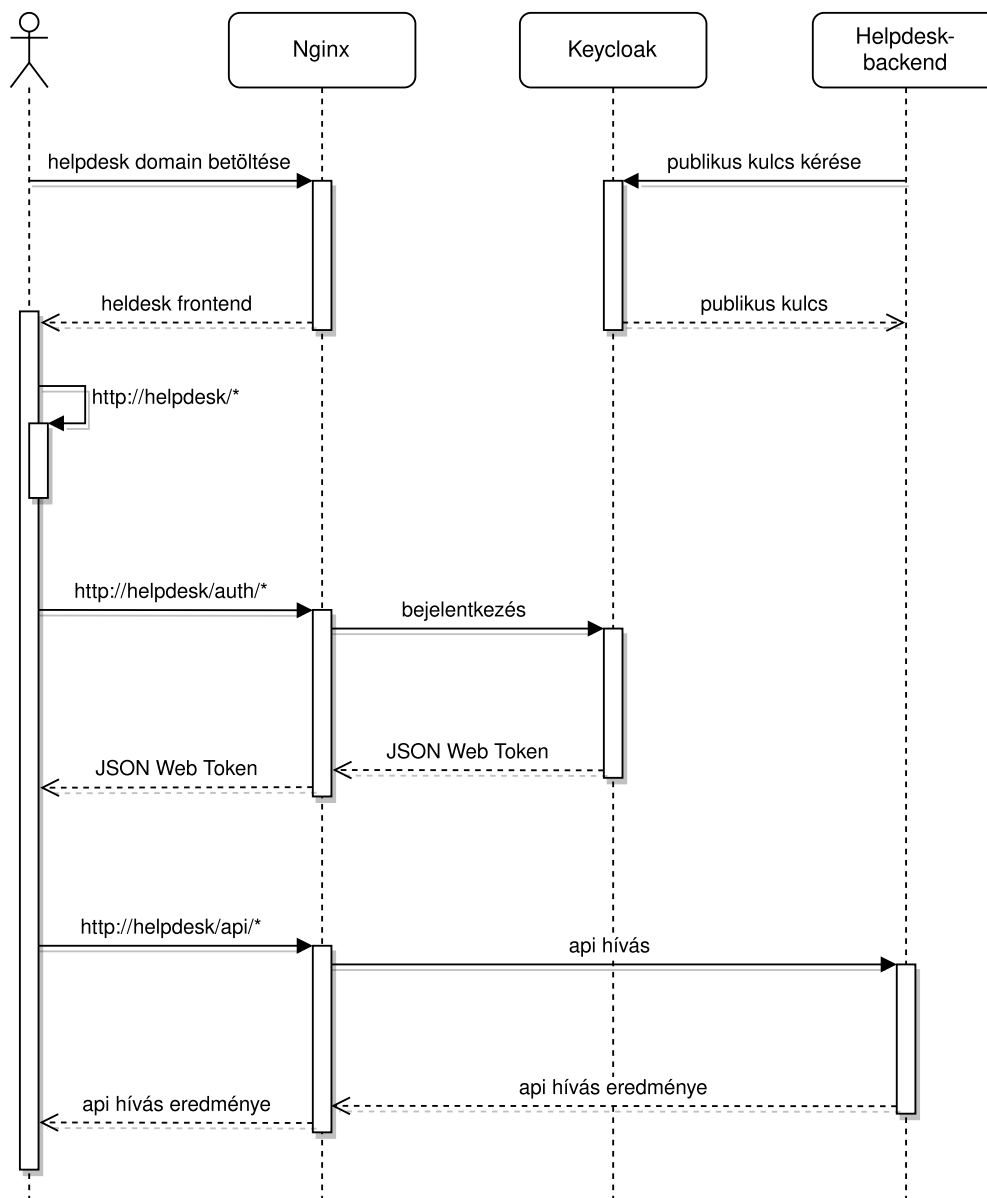
A helpdesk alkalmazásban feladata a felhasználók azonosítása, és adataiknak nyilvántartása. Különálló mikroszervizként, saját adatbázissal rendelkezik.

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

5.6.1. 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.

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.



5.3. ábra. Helpdesk frontend szekvencia diagramja

Forrás: saját ábra

5.6.2. JSON Web Token

A jogosultságkezelés technikai alapját az *rfc7519*-es szabványban [18] 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 5.3 ábra), és a megfelelő jogosultság megléte esetén engedélyezi a hozzáférést az erőforráshoz.

5.7. 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 (5.8) miatt a felhasználók adatai egy külön kafka *topic*-ban érhetőek el. Bármelyik mikroszerviznek szüksége lenne valamilyen felhasználóval kapcsolatos információra, azokat a *topic* végigolvasásával megkaphatja.

5.8. Helpdesk backend és a Keycloak elkülönítése

A felhasználók adataiért a Keycloak (5.6), az e-mail szálakért pedig a backend (5.4) 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 (5.6.2) 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 (5.7) *topic*-ban követhetőek nyomon.

A Keycloak Kafka integrációjának céljából hoztam létre a *keycloak-plugin* (6.1 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, 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-ot,
- a *travel* sorhoz tarozó helpdesk.gdf.travel@yandex.com-ot,
- és a *theater* sorhoz tarozó h.gdf.theater@gmx.com-ot.

6.1. Alkalmazás elindítása

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

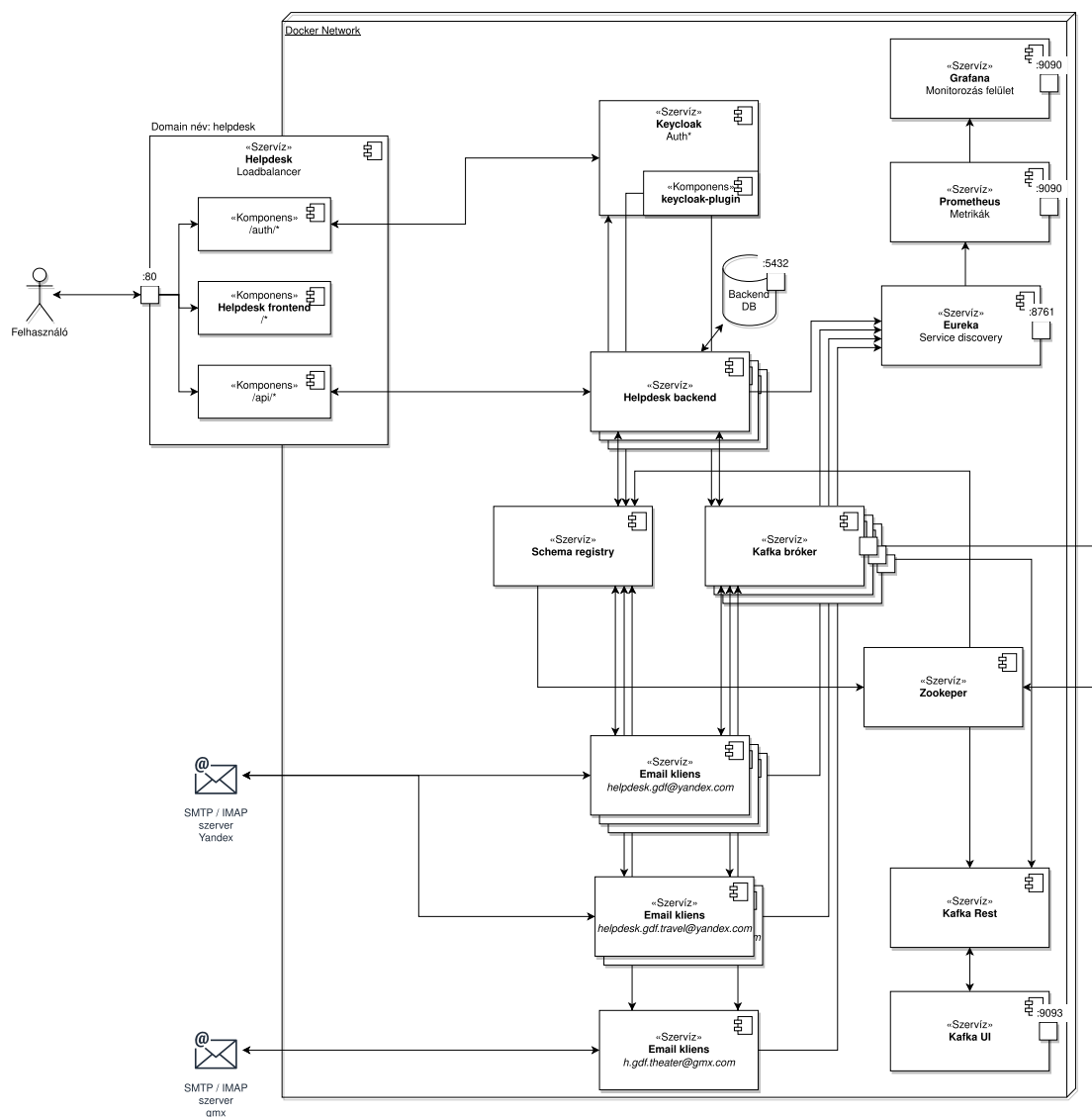
1. a *docker-compose* paranccsal elindítja a docker *containereket* (5.2.1 pont),
2. „helpdesk” domain névvel hozzáadja az Nginx (5.2 pont) IP címét a */etc/hosts* állományhoz.

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

6.2. Deployment

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

A *docker-compose* (6.1) által elindított *containereket* a 6.1. ábrán foglaltam össze. Az ábrán feltüntettem, hogy az adott *containert* a *localhost* melyik portján lehet elérni.



6.1. ábra. Deployment diagram

Forrás: saját ábra

6.3. 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 (6.6) szintén három darab.

A példányok metrikáit (5.2.2 pont) nyomon lehet követni az erre a célra létrehozott Grafana oldalon (6.2c ábra). Az oldal elérhető a *Spring metrics* menüpont alatt.

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.

6.4. E-mail fogadásának és küldésének folyamata

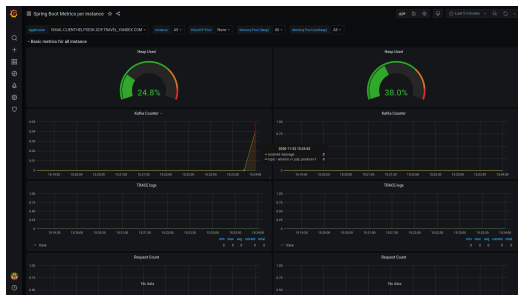
A 3. fejezetben a 3.3. ábrán bemutattam egy e-mail fogadásának elméleti útját. Most a 6.2. ábrán bemutatom hogyan követhető végig a rendszerben egy e-mail valódi útja.

E-mail fogadása:

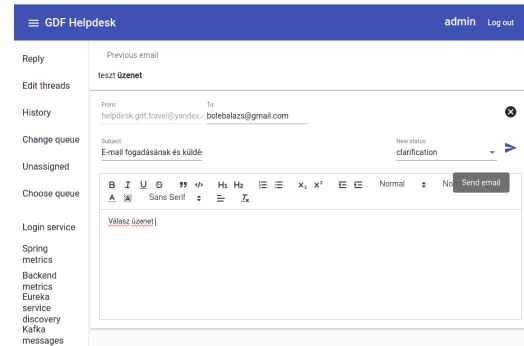
1. Egy teszt üzenet érkezik a helpdesk.gdf.travel@yandex.com címre *E-mail fogadásának és küldésének folyamata* tárggyal.
2. Az e-mail kliens kafka üzenetként publikálja az üzenetet az *email.in.v1.pub* topicba (6.2a. ábra).
3. A backend megkapja a kafka üzenetet (6.2c. ábra).
4. A backend elmenti az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

E-mail küldése:

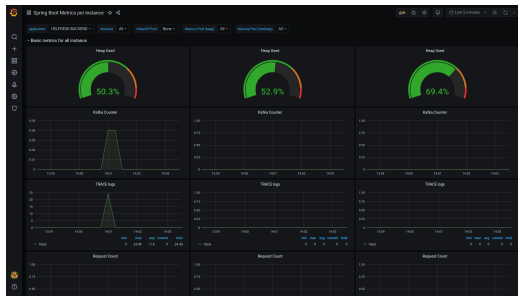
1. A felhasználó elküldi a válaszát a felhasználói felületen (6.2b. ábra).
2. A backend megkapja az üzenetet és eltárolja az adatbázisba (6.2d. ábra).
3. A *h.gdf.theater_gmx.com.v1.pub* topicban megjelenik. (6.2f. ábra) a backend által publikált kafka üzenet.
4. A topicra feliratkozott e-mail kliens fogadja és továbbítja az üzenetet. (6.2h. ábra).



(a) Az egyes instance kafka üzenet küld



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



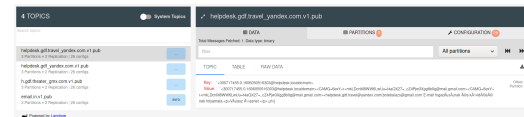
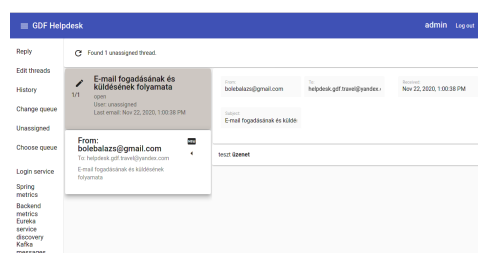
(c) Az egyes instance kafka üzenet fogad



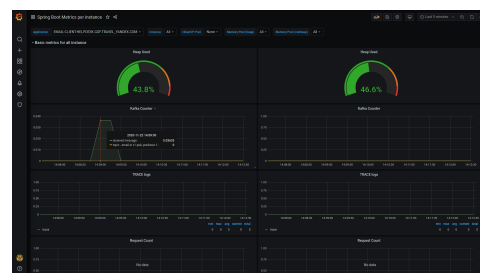
(e) Az új e-mail az adatbázisban



(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

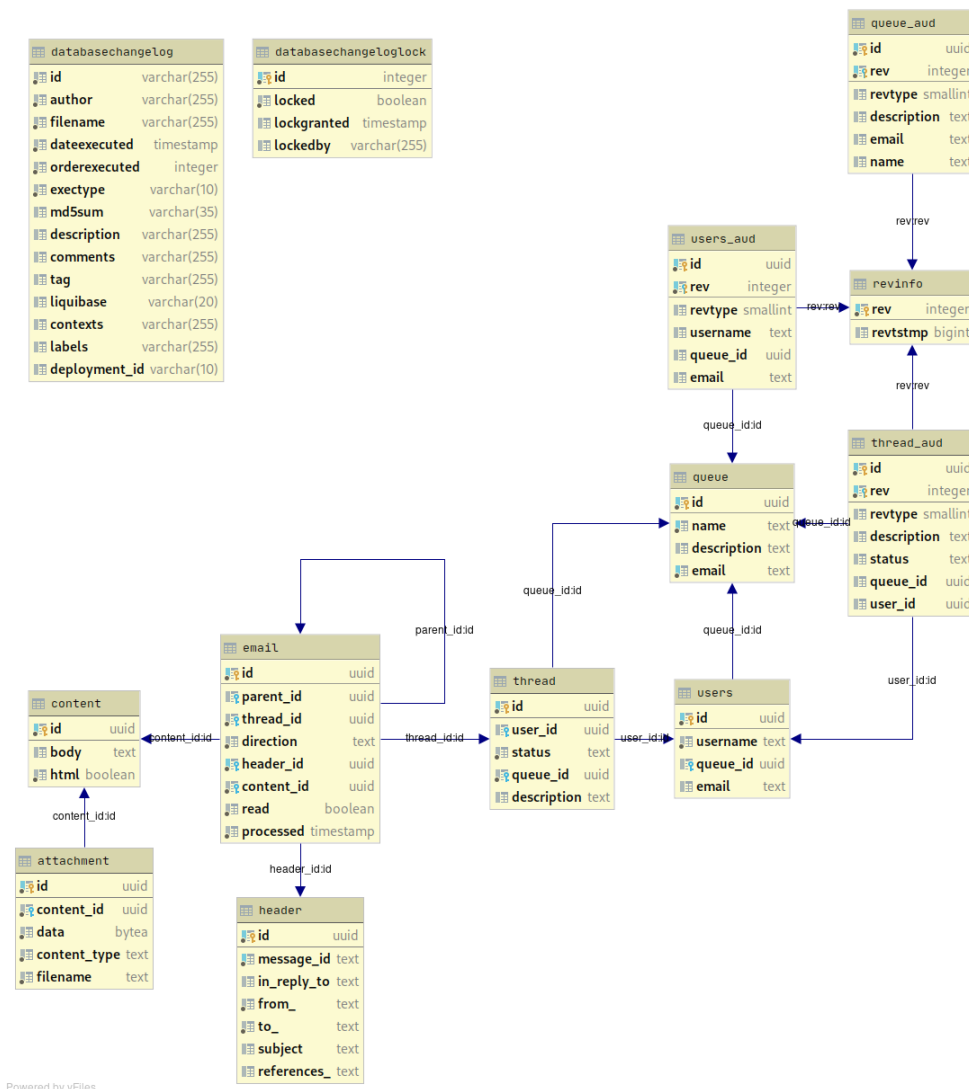


(h) Az egyes instance kafka üzenet fogad

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

6.5. Adatbázistáblák

A helpdesk backend adatbázistábláit a 6.3. ábra tartalmazza.



6.3. ábra. A backend összes adatbázistáblája

Forrás: saját ábra

6.5.1. Liquibase

A *databasechangelog* és *databasechangeloglock* táblákat a Liquibase (5.4.2. 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 szerviz 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 feladatnak ellátásához szükséges szinten férjen hozzá az alkalmazáshoz, és hogy futása során ne módosíthassa a táblákat.

6.5.2. 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.

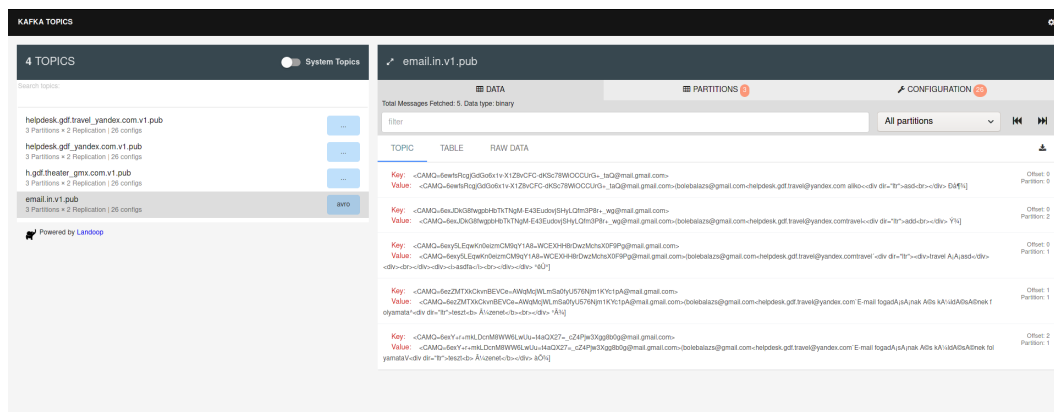
_aud tábla tartalmazza a módosítás típusát, sorszámát és az entitás új értékeit.

Az Envers – a *Hibernate Event system*en 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.

6.6. Apache Kafka

A kafka *topicok* és üzenetek elérhetőek és követhetőek a *Kafka messages* menü pontja alatti Kafka Topics UI (6.4. 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 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,

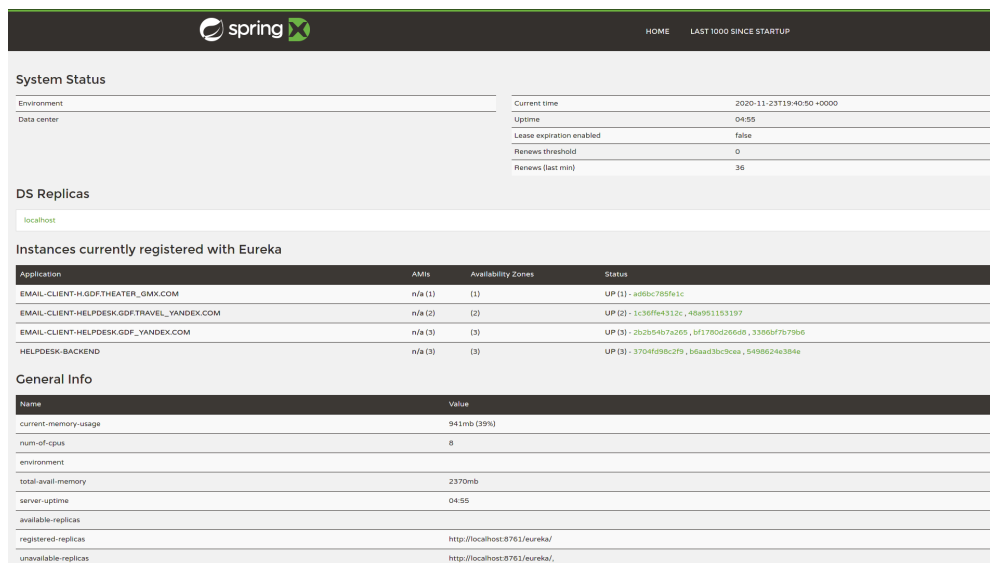
_schemas a Schemaregistry ebben a *topic*ban tárolja az alkalmazásban használt Avro schemákat.

Az alkalmazás három kafka brókert futtat 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.

6.7. Eureka

Az 5.2.2. pontban említett szervíz felderítésre az Eureka szerveret használom. A helpdesk backend és az e-mail kliens az elindításuk után közvetlenül beregisztrálják magukat az Eureka szervízbe.

Az Eureka szerveren keresztül megtekinthető és más szervizek 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 services.

Application	AMIs	Availability Zones	Status
EMAIL-CLIENT-H.GDFTHEATER_GMX.COM	n/a (1)	(1)	UP (1) - a09bc705fe1c
EMAIL-CLIENT-HELPDESK.GDFTTRAVEL_YANDEX.COM	n/a (2)	(2)	UP (2) - 1c30ffe4312c, 40e951153197
EMAIL-CLIENT-HELPDESK.GDF_YANDEX.COM	n/a (3)	(3)	UP (3) - 2b2b54b7a205, bf1700d266d0, 33060f7b79b6
HELPDESK-BACKEND	n/a (3)	(3)	UP (3) - 3704fd95c2f9, b6aad30c3cea, 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-uptime	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

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

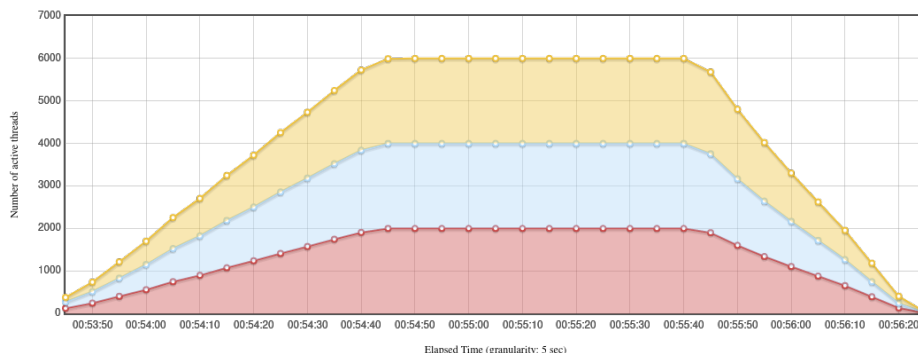
Az Apache Jmeter egy nyílt forrású terheléses teszt végrehajtására alkalmas eszköz. Számos protokollt képes kezelni, a helpdesk alkalmazás vizsgálatánál HTTP üzenetek küldésével modelleztem a párhuzamos felhasználókat.

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 azonosíthatók 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.

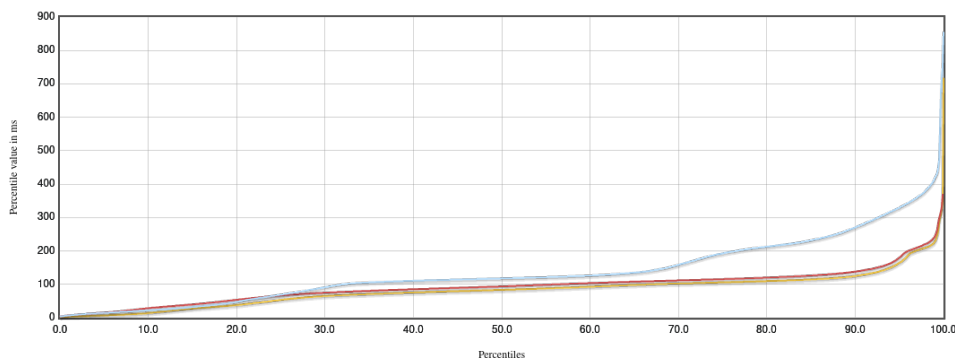
Átlag	Válaszidő [<i>ms</i>]				Áteresztőképesség [Tranzakció/ <i>s</i>]	Válasz [%]	
	Max.	Medián	P90	P95		< 3 <i>s</i>	< 6 <i>s</i>
99,32	856	88,00	223,00	340,00	900,01	100	100

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

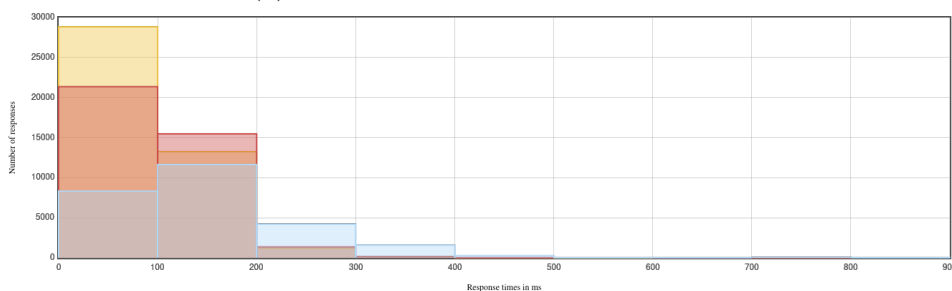
A 7.2 ábrán és a 7.1 táblázat adataiból látszódik, hogy:

- 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 terheléses teszt eredménye 100 felhasználóval

Forrás: saját ábra

7.4. Csúcsteljesítmény vizsgálata

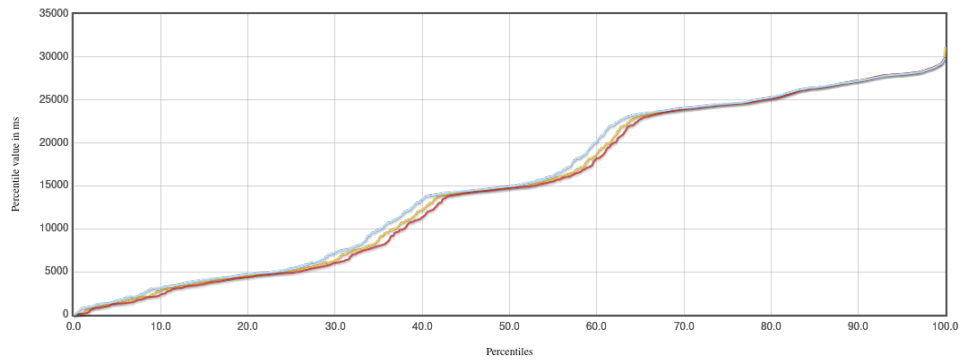
A helpdesk backend egy önálló példányát vizsgáltam 6 000 párhuzamos felhasználó modellezésével. Mint ahogy a teszt során használt párhuzamos szálakról készített a 7.1 ábrán is látszódik, a felfutási idő 60 másodperc volt. Az ábrán látható ahogy a JMeter a felfutási idő után egyszerre 6 000 szálon futtatja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

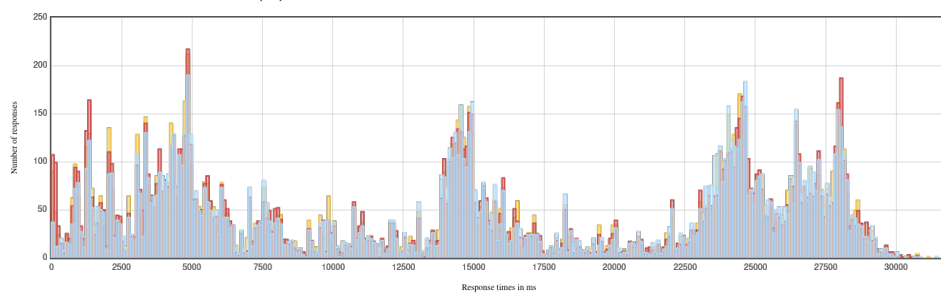
Átlag	Válaszidő [ms]				Áteresztőképesség [Tranzakció/s]	Válasz [%]	
	Max.	Medián	P90	P95		< 3s	< 6s
15 133,81	31 732	24 553,00	28 011	28 407	281,72	10,19	27,78

7.2. táblázat. Egy példányban futó helpdesk backend terheléses 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

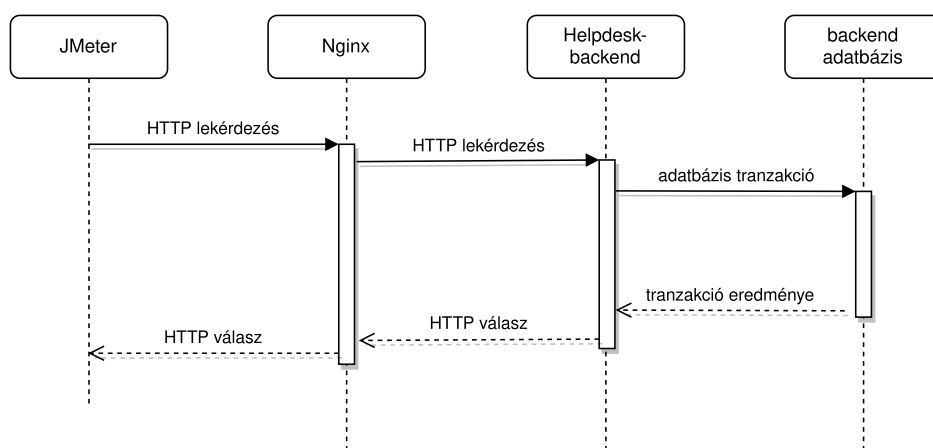
7.3. ábra. Egy példányban futó helpdesk backend terheléses teszt eredménye 6 000 felhasználóval

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.

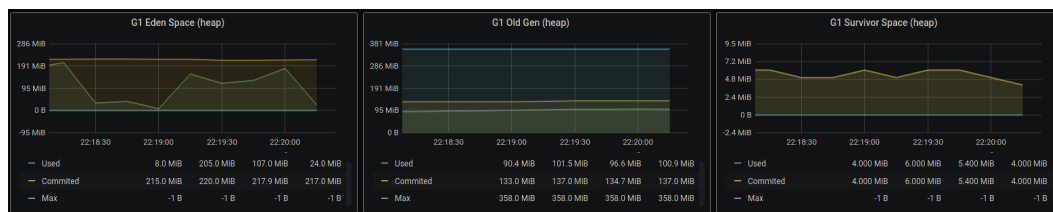


7.4. ábra. A terheléses tesztben résztvevő alkalmazások

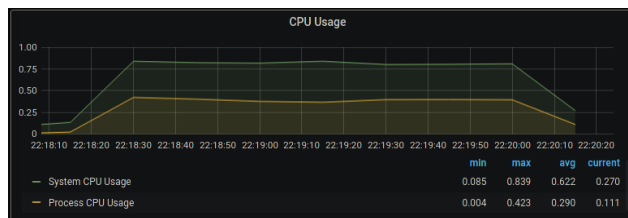
Forrás: saját ábra

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

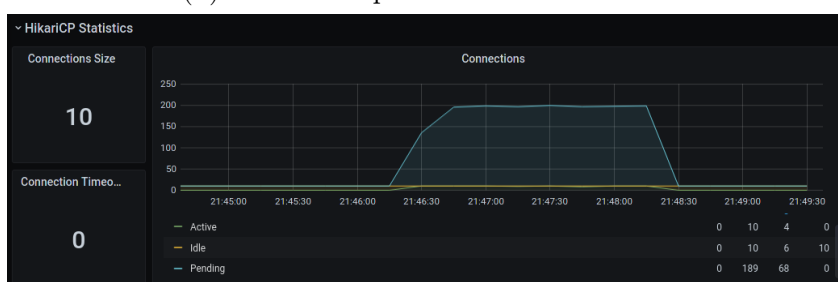
- A backend memóriaigénye nem nőtt meg jelentősen (7.5a ábra).
- A processzorigény szignifikánsan megemelkedett (7.5b ábra). A rendszer CPU felhasználása 80%-ra a backendé 40%-ra nőtt.
- Az adatbázis kapcsolatok fenntartására használt HikariCP-ben (5.4.2 pont) lényegesen feltorlódtak a lekérdezések (7.5c 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 felhasználót¹, 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

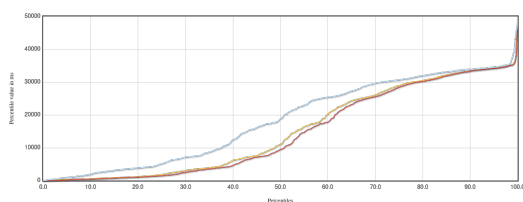
- az átlagos és a medián válaszidő, valamint áteresztőképesség nem változott,
- a három másodperc alatt megérkező válaszok aránya – az első esethez (7.4) képest – meg két és félszereződött, 26,41%-ra nőtt.

¹Az alapbeállítás 1 024 kapcsolat

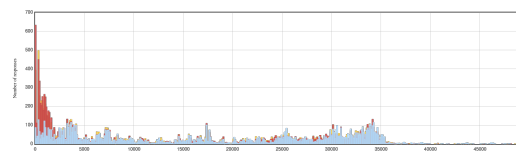
Átlag	Válaszidő [ms]					Áteresztőképesség [Tranzakció/s]	Válasz [%]	
	Max.	Medián	P90	P95	P95		< 3s	< 6s
15 772,71	49 367	29 450,00	34 256	34 898	34 898	265,72	26,41	36,78

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

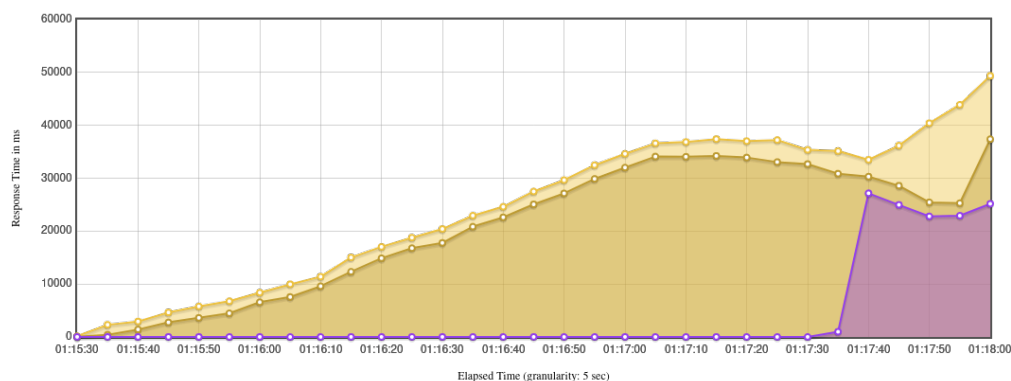
Ugyanez jelenik meg a 7.6c ábrán is, a teszt első háromnegyedében a backend néhány kérésre azonnal válaszol, míg a többségre – lásd medián – csak sokkal később.



(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

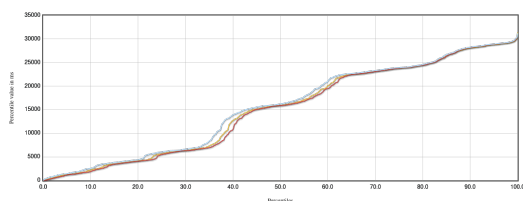
A 7.5c ábrán látszódik hogy a HikariCP-ben beállított 10 kapcsolat hamar elfogy, már mérés elején 189-re nő, és tartósan ott is marad a poolra várakozók száma.

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

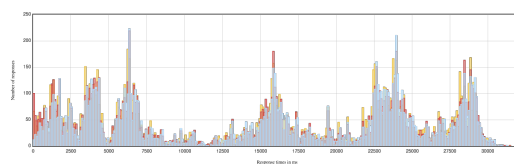
Ezért a megismételt a JMeter tesztben a backendben megháromszoroztam a HikariCP pooljának a méretét. A mérési eredmények – 7.4 táblázat – nem mutatnak eltérést az első, kiinduló állapothoz képest.

Válaszidő [ms]					Áteresztőképesség [Tranzakció/s]	Válasz [%]	
Átlag	Max.	Medián	P90	P95		< 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



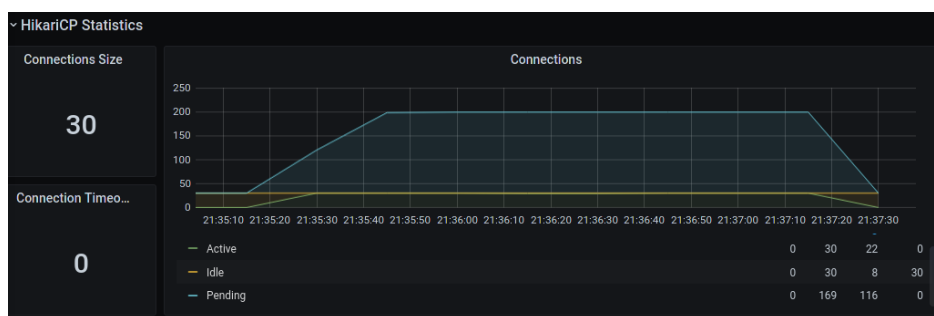
(b) A válaszidők eloszlása

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.

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

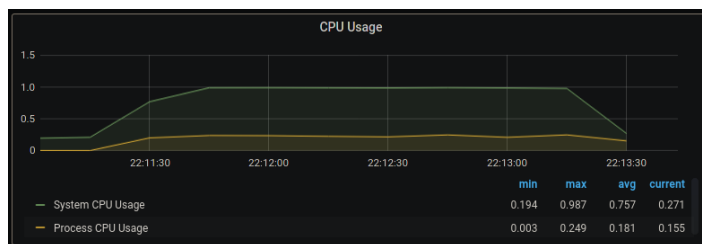
7.5. táblázat. Három példányban futó helpdesk backend terheléses teszt eredménye 6 000 felhasználóval

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.

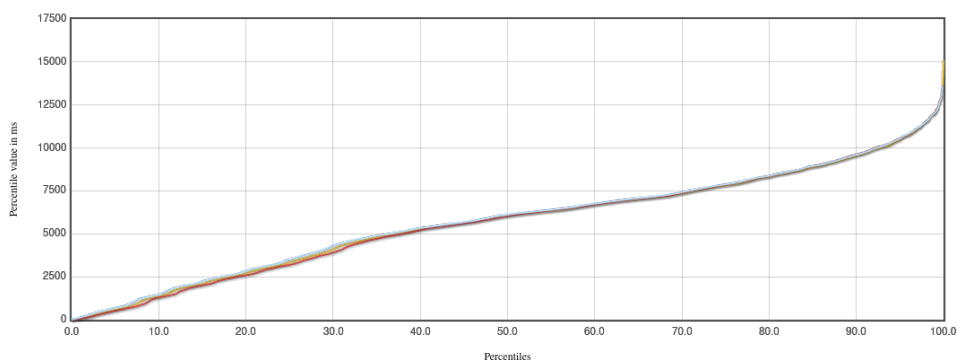
A 7.10 ábrán megjelenített mérési eredményeken látszódik hogy a válaszidők szórása nagy mértékben csökkent, a várható bizonytalanság így már sokkal inkább tolerálható.

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.

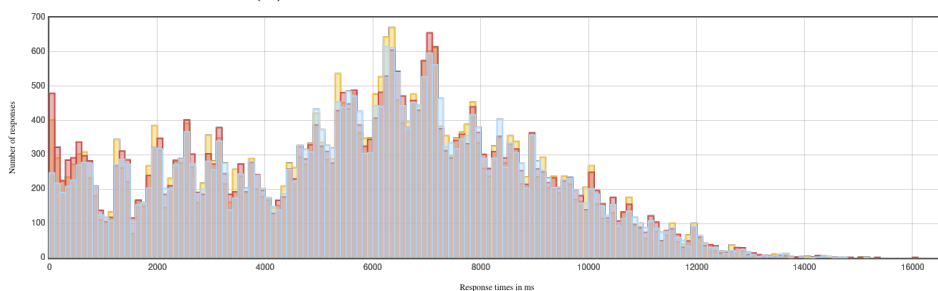


7.9. ábra. A backend egy példányának a processzor használata a Grafana felületén

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 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.

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 *clusterre* migrálása. A docker natívan támogatja a Microsoft Azure és az Amazon [19] szolgáltatókat. Így tehát a kód és a beállítások módosítása nélkül lehetséges az alkalmazás *clusteresítése* 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 szerviz egy feladattal foglalkozik, ha például le kell cserélni a frontendet, 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 szervizek, 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.

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

A moduláris felépítés, és a szervizek 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 szerviz 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.

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 választhatam 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.

Terheléses 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.
- [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] Keycloak. Letöltve: 2020. Nov. 18. URL: <https://www.keycloak.org/>.
- [18] Json web token (jwt). Letöltve: 2020. Nov. 18. URL: <https://tools.ietf.org/html/rfc7519>.
- [19] 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 : 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.	< 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
```

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. Headers : ETag (string) : The version of the EmailThread.	EmailThreadVersion
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)
Query	ifMatch <i>required</i>	If-Match header	string
Body	body <i>required</i>	EmailThread containing the new properties	EmailThread

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	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
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	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
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

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

Name	Description	Schema
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

EmailThreadVersion

EmailThread with version number

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

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