Gábor Dénes Főiskola

Mérnökinformatikus alapképzés

Helpdesk rendszer megvalósítása mikroszerviz alapú elosztott alkalmazással

Bőle Balázs

Konzulens:

Dr. Nagy Elemér Károly

Szoftverfejlesztés szakirány

2020 november

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 problémasorba kerül, ahonnan a sorhoz hozzárendelt csoport valamelyik ilyen jogokkal felruházott tagja felelőst vagy felelősöket rendelhet az adott kérdéshez, illetve a kérést más problémasorba 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 alklamazás. A buildhez felhasznált programok: maven, nodeJs, npm, angular-cli.
Az authentikációért és authorizációért dedikált keycloak szerver felel. A frontend és a backend között REST alapú, a bakcend 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 készítem 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.

GÁBOR DÉNES FŐISKOLA

Budapest, 2020 november.

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

Ábrák jegyzéke

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

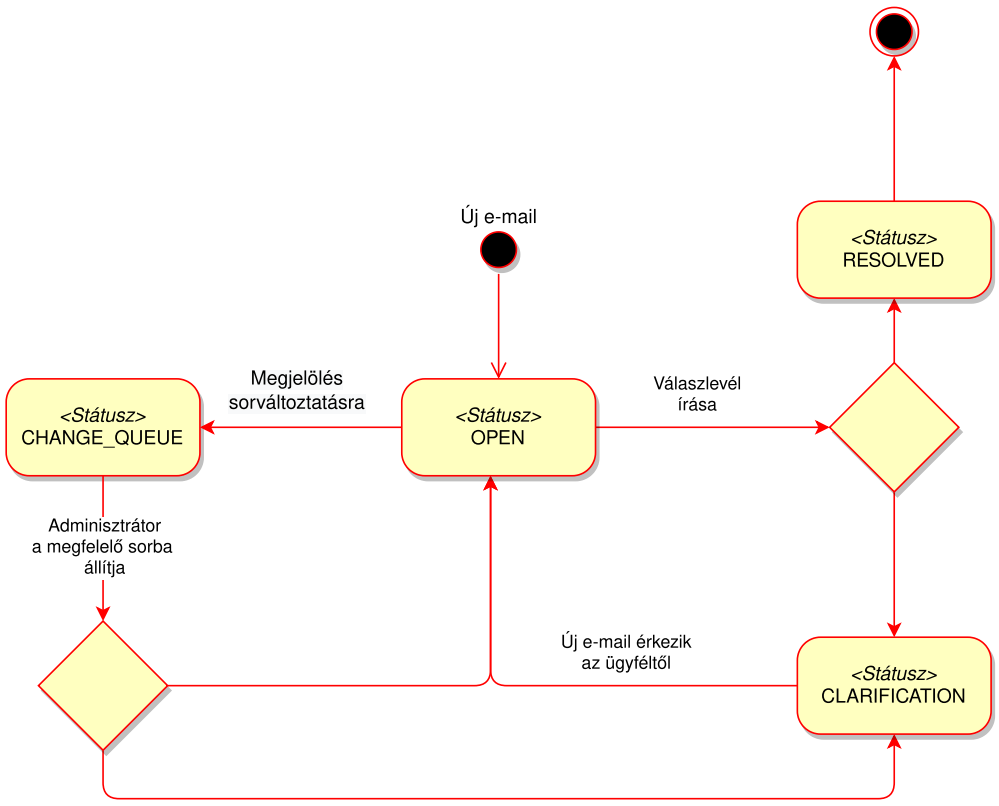
A rendszer által kezelt üzenetek szálakba rendezve érhetőek el. Egy szál az ügyfél és a felhasználó közötti üzenetváltásokból épül fel.

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 emailszálait 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.

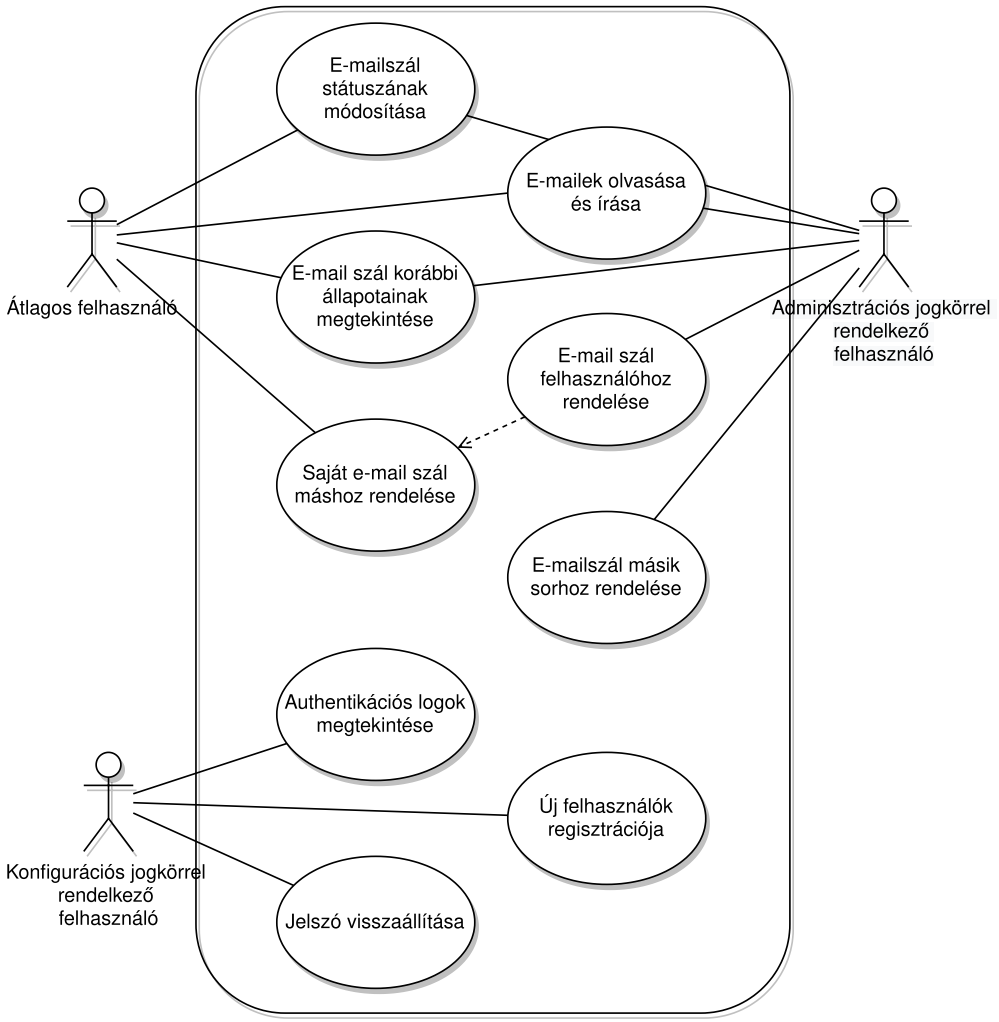
A konfigurációs jogkörrel rendelkező felhasználó feladata más felhasználók regisztrálása, valamint az alakalmazásban használt jogkörök (*role*-ok) kezelése. Lehetősége van továbbá authentikációs logok megtekintésére, jelszó visszaállítására és más felhasználók megszemélyesítésére (*impersonate*).

A felhasználói felületen elérhető funkciókat az 1.2 ábra foglalja össze.

1.2. Nem funkcionális igények

1.2.1. Horizontális skálázhatóság

A kiszolgálandó 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 legaktívabbak.

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

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

1.2.3. Mérhető indikátorok

A rendszernek átlagosan 100 felhasználót kell kiszolgálnia másodpercenként. A várható csúcsteljesítmény 10 000 lekérdezés másodpercenként. A tolerálható legnagyobb

válaszidő 1 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 (2.6), a backend és az e-mail kliens Spring Boot-ot (2.7) használ. A alkalmazáson belüli események kezelésére és tárolására Apache Kafkát (2.5) 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 vertikális[1] és horizontális skálázhatóság[2].

[1] több processzor vagy memória bevonása

[2] újabb példányok futtatása

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

erőforrás felderítés a rendszer által nyújtott erőforrások automatikus felfedezhetősége[3].

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

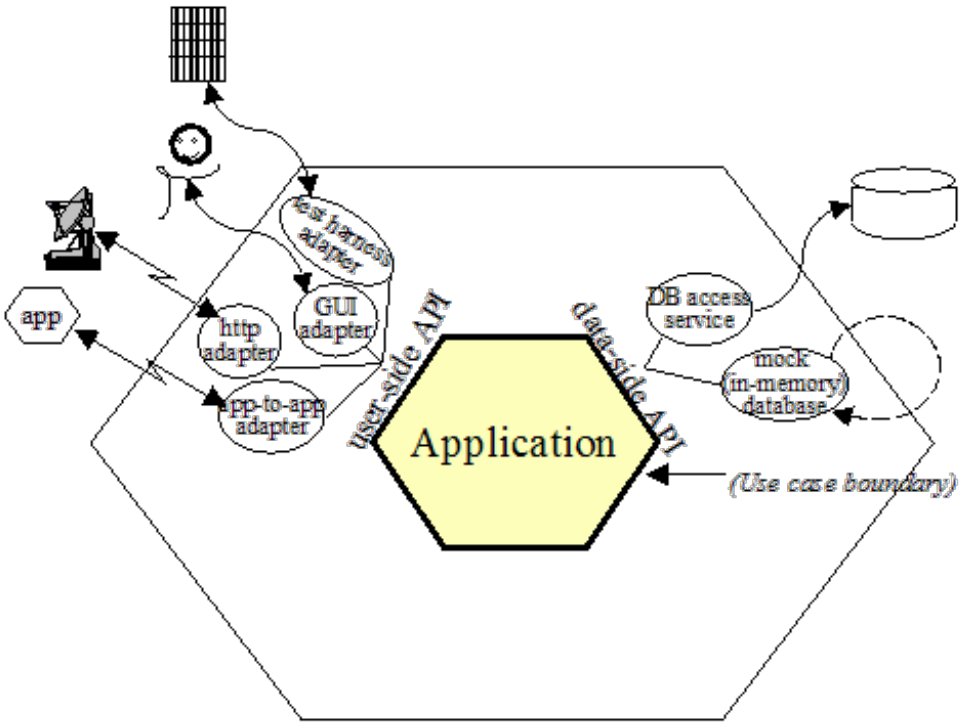
A hexagonális architektúra –vagy más néven portok és adapterek architektúrája– egy Alistair Cockburn által létrehozott [4] szoftverterezési minta. Nevét a cikkben felrajzolt hatszögletű rendszerábrázolásról kapta (2.1 ábra), ami szembemegy a korábban elterjedt réteges elrendezéssel.

Az eredeti szándék mögötte az alkalmazás függetlenítése mindennemű külső függőségtől[4], 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 port-adapter 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.
- Keretrendszertől függetlenül is megvalósíthatóak. A megvalósítás nem függ semmilyen könyvtártól vagy egyéb tulajdonságtól.

[3] angolul *service discovery*-nek hívják
[4] például adatbázis, felhasználók, automatizált tesztek



2.1. ábra. A hexagonális alkalmazás külső függőségeinek elszeparálása

Forrás: Alistair Cockburn [4]

2.3. Rétegek szeparálása

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

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. 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 CQRS[5] 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 újrajátszásával bármikor megkapható.

Az Apache Kafkát (2.5) 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.

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

2.5. 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 üzenetbróker-en keresztül egy (*topic*)-ba teszi közzé. A fogadó fél hogy megkaphassa az üzenetet, feliratkozik az adott témára.

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

A üzenetbórkerek összehaangolását a Zookeper szerviz végzi. MIvel minden kafka bróker beregisztálja magát a szervízbe, a Zookeper mindig naprakész információval rendelkezik az üzenetbrókerekről.

[5]Command Query Responsibility Segregation

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.

2.6. 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 mértében a kliens oldalon fut, így a szerver oldalon elegendő egy egyszerű, statikus HTML-oldalt visszaadó alkalmazásszerver használata.

2.7. Spring Boot

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

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

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

[6] Angolul *Dependency Injection*

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

[8] Például: metrikák, biztonság, adattárolás

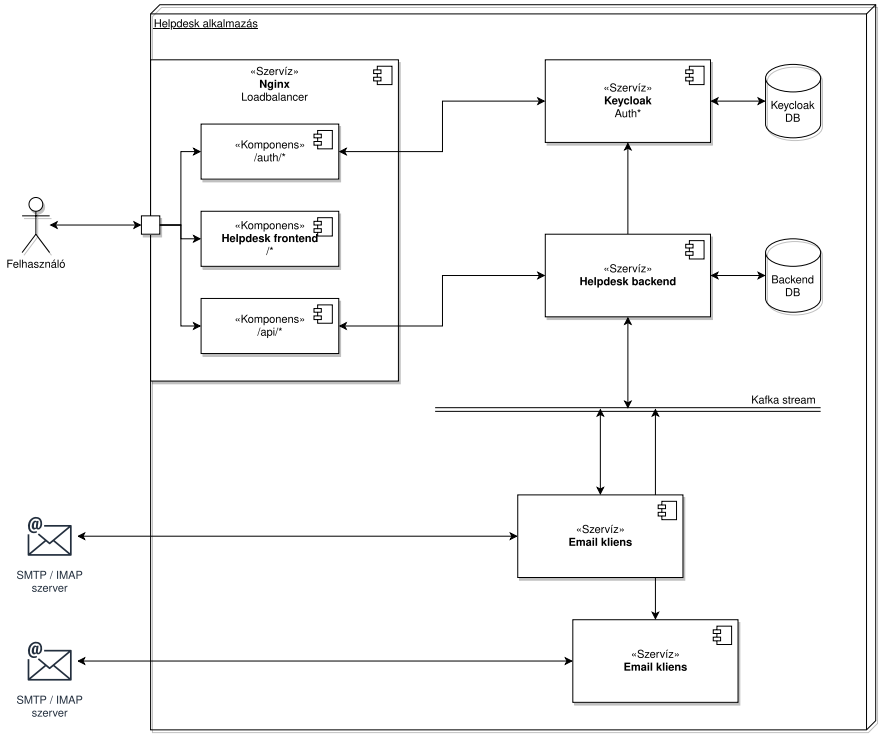
3. fejezet

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.



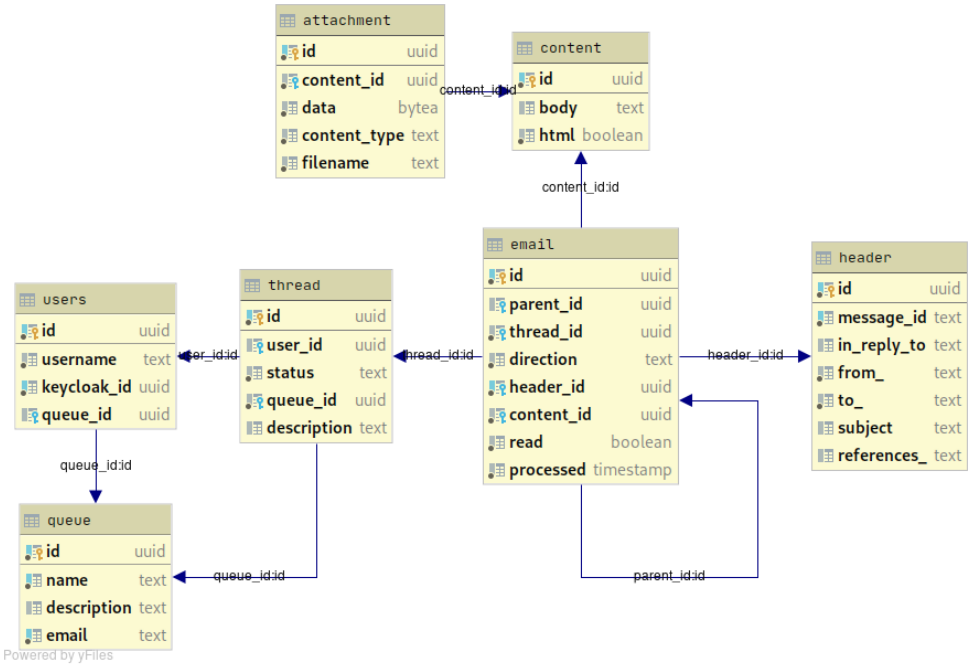
3.1. ábra. A legfontosabb komponensek

Forrás: saját ábra

- A felhasználó az nginx-en (4.1.1 pont) keresztül éri el a heldesk alkalmazást.
- Az nginx dönti el, hogy melyik URL-t melyik szerviz szolgálja ki.
- Az email kliens és a heldpesk 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

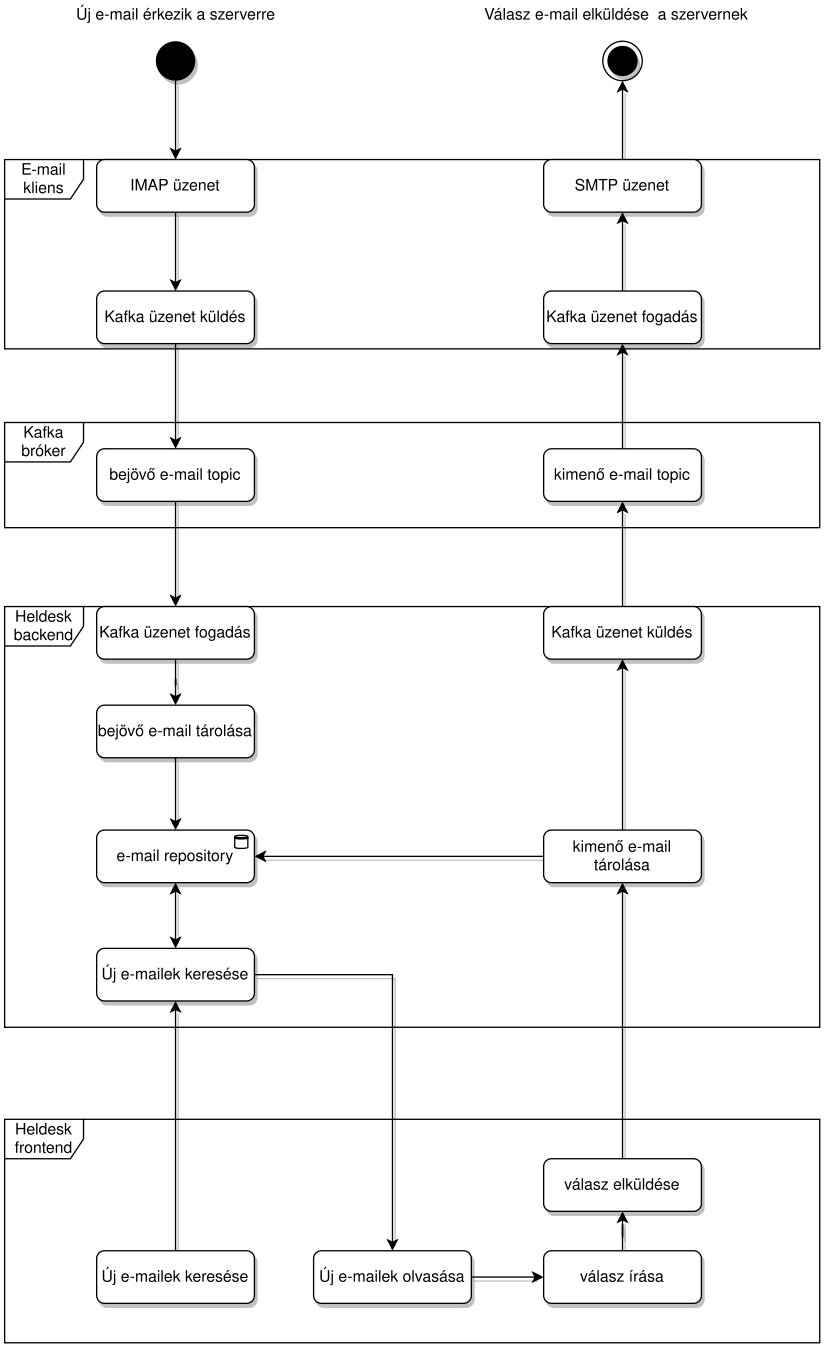
A könnyebb átláthatóság érdekében, a folyamatokat egy e-mail szemszögéből mutatom be a 3.3. ábrán. Az e-mail fogadása során:

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 *topic*ra feliratkozott helpdesk backend megkapja a kafka üzenetet.
4. A helpdesk backend eltárolja az új üzenetet az adatbázisban
5. A felhasználó a frontend segítségével lekérdezi az újonnan beérkezett e-maileket.
6. A helpdesk backend a kérésre elküldi az újonnan fogadott e-mailt.

Az e-mail küldése 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 *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

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 2.6 pont)
- Routingot valósít meg, rajta kerestül érhető el a helpdesk backend és a keycloak szerviz
- HTTP cahce-ként működik a frontend és a backend között, illetve a frontend és a keycloak között

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

4.1.2. Docker konténerizáció

Az alakalmazás 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 alkamazásaim egy-egy HTTP endpointon keresztül érhetőek el a prometheus számára (*/actuator/proemtheus*) és induláskor beregisztrálják magukat az eureka[1] szerverbe.

A prometheus[2] az eurekán keresztül találja meg az instanceokat, és gyűjti össze a metrikákat. Az alkalmazások információt küldenek a Kafka konnektorukról, REST interfészükről és az adatbázis kapcsolatukról[3].

A Prometheus által összegyűjtött adatokat grafanában[4] 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,

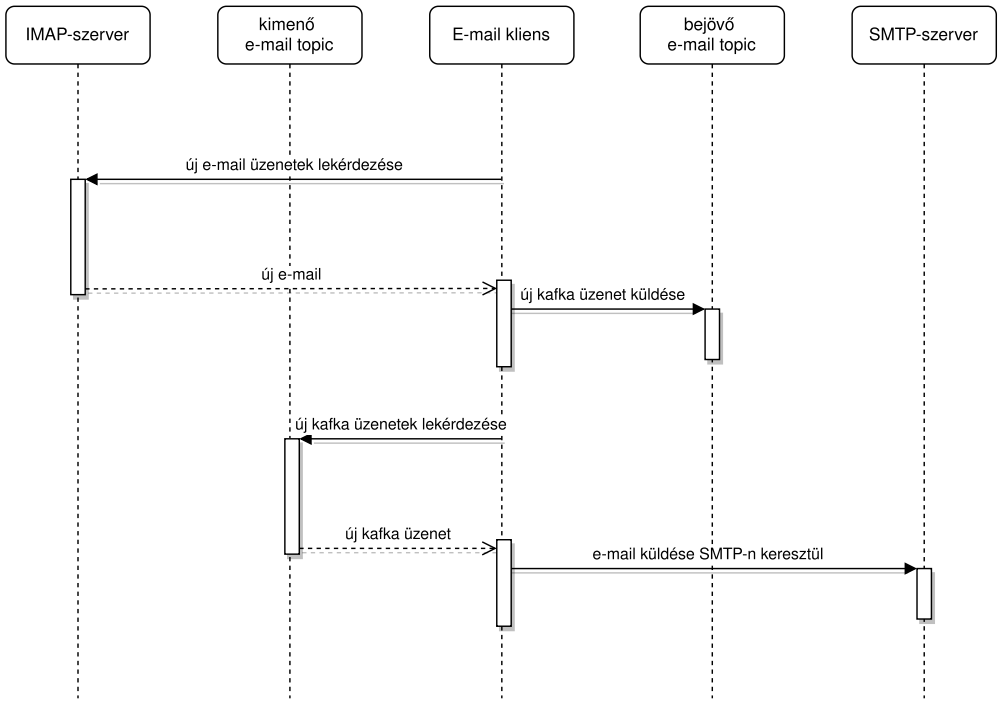
References azonosítja az üzenet szálat, értéke az eredeti üzenetek *Message-ID*-jai vesszővel elválasztva.

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

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

[3] HikariCP-t használok JDBC kapcsolathoz

[4] A Grafana egy open source elemző és megjelenítő web alkalmazés



4.1. ábra. E-mail kliens szekvencia diagramja

Forrás: saját ábra

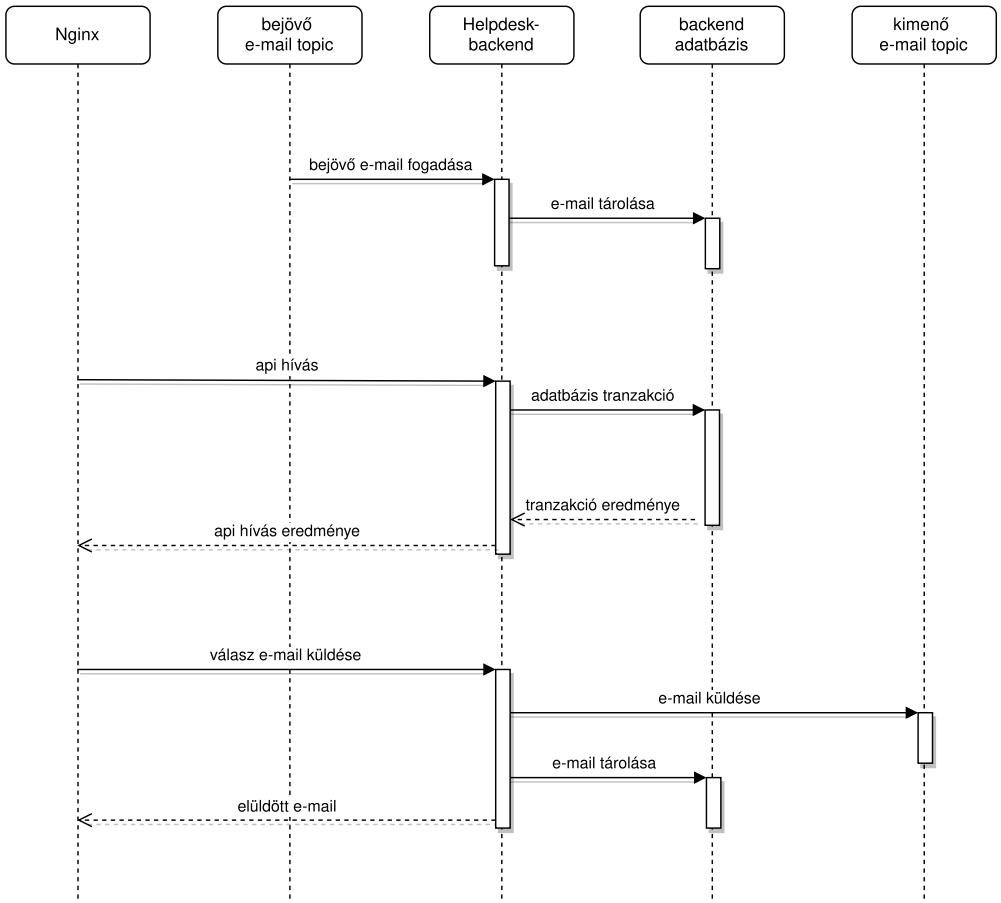
4.3. Helpdesk backend

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 (2.7 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 controller*ekhez, 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 Hibernate[5]-en keresztül, az adatbázis verziókövetését Liquibase-en keresztül valósítom meg.

Az e-mail szálak audit információinak és verzióinak követésére a Hibernate Envers eszközét használom. Az Envers a neki létrehozott táblában automatikusan követi a megjelölt Hibernate objektumok állapotát.

4.3.3. 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 dokumnetációt készít. A dokumentációt az A függelékben csatoltam a dolgozatomhoz.

[5]A Hibernate egy JPA implementáció, ami objektum relációs leképzést valósít meg

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 dokumnetációból (4.3.3) 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 alapjának 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 lérehozására az open source Quill szövegszerkesztőt használtam. Egyszerűen beilleszthető az Angular környezetbe, és a felhasználó számára intuitív kezelőfelülettel rendelkezik.

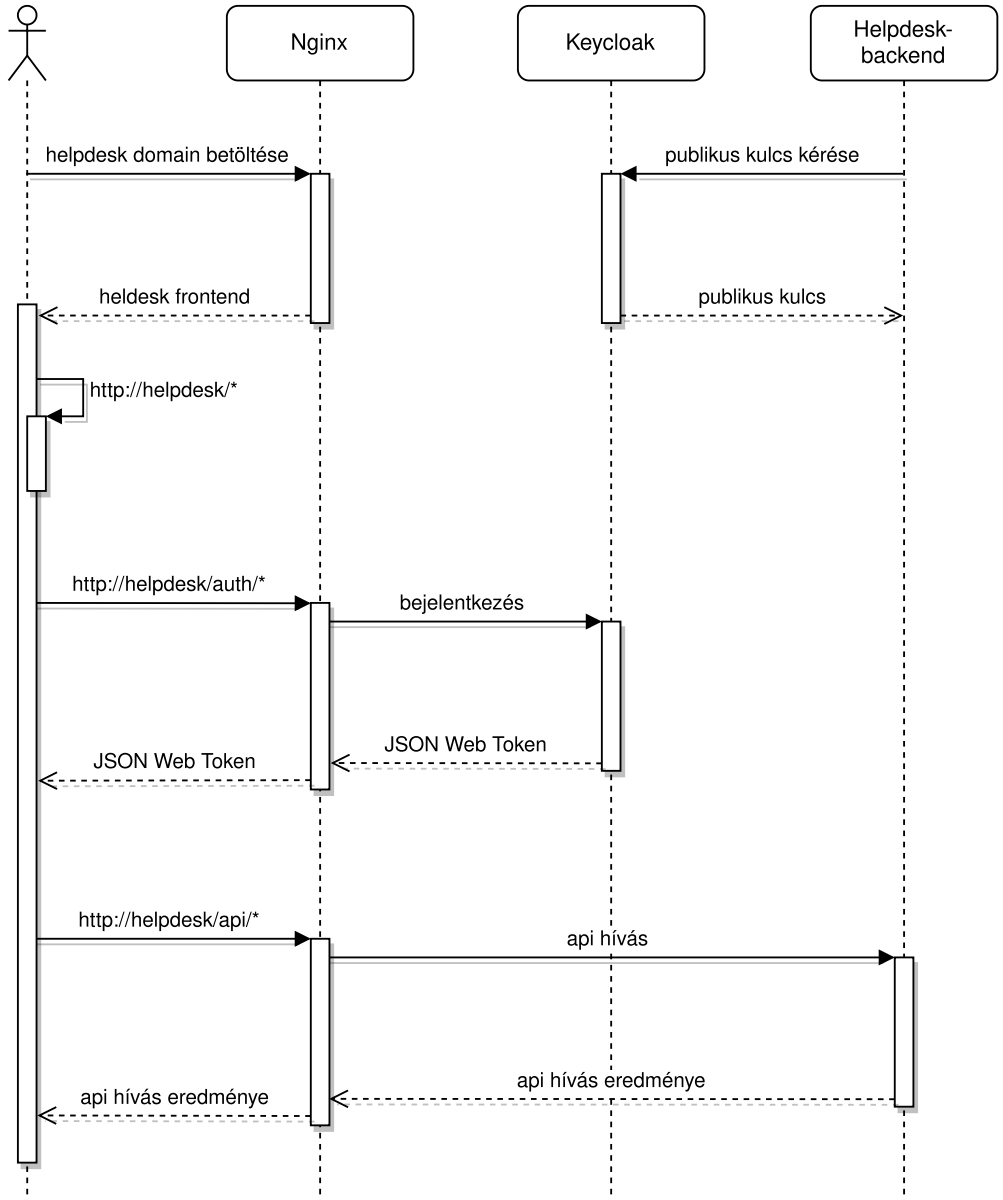
4.4.3. Futtatási környezet

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].



4.3. ábra. Helpdesk frontend szekvencia diagramja

Forrás: saját ábra

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.

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 alakalmazás funkcionális (1.1.3) feladatiért felelős.

A *helpdesk realm*on belül további két jogkört külöböztetek meg. Az *admin_user* szerepbe tartozó felhasználók képesek más e-mail szálait 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 tokent 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 (2.5 pont) mennek keresztül. A szeparációval függetlenné teszem egymástól a két rendszer működését, ami lehetővé teszi az eltérő igénybevételnek (1.2.2 pont) megfelelő skálázhatóságot.

4.7. Helpdesk backend és a Keycloak elkülönítése

A felhasználók adataiért a Keycloak (4.5), az e-mail szálakért pedig a backend (4.3) felelős. Az üzleti igény megköveteli hogy a felhasználók e-mail sorokhoz, és az e-mail szálak felhasználókhoz legyenek rendelve. A helpdesk backendnek éppen ezért tárolnia kell a fennálló kapcsolatot.

A felhasználó Keycloaktól kapott egyedi azonosítóját tárolom a backenden. Ez alapján azonosítom a JSON Web Tokenben (4.5.2) érkező felhasználót.

A felhasználók bejelentkezésről értesül a backend és eltárolja az új felhasználók egyedi adatait. Ez a megoldás addig használható, míg a felhasználókról tárolt mezők nem változnak meg, és a felhasználók nem törölhetőek. Amint ezek a feltételek megváltoznak, vagy más, felhasználókkal kapcsolatos igények jelennek meg, akkor a 2.4 pontban megismertetett CQRS utat kell megvalósítani.

A Keycloakban regisztrálni kell egy Kafka event providert, és a kafka *topic*on keresztül kell követni a felhasználók adatait. Ezzel is elválasztva egymástól a két mikroszerviz működését.

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

1. a *docker-compose* parancssal elindítja a docker *container*eket (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 könyebb bemutathatóság érdekében a szemléletesebb szervizeket –hogy ne a docker daemon által kiosztott IP címen keresztül kelljen elérni– a docker hálózaton kívül is elérhetővé tettem.

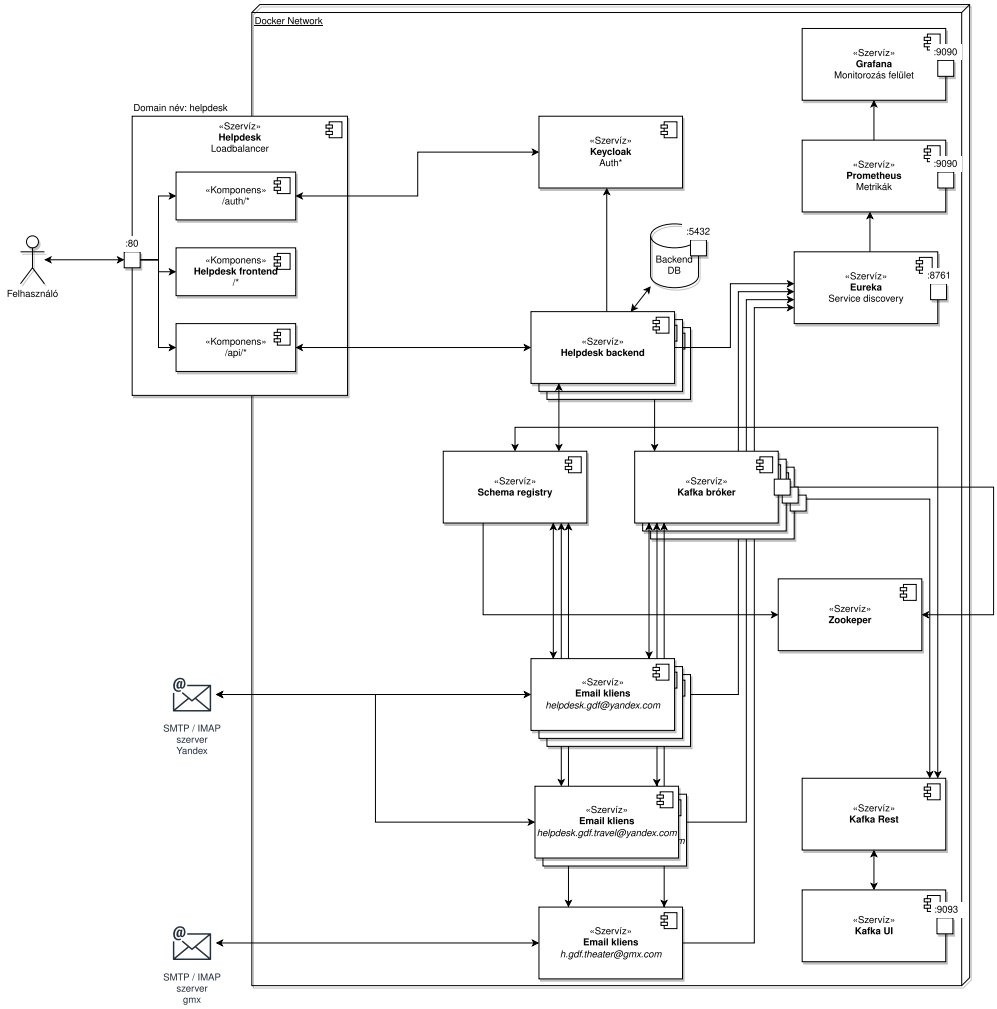
A *docker-compose* (5.1) által elindított *container*eket az 5.1. ábrán foglaltam össze. Az ábrán feltüntettem, hogy az adott *container*t a *localhost* melyik portján lehet elérni.

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

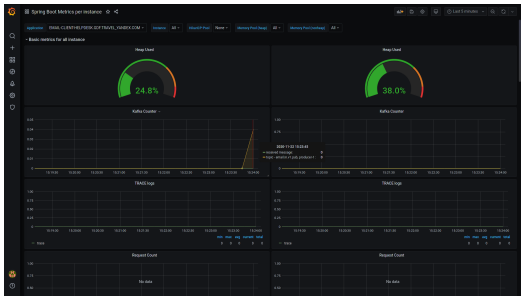
5. A felhasználói felületen (5.2g. ábra) elérhető az új üzenet.

E-mail küldése:

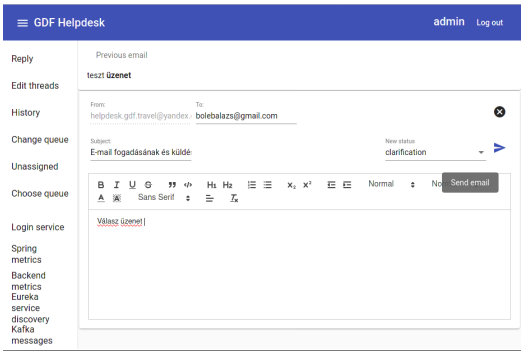
1. A felhasználó elküldi a válaszát a felhasználói felületen (5.2b. ábra).
2. A backend megkapja az üzenetet és eltárolja az adatbázisba (5.2d. ábra).
3. A *h.gdf.theater_ gmx.com.v1.pub* topicban megjelenik (5.2f. ábra) a backend által publikált kafka üzenet
4. A topicra feliratkozott e-mail kliens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

5.5. Adatbázis táblák

A helpdesk backend



(a) Az egyes instance kafka üzenet küld

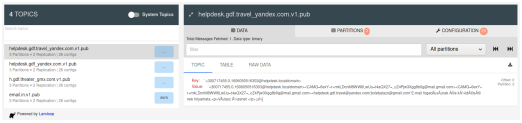


(b) A felületen válasz e-mailt küld a felhasználó

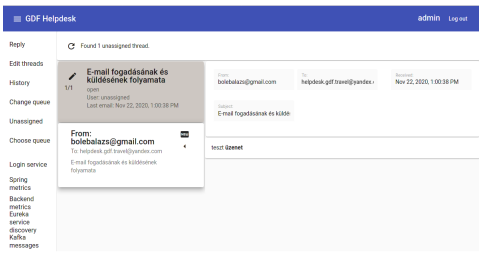
(c) Az egyes instance kafka üzenet fogad

(d) A válasz e-mail az adatbázisban

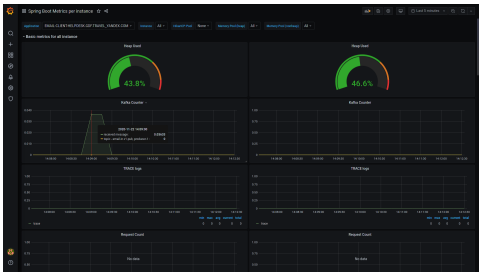
(e) Az új e-mail az adatbázisban



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

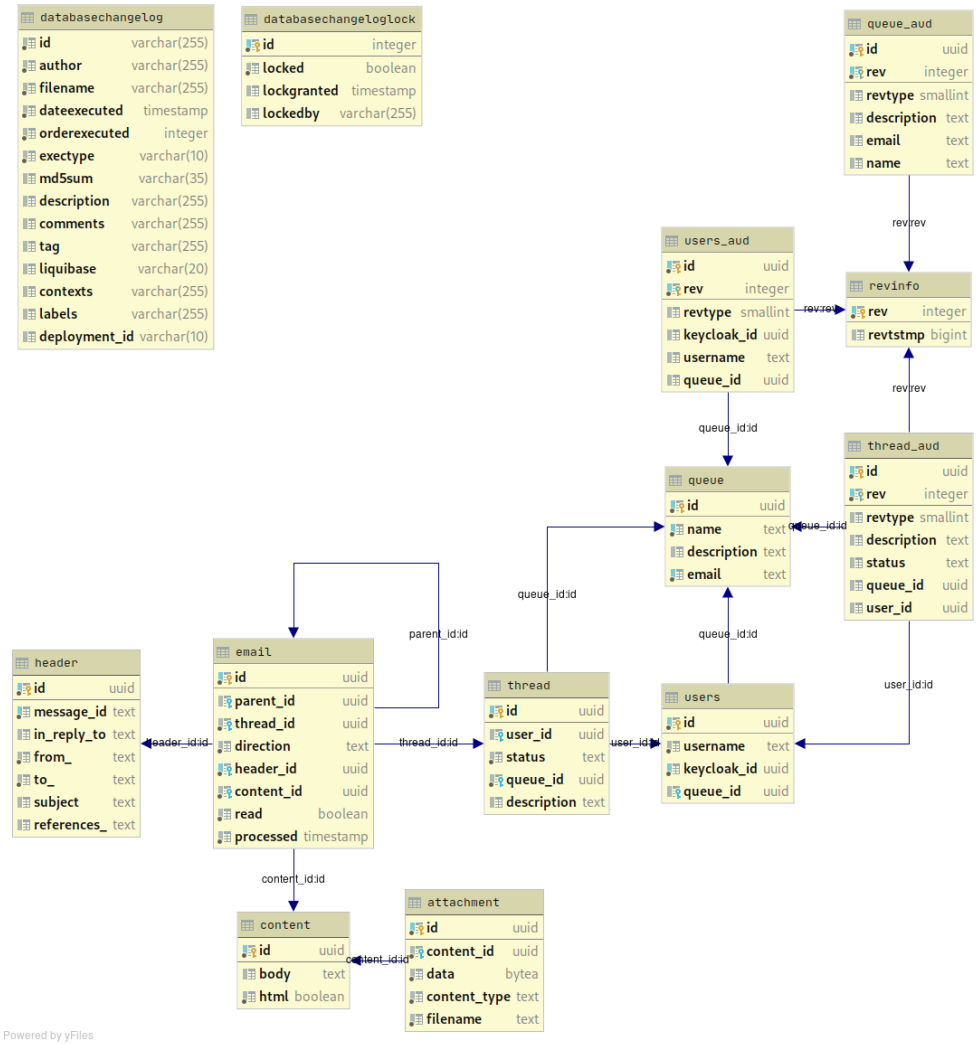


(g) A felületen elérhető az új e-mail



(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

5.6. Apache Kafka

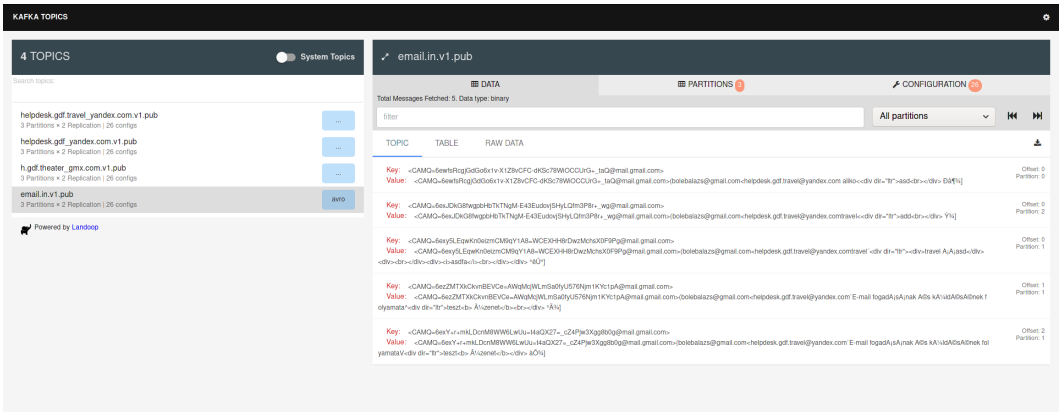
A kafka *topic*ok és üzenetek elérhetőek és követhetőek a *Kafka messages* menü pontja alatti Kafka Topics UI (5.4. ábra) eszközzel.

A helpdesk alakalmazás összesen öt *topic*-ot használ:

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.



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

Forrás: saját ábra

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 particióval és kettes replikációs faktorral lett létrehozva. Így a kafka cluster egy bróker kiesése, vagy egy partíció sérülése esetén is működőképes marad.

5.7. frontend

egy-egy érdekesebb problémásabb felület képenyőképek? valami how to dokumentáció gondolok itt az urlben állapottárolásra meg a user keresés leütöm vallback meg

5.8. eureka

6. fejezet

Terheléses tesztelés

6.1. Load test

Jmeter load test

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 költséghatékonnyá 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 *cluster*re 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 *cluster*esítése 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 fontendet, 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.

8. fejezet

Tapasztalatok

Szeretném összefoglalni a szakdolgozat készítése során nyert tapasztalataim.

meg miket tanultam: normalizálás adatbázis spring angular dependency injection multimodul maven kafka működése nem funkcionális technológiák, mint git, latex, diagrammok

kutatómunka a tervezés hogy mwnnyi minden

a problémát analizálni és megfeelő megoldást keresni rá

a végé pedig hogy ez mennyire csodás érzés hogy minden úgy működik ahogy kitaláltam.

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, az ügyfél igényeit 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 dokumnetáció

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** *required*	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** *optional*	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** *optional*	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** *optional*	Pagination page	integer (int32)	`0`
Query	**size** *optional*	Pagination size	integer (int32)	`1`

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** *required*	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** *required*	Id of the emailThread	string (uuid)
Body	**body** *required*	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** *required*	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** *required*	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** *required*		string (uuid)
Body	**body** *required*	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** *required*	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** *required*	Queue id	string (uuid)
Query	**username** *optional*	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 *required*	The attachment is part of this email content.	Content

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

ByteArrayOutputStream

Type : object

Content

Full DTO for content

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

Email

Full DTO for email

Name	Description	Schema
content *required*	The content of the email.	Content
direction *optional*	The direction of the email.	enum (IN, OUT)

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

EmailThread

Full DTO for thread

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

EmailThreadAudit

Full DTO for thread audit

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

Header

Full DTO for header

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

Name	Description	Schema
subject *optional*	The subject of the mail.	string
to *required*	The email sent to this address.	string

Queue

Full DTO for queue

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

User

Full DTO for users

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