Gábor Dénes Főiskola

Mérnökinformatikus alapképzés

Helpdesk rendszer megvalósítása mikroszerviz alapú elosztott alkalmazással

Bőle Balázs

Konzulens:

Dr. Nagy Elemér Károly

Szoftverfejlesztés szakirány

2020 november

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

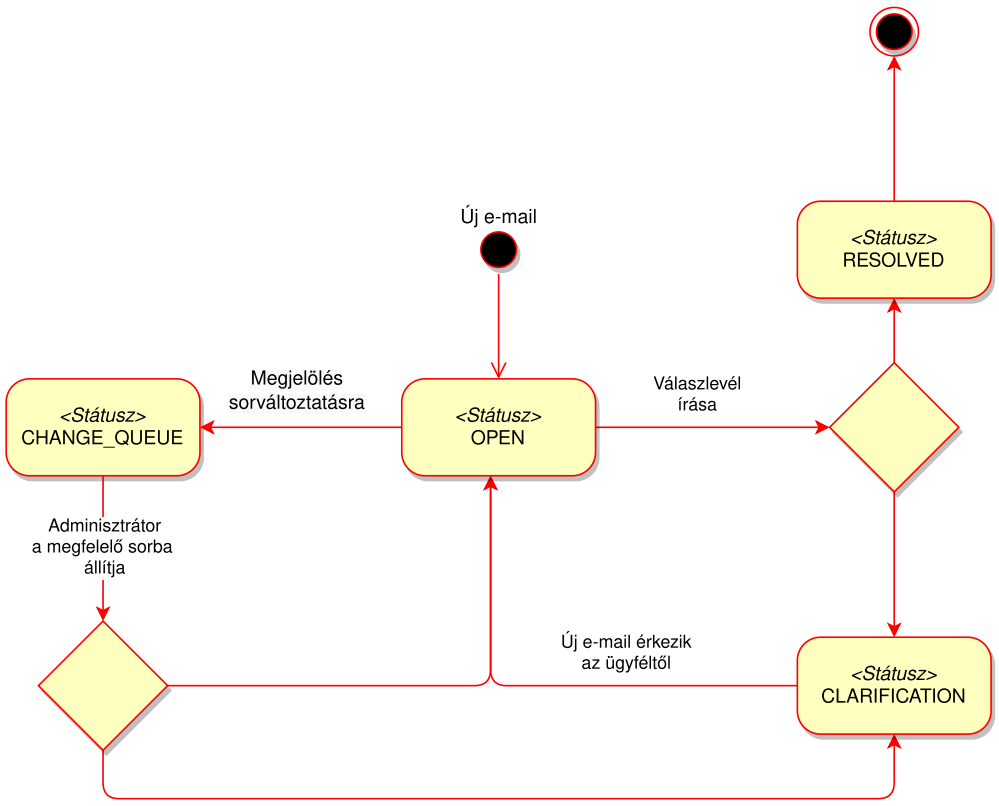
A rendszer által kezelt üzenetek szálakba rendezve érhetőek el. Egy szál az ügyfél és a felhasználó közötti üzenetváltásokból épül fel.

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.

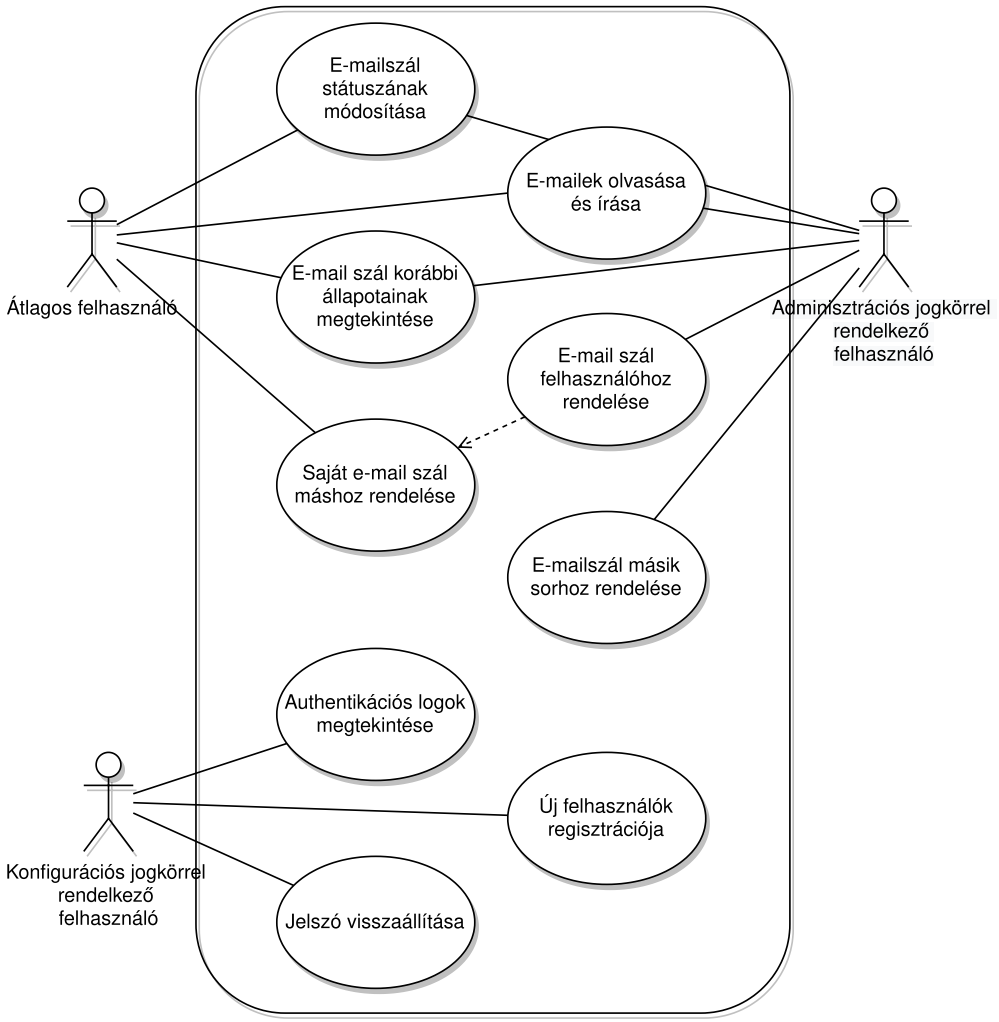
A konfigurációs jogkörrel rendelkező felhasználó feladata más felhasználók regisztrálása, valamint az alakalmazásban használt jogkörök (*role*-ok) kezelése. Lehetősége van továbbá authentikációs logok megtekintésére, jelszó visszaállítására és más felhasználók megszemélyesítésére (*impersonate*).

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 (2.5), a backend és az e-mail kliens Spring Boot-ot (2.6) használ. A alkalmazáson belüli események kezelésére és tárolására Apache Kafkát (2.4) 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
- 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

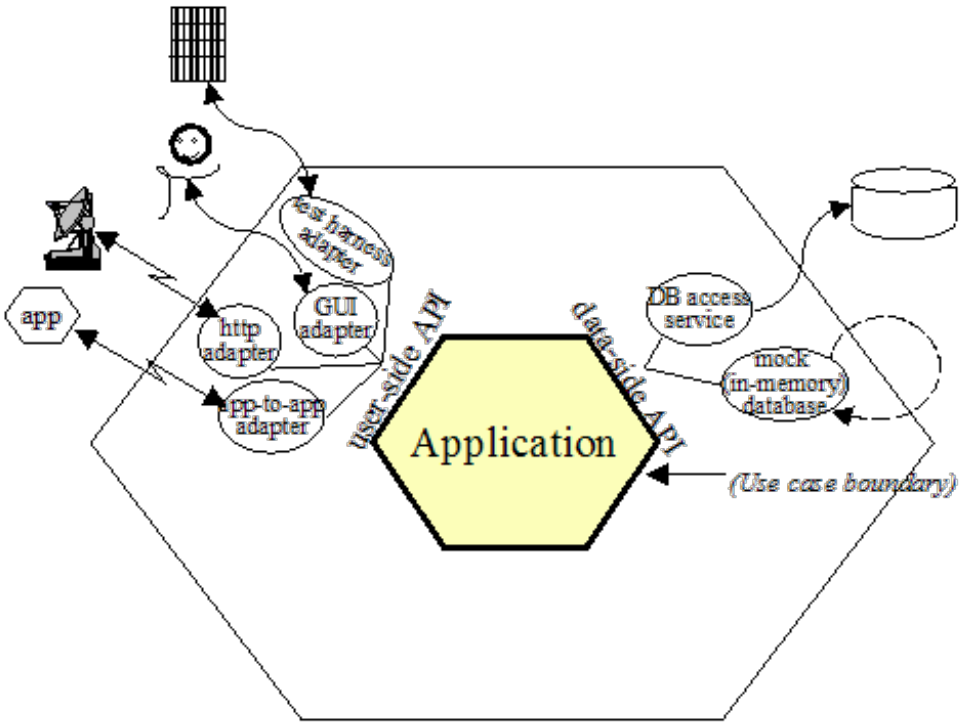
A hexagonális architektúra –vagy más néven portok és adapterek architektúrája– egy Alistair Cockburn által létrehozott [4] szoftverterezési minta. Nevét a cikkben felrajzolt hatszögletű rendszerábrázolásról kapta (2.1 ábra), ami szembemegy a korábban elterjedt réteges elrendezéssel.

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* [8] 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. Apache Kafka

Az apache kafka egy üzenet tárolásra és továbbításra kifejlesztett hibatűrő, magas áteresztő képességű, open source alkalmazás [6].

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 partícióra van elosztva, és ezen felül minden partició replikálva is van [7]. 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.

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.

2.5. Angular

Az Angular egy a Google által fejlesztett TypeScript alapú platform és keretrendszer [9]. 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.

2.6. Spring Boot

A Spring Boot egy a Springre épülő keretrendszer. Mindkét rendszer alapja a függőség befecskendezése[5], ami egy a 2.3 pontban említett tiszta kód [8] eszköze.

A Spring Boot [10] célja hogy gyorsan és egyszerűen lehessen önálló, magas minőségű alkalmazásokat fejleszteni:

- az alapbeállítástól való eltérést kell meghatározni[6] ezzel lecsökkentve a konfigurációval töltött időt,
- valamint sok gyakran visszatérő problémára[7] nyújt könnyen elérhető megoldást.

[5] Angolul *Dependency Injection*

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

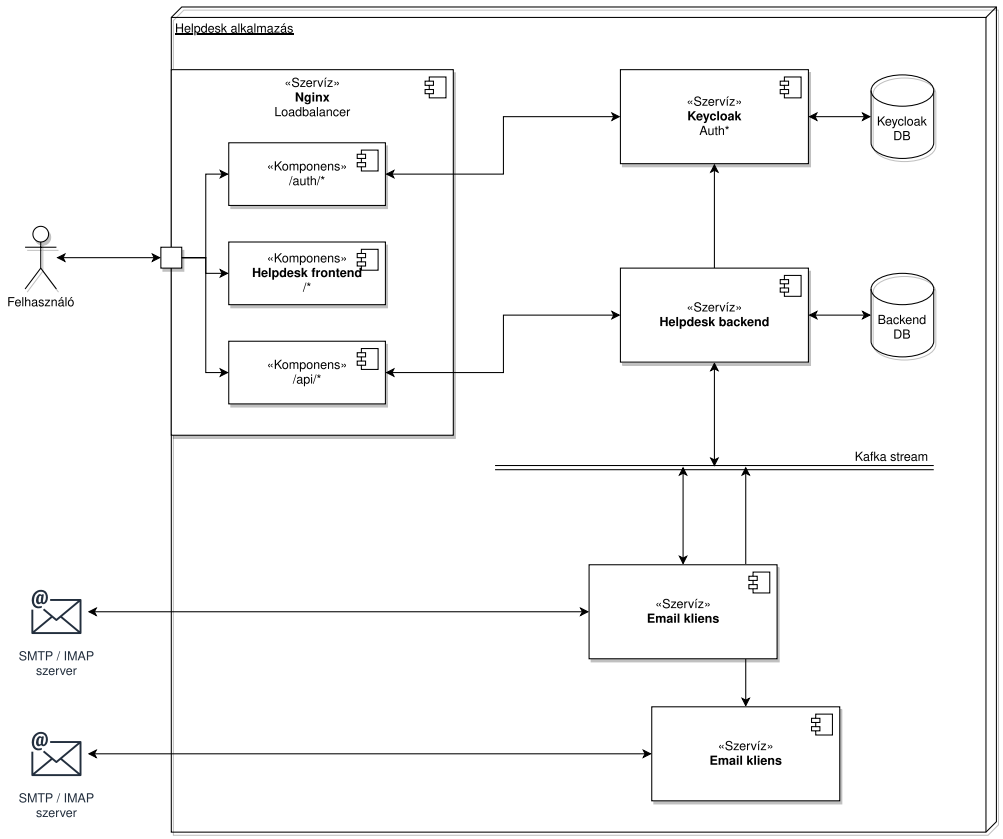
[7] Például: metrikák, biztonság, adattárolás

3. fejezet

Az alkalmazás felépítése

Ebben a fejezetben szeretnék egy átfogó képet adni a helpdesk alkalmazásról.

3.1. Component Diagram



3.1. ábra. A legfontosabb komponensek

Forrás: saját ábra

3.2. Sequence UML Diagram

rajta legyen az összes kommunikáció majdnem mind, le kellene írni hogy hogyan megy a kommunikáció

3.3. Adatbázis UML ábra

audittal együtt lehet

3.4. Átfogó áttekintés az implementálás bemutatása előtt

4. fejezet

Implementáció

Ez a miért és hogy csináltam rész

4.1. Mikroszerviz infrastruktúra

4.1.1. Nginx

Az nginxnek három elkülönülő szerepe van:

- A helpdesk frontend alkalmazásszervereként működik (lásd 2.5 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 [11] meghatározott mezőkre:

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

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

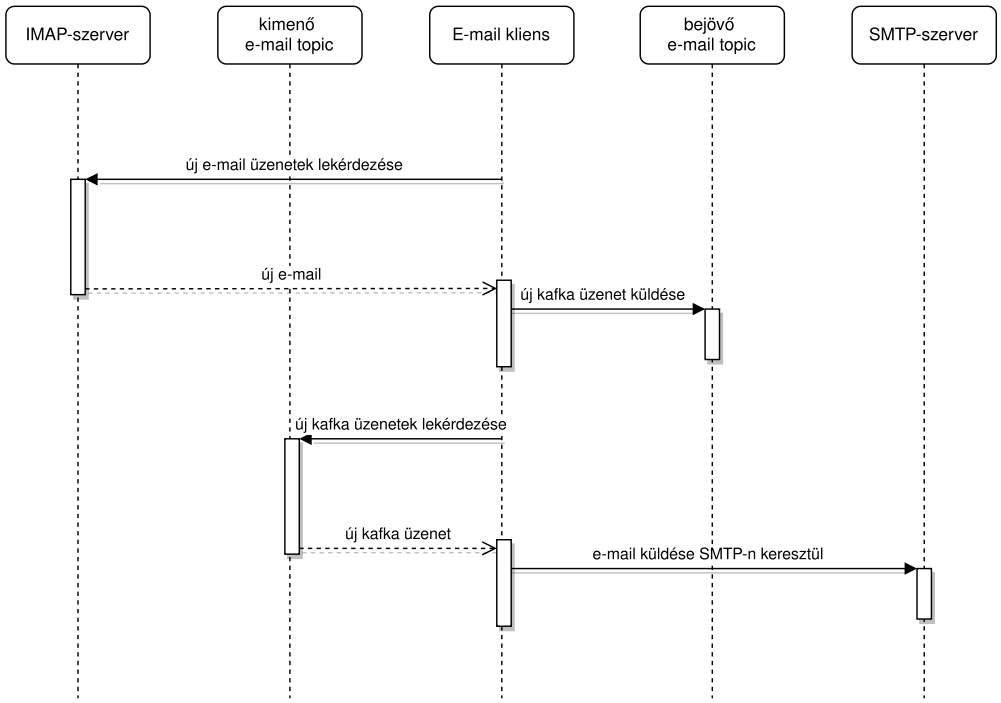
References azonosítja az üzenet szálat, értéke az eredeti üzenetek *Message-ID*-jai vesszővel elválasztva.

[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

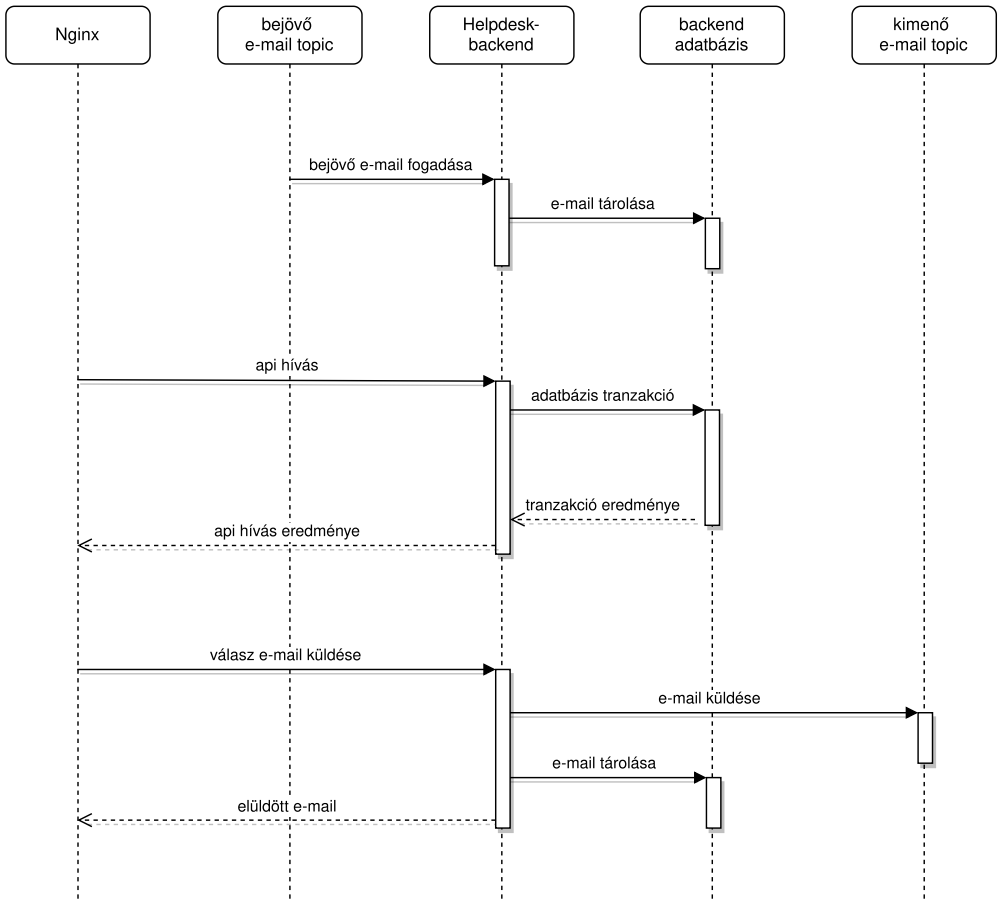
4.3. Helpdesk backend

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 (2.6 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.

[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 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 [12].

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 különböztetek 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.

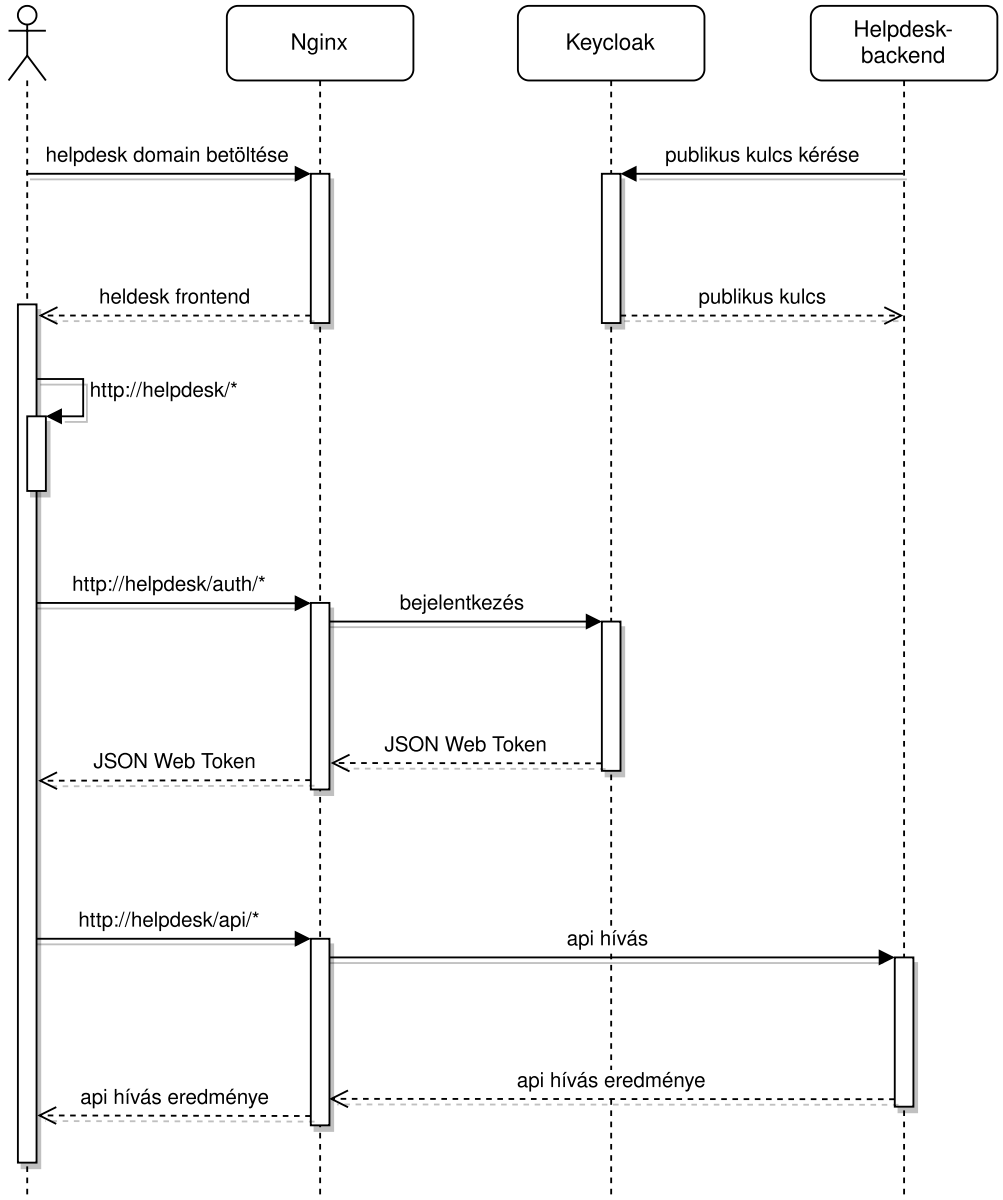
4.5.2. JSON Web Token

A jogosultságkezelés technikai alapját az *rfc7519*-es szabványban [13] 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 (2.4 pont) mennek keresztül. A szeparációval függetlenné teszem egymástól a két rendszer működését, ami lehetővé teszi az eltérő igénybevételnek (1.2.2 pont) megfelelő skálázhatóságot.



4.3. ábra. Helpdesk frontend szekvencia diagramja

Forrás: saját ábra

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 szintén három darab.

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

Az todo ábrán csak a Grafana oldal legfelső néhány panele látható, az instance-okra lebontott legfontosabb mérőszámok:

- a legfelső sorban a Java Virtual Machine, által akutálisan felhasznált Heap space,
- alatta az aktuális REST lekérések száma,
- a harmadik sorban a feldolgozott Kafka üzenetk száma,
- míg az utolsó sorban a Trace logüzenetek 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. Load test

Jmeter load test

5.5. Egy e-mail útja

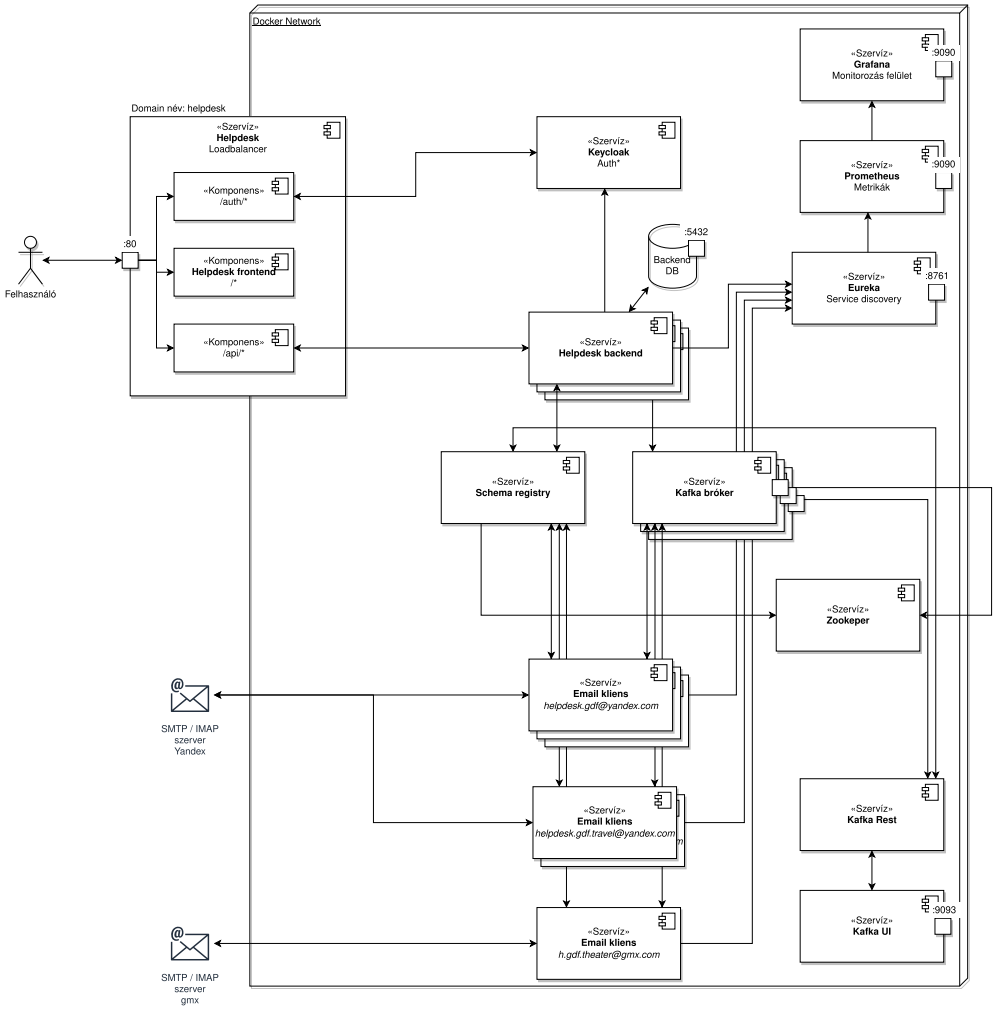
message flow diagram vagy valami hasonló a lényeg hogy vertikális legyen

5.6. Felhasználói felület funkciói

képenyőképek? valami how to dokumentáció

5.7. Kafka topicok?

a replication factorról meg a három brókerről topicok meg egyebek



5.1. ábra. Deployment diagram

Forrás: saját ábra

6. fejezet

Továbbfejlesztési lehetőségek

docker swarm kubernetes

Összefoglalás

összefoglalás a végén

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] 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.

[7] 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.

[8] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*, chapter 11, Systems. Prentice Hall, 1 edition, 2008.

[9] Introduction to angular concepts. Letöltve: 2020. Nov. 16. URL: https://angular.io/guide/architecture.

[10] Introducing spring boot. Letöltve: 2020. Nov. 16. URL: https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started-introducing-spring-boot.

[11] Identification fields. Letöltve: 2020. Nov. 16. URL: https://tools.ietf.org/html/rfc5322#section-3.6.4.

[12] Keycloak. Letöltve: 2020. Nov. 18. URL: https://www.keycloak.org/.

[13] Json web token (jwt). Letöltve: 2020. Nov. 18. URL: https://tools.ietf.org/html/rfc7519.