MISKOLCI EGYETEM
Gépészmérnöki és Informatikai Kar
Általános Informatikai Tanszék

Oktatást segítő alkalmazás kibővítése felhasználók közötti csevegéssel és részletes kereséssel

DIPLOMAMUNKA

Vécsi Ádám
IZBTF9

Miskolc, 2020

Eredetiségi nyilatkozat

Alulírott *Vécsi Ádám.* (neptun kód: IZBTF9)
a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy a

Oktatást segítő alkalmazás kibővítése felhasználók közötti csevegéssel és részletes kereséssel

című diplomamunkám saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályi szerint történt.

Tudomásul veszem, hogy plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén a szakdolgozat visszavonásra kerül.

Miskolc, 2020.10.20.

Hallgató aláírása

Tartalomjegyzék

1. Bevezetés

A mai világban az egyik legjobban használt kommunikációs forma az interneten való csevegés. Használjuk munkahelyen és otthon is, akár számítógépen, mobiltelefonon, tábla gépen vagy okos órán. Talán a legismertebb alkalmazások a Skype, Slack, WhatsApp, Viber, de a legtöbb közösségi média platform lehetővé teszi számunkra ezt a kommunikációs eszközt. Az Instagram, Facebook Messenger, Snapchat, LinkedIn is ad ilyen lehetőséget így a barátainkkal, ismerőseinkkel tudunk csevegni. Nem csak magán jellegű beszélgetésekre alkalmas ez, sokan használjuk munkahelyen is információ csere céljából.

Legnagyobb előnye, hogy valós időben történik az üzenetváltás bizonyos esetekben akár kép és videó megosztás, fájl csere is. Ami fontos még a chat alkalmazásokban, hogy az üzenetek naplózva lesznek, így korábbi beszélgetéseket is vissza lehet olvasni és nem vész el az információ. Ezen kívül általában keresni is lehet a beszélgetésekben.

Egy Egyetemi óra keretein belül diákok dolgoznak egy olyan oktatást segítő alkalmazáson, ami megkönnyíti a diákok órai munkáját. Ehhez gondoltam, hogy készítek egy olyan alkalmazás modult, amivel valós időben tudnak csevegni a diákok egymással és tanáraikkal. Terveim szerint tantárgyanként lehetne létrehozni szobákat, a diákok és tanárok oda csatlakozhatnak be és cseveghetnek. A fejlesztés párhuzamosan zajlik a két alkalmazás között és mind a kettő tud teljes értékűen működni a másik nélkül.

Két éve dolgozok a Shiwaforce.com Zrt-nél, közép és nagyvállalatoknak fejlesztünk webalkalmazásokat. Olyan cégeknek fejlesztünk, mint például a Budapest Bank, OTP Bank, Telekom, Telenor.

Java backend fejlesztőként már vettem részt olyan projekten, ahol készítettünk hasonló chat alkalmazást, amit egy magyar nagy vállalat azóta is használ ügyfelekkel való kommunikációra. Úgy gondoltam, hogy egy oktatást segítő alkalmazáshoz nagyon hasznos lenne egy ilyen funkció, akár tanórán való feladat elakadásakor, otthoni beadandó írása esetén.

Dolgozatomban egy chat alkalmazás működését szeretném bemutatni, hogyan jutnak el az üzenetek a küldőtől a célig, hogyan történik az üzenetek naplózása, valamint az üzenetekben való keresés.

Szeretnék készíteni egy chat alkalmazást, egy oktatást segítő alkalmazás kiegészítéseként, melyben pontosan az előbb említett funkciókat valósítanám meg. Backend oldalon Java, Spring boot, Apache Kafka és Elasticsearch segítségével. A frontend-nek nagyon egyszerű megoldást gondoltam, HTML-t, JavaScript-et szeretnék használni JQuery segítségével. Használt kommunikációs formák a Websocket és REST. Úgy gondolom, hogy egy kicsit a konténerizációt is használnám az alkalmazás függőségeihez. Így a Dockert is bemutatnám a következőkben.

Azért ezekre a technológiákra gondoltam, mert mint szoftverfejlesztő úgy látom elfogadott és bevált módszerek, és a mai szoftverfejlesztési trendeknek is megfelelnek. Segítségükkel biztonságos, hibatűrő alkalmazásokat lehet készíteni, adatvesztés minimalizálással.

A következő fejezetekben ezen technológiákat szeretném bemutatni részletesen, majd az általam implementált Chat alkalmazást.

2. Chat alkalmazáshoz használt technológiák bemutatása

Bevezetésben említettem, hogy Java webalkalmazást szeretnék készíteni, Spring Boot segítségével. Az üzenetek küldéséért és fogadásáért az Apache Kafka streaming platform felel. A kereséshez Elasticsearch-ben szeretném tárolni az üzeneteket. A szobákat, felhasználókat egy relációs adatbázisban tárolnám. REST-en keresztül lehet felhasználókat, szobákat létrehozni, lekérdezni, módosítani és törölni. Ahhoz, hogy valós időben jusson el az üzenet a felhasználótól a szobában websocket-et fogok használni. A Chat alkalmazáshoz szükséges függőségeket Dockerben szeretném konténerekben futtatni, itt a Apache Kafka brókerre és a hozzá tartozó ZooKeeper-re, az Elasticsearch-re és a hozzá tartozó Kibanára gondolok. Ebben a fejezetben sorban szeretnék ezen technológiák működéséről írni.

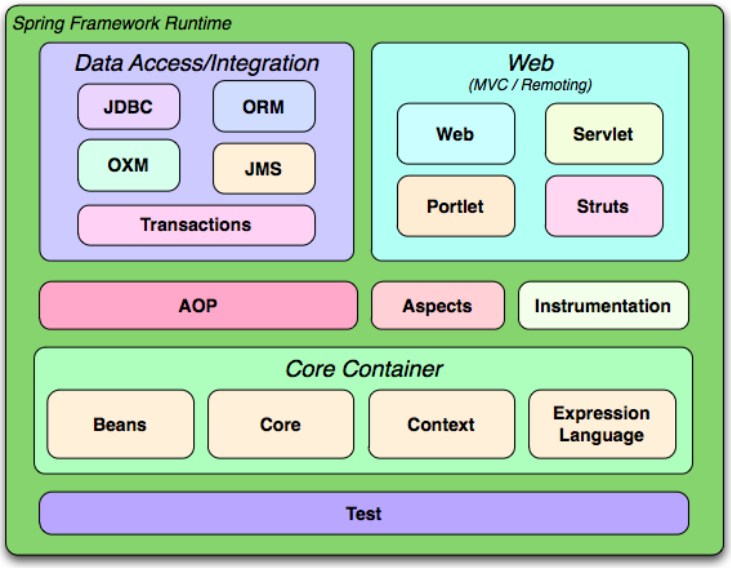
2.1. Spring Boot keretrendszer

Ahhoz, hogy a Spring Boot-ot megértsük, hogy miért is nagyon elterjedt és könnyen használható szerver oldali java alkalmazások készítésére, először a Spring-et kell megvizsgálnunk.

2.1.1. Spring bemutatása

A Spring egy nyílt forráskódú keretrendszer. Széles körben használják skálázható alkalmazások létrehozására. A Spring rendszer körülbelül 20 modulba rendezett szolgáltatást tartalmaz. A leggyakrabban használt modul a Spring MVC, de a Data Access/Integration modul is nagyon hasznos, ha az alkalmazás valamilyen adatbázishoz csatlakozik. A *2.1*. ábrán láthatók a Spring modulok.

A Spring projektek fő hátránya, hogy a konfiguráció valóban időigényes és bonyolult a kezdő vagy új fejlesztők számára. Az alkalmazás fejlesztése sok időt vesz igénybe, ha még a fejlesztő nem ismeri jól a Spring-et. Ennek megoldása a Spring Boot [1].



2.1. ábra Spring keretrendszer

https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/overview.html

2.1.2. Spring Boot bemutatása

Spring boot egy Spring-re épülő keretrendszer és tartalmazza összes tulajdonságát. Manapság a backend fejlesztők egyik kedvenc keretrendszere mert, gyors fejlesztésre kész környezet biztosít, amely lehetővé teszi a fejlesztők számára, hogy közvetlenül a logikára koncentráljanak ahelyett, hogy a konfigurációval és a beállításokkal küszködnének. Annotációk segítségével könnyíti meg a konfigurációt. A Spring Boot alkalmazásokat nem szükséges .war fájlként deploy-olni, .jar-ként is lehet. Önállóan futó alkalmazás hozható létre vele és rendelkezik beépített Tomcat-tel is. Nem igényel egyáltalán XML konfigurációt, így a testreszabása és kezelése is könnyebb, mint a Spring-nek [2].

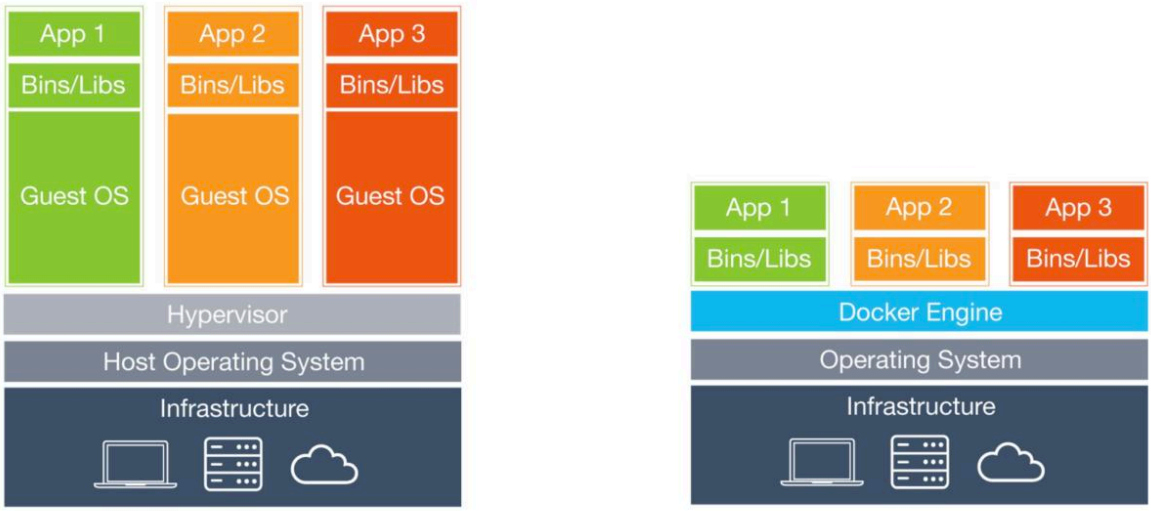
2.2. Docker

A Docker egy szoftverplatform, konténereken alapuló alkalmazások felépítéséhez. A konténerek olyan kicsi és könnyű végrehajtási környezetek, amelyek közösen használják az operációs rendszer kerneljét, de egyébként egymástól elkülönítve futnak.

Míg a konténer, mint fogalom már egy ideje létezik, a Docker egy 2013-ban elindított nyílt forráskódú projekt, elősegítette a technológia népszerűsítését és elősegítette a konténerezés és a mikro szolgáltatások irányába mutató tendenciát a szoftverfejlesztésben.

2.2.1. Docker konténer

A modern szoftverfejlesztés egyik célja az, hogy az ugyanazon host gépen vagy cluseter-ben lévő alkalmazásokat egymástól elkülönítve tartsák, így nem zavarják indokolatlanul egymás működését vagy karbantartását. Ez nehéz lehet a futtatásukhoz szükséges csomagoknak, könyvtáraknak és egyéb szoftverelemeknek köszönhetően. Az egyik megoldást erre a problémára a virtuális gépek jelentették, amelyek teljesen elkülönítik az ugyanazon hardveren lévő alkalmazásokat, és minimálisra csökkentik a szoftverkomponensek közötti konfliktusokat és a hardver erőforrásokért folytatott versenyt. De a virtuális gépek terjedelmesek, mindegyikhez saját operációs rendszer szükséges, így általában gigabájt méretű is, és nehéz fenntartani és frissíteni.



2.2. ábra VM vs. Docker

https://www.upguard.com/blog/docker-vs-vmware-how-do-they-stack-up

A konténerek ezzel szemben elkülönítik az alkalmazások végrehajtási környezeteit, de megosztják az alapul szolgáló operációs rendszert. Általában megabájtban mérik őket, sokkal kevesebb erőforrást használnak, mint a virtuális gépek, és szinte azonnal elindulnak. Sokkal sűrűbben csomagolhatók ugyanarra a hardverre, és tömegesen indíthatók, leállíthatók kevesebb erőforrást használva. A konténerek rendkívül hatékony és részletgazdag mechanizmust kínálnak a szoftverkomponensek egyesítéséhez [3].

2.2.2. Dockerfile fájl

Minden Docker konténer egy Dockerfile fájlal kezdődik. A Dockerfile egy könnyen érthető szintaxissal írt szövegfájl, amely tartalmazza a Docker image elkészítésének utasításait. A Dockerfile meghatározza a konténer alapjául szolgáló operációs rendszert, valamint a szükséges nyelveket, környezeti változókat, fájlok helyét, hálózati port-jait és egyéb összetevőit és természetesen azt is, hogy a konténer valójában mit fog csinálni, ha futtatjuk. Dockerfile példa:

```
FROM openjdk:11
COPY war/chat-1.0.0-SNAPSHOT.war /usr/app/
WORKDIR /usr/app
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "chat-1.0.0-SNAPSHOT.war"]
```

2.2.3. Docker image

Miután elkészül a Dockerfile, a Docker build segédprogram meghívásával elkészül egy Docker image. A Docker image egy csak olvasható sablon, amely utasításokat tartalmaz a Docker konténer létrehozásához. Gyakran egy image egy másik image-en alapul, további testre szabással.

Létrehozhatunk saját image-eket, vagy mások által létrehozott és a rendszerleíró adatbázisban közzétett image-ket is használhatjuk. Saját image készítéséhez egy egyszerű szintaxissal létre kell hozni egy Dockerfile-t, amely meghatározza az image létrehozásához és futtatásához szükséges lépéseket. A Dockerfile minden egyes utasítása létrehoz egy réteget az image-ben. A Dockerfile megváltoztatásakor és az image újra build-elésekor csak azok a rétegek épülnek újra, amik megváltoztak. Ez az, ami az image-ket olyan könnyűvé, kicsivé és gyorssá teszi, összehasonlítva más virtualizációs technológiákkal [3].

2.2.4. Docker compose

A Docker compose egy eszköz a több konténeres Docker alkalmazások meghatározásához és futtatásához. A Docker compose YAML fájlt használ az alkalmazás szolgáltatásainak konfigurálásához. Ezután egyetlen paranccsal létrehozza és elindítja az összes szolgáltatást a konfigurációjából [4].

A Docker compose használata alapvetően három lépésből áll:

1. Dockerfile segítségével határozza meg alkalmazásának környezetét.
2. A docker-compose.yml fájlban definiáljuk az alkalmazásokat és a hozzájuk tartozó konfigurációkat, hogy azok elszigetelt környezetben futtathatók legyenek.
3. docker-compose up parancs segítségével indítjuk és futtatjuk az összes alkalmazást.

Docker compose példa:

```yaml
version: '3'
services:
  zookeeper:
    image: wurstmeister/zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka
    container_name: kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

2.3. Liquibase

Fentebb nem említettem a Liquibase-et, de mivel az alkalmazásban használni szeretném ezt a technológiát adatbázis táblák létrehozásához és alapadatokkal való feltöltéséhez, úgy gondoltam, hogy ebben a fejezetben elméleti szinten írok róla.

A Liquibase egy nyílt forráskódú megoldás az adatbázis-séma szkriptjeinek verzióinak kezelésére. Különböző típusú adatbázisokban működik, és különféle fájlformátumokat

támogat az adatbázis struktúra meghatározásához. A Liquibase legjobb tulajdonsága, hogy képes a változásokat követni.

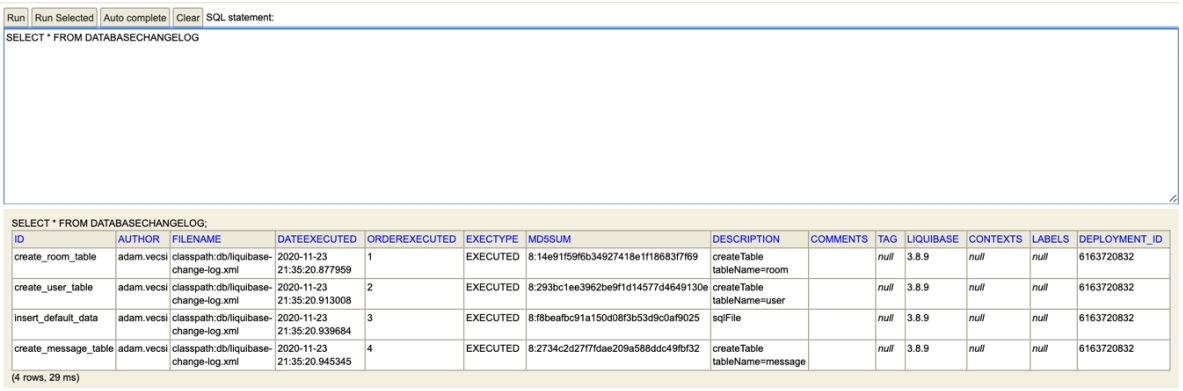
A Liquibase szkripteket használ, amiket changeSet-nek nevezünk és changeLog fájlban definiáljuk ezeket. Ezt a fájlt használja az adatbázison végrehajtott változtatások kezelésére. A changeLog fájlok különböző formátumok lehetnek, XML, JSON, YAML és SQL fájlokat is támogat. ChangeSet-ben tudunk létrehozni, törölni, módosítani táblákat és rekordokat, de ezen kívül minden olyan műveletre képes, amit SQL-lel megoldhatunk. Sőt akár natív SQL-t is használhatunk benne.

Itt látható egy példa táblalétrehozásra.

```
<changeSet author="adam.vecsi" id="create_user_table">
        <createTable tableName="user">
                <column autoIncrement="true" name="user_id"
type="bigint">
                        <constraints primaryKey="true"
primaryKeyName="user_pkey"/>
                </column>
                <column name="neptun" type="varchar(255)"/>
                <column name="name" type="varchar(255)"/>
                <column name="email" type="varchar(255)"/>
                <column name="role" type="varchar(255)"/>
        </createTable>
</changeSet>
```

A Liquibase megvizsgálja az adatbázis aktuális állapotát és azonosítja, hogy mely változások történtek már meg. Ehhez a vizsgálathoz egy databasechangelog táblát használja, itt tárolja az már lefutott szkripteket. Ez után lefuttatja a többi változtatást azokon a táblákon, amik a changelog-ban voltak, és visszamenti a changeSet-ek azonosítóját a databasechangelog táblába. Így tudja mindig az aktuális állapotokat.

A databasechangelog táblán kívül a Liquibase létrehoz még egy táblát, amit databasechangeloglock-nak hív, ez biztosítja, hogy egyszerre csak egy Liquibase példány fusson. Ez azért fontos mert így egyszerre két példány nem módosíthatja Liquibase változásokat [5].


Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM DATABASECHANGELOG
```

SELECT * FROM DATABASECHANGELOG;

ID	AUTHOR	FILENAME	DATEEXECUTED	ORDEREXECUTED	EXECTYPE	MD5SUM	DESCRIPTION	COMMENTS	TAG	LIQUIBASE	CONTEXTS	LABELS	DEPLOYMENT_ID
create_room_table	adam.vecsi	classpath:db/liquibase-change-log.xml	2020-11-23 21:35:20.877959	1	EXECUTED	8:14e91f59f6b34927418e1f18683f7f69	createTable tableName=room		null	3.8.9	null	null	6163720832
create_user_table	adam.vecsi	classpath:db/liquibase-change-log.xml	2020-11-23 21:35:20.913008	2	EXECUTED	8:293bc1ee3962be9f1d14577d4649130e	createTable tableName=user		null	3.8.9	null	null	6163720832
insert_default_data	adam.vecsi	classpath:db/liquibase-change-log.xml	2020-11-23 21:35:20.939684	3	EXECUTED	8:f8beafbc91a150d08f3b53d9c0af9025	sqlFile		null	3.8.9	null	null	6163720832
create_message_table	adam.vecsi	classpath:db/liquibase-change-log.xml	2020-11-23 21:35:20.945345	4	EXECUTED	8:2734c2d27f7fdae209a588ddc49fbf32	createTable tableName=message		null	3.8.9	null	null	6163720832

(4 rows, 29 ms)

2.3. ábra Liquibase databasechangelog tábla

2.4. Apache Kafka elosztott streaming platform

A Kafka elosztott rendszer szerverekből és kliensekből áll, amelyek nagy teljesítményű TCP hálózati protokollon keresztül kommunikálnak.

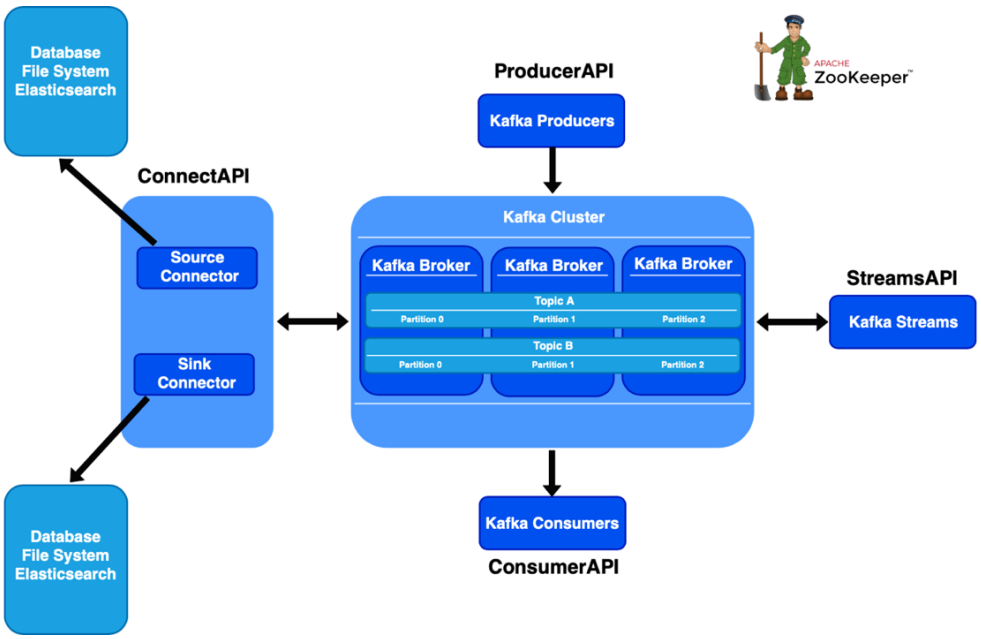
Az Apache Kafka Scala és Java nyelven íródott, és a korábbi LinkedIn adatmérnökök alkotása. Már 2011-ben a technológiát erősen skálázható üzenetküldő rendszerként adták át, amely nyílt forráskódú.

A mai összetett rendszerekben szereplő adatokat és naplókat feldolgozni, újra feldolgozni, elemezni és kezelni kell, gyakran valós időben. Az Apache Kafka jelentős szerepet játszik az üzenet streaming környezetében. A Kafka kulcsfontosságú tervezési alapelveit az egyre növekvő igény alapján alakítják ki, a nagy teljesítményű architektúrák, amelyek könnyen skálázhatóak, és lehetővé teszik az adatok tárolását, feldolgozását és újrafeldolgozását.

2.4.1. Apache Kafka Architektúra

Egy Kafka architektúra legalább egy Kafka szerverből vagy másnéven Kafka brókerből áll, ami a konfigurációját kötelezően a ZooKeeper nevű elosztott konfigurációs management rendszerben tárolja. Általában cluster-ben futnak a brókerek. A ZooKeeper feladata ezek konfigurációja. A ZooKeeper folyamatosan ellenőrzi a brókerek állapotát, ha változik a konfiguráció értesíti a brókereket, vagy ha esetleg valamelyik bróker kiesik, akkor az ő hozzá irányított üzenetek másikhoz kerülnek, nyilván ez akkor fordulhat elő, ha több brókert

kötünk az alkalmazásba. A Kafka brókerhez csatlakoznak a producer-ek és consumer-ek. A Kafka brókerben úgynevezett topic-ok találhatók.



2.4. ábra Kafka architektúra

A producer mindig egy dedikált topic-ra küldi az üzeneteket, és a consumer-ek mindig egy dedikált topic-ból olvasnak, tehát a topic az a logikai egység, ami egy producer-consumer páros számára az üzeneteket tárolja és továbbítja. A producer-ek mindig egy brókerhez csatlakoznak. A ZooKeeper tudja értesíteni a klienseket, ha a konfiguráció változik, ezért hamar elterjed a hálózaton a változás [6].

Nem csak Producers API-n és Consumers API-n lehet üzeneteket beküldeni és kiszedni egy topic-ból, hanem a Connect API segítségével, külső fájlrendszerekről, adatbázisokból, Elasticsearch-ből tudunk kiszedni és betenni adatot egy-egy topic-ba. Ezek valójában third party komponensek, amik megkönnyítik a munkánkat, mivel nem nekünk kell megírni a Producer API-t.

A Kafka Connect egy keretrendszer a Kafka és külső rendszerek, például adatbázisok, kulcsérték-tárolók, keresési indexek és fájlrendszerek összekapcsolására, ezt Connectors-nak nevezzük.

A Kafka Connectors használatra kész komponensek, amelyek segíthetnek abban, hogy külső rendszerekből származó adatokat importáljunk a Kafka topic-ba, és adatokat exportáljunk a Kafka topic-ból külső rendszerekbe. Használhatjuk a meglévő connector megvalósításokat általános adatforrásokhoz, vagy megvalósíthatjuk saját connector-unkat [7].

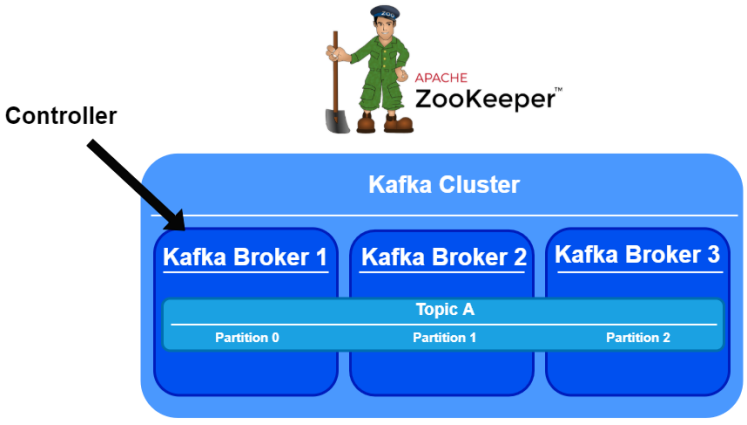
A *Source connector* összegyűjti az adatokat a rendszerből. A forrásrendszerek lehetnek teljes adatbázisok, adatfolyam-táblák vagy üzenetközvetítők. A source connector metrikákat is gyűjthet az alkalmazás szerverektől Kafka topic-ba, így az adatok alacsony késleltetéssel elérhetővé válhatnak az adatfolyamok feldolgozásához.

A *Sink connector* a Kafka topic adatait szállítja más rendszerekbe, amelyek lehetnek indexek, például Elasticsearch, kötegelt rendszerek, például Hadoop, vagy bármilyen adatbázis.

A Kafka Streams-t az Apache Kafka készítői tervezték. A szoftver elsődleges célja, hogy lehetővé tegye a programozók számára, hogy hatékony, valós idejű, streaming alkalmazásokat hozzanak létre, amelyek Microservice-ként működhetnek. A Kafka Streams lehetővé teszi, Kafka topic-ból ki szedett adatok elemzését, vagy átalakítását, és esetleg egy másik Kafka topic-ba való elküldését.

2.4.2. Kafka Broker

Egy Kafka Cluster több Kafka brókerből állhat. Fentebb említettem Zookeeper, azért felelős, hogy a cluster-ban lévő brókerek megfelelően működjenek. Az előnye a clusteres működésnek, hogy ha esetlegesen hiba fordul elő valamelyik Kafka példánnyal a ZooKeeper amint észreveszi a hibát egy másik bróker felé irányítja az üzeneteket. Ezzel megelőzve az adatvesztést. A Kafka brókerben találhatók a topic-ok, egy brókerben több topic is lehet. Több példány esetén a topic megosztás úgy néz ki, hogy az egyik bróker controller-ként viselkedik. Amikor létrejön egy új topic akkor a ZooKeeper elküldi a controller-nek és az szétosztja a topic partíciókat a brókerek között.



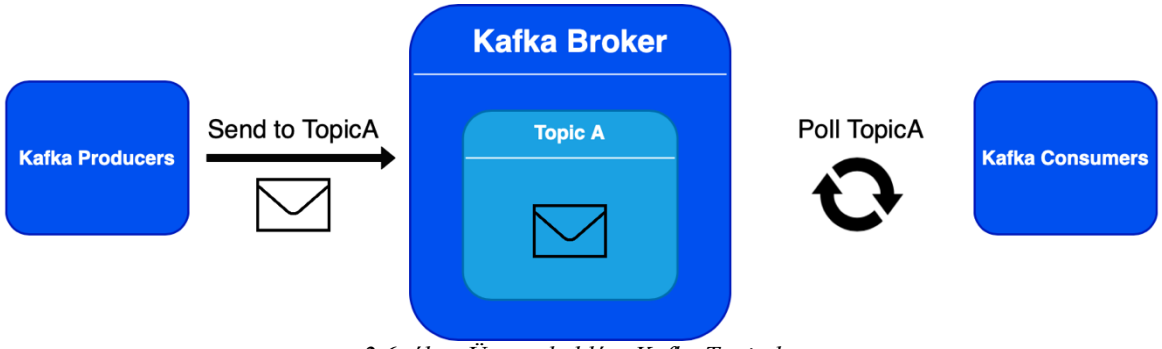
2.5. ábra Kafka-particiók kiosztása a brókerek között

A *2.5.* ábrán látható erre egy példa, ha három darab Kafka bróker van, ebben az esetben a „Kafka Broker 1” lesz a controller a három közül. Amikor küld a Kafka Producer egy üzenetet átkerül a Partitioner-hez, ami megnézi, hogy melyik partícióhoz kell annak az üzenetnek kerülni, és ahhoz a brókerhez osztja ki.

A következő szekcióban a Kafka topic-ról fogok írni.

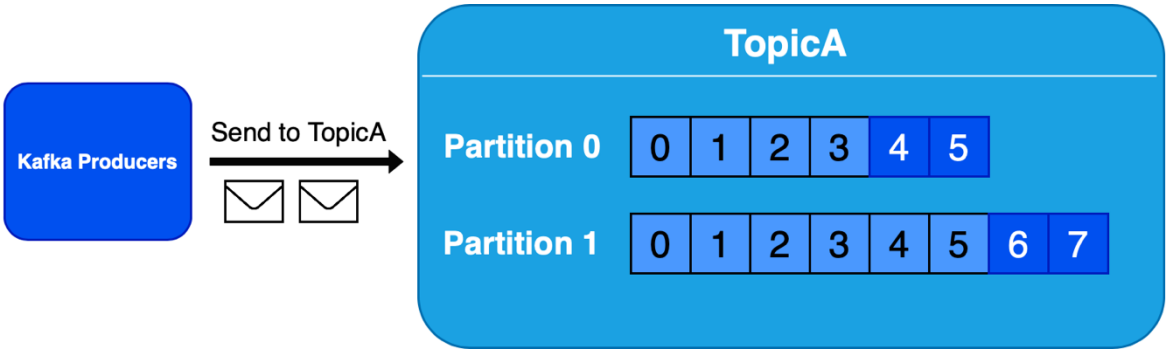
2.4.2.1. Topic és partíciók

A topic-ot úgy kell elképzelni, mint egy táblát egy adatbázisban. A működés a *2.6.* ábrán látszik igazán. A Producer küld egy üzenetet a „Topic A”-ba. Közben a Consumer folyamatosan poll-ozza (vizsgálja) a „Topic A”-t, hogy van-e újabb üzenet, ha van akkor kiolvassa a topic-ból.



2.6. ábra Üzenet küldése Kafka Topic-ba

A partíció az, ahol egy adott üzenet él a topic-ban. Minden topic-nak egy vagy több partíciója lehet, ezek a partíciók rendezett rekordok listája. A *2.7.* ábrán látszik, hogy egy üzenet az adott topic valamelyik partíciójában tárolódik. A producer tudja, hogy melyik üzenet melyik partícióban van. Ezek függetlenek egymástól és minden rekordnak van egy szekvencia száma, amit offset-nek nevezzük. Egy partíció az a logikai egység, aminek rá kell férnie egy lemezre. A topic-kot úgy kell felskálázni, hogy egyre több partíciót adunk hozzá, amik különböző brókereken fognak létrejönni. Minden partíciónak lehet egy vagy több replikája, amik biztonsági másolatok. A partíciók új üzenete mindig a partíció végére íródik.



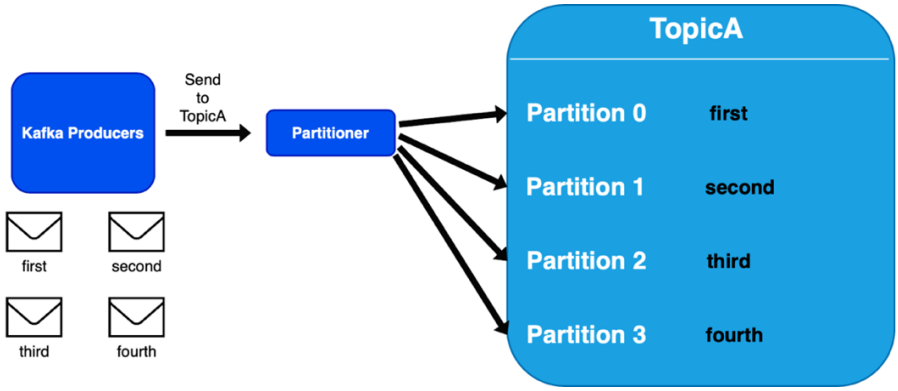
2.7. ábra Topic partíciókra bontása

Mikor a consumer kiolvas egy üzentet a partícióból, attól az még ott marad a partícióba egészen addig, amíg le nem jár az ideje, ez alapértelmezetten ez egy nap, de konfigurálható. A Kafka nyilvántartja, hogy melyik consumer egy adott partícióban melyik offset-nél tartott. Ezt egy speciális topic-ban tartja nyilván, így, ha újra is indul a bróker, akkor is tudni fogják a consumer-ek, hogy hol tartottak, és onnan folytatják.

Egy Kafka üzenet két részből áll, egy kulcsból és egy értékből. A kulcs opcionális, de ha nem küld a producer kulcsot az üzenethez nem garantálható azok sorrendje. A producer és a topic között van egy „partíció” réteg (Partitioner), ami megkapja az üzenetet és megnézi van-e hozzátartozó kulcs.

Ha nem talál kulcsot az üzenethez, akkor round robin-t használ arra, hogy eldöntse melyik partícióba kerüljön az üzenet. Ezzel az a gond, hogy nem garantálható, hogy az üzenet a beküldött sorrendben jelenik meg.

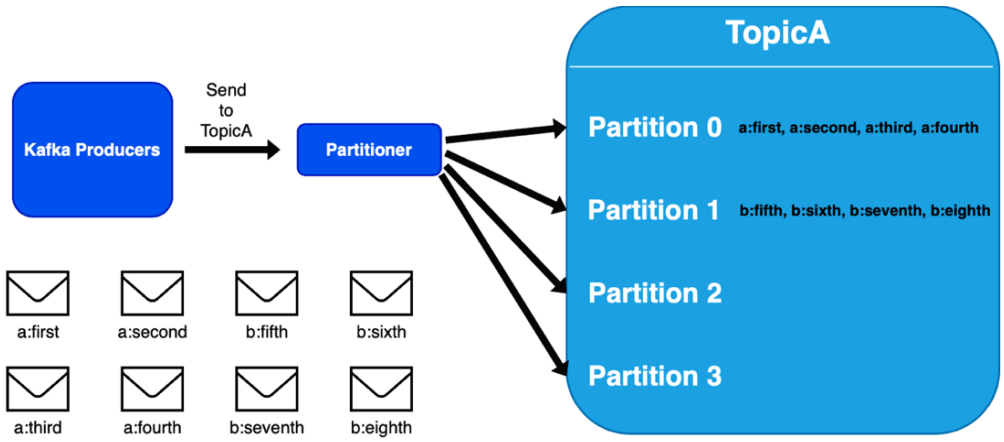
Erre láthatunk példát a *2.8.* ábrán. Látható, hogy nem érkezik, egyik üzenethez sem kulcs, amivel azonosítható, hogy melyik partícióhoz tartozik, így round robin-nal osztja ki, hogy melyik üzenet hova kerül.



2.8. ábra Üzenet küldése kulcs nélkül

Viszont, ha partitioner talál kulcsot, az üzenet mellett, a kulcsokat is kiosztja partíciókhoz és ugyan azzal a kulccsal érkező üzenetek mindig ugyanahhoz a partícióhoz kerül. Ezzel biztosítva a sorrendhelyes érkezést a consumer-hez. A kulcsot egy hashing technikával titkosítja.

A *2.9.* ábrán látható erre egy példa. Látszik, hogy sorban érkezik nyolc darab üzenet. Az első négy (first, second, third, fourth) üzenet egy „a” kulccsal érkezik, és az ábrán látható az is, hogy mind a négy üzenet a „Partition 0” partícióhoz kerül kiosztásra. A második négy (fifth, sixth, seventh, eighth) üzenet viszont a „Partition 1”-hez került kiosztásra. Ezzel garantált a sorrendhelyes üzenet érkezés.



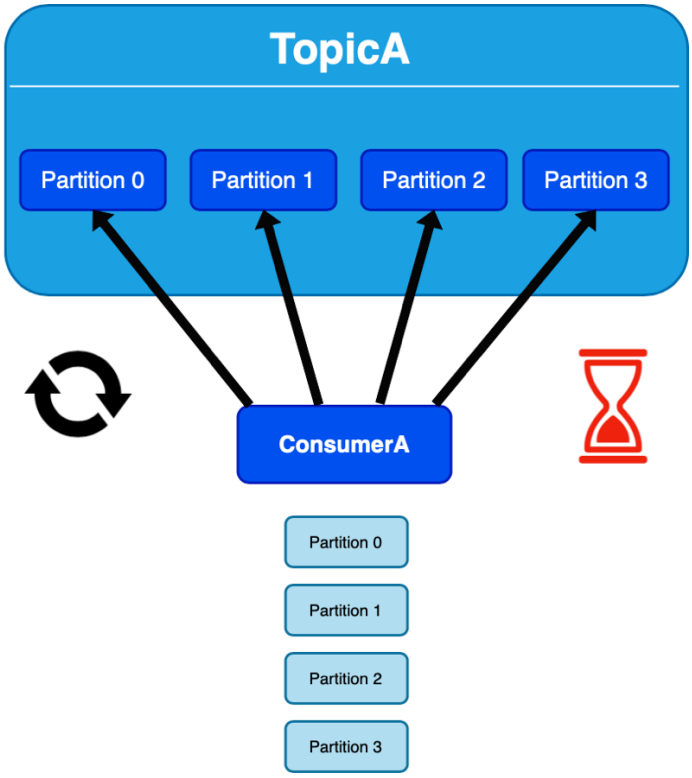
2.9. ábra Üzenet küldése kulccsal

A Kafka egyébként nem tudja értelmezni sem a kulcsot, sem az üzenetet, számára egy bájt tömb. Az, hogy egy objektumból, hogy lesz bájt tömb kulcs és bájt tömb üzenet a producer-ben lévő szerializátor dolga. A consumer-ben pedig a deszerializátor dolga, hogy a bájt folyamból újra értelmes objektumot állítson elő.

2.4.3. Consumer-ek és consumer csoportok

Azok az alkalmazások, amik topic-okból olvasnak üzeneteket Kafka Consumer-nek nevezik. Ezek feliratkoznak egy Kafka topic-ra és folyamatosan figyelik, hogy van-e olyan üzenet, amit fogadnia kell.

A *2.10.* ábrán látható, hogy van egy „Topic A”, amiben négy partíció van. Ezen kívül van egy „Consumer A” Kafka consumer, ami mind a négy partícióból olvas.



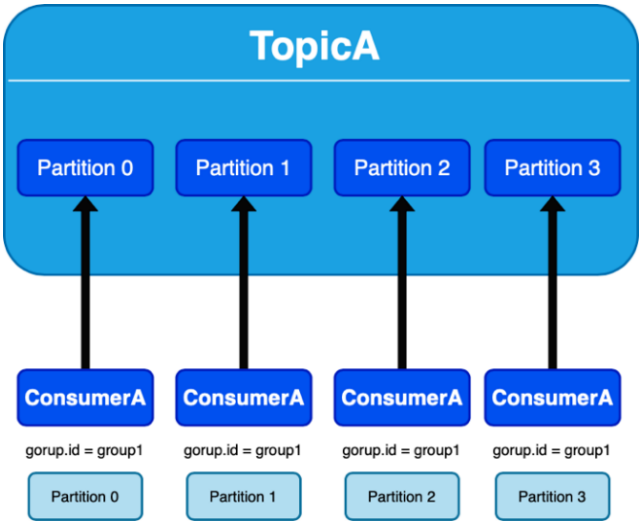
2.10. ábra Kafka consumer működése

Ez egy ideig jól működhet, de mi van akkor, ha a „Topic A”-ba olyan mennyiségű üzenetek érkeznek, hogy már a „Consumer A” nem tudja tartani a olvasási sebességet?

Ha egyetlen consumer-re korlátozódik az adatok olvasása és feldolgozása, az alkalmazás egyre távolabb eshet, és nem képes lépést tartani a bejövő üzenetek arányával. Nyilvánvaló, hogy a consumer-eket topic-ok szerint kell skálázni. Ahogy több producer is írhat ugyanarra a topic-ra, lehetővé kell tennünk, hogy több consumer is olvashasson ugyanabból a topic-ból, felosztva az adatokat közöttük. Erre valók az úgynevezett Consumer csoportok.

A consumer-ek tipikusan egy Consumer csoport tagjai. Ha több consumer feliratkozik egy topic-ra, és ugyanahhoz a consumer csoporthoz tartoznak, akkor a csoport egyes tagjai abból a partícióból olvasnak, ami ki lett nekik osztva.

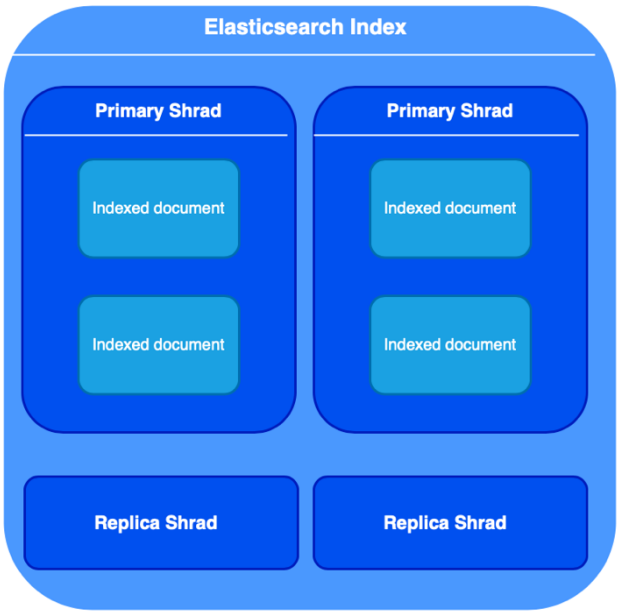
A *2.11.* ábrán jól látszik, erre egy példa ugyan úgy, mint a fenti példában, van egy „Topic A”, amiben négy partíció van, de ahelyett, hogy egy consumer fogadná mind a négy partíció üzenetét, egy consumer csoport csinálja ezt. Látható, hogy minden partícióhoz tartozik egy „Consumer A” és ezekben az a közös, hogy a group id-k megegyeznek. Gyakran előfordul, hogy a Kafka consumer nagy késleltetésű műveleteket végeznek, például írnak egy adatbázisba vagy időigényes számítást végeznek az adatokkal [12].



2.11. ábra Consumer csoportok

2.5. Elasticsearch

Az Elasticsearch egy skálázható, nyílt forrású, teljes szövegű kereső és elemző motor. Ez lehetővé teszi a nagy mennyiségű adat gyors tárolását, keresését és elemzését, gyors és szinte valós időben. Általában olyan alkalmazások használják motorként, amely bonyolult keresési funkciókkal és követelményekkel rendelkeznek. Séma nélküli, néhány alapértelmezett értéket használ az adatok indexeléséhez a keresés gyorsítása miatt.

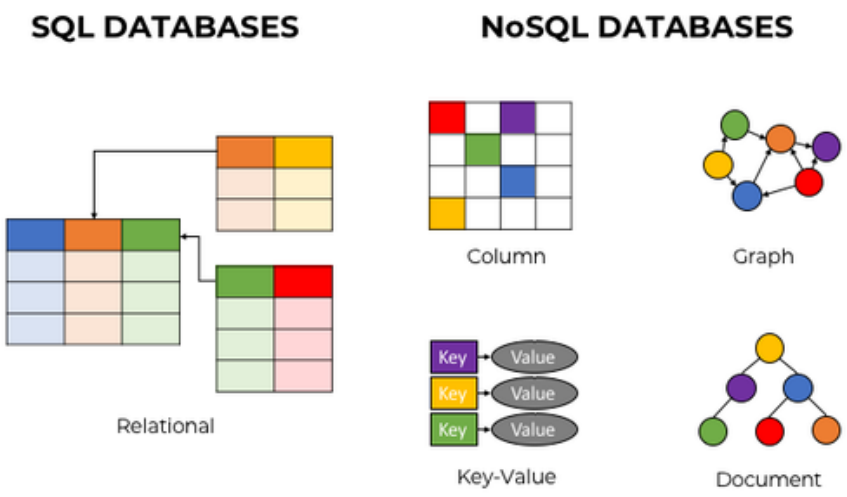


2.12. ábra Elasticsearch architektúra

A Relációs adatbázis (RDBMS) viszonylag lassan működik hatalmas adatkészletek esetében, ami lassabb keresési eredményeket eredményez. Természetesen az RDBMS optimalizálható, de ez magában foglalja a korlátozások halmazát is, például, hogy minden mezőt nem lehet indexelni, és a sorok frissítése erősen indexált táblázatokba hosszadalmas folyamat. A vállalkozások manapság alternatív módszereket keresnek, ahol az adatokat olyan módon tárolják, hogy a visszakeresés gyors. Ez úgy érhető el, ha az adatok tárolására az RDBMS helyett NoSQL-t alkalmazunk.

2.5.1. NoSQL adatbázis bemutatása

A NoSQL adatbázis egy nem relációs adatkezelő rendszer, amelyhez nincs szükség fix sémára. Kerüli a join-okat, és könnyen skálázható. A NoSQL adatbázis használatának fő célja az elosztott adattárolók, amelyeknek nagy az adattárolási igénye. A NoSQL-t nagy adatokhoz és valós idejű webalkalmazásokhoz használják [8].



2.13. ábra RDBMS vs. NoSql
https://www.quora.com/Why-and-when-should-I-use-NoSQL-instead-of-SQL

Az RDBMS adatbázis függőlegesen skálázható. Ha az RDBMS adatbázis terhelése növekszik, az adatbázist úgy méretezzük, hogy növeljük a kiszolgáló hardver teljesítményét. A NoSQL adatbázisok pedig vízszintesen skálázhatók, ami azt jelenti, hogy további gépek hozzáadásával bővíthető.

A NoSQL adatbázisok olcsók és nyílt forráskódúak. A NoSql adatbázis-implementáció egyszerű és általában olcsó szervereket használ az adatok és tranzakciók kezelésére, míg az RDBMS-adatbázisok drágák, és nagy kiszolgálókat és tárolórendszereket használnak. Tehát a NoSQL esetében az adatok tárolása és feldolgozása gigabyte-onként sokszor alacsonyabb lehet, mint az RDBMS költsége.

Az Elasticsearch egy ilyen NoSQL adatbázis. Rugalmas adatmodelleken alapszik és kis késleltetésű, majdhogy nem valós idejű keresést tesz lehetővé.

2.5.2. Elasticsearch műveletek

Az adatok Elasticsearch-be juttatása és lekérdezése REST-en API-n keresztül HTTP kérések formájában történik. Mivel REST-en lehet létrehozni, törölni, módosítani és adatokat lekérdezni, így egyszerű CURL parancssori eszközzel is kezelhető. A Kibana egy adatmegjelenítő és kezelő eszköz hozzá, erről részletesebben a 2.5.3. szekcióban írok. A következő fejezetekben egy pár szóban az Elasticsearch műveletekről szeretnék írni.

2.5.2.1. Index létrehozás

Elasticsearch-ben az index úgy viselkedik, mint egy relációs adatbázisban a tábla. Index létrehozása két féle módon történhet [9].

Explicit mapping: pontosan meg kell adni, hogy milyen adattípusok lesznek az indexben.

```
curl -XPUT -H "Content-Type: application/json" localhost:9200/message -d
'
{
        "settings": {
                "index": {
                        "number_of_shards": 3,
                        "number_of_replicas": 2
                }
        },
        "mappings": {
                "properties": {
                        "messageId": {
                                "type": "text"
                        },
                        "message": {
                                "type": "text"
                        },
                        "dateTime": {
                                "type": "date"
```

```
                        },
                        "roomId": {
                                "type": "long"
                        },
                        "userId": {
                                "type": "long"
                        }
                }
        }
}'
```

Dynamic mapping: az Elasticsearch-re bízzuk, hogy automatikusan hozza létre az indexet.

Tulajdonképpen ez azt jelenti, hogy egy sima „insert” utasítással, ha nem létezik az index akkor létrehozza az adattagok alapján az indexet is.

```
curl -XPOST -H 'Content-Type: application/json'
localhost:9200/message/_doc -d
'{
        "messageId": "fbb50f08-0d32-401a-a645-f6a24526d8f4",
        "message": "Üzenet",
        "dateTime": "2020-04-11T12:34:56.789Z",
        "roomId": 1,
        "userId": 1
}'
```

Mapping lekérdezése:

```
curl -XGET localhost:9200/message/_mapping
```

Response:

```
{
   "message":{
      "mappings":{
         "properties":{
            "dateTime":{
               "type":"date"
            },
            "message":{
               "type":"text"
            },
            "messageId":{
               "type":"text"
            },
            "roomId":{
               "type":"long"
            },
            "userId":{
               "type":"long"
            }
         }
      }
   }
}
```

Index beállításainak lekérdezése:

```
curl -XGET localhost:9200/message/_settings
```

Response:

```
{
  "message" : {
    "settings" : {
      "index" : {
        "creation_date" : "1604945360197",
        "number_of_shards" : "3",
        "number_of_replicas" : "2",
        "uuid" : "WvWHKuiQTw-JGQ-QPLplrg",
        "version" : {
          "created" : "7090299"
        },
        "provided_name" : "message"
      }
    }
  }
}
```

Itt látható, hogy van egy `"number_of_shards"` és egy `"number_of_replicas"` adattag. Az indexet „shard” -okra tudjuk darabolni. A shrad-okban tárolódik egy-egy dokumentum. Dokumentum írásnál először a primary shard-okba kerülnek az adatok majd utána a replikákba. Olvasásnál mind a primary, mind a replikából lehet kiolvasni dokumentumot. Ezeket explicit index létrehozáskor lehet megadni. Utólag is lehet módosítani, de elég körülményes, a replika hozzáadása egyszerűbb, de ez csak akkor jó, ha inkább az olvasási műveletek vannak túlnyomóan.

2.5.2.2. Egyszerű lekérdezések

Az összes dokumentum lekérdezése a „message” indexből:

```
curl -XGET -H 'Content-Type: application/json'
localhost:9200/message/_search
```

Azon dokumentumok lekérdezése a „message” indexből, melynek a message adattagja tartalmazza az „Üzenet” szót:

```
curl -XGET -H 'Content-Type: application/json'
localhost:9200/message/_search -d '{
  "query": {
    "match": {
      "message": "Üzenet"
    }
  }
}'
```

Az összes olyan dokumentum lekérdezése a „message” indexből, ami az 1-es szobában az Üzenet szót tartalmazza.

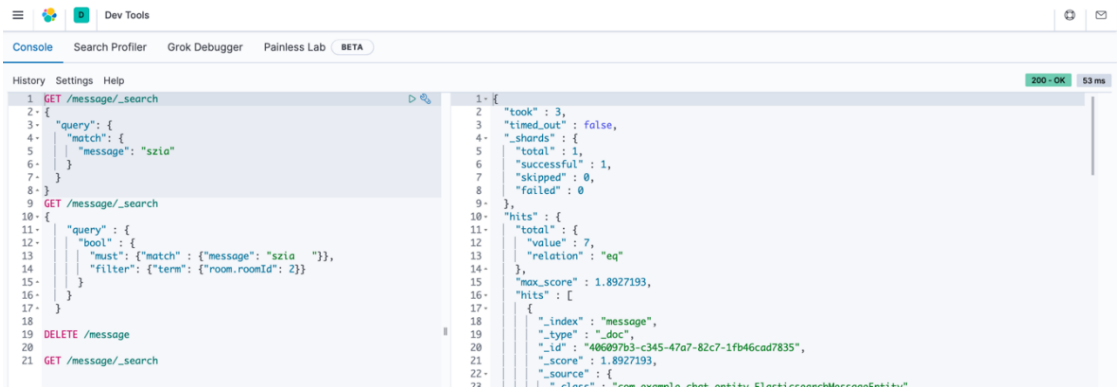
```
curl -XGET -H 'Content-Type: application/json'
localhost:9200/message/_search -d '
{
        "query": {
                "bool": {
                        "must": {
                                "match": {
                                        "message": "Üzenet"
                                }
                        },
                        "filter": {
                                "term": {
                                        "roomId": 1
                                }
                        }
                }
        }
}'
```

Ezek mind nagyon egyszerű lekérdezések, az Elasticsearch sokkal bonyolultabb lekérdezésekre is képes, ebben a dolgozatban nem szeretnék kitérni rájuk. A fent említett mind három lekérdezés eredménye a következő JSON.


```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "message",
        "_type" : "_doc",
        "_id" : "-V1IrnUB3hAnWBOVrTEi",
        "_score" : 1.0,
        "_source" : {
          "messageId" : "fbb50f08-0d32-401a-a645-f6a24526d8f4",
          "message" : "Üzenet",
          "dateTime" : "2020-04-11T12:34:56.789Z",
          "roomId" : 1,
          "userId" : 1
        }}]}}
```

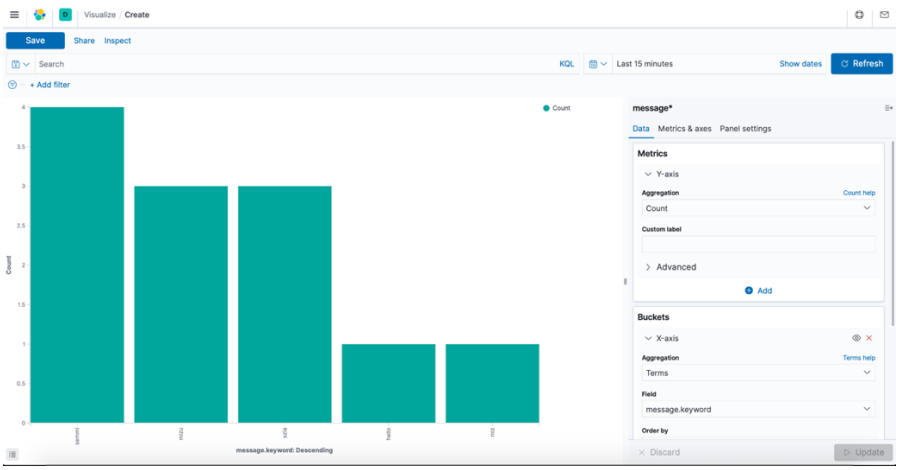
2.5.3. Kibana

A Kibana egy adatmegjelenítő és kezelő eszköz az Elasticsearch számára, amely valós idejű hisztogramokat, vonaldiagramokat, kördiagramokat biztosít a felhasználó számára. Ez lehetővé teszi az Elasticsearch adatok egyszerű megjelenítését. A Kibana remek módja az index-en belüli keresésére és megjelenítésére egy hatékony és rugalmas felhasználói felülettel. Tehát ha szigorúan eltérő adatokkal rendelkező indexekkel rendelkezik, akkor mindegyikhez külön megjelenítést kell létrehoznia.



2.14. ábra Kibana - Dev Tools

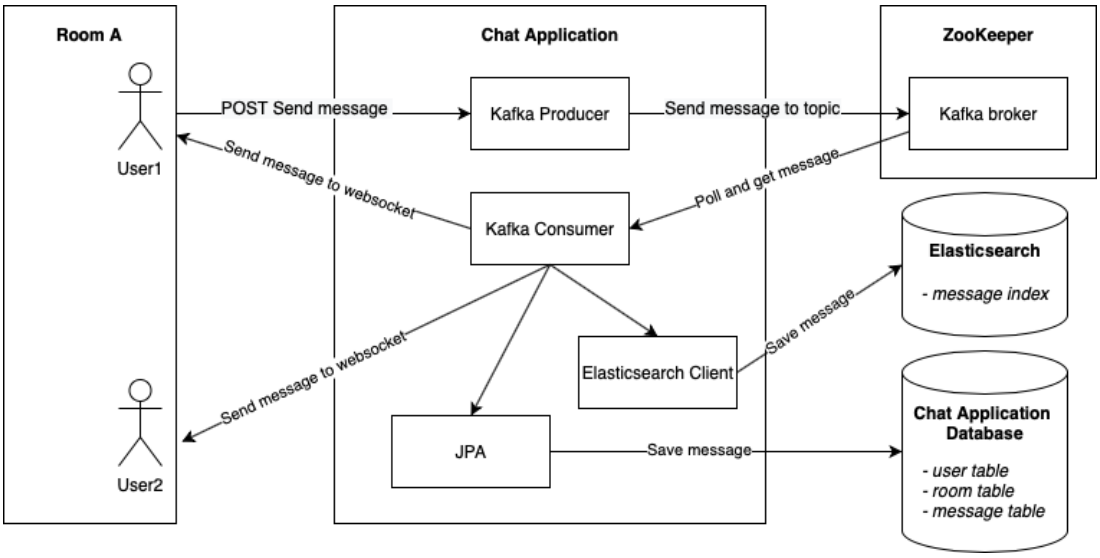
A Visualize menüpont alatt könnyedén készíthetők diagrammok, hisztogramok, kördiagramok a *2.15.* ábrán látható erre egy nagyon egyszerű példa. Üzenet szövegeinek összeszámlálását láthatjuk. az összes üzenet közül a „semmi” szó az, ami négy alkalommal is szerepel, utána három alkalommal a „mizu” és a „szia”, és egy alkalommal a „hello” és „miz” szavak.



2.15. ábra Kibana – Visualize

3. Chat alkalmazás implementáció

Ebben a fejezetben a Chat alkalmazás részeit szeretném részletesen bemutatni. Ez egy Java alkalmazás, részekre bontva mutatom be a komponenseit. A *3.1.* ábrán az alkalmazás komponenseit láthatjuk, valamint egy üzenet küldés-fogadás útját.



3.1. ábra. Chat alkalmazás üzenet küldés-fogadás

A folyamat úgy néz ki, hogy egy felhasználó (User1) üzenetet küld egy másik felhasználónak (User2), akkor http REST végponton keresztül elküldi, az üzenet tartalmát, és a szoba azonosítóját (pl.: „Room A"). A végpont létrehoz egy új Kafka üzenetet, és behelyezi azt a megfelelő Kafka topic-ba. Ezen REST végpont meghívása után csak egy eredményt kap (http 200) a kliens oldali Javascript, az eseményt a rendszer befogadta. Ez az üzenet a Kafka Producer-hez érkezik, ami továbbítja a Kafka Bróker felé, az új üzenet bekerül a megfelelő üzenetsorba. A Kafka bróker a példakód szerint a localhost:9092 url-en elérhető.

A beérkező chat üzenet megjelenítése funkció az előző funkció hatására lép működésbe aszinkron módon. Az általam implementált Kafka Consumer folyamatosan kiolvassa a Kafka bróker üzenet sorából a soron következő üzeneteket. Az új üzenetet három irányba továbbítja.

Először elküldi az Elasticsearch felé és letárolja az üzenetet egy indexben. Az Elasticsearch-csel való integráció nagyon egyszerű, egy Spring data elastichsearch teljesen úgy működik,

mint bármilyen repository, így a kommunikációsforma teljesen transzparens a fejlesztőknek. Ugyan úgy vannak save(), findAll(), findById(id), stb. metódusok. Az Elasticsearch a példakód szerint a localhost:9200 url-en elérhető.

Másrészt a Consumer továbbítja az üzenetet és tárolja egy relációs adatbázis táblájába is. Ez igazából egy biztonsági mechanizmus, nem volt része a feladatnak. Nem készült el a teljes ez fejlesztés, de arra gondoltam, hogy lehetne egy olyan funkciót is készíteni, hogy ha az Elasticsearch-ben bármi probléma történik, akkor újra lehesse az indexet építeni Elasticsearch-ben a relációs adatbázis „message” táblájából. Addig jutottam ezzel, hogy redundánsan tárolom mind a két féle adatbázisban az üzeneteket.

Ezeken kívül WebSocketen továbbítja a „Room A” szobába is. Így jelenik meg minden olyan felhasználónál az üzenet, aki ebben a szobában tartózkodik. Ahogy fentebb említettem a „Room A” magát a böngészőt reprezentálja. Ez a dolgozat inkább a backend megoldásokról szól, de a dolgozat későbbi részében látható, hogy egy minimális felhasználói felületet is készítettem.

A kereséshez végpontokat készítettem, amik REST-en hívhatók, ezeket nem vezettem ki a felületre. Részletesebb kereséshez, vagy metrikák diagrammok létrehozásához viszont a Kibana a legjobb megoldás ez a localhost:5601 url-en elérhető.

Szobák és felhasználók létrehozásához, módosításához, törléséhez és lekérdezéséhez is külön végpontokat készítettem, amik REST-en hívhatók, ezek sincsenek kivezetve a felületre. A Chat alkalmazásban lévő Spring data jpa segítségével tároljuk őket a relációs adatbázisban.

Ahhoz, hogy az alkalmazás működőképes legyen szükség van arra, hogy megteremtsük az előfeltételeket. Kell, hogy fusson Elasticsearch, Apache Kafka bróker, Zookeeper és végül nem feltétlenül szükséges, de a könnyű tesztelhetőség kedvéért egy Kibana is. Mivel bonyolult lenne ezt mind telepíteni, konfigurálni úgy döntöttem, hogy készítek hozzá egy Docker compose fájlt.

3.1. Docker Compose megvalósítása

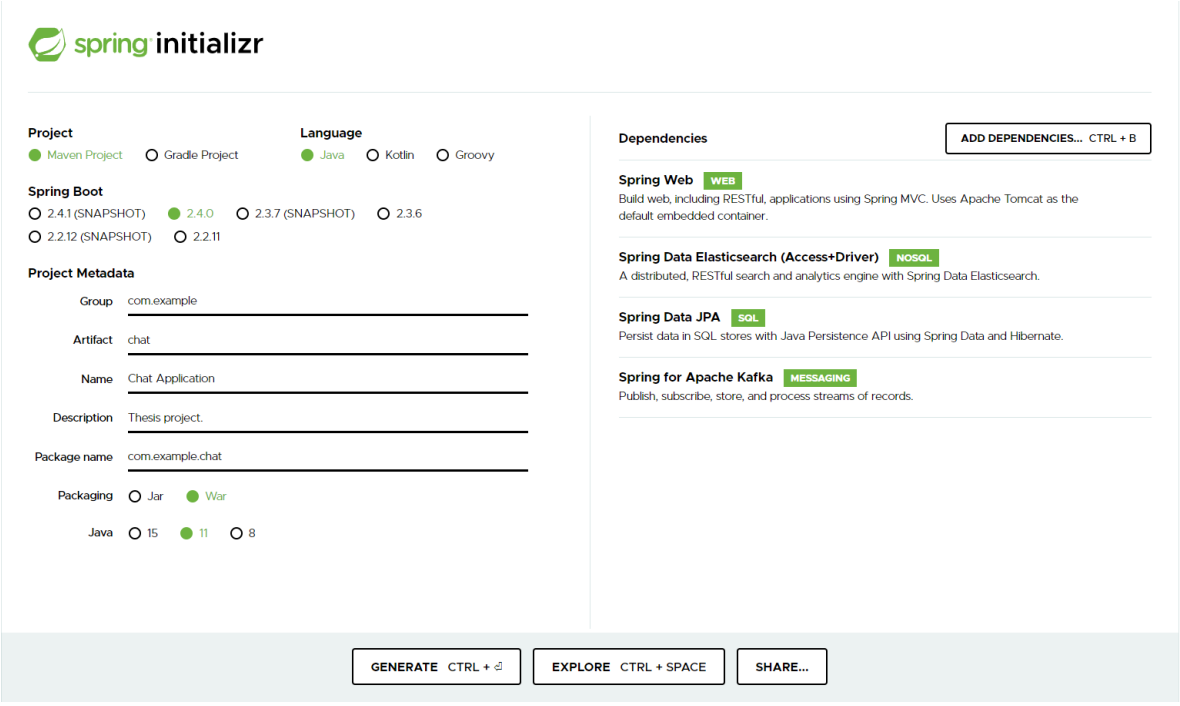
A *2.2.4.* fejezetben már részleteztem, hogy hogyan működik a Docker compose. Ezzel indítjuk a ZooKeeper-t, Apache Kafka brókert, Elasticsearch-et, és Kibana-t. A services alatt soroljuk fel az elindítandó alkalmazásokat és a hozzájuk tartozó konfigurációkat. Itt adjuk meg, hogy az adott szolgáltatás milyen Docker image-et használjon, mi legyen a konténer neve, milyen port-on legyen elérhető, és az esetleges környezeti változókat, ha szükségesek.

```yaml
version: '3'
services:
  zookeeper:
    image: wurstmeister/zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"

  kafka:
    image: wurstmeister/kafka
    container_name: kafka
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.9.2
    container_name: elasticsearch
    environment:
      - node.name=elasticsearch
      - discovery.seed_hosts=elasticsearch
      - cluster.initial_master_nodes=elasticsearch
      - cluster.name=docker-cluster
      - bootstrap.memory_lock=true
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
    volumes:
      - esdata1:/usr/share/elasticsearch/data
    ports:
      - 9200:9200

  kibana:
    image: docker.elastic.co/kibana/kibana:7.9.2
    container_name: kibana
    environment:
      ELASTICSEARCH_URL: "http://elasticsearch:9200"
    ports:
      - 5601:5601
```

```
    depends_on:
      - elasticsearch
volumes:
  esdata1:
    driver: local
```

3.2. Alap Spring Boot alkalmazás generálása

A Spring hivatalos oldalán (https://start.spring.io/) található egy Spring Boot alkalmazás generátor, amivel nagyon egyszerűen készíthetünk egy alap alkalmazást. A *3.2.* ábrán jól látható ez a felület. Meg kell adni, hogy milyen projektet szeretnének készíteni, és milyen nyelven szeretnénk legenerálni az alap projektet. Milyen Spring Boot verziót szeretnénk használni, hányas java verzióval és kiválaszthatjuk, hogy az alkalmazást .jar vagy .war-ba szeretnénk majd becsomagolni. A projekt meta adatait szükséges megadni, ezen belül a gorup-ot, az artifact id-t, a nevet, package nevét a projekt generáláshoz. Ezzel már generálható egy alap alkalmazás Generate gomb segítségével, ami hatására egy .zip-ben letöltésre kerül az alkalmazás. Van arra is lehetőség, hogy függőségeket is generáljuk, az alkalmazásunkba, az Add dependencies gomb megnyomásánál előjön egy pop up ablak, amiben válogathatunk a függőségek közül, amikről már előre tudjuk, hogy szükségesek az alkalmazásba.



3.2. ábra. Spring initializr

Ha a legenerált zip tartalmát kicsomagoljuk és betöltjük egy IDE-be. Akkor egy osztályt találunk, ami tartalmazza a `@SpringBootApplication` annotációt és a main metódust, ez lesz a Spring Boot alkalmazás „belépési pontja”. A `@SpringBootApplication` annotáció tartalmazza a `@EnableAutoConfiguration`, ami automatikusan konfigurálja az alkalmazást a projekthez hozzáadott függőségek alapján, valamint a `@ComponentScan`, ami az összes komponenst behúzza automatikusan.

```
@SpringBootApplication
public class ChatApplication {

	public static void main(String[] args) {
		SpringApplication.run(ChatApplication.class, args);
	}
}
```

3.3. Felhasználók, szobák, üzenetek a relációs adatbázisban

Először a szobák és felhasználók implementálásával foglalkoztam. Ehhez a Spring data jpa-t, liquibase-t és egy H2 embedded adatbázist használtam. REST végpontokat alakítottam ki a felhasználóknak és szobáknak, amik segítségével létrehozhatók, módosíthatók, törölhetők és lekérdezhetők az adatbázisból. Arra gondoltam, hogy a biztonság kedvéért a felhasználók közötti üzenetet nem csak egy Elasticsearch indexben tárolom, hanem egy message táblában is. A *3.3.* ábrán látható a három adatbázis tábla, és a benne lévő oszlop nevek és típusuk.

room	
PK	room_id (bigint)
	room_name (varchar(255))
	subject_id (varchar(255))
	subject_name (varchar(255))
	description (varchar(25

user	
PK	user_id (bigint)
	neptun (varchar(255))
	name (varchar(255))
	email (varchar(255))
	role (varchar(255))

message	
PK	message_id (uuid)
	room_id (bigint)
	user_id (bigint)
	message (varchar(5000))
	datetime (datetime)

3.3. ábra. Relációs adatbázis táblák

Ahhoz, hogy a szobákat, felhasználókat, üzeneteket tudjuk tárolni adatbázisban szükség van több függőségre is. Részlet a pom.xml-ből, a teljes pom.xml megtekinthető az 1. mellékletben.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
        <groupId>org.liquibase</groupId>
        <artifactId>liquibase-core</artifactId>
</dependency>
<dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
</dependency>
```

H2 embedded adatbázis egy olyan eszköz, amit igazából, csak fejlesztési szakaszban jó használni, mivel addig őrzi meg az adatokat, amíg fut az alkalmazás. Az alkalmazás indulásakor a Liquibase segítségével jönnek létre a táblák és az alapadatok.

A Liquibase-ről már a *2.3.* fejezetben írtam, ebben a részben az implementációhoz szükséges teendőket ismertetem. A lentebbi példában kettő changeSet látható, a user tábla létrehozása, és egy sql file beolvasása. Ebben a data.sql fájlban natív sql insertek találhatók.

Részlet a liquibase-change-log.xml-ből:

```
<changeSet author="adam.vecsi" id="create_user_table">
        <createTable tableName="user">
                <column autoIncrement="true" name="user_id"
type="bigint">
                        <constraints primaryKey="true"
primaryKeyName="user_pkey"/>
                </column>
                <column name="neptun" type="varchar(255)"/>
                <column name="name" type="varchar(255)"/>
                <column name="email" type="varchar(255)"/>
                <column name="role" type="varchar(255)"/>
        </createTable>
</changeSet>
<changeSet author="adam.vecsi" id="insert_default_data">
        <sqlFile path="db/data.sql"/>
</changeSet>
```

Részlet a data.sql fájlból:

```
INSERT INTO user (user_id, neptun, name, email, role) VALUES (1,
'IZBTF9', 'Vécsi Ádám', 'vecsi1994@hotmail.com','STUDENT');
INSERT INTO user (user_id, neptun, name, email, role) VALUES (2,
'XYZ232', 'Kovács Béla', 'asdsa@asdas.com','STUDENT');
```

```
INSERT INTO room (room_id, room_name, subject_id, subject_name,
description) VALUES (2, 'Mobil programozás szoba', 'GEIAL51AML','Mobil
programozás','Chat szoba a Mobil programozáshoz.');
INSERT INTO room (room_id, room_name, subject_id, subject_name,
description) VALUES (3, 'Adatelemzés és adatbányászati módszerek szoba',
'GEIAL526ML','Adatelemzés és adatbányászati módszerek','Chat szoba a
Adatelemzés és adatbányászati módszerekhez.');
```

A Liquibase és a H2 embeded database is az application.properties fájlban konfigurálható. Részlet az application.properties-ből, a 2. mellékletben megtalálható a teljes application.properties.

```
#DB
spring.h2.console.enabled=true

spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect


#liquibase
spring.liquibase.enabled=true
spring.liquibase.change-log=classpath:db/liquibase-change-log.xml
```

Itt tudjuk engedélyezni, hogy a H2 konzol elérhető legyen, valamint be tudjuk állítani, hogy milyen url-en, milyen felhasználóval, milyen jelszóval, legyen elérhető.

Liquibase-t is engedélyezni kell, és meg kell adni, hogy hol található az change-log fájl. Ezekkel a beállításokkal, már alkalmazás indulásakor létrejön egy adatbáziskapcsolat, létrejönnek a táblák, és alap adatok is kerülnek bele. Természetesen nem csak így lehet az adatbázisba adatot felhasználókkal és szobákkal kapcsolatos adatokat küldeni, törölni, módosítani.

Készítettem erre külön végpontokat, amik a Spring data jpa segítségével juttatja be és ki az adatokat az adatbázisból, Repository interfészek segítségével. Ezek a repository interfészek a JpaRepository-t származtatják, így könnyedén történik a mentés, módosítás, törlés és lekérdezés. A hívás a Controller rétegbe érkezik be. Például a UserController osztályba, ami `@RestController` annotációval van ellátva, a lentebbi példában a `/chat/user` végpontra. Ez az osztály injektálja a Service rétegben lévő UserService interface-t, amit implementáltam a UserServiceImpl osztályban és ez megkapta a `@Service` annotációt. Ez

az a réteg, ahol az üzleti logika van általában megvalósítva. A Service rétegből hívunk át a Repository rétegbe, ami az adatbázissal „beszélget”.

A következő példában bemutatom a felhasználó létrehozást és végig vezetem mindhárom rétegen.

Controller réteg:

```
@RestController
@RequiredArgsConstructor
@RequestMapping("/chat")
public class UserController {

        private final UserService userService;

        @PostMapping("/user")
        public ResponseEntity<Void> createUser(@RequestBody User user) {
                userService.createUser(user);
                return new ResponseEntity<>(HttpStatus.CREATED);

        }
}
```

Service réteg:

```
@Service
@RequiredArgsConstructor
public class UserServiceImpl implements UserService {

        private final UserRepository userRepository;

        @Override
        public void createUser(User user) {
                user.setNeptun(user.getNeptun().toUpperCase());
                userRepository.save(UserMapper.userToEntity(user));

        }
}
```

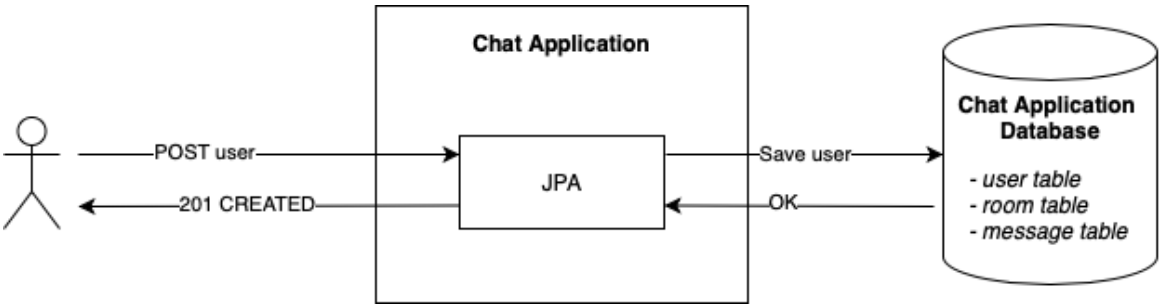
Repostiroy réteg:

```
public interface UserRepository extends JpaRepository<UserEntity, Long> {

}
```

User entitás:

```
@Data
@Entity
@Table(name = "user")
@NoArgsConstructor
@AllArgsConstructor
public class UserEntity {
```

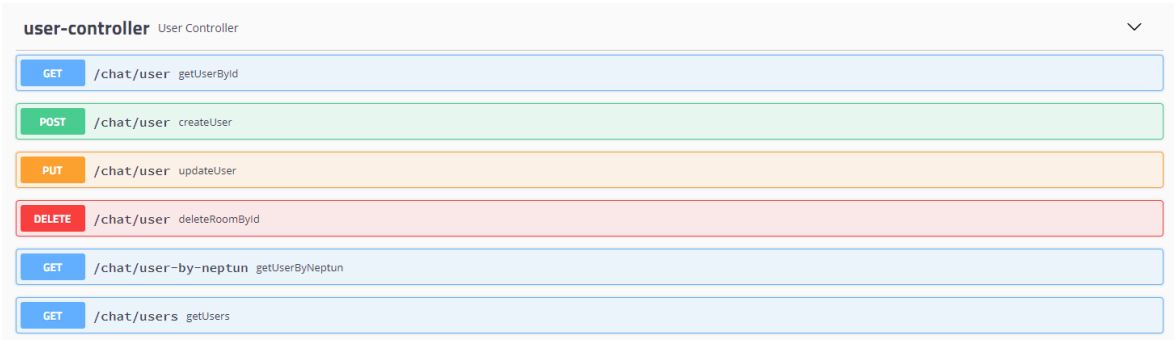
```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_id")
    private Long userId;

    @Column(name = "neptun")
    private String neptun;

    @Column(name = "name")
    private String name;

    @Column(name = "email")
    private String email;

    @Column(name = "role")
    @Enumerated(EnumType.STRING)
    private Role role;

    @OneToMany(mappedBy = "senderUser")
    private List<DatabaseMessageEntity> messages;
}
```



3.4. ábra. Felhasználó mentése

Ennek mintájára hoztam létre a lentebb említett, felhasználókhoz és szobákhoz kapcsolódó végpontokat.

Felhasználókhoz kapcsolódó végpontok:



3.5. ábra Swagger UI – Felhasználókhoz köthető végpontok

Felhasználó lekérdezése id alapján

GET localhost:8082/chat/user?userId={userId}

Response:


```
{
  "userId": 1,
  "neptun": "IZBTF9",
  "name": "Vécsi Ádám",
  "email": "vecsi1994@hotmail.com",
  "role": "STUDENT"
}
```


Felhasználó létrehozása:

POST localhost:8082/chat/user

Request:


```
{
  "neptun": "IZBTF9",
  "name": "Vécsi Ádám",
  "email": "vecsi1994@hotmail.com",
  "role": "STUDENT"
}
```


Response: 201 - Create

Felhasználó módosítása:

PUT localhost:8082/chat/user

Request:


```
{
  "userId": 1,
  "neptun": "IZBTF9",
  "name": "Vécsi Ádám 2",
  "email": "vecsi1994@hotmail.com",
  "role": "STUDENT"
}
```


Response: 204 – No content

Felhasználó törlése id alapján:

DELETE localhost:8082/chat/user?userId={userId}

Response: 202 – Accepted

Felhasználó lekérdezése neptun kód alapján:

GET localhost:8082/chat/user-by-neptun?neptun={neptun}

Response: 200 - Ok


```
{
  "userId": 1,
  "neptun": "IZBTF9",
  "name": "Vécsi Ádám",
  "email": "vecsi1994@hotmail.com",
  "role": "STUDENT"
}
```

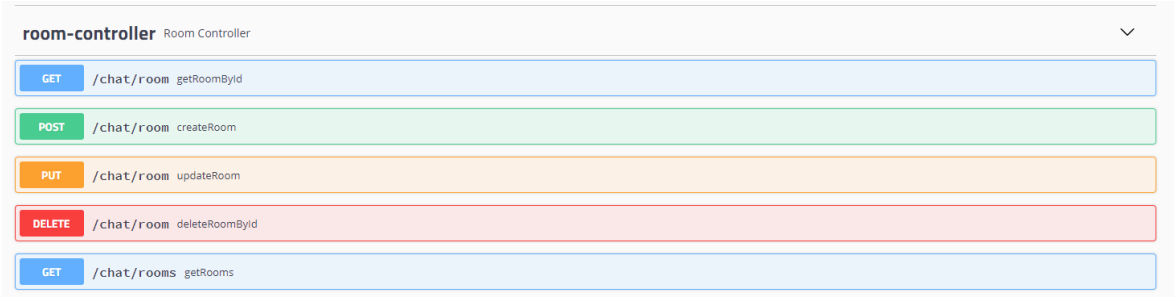

Összes felhasználó lekérdezése:

GET localhost:8082/chat/users

Response: 200 - Ok


```
[
  {
    "userId": 1,
    "neptun": "IZBTF9",
    "name": "Vécsi Ádám",
    "email": "vecsi1994@hotmail.com",
    "role": "STUDENT"
  },
  {
    "userId": 2,
    "neptun": "XYZ232",
    "name": "Kovács Béla",
    "email": "asdsa@asdas.com",
    "role": "STUDENT"
  }
]
```


Szobákhoz kapcsolódó végpontok:



3.6. ábra Swagger UI – Szobákhoz köthető végpontok

Szoba létrehozása:

POST localhost:8082/chat/room

Request:

```
{
  "roomName": "Diplomatervezés II. szoba",
  "subjectId": "GEIAL536ML",
  "subjectName": "Diplomatervezés II.",
  "description": "Chat szoba a diplomatervezéshez."
}
```

Response: 201 - Created

Szoba módosítása:

PUT localhost:8082/chat/room

Request:

```
{
  "roomId": 1,
  "roomName": "Diplomatervezés I. szoba",
  "subjectId": "GEIAL536ML",
  "subjectName": "Diplomatervezés I.",
  "description": "Chat szoba a diplomatervezéshez."
}
```

Response: 204 – No content

Szoba törlése id alapján:

DELETE localhost:8082/chat/room?roomId={roomId}

Response: 202 - Accepted

Szoba lekérdezése id alapján:

GET localhost:8082/chat/room?roomId={roomId}

Response: 200 – Ok

```
{
  "roomId": 1,
  "roomName": "Diplomatervezés II. szoba",
  "subjectId": "GEIAL536ML",
  "subjectName": "Diplomatervezés II.",
  "description": "Chat szoba a diplomatervezéshez."
}
```

Összes szoba lekérdezése:

GET localhost:8082/chat/rooms

Response: 200 – Ok

```
[
  {
    "roomId": 1,
    "roomName": "Diplomatervezés II. szoba",
    "subjectId": "GEIAL536ML",
    "subjectName": "Diplomatervezés II.",
    "description": "Chat szoba a diplomatervezéshez."
  }
]
```

3.4. Apache Kafka integráció

A *2.4.* fejezetben már említettem ahhoz, hogy az alkalmazás elinduljon szükséges egy Zookeeper és egy Kafka Bróker ezt a Docker compose segítségével nagyon egyszerűen indítható. De ahhoz, hogy az alkalmazás tudjon kapcsolódni a Kafka brókerhez és tudjon a topic-ba üzenetet küldeni, és azt ki is tudja venni szükség van Producer és Consumer beállításokra. A pom.xml-be a következő függőségekre van szükség:

```
<dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
</dependency>
```

3.4.1. Topic létrehozása és csatlakozás hozzá

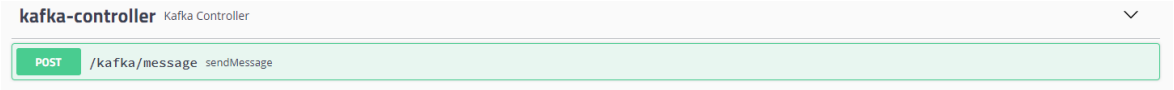
Amikor készítettem az alkalmazást úgy döntöttem, hogy a topic-ot is az alkalmazás hozza létre erre egy egyszerű konfigurációs osztály készült. Itt annyi történik, hogy létrejön egy „chat-rooms” topic, egy partícióval és egy replikával.

```
@Configuration
public class KafkaTopicConfig {

        @Bean
        public NewTopic createTopic() {
                return TopicBuilder.name("chat-rooms")
        .partitions(1).replicas(1).build();
        }
}
```

3.4.2. Producer

Kafka producer konfigurációjában meg kell adnunk, a Kafka bróker elérését, ami ebben az esetben a `localhost:9092` és azt is meg kell adnunk, hogy milyen kulccsal küldünk be üzenetet és milyen értékkel. Ebben az esetben a kulcs szerializálása Long, az értéké pedig String. Ezt az application.properties-ben tehetjük:

```
#Kafka producer
spring.kafka.producer.bootstrap-servers=localhost:9092
spring.kafka.producer.key-
serializer=org.apache.kafka.common.serialization.LongSerializer
spring.kafka.producer.value-
serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.admin.properties.bootstrap.servers=localhost:9092
```



3.7. ábra Swagger UI – Kafka üzenet végpont

Készítettem egy végpontot, amivel üzenetet lehet POST-olni a Kafka topic-ba itt látható, hogy a metódus, ami paraméterként egy szoba id-t és egy üzenet objektumot vár.

POST localhost:8082/kafka/message?roomId={roomId}

Request:

```
{
    "room": {
        "roomId": 1,
        "roomName": "Diplomatervezés II. szoba",
        "subjectId": "GEIAL536ML",
        "subjectName": "Diplomatervezés II.",
        "description": "Chat szoba a diplomatervezéshez."
    },
    "senderUser": {
        "userId": 1,
        "neptun": "IZBTF9",
        "name": "Vécsi Ádám",
        "email": "vecsi1994@hotmail.com",
        "role": "STUDENT"
    },
    "message": "Szia"
}
```

Response: 201 - Created

Létrehoz egy olyan „üzenetet”, amiben van egy generált message id, és maga az üzenetben lévő többi adattagot. Átalakítja json formátummá ezt az objektumot, és beleteszi egy String-be. Egy builder segítségével összekészít egy ProducerRecord objektumot, amihez a topic neve, a kulcs, ami itt a roomId és String-gé alakított üzenet szükséges. KafkaTemplate segítségével, beküldi az üzenetet a topic-ba.

```
@Autowired
private KafkaTemplate<Long, String> kafkaTemplate;

public void sendMessage(Long roomId, Message message) throws
JsonProcessingException {

        MessageWithId messageWithId = new MessageWithId();
        messageWithId.setMessageId(UUID.randomUUID());
        messageWithId.setDateTime(LocalDateTime.now());
        messageWithId.setMessage(message.getMessage());
        messageWithId.setRoom(message.getRoom());
        messageWithId.setSenderUser(message.getSenderUser());

        String value = objectMapper.writeValueAsString(messageWithId);

        ProducerRecord<Long, String> producerRecord =
buildProducerRecord(roomId, value,”chat-rooms”);
        ListenableFuture<SendResult<Long, String>> listenableFuture =
kafkaTemplate.send(producerRecord);
        listenableFuture.addCallback(new
ListenableFutureCallback<SendResult<Long, String>>() {

                @Override
                public void onSuccess(SendResult<Long, String> result) {
                        handleSuccess(roomId, value, result);
                }

                @Override
                public void onFailure(Throwable ex) {
                        handleFailure(roomId, value, ex);
                }
        });
}
private ProducerRecord<Long, String> buildProducerRecord(Long key, String
value, String kafkaTopic) {
        return new ProducerRecord<Long, String>(kafkaTopic, null, key,
value, null);
        }
```

3.4.3. Consumer

Ahhoz, hogy ezeket az üzeneteket fogadjuk, szükség van egy Consumer-re is. A következő beállítások szükségesek az application.properties-ben. Itt is meg kell adni, a Kafka bróker elérését, és hogy mivel szeretnénk deszerializálni a kulcs-érték párokat. Mivel a Producerben

Long és String-et adtunk meg, itt annak a párját kell megadni. Létrehoztam egy „chat-listener-group”-pot is, ami akkor szükséges, hogy ha több consumer fut. Ebben az alkalmazásban, együtt van a Producer és a Consumer így ez nem releváns.

```
#Kafka consumer
spring.kafka.consumer.bootstrap-servers=localhost:9092
spring.kafka.consumer.key-
deserializer=org.apache.kafka.common.serialization.LongDeserializer
spring.kafka.consumer.value-
deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.group-id=chat-listener-group
```

A Consumer konfigurációja nagyon egyszerű, legalább is az alapkonfiguráció `@EnableKafka` annotáció-ban alapból minden szükséges dolog benne van, amit használtam.

```
@Configuration
@EnableKafka
public class KafkaConsumerConfig {
}
```

Az üzenet fogadása is nagyon egyszerű a `@KafkaListener` annotációnak megadjuk, hogy milyen topic az, poll-ozunk és várjuk, hogy üzenet kerüljön bele. Ha talál üzenetet kiveszi és meghívja a `messageSaveService.sendAndSaveMessage(consumerRecord.value())` metódust, ennek hatására letárolja az üzenetet Elasticsearch-ben, H2 adatbázisban és végül továbbítja websocket-en a megfelelő szobába.

```
@KafkaListener(topics = { "chat-rooms" })
public void onMessage(ConsumerRecord<Long, String> consumerRecord)
                throws JsonMappingException, JsonProcessingException {
        log.info("ConsumerRecord : {}", consumerRecord);
        messageSaveService.sendAndSaveMessage(consumerRecord.value());

}
```

3.5. Elasticsearch integráció

Az alkalmazás indulásához és helyes működéséhez elengedhetetlen az Elasticsearch, mivel itt is tárolódik az üzeneteket, igaz tárolódik H2 adatbázisban is, de a keresés és szűrési lehetőségek az Elasticsearch-öt használjuk. Az alkalmazás konfigurációban, az application.properties-ben egy dolgot szükséges felvenni, az Elasticsearch elérési url-jét.

```
#Elasticsearch
chat.elasticsearch.url=localhost:9200
```

A pom.xml-ben a következő függőséget szükséges felvenni:

```
<dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-elasticsearch</artifactId>
</dependency>
```

A konfigurációjában engedélyezni szükséges az Elasticsearch repository-t és be kell állítani, hogy melyik package-ben található az elasticsearch repository class. Két dolgot szükséges behúzni bean-ként, egy RestHighLevelClient-et kell létrehozni és ott megadni azt az url-t, amit az application.properties-ben állítottunk be. A másik bean-ben egy elasticsearchTemplat-et hoz létre, a client segítségével.

```
@Configuration
@EnableElasticsearchRepositories(basePackages =
"com.example.chat.repository")
public class ElasticsearchConfig {

        @Value("${chat.elasticsearch.url}")
        private String elasticsearchUrl;

        @Bean
        public RestHighLevelClient client() {
                ClientConfiguration clientConfiguration =
ClientConfiguration.builder().connectedTo(elasticsearchUrl).build();
                return RestClients.create(clientConfiguration).rest();
        }
        @Bean
        public ElasticsearchOperations elasticsearchTemplate() {
                return new ElasticsearchRestTemplate(this.client());
        }
}
```

3.5.1. Index létrehozása

Az index létrehozása annotációkkal történik, egy entitást hoztam létre hozzá, ami az üzenetek tárolásáért felelős. A `@Document` annotációnál szükséges megadni, hogy milyen indexet szeretnénk létrehozni az Elasticsearch-ben. Itt lehet megadni, hogy hány darab primary shards és hány darab replika legyen. Ebben az esetben egy shard és egy replica elég lesz. Nincs egzakt mondás arra, hogy mennyi az ideális shard és replika, függ attól, hogy milyen terhelése van az Elasticsearh-nek, tehát, hogy mennyi az írás és mennyi az olvasás. Ugyan úgy, mint a spring data jpa-nál itt is meg kell adni egy id-t kötelezően, és az összes egyéb adattagra egy `@Field` annotációt, amiben meg kell adni, hogy az adott mezőt, hogyan

kezelje az Elasticsearch. Lentebb látható, hogy teljesegészében tárolja a szoba entitást és a felhasználó entitást is. A dátumnál megadható egy formátum és egy dátum „pattern”. Azokra a mezőkre, amiknél egyértelmű a típus, mint például a message, annál nem szükséges ez az annotáció.

```
@Document(indexName = "message", shards = 1, replicas = 1)
public class ElasticsearchMessageEntity {

        @Id
        private UUID messageId;

        @Field(type = FieldType.Nested, includeInParent = true)
        private RoomEntity room;

        @Field(type = FieldType.Nested, includeInParent = true)
        private UserEntity senderUser;

        private String message;

        @Field(type = FieldType.Date, format = DateFormat.custom, pattern
= "uuuu-MM-dd'T'HH:mm:ss.SSSSSSZ")
        private LocalDateTime dateTime;
}
```

3.5.2. Üzenetek mentése és lekérdezése

Ami nagyon egyszerűvé teszi az Elasticsearch-el való integrációt, hogy teljesen úgy működik, mint bármelyik repository hívás jpa segítségével. Egy repository interface szükséges a mentéshez és lekérdezéshez, ami származtat egy ElasticsearchRepository-t és meg kell adni, hogy milyen entitásokat használ és milyen id-ja van.

Ha a lentebb látható lekérdezéseket nézzük, három darab query-t készítettem. Az elsőnél, az utolsó öt üzenetet listázza, „roomId” alapján beküldési időpont szerinte csökkenő sorrendbe rendezve. Látható, hogy ez nem egy összetett és bonyolult lekérdezés. Ez után egy szűréshez szükséges lekérdezés látható, ami az összes olyan üzenetet listázza, ami két dátum közé esik, amit a „from” és „to” paraméterekben lehet megadni, „roomId” -val együtt. Az utolsó lekérdezés kicsit már összetettebb ezért ott már szükség volt `@Query` annotációra, és megadjuk benne, hogy melyik szobában szeretnénk keresni az üzenetekben, és meg kell adni egy keresőszót is. Ez alapján listázza az össze olyan üzenetet, amiben megtalálható az adott keresőszó. Látható, hogy a lekérdezéseket nagyon megkönnyíti a Spring data.

```
public interface MessageElasticsearchRepository extends
ElasticsearchRepository<ElasticsearchMessageEntity, UUID> {

        List<ElasticsearchMessageEntity>
findTop5ByRoomRoomIdOrderByDateTimeDesc(Long roomId);

        List<ElasticsearchMessageEntity>
findAllByDateTimeBetweenAndRoomRoomId(LocalDateTime from, LocalDateTime
to, Long roomId);

        @Query("{\"bool\":{\"must\":{\"match\":{\"message\":\"?1\"}},\"fi
lter\":{\"term\":{\"room.roomId\":?0}}}}")
        List<ElasticsearchMessageEntity> findByMessageAndRoomRoomId(Long
roomId, String search);

}
```

A mentést is megkönnyíti a Spring data, amikor a Kafka Consumer-be elkap egy üzenetet fentebb említettem, hogy meghív egy metódust.

```
messageSaveService.sendAndSaveMessage(consumerRecord.value());
```

Az a metódus tulajdonképpen, áthív az Elasticsearch-be, és így továbbítja az üzenetet. Az üzenetet, amit a Kafka consumer elkapott átalakítja egy ElasticsearchMessageEntity-vé egy objectMapper segítségével. Ezután hív a repository-n egy `elasticsearchRepository.save(elasticsearchMessage);` metódust, amivel letárolódik az üzenet az Elasticsearch-ben. A következőkben bemutatom a `sendAndSaveMessage(String value)` metódust és kiemeltem azokat a részeket, amik ide tartoznak. Fentebb már említettem, ez a metódus nem csak az Elasticsearch-be juttatja el az üzenetet, hanem eltárolja a H2 embedded adatbázisban és websocketen is továbbítja.

```
public void sendAndSaveMessage(String value) {
        ElasticsearchMessageEntity elasticsearchMessage;
        try {
                elasticsearchMessage = objectMapper.readValue(value,
ElasticsearchMessageEntity.class);
        } catch (JsonProcessingException e) {
                log.error("Failed to parse json.", e);
                throw new IllegalArgumentException("Failed to parse
json.");
        }
        elasticsearchRepository.save(elasticsearchMessage);
}
```

3.6. Websocket és JQuery használata

Ez a dolgozat nem kifejezetten a felhasználói felületről szól, hanem inkább a backend oldali implementálásáról, de úgy gondoltam, hogy egy minimális felhasználói felületet kialakítok azért, hogy prezentálható legyen a szobába való csatlakozás, valamint az üzenet beküldés-fogadása.

A pom.xml-ben a következő függőségeket szükséges felvenni:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
<dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator-core</artifactId>
</dependency>
<dependency>
        <groupId>org.webjars</groupId>
        <artifactId>sockjs-client</artifactId>
        <version>1.0.2</version>
</dependency>
<dependency>
        <groupId>org.webjars</groupId>
        <artifactId>stomp-websocket</artifactId>
        <version>2.3.3</version>
</dependency>
<dependency>
        <groupId>org.webjars</groupId>
        <artifactId>bootstrap</artifactId>
        <version>3.3.7</version>
</dependency>
<dependency>
        <groupId>org.webjars</groupId>
        <artifactId>jquery</artifactId>
        <version>3.1.1-1</version>
</dependency>
```

A websocket szerver oldali konfigurálásával és implementálásával kezdem, majd a JavaScript segítségével kialakított frontend-ről fogok írni. Egy konfigurációs osztályra van szükség, szerver oldalon, hogy a websocket kapcsolat felépüljön. Itt először származtatni szükséges a `WebSocketMessageBrokerConfigurer` interfészt és implementálnunk kell kettő metódust a websocket bróker konfigurálásához.

Először az `enableSimpleBroker` meghívásával engedélyezheti, hogy egy egyszerű memória alapú üzenetközvetítő visszavigye az üzeneteket az klienshez a /topic előtaggal ellátott célállomásokon. Ezenkívül kijelöli az /app előtagot azokhoz az üzenetekhez,

amelyek a `@MessageMapping` annotációval vannak ellátva. Ezzel az előtaggal fogják meghatározni az összes üzenet-hozzárendelést. A **`registerStompEndpoints`** metódus regisztrálja a `"/chat-websocket"` végpontot, lehetővé téve a SockJS opcióit, hogy alternatív „szállítások” is használhatók legyenek, ha a WebSocket nem érhető el. A SockJS kliens megpróbál csatlakozni a `"/chat-websocket"`-hez [10].

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer
{
        @Override
        public void configureMessageBroker(MessageBrokerRegistry config){
                config.enableSimpleBroker("/topic");
                config.setApplicationDestinationPrefixes("/app");
        }
        @Override
        public void registerStompEndpoints(StompEndpointRegistry
registry) {
                registry.addEndpoint("/chat-websocket").withSockJS();
        }
}
```

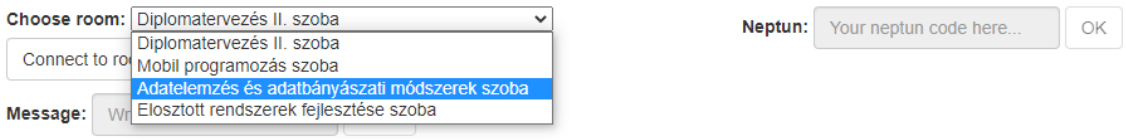
Amikor a Kafka Consumer kap egy új üzenetet fentebb már említettem, hogy először az Elasticsearch-be, aztán a H2 adatbázisba kerül be, ezek után kerül továbbításra az adott szobába. A `SimpleMessagingTemplate` osztály `convertAndSend("/topic/chat/{roomId}")` metódusával küldi az üzenetet a megfelelő id-val rendelkező szobába.

```
@Autowired
private SimpMessagingTemplate template;
@MessageMapping("/chat/{roomId}")
public void sendAndSaveMessage(String value) {
        ElasticsearchMessageEntity elasticsearchMessage;
        DatabaseMessageEntity jpaMessage;
        try {
                elasticsearchMessage = objectMapper.readValue(value,
ElasticsearchMessageEntity.class);
                jpaMessage = objectMapper.readValue(value,
DatabaseMessageEntity.class);
        } catch (JsonProcessingException e) {
                log.error("Failed to parse json.", e);
                throw new IllegalArgumentException("Failed to parse
json.");
        }
        elasticsearchRepository.save(elasticsearchMessage);
        jpaRepository.save(jpaMessage);
        template.convertAndSend("/topic/chat/" +
elasticsearchMessage.getRoom().getRoomId(), value);
}
```

A kliens rész HTML, JavaScript, JQuery segítségével készült. Amikor meglátjuk a felületet először ki kell, hogy válasszuk, hogy melyik szobához szeretnénk csatlakozni.

3.8. ábra Felhasználói felület

Ehhez a *3.3.* fejezetben említett /chat/rooms végpontot hívjuk, ami a getRooms() metódust hívja és vissza adja az összes H2 adatbázisban létező szobát.

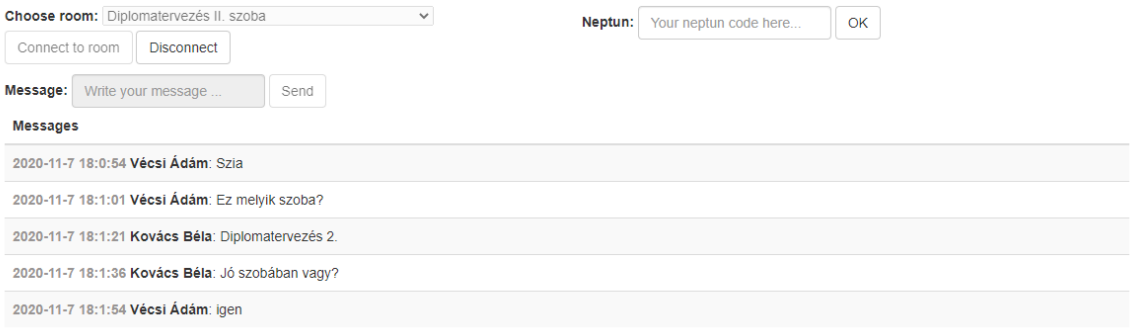


3.9. ábra Felhasználói felület - szoba választás drop down

Amikor a kliens betöltődik egy egyszerű ajax hívással lekérdezi és belerakja egy drop down-ba.

```
$(document).ready(function() {
        $("#neptunButton").prop("disabled", true);
        $("#sendButton").prop("disabled", true);
        $("#disconnectButton").prop("disabled", true);
        $("#neptun").prop("disabled", true);
        $("#message").prop("disabled", true);
    $.ajax({
        url: "http://localhost:8082/chat/rooms"
    }).then(function(data) {
        for(a of data){
                $("#rooms").append("<option value="+a.roomId+">" +
a.roomName + "</option>")
        }
    });
});
```

Miután kiválasztjuk a kívánt szobát a Connect to room gomb segítségével, csatlakozik a websocket-hez és feliratkozik az adott szobára, valamint az utolsó öt üzenet betöltődik az adatbázisból.



3.10. ábra Felhasználói felület – Utolsó öt üzenet megjelenítése

```
$("#roomButton").click(function() {
        getLastMessages();
        connect();
        $("#neptunButton").prop("disabled", false);
        $("#neptun").prop("disabled", false);
        $("#disconnectButton").prop("disabled", false);
        $("#roomButton").prop("disabled", true);
        $("#rooms").prop("disabled", true);
});
```

Lekérdezzük, a kiválasztott szobában lévő utolsó öt üzentet az elasticsearh-ből, és megjelentjük „Messages” alatt.

GET localhost:8082/chat/search/last-5-messages?roomId={roomId}

Response: 200 – Ok

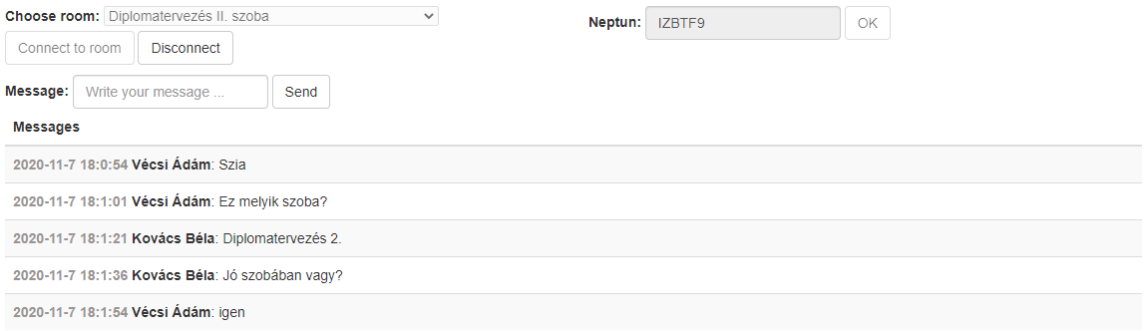

```
[
  {
    "messageId": "112f93e0-6a1f-44c4-98bb-bb4f15ea93d9",
    "room": {
      "roomId": 1,
      "roomName": "Diplomatervezés II. szoba",
      "subjectId": "GEIAL536ML",
      "subjectName": "Diplomatervezés II.",
      "description": "Chat szoba a diplomatervezéshez.",
      "messages": null
    },
    "senderUser": {
      "userId": 1,
      "neptun": "IZBTF9",
      "name": "Vécsi Ádám",
      "email": "vecsi1994@hotmail.com",
      "role": "STUDENT",
      "messages": null
    },
    "message": "szia",
    "dateTime": "2020-10-31T20:38:46.64995"
  }
]
```

```
function getLastMessages() {
        var roomId = $("#rooms").val();
         $.ajax({
                url: "http://localhost:8082/chat/search/last-5-
messages?roomId=" + roomId
            }).then(function(data) {
                for(a of data){
                        showMessages(a)
                }
            });
}
```

Csatlakozás a websocket-hez és a kiválasztott szobára feliratkozás.

```
function connect() {
        var socket = new SockJS('/chat-websocket');
        stompClient = Stomp.over(socket);
        stompClient.connect({}, function(frame) {
                setConnected(true);
                console.log('Connected: ' + frame);
                stompClient.subscribe('/topic/chat/' + room.roomId,
function(message) {
                showMessages(JSON.parse(message.body));
                });
        });
}
```

Miután a szoba kiválasztásra került, egy neptun kód megadása szükséges, ami alapján a felhasználó adatait lekérdezzük a H2 adatbázisból, a *3.3*. fejezetben említett /chat/user-by-neptun/{neptun} végpontról.



3.11. ábra Felhasználói felület – Neptun kód megadása

```
var senderUser=null;
function getNeptunCode() {
        var neptun = $("#neptun").val();
        $.ajax({
                url: "http://localhost:8082/chat/user-by-neptun?neptun="
+ neptun
            }).then(function(data) {
                senderUser = data;
            });
        console.log(senderUser)
}
```

Így a kliensnek már meg van a felhasználó és a szoba összes adata, amivel már össze tudja állítani az üzenetet és beküldeni a Kafka producer-be. A Message input mező kitöltése után a Send gomb segítségével küldjük el az üzenetet.

```
function sendMessage() {
        $.ajax({
            contentType: 'application/json',
            data: JSON.stringify({ "message": $("#message").val(), room,
senderUser }),
            dataType: 'json',
            type: 'POST',
            url: "http://localhost:8082/kafka/message?roomId=" +
room.roomId
        }).then(function(data) {
                console.log(data)
    });
}
```

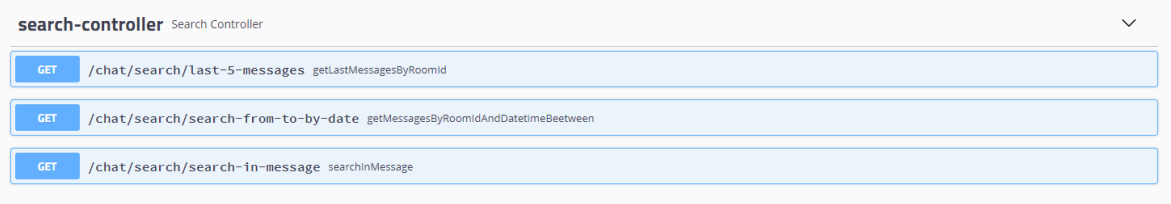
Miután bekerült az üzenet lementődik az Elasticsearch-be és a H2 adatbázisba, és mindenki számára látható, aki az adott szobába van becsatlakozva.

3.7. Keresés, szűrés implementálása

Fentebb a *3.6.* fejezetben, arról írok, hogy készült egy nagyon egyszerű felhasználói felület a Chat alkalmazáshoz, viszont a keresés és szűrési lehetőségek nincsennek a felületre kivezetve, ezek elérése, Swagger UI-ből, Postman-ből, parancssorból curl parancs segítségével, vagy esetleg böngészőből lehetséges. Fentebb a *2.5.2.2.* fejezetben bemutattam, hogyan lehet lekérdezni az Elasticsearch-ből. Ezeket a lekérdezéseket, kivezettem egy-egy végpontra. A következő kódsorokban a `SearchController` osztály látható, ami egy service rétegen keresztül meghívja a fent említett lekérdezéseket.

```
@GetMapping("/search/last-5-messages")
public ResponseEntity<List<ElasticsearchMessageEntity>>
getLastMessagesByRoomId(
@RequestParam @ApiParam(required = true, example = "1") Long roomId) {
        return new
ResponseEntity<>(searchService.getLastMessagesByRoomId(roomId),
HttpStatus.OK);
}

@GetMapping("/search/search-from-to-by-date")
public ResponseEntity<List<ElasticsearchMessageEntity>>
getMessagesByRoomIdAndDatetimeBeetween(
@RequestParam @ApiParam(required = true, example = "1") Long roomId,
```

```
@RequestParam @ApiParam(example = "2020-10-21T17:09:42.411", required =
true) String from,
@RequestParam @ApiParam(example = "2020-11-20T17:09:42.411", required =
true) String to) {
        return new
ResponseEntity<>(searchService.getMessagesByRoomIdAndDatetimeBetween(room
Id,LocalDateTime.parse(from), LocalDateTime.parse(to)), HttpStatus.OK);
}

@GetMapping("/search/search-in-message")
public ResponseEntity<List<ElasticsearchMessageEntity>> searchInMessage(
@RequestParam @ApiParam(required = true, example = "1") Long roomId,
@RequestParam @ApiParam(required = true, example = "szia") String search)
{
        return new ResponseEntity<>(searchService.searchInMessage(roomId,
search), HttpStatus.OK);
}
```



3.12. ábra Swagger UI – Keresés végpontok

Keresés üzenet szövegében, az egyes szobában:

GET localhost:8082/chat/search/search-in-message?roomId={roomId}&search={search}

Response: 200 - Ok


```
[
  {
    "messageId": "112f93e0-6a1f-44c4-98bb-bb4f15ea93d9",
    "room": {
      "roomId": 1,
      "roomName": "Diplomatervezés II. szoba",
      "subjectId": "GEIAL536ML",
      "subjectName": "Diplomatervezés II.",
      "description": "Chat szoba a diplomatervezéshez.",
      "messages": null
    },
    "senderUser": {
      "userId": 1,
      "neptun": "IZBTF9",
      "name": "Vécsi Ádám",
      "email": "vecsi1994@hotmail.com",
      "role": "STUDENT",
      "messages": null
    },
```

```
    "message": "szia",
    "dateTime": "2020-10-31T20:38:46.64995"
  },
    "senderUser": {
      "userId": 2,
      "neptun": "XYZ232",
      "name": "Kovács Béla",
      "email": "asdsa@asdas.com",
      "role": "STUDENT",
      "messages": null
    },
    "message": "szia helló",
    "dateTime": "2020-10-31T20:39:03.511322"
  }
]
```


Az összes üzenet lekérdezése szoba id alapján a megadott dátumok között.

GET localhost:8082/chat/search/search-from-to-by-date?from={fromDate}&roomId={roomId}&to={toDate}

Response:


```
[
  {
    "messageId": "75005895-cf64-4711-ac10-5405b38e17f3",
    "room": {
      "roomId": 1,
      "roomName": "Diplomatervezés II. szoba",
      "subjectId": "GEIAL536ML",
      "subjectName": "Diplomatervezés II.",
      "description": "Chat szoba a diplomatervezéshez.",
      "messages": null
    },
    "senderUser": {
      "userId": 1,
      "neptun": "IZBTF9",
      "name": "Vécsi Ádám",
      "email": "vecsi1994@hotmail.com",
      "role": "STUDENT",
      "messages": null
    },
    "message": "naaa",
    "dateTime": "2020-10-31T18:39:59.74611"
  },
  {
    "messageId": "112f93e0-6a1f-44c4-98bb-bb4f15ea93d9",
    "room": {
      "roomId": 1,
      "roomName": "Diplomatervezés II. szoba",
      "subjectId": "GEIAL536ML",
      "subjectName": "Diplomatervezés II.",
      "description": "Chat szoba a diplomatervezéshez.",
```

```
      "messages": null
    },
    "senderUser": {
      "userId": 1,
      "neptun": "IZBTF9",
      "name": "Vécsi Ádám",
      "email": "vecsi1994@hotmail.com",
      "role": "STUDENT",
      "messages": null
    },
    "message": "szia",
    "dateTime": "2020-10-31T20:38:46.64995"
  }
]
```

4. Összegzés

A dolgozat egy chat alkalmazás működését mutatja be Java nyelven, Spring boot keretrendszer, Apache Kafka és Elasticsearch segítségével.

A bevezetésben említettem, hogy munkám során már találkoztam ezekkel a technológiákkal, de ilyen részletességgel nem foglalkoztam még vele. Az első pár hétben az Apache Kafka és az Elasticsearch technológiákat tanulmányoztam, valamint a WebSocket működését.

Mint az előző fejezetekben olvasható, részletesen bemutatom, hogy hogyan is működnek ezek a technológiák példákkal együtt. Egy komplett Chat alkalmazást készítettem, ami tartalmaz egy Kafka Producer-t és egy Kafka Consumer-t, ami az üzenetek küldéséért és fogadásáért felelős, az üzenetek WebSocket-en mennek. Az üzeneteket Elasticsearch-ben tárolom egy message index-ben. Ahhoz, hogy az alkalmazás elinduljon először is kell, hogy fusson, a Zookeeper, Kafka és az Elasticsearch, ezek elengedhetetlenek, erre készítettem egy Docker compose fájlt, ami elindítja a szükséges környezetet.

Az alkalmazás induláskor létrehoz és csatlakozik egy Kafka topic-hoz. Csatlakozik és létrehoz indexet az Elasticsearc-ben. Létrehoz táblákat és feltölti default adatokkal Liquibase segítségével egy H2 embedded relációs adatbázisban. User-eket, Room-okat lehet létrehozni, lekérdezni, módosítani dedikált végpontokon. Létrehoz egy socket kapcsolatot, és csatlakozik hozzá. A létrehozott felhasználók be tudnak csatlakozni szobákba, amiket tantárgyakhoz rendelek. Készítettem egy felhasználói felületet is JQuery segítségével, de nem kifejezetten erről szól ez a projekt, inkább a backend oldalról. A felületen ki lehet választani szobát, ahol meg kell adni egy neptun kódot és már működik is a csevegés. Amikor elküldünk egy üzenetet, először letároljuk egy Elasticsearch index-ben, majd a H2 adatbázis message táblájában és csak ezek után érkezik meg a másik oldalra. A csevegési előzményekben való kereséshez az Elasticsearchet használom. A keresést nem vezettem ki a felületre, ezekre külön végpontokat hoztam létre.

A dolgozathoz készített alkalmazáshoz használtam Git verzió kezelő rendszert. Az alkalmazás forrás kódja elérhető a github.com-on. https://github.com/vecsiadam/diplomaterv.

5. Summary

The dissertation presents the operation of a chat application in Java using Spring boot framework, Apache Kafka and Elasticsearch.

I mentioned in the introduction that I had already encountered these technologies in my work, but I had not dealt with them in such detail yet. For the first few weeks, I studied Apache Kafka and Elasticsearch technologies.

As you can read in the previous chapters, I detail how these technologies works together with examples. I created a complete Chat Application that includes a Kafka Producer and a Kafka Consumer, which is responsible for sending and receiving messages, the messages go on a websocket. I store messages in Elasticsearch index. To run Chat application, you need to run Zookeeper, Kafka and Elasticsearch first, these are essential, I created a docker compose file to run the required environment.

The application starts and connects a Kafka topic at startup. Connects and create an Elasticsearch index. Creates tables and uploads default data to an H2 embedded relational database. Users and Rooms can be created, queried, modified. Creates a socket connection and connects to it. Created users can join rooms that I assign to subjects. I also created a user interface using JQuery, but that’s not what this project is about, it’s more about the backend side. In the interface you can select a room and enter a neptun code and the chat will work. When we send a message, it is first stored in an Elasticsearch index, then in the message table of the H2 database, and only then does it arrive at the other page from websocket. I use Elasticsearch to search chat history. I did not create search user interface, I created separate endpoints for these.

I used the Git version management system for the application. The source code for the application is available at github.com. https://github.com/vecsiadam/diplomaterv.

Irodalomjegyzék

[1] Spring hivatalos oldala - Introduction to Spring Framework
https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/overview.html

[2] Rahul Gunkar - Introduction to Spring Boot
https://www.geeksforgeeks.org/introduction-to-spring-boot/

[3] Serdar Yegulalp - What is Docker? The spark for the container revolution
https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html

[4] Docker hivatalos oldala
https://docs.docker.com/compose/

[5] Shay Shmeltzer - Introduction to Liquibase and Managing Your Database Source Code
https://dzone.com/articles/introduction-to-liquibase-and-managing-your-databa

[6] Berki Ádám - Apache Kafka
https://wiki.berki.org/index.php/Apache_Kafka#Kafka_bemutat.C3.A1sa

[7] Markus Gulden - Introduction to Kafka Connectors
https://www.baeldung.com/kafka-connectors-guide

[8] Guru99.com - NoSQL Tutorial: Types of NoSQL Databases, What is & Example
https://www.guru99.com/nosql-tutorial.html

[9] Frank Kane - Elasticsearch 7 and the Elastic Stack: In Depth and Hands On
https://www.udemy.com/course/elasticsearch-7-and-elastic-stack/

[10] Spring hivatalos oldala
https://spring.io/guides/gs/messaging-stomp-websocket/

[11] Dilip S. - Apache Kafka for Developers using Spring Boot
https://www.udemy.com/course/apache-kafka-for-developers-using-springboot

[12] Neha Narkhede, Gwen Shapira, and Todd Palino: Kafka: The Definitive Guide, Real-Time Data and Stream Processing at Scale. O'Reilly Media, Inc., 2017.

[13] Lovisa Johansson - Apache Kafka for beginners - What is Apache Kafka?
https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html

[14] Maverick - Apache Kafka Architecture and Components
https://programmertoday.com/apache-kafka-architecture-and-components/

[15] Elasticsearch hivatalos oldala
https://www.elastic.co/what-is/elasticsearch

[16] Tutorialspoint.hu - Spring Boot - Introduction
https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm

[17] Team LoginRadius - RDBMS vs NoSQL
https://www.loginradius.com/engineering/blog/relational-database-management-system-rdbms-vs-nosql/

[18] Opensource.com - What is Docker?
https://opensource.com/resources/what-docker

Linkek utoljára ellenőrizve: 2020.12.02.

1. melléklet

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.3.3.RELEASE</version>
        </parent>

        <artifactId>chat</artifactId>
        <name>Chat Application</name>
        <packaging>war</packaging>
        <description>Thesis project.</description>
        <version>1.0.0-SNAPSHOT</version>

        <properties>
                <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
                <maven.compiler.source>11</maven.compiler.source>
                <maven.compiler.target>11</maven.compiler.target>
                <java.version>11</java.version>
                <swagger.version>2.9.2</swagger.version>
        </properties>

        <dependencies>

                <dependency>
                        <groupId>org.springframework.kafka</groupId>
                        <artifactId>spring-kafka</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-
jpa</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.data</groupId>
                        <artifactId>spring-data-elasticsearch</artifactId>
```

```
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-
websocket</artifactId>
		</dependency>

		<dependency>
			<groupId>org.webjars</groupId>
			<artifactId>webjars-locator-core</artifactId>
		</dependency>

		<dependency>
			<groupId>org.webjars</groupId>
			<artifactId>sockjs-client</artifactId>
			<version>1.0.2</version>
		</dependency>

		<dependency>
			<groupId>org.webjars</groupId>
			<artifactId>stomp-websocket</artifactId>
			<version>2.3.3</version>
		</dependency>

		<dependency>
			<groupId>org.webjars</groupId>
			<artifactId>bootstrap</artifactId>
			<version>3.3.7</version>
		</dependency>

		<dependency>
			<groupId>org.webjars</groupId>
			<artifactId>jquery</artifactId>
			<version>3.1.1-1</version>
		</dependency>

		<dependency>
			<groupId>org.projectlombok</groupId>
			<artifactId>lombok</artifactId>
			<scope>provided</scope>
		</dependency>

		<dependency>
			<groupId>io.springfox</groupId>
			<artifactId>springfox-swagger2</artifactId>
			<version>${swagger.version}</version>
		</dependency>

		<dependency>
			<groupId>io.springfox</groupId>
			<artifactId>springfox-swagger-ui</artifactId>
```

```
                        <version>${swagger.version}</version>
                </dependency>

                <dependency>
                        <groupId>org.liquibase</groupId>
                        <artifactId>liquibase-core</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                        <exclusions>
                                <exclusion>

        <groupId>org.junit.vintage</groupId>
                                        <artifactId>junit-vintage-
engine</artifactId>
                                </exclusion>
                        </exclusions>
                </dependency>

                <dependency>
                        <groupId>com.h2database</groupId>
                        <artifactId>h2</artifactId>
                        <scope>runtime</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>

        <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-
plugin</artifactId>
                        </plugin>
                </plugins>
        </build>
</project>
```

2. melléklet

application.properties

```
server.port=8082

#DB
spring.h2.console.enabled=true

spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

#hibernate
spring.jpa.hibernate.ddl-auto=none

#liquibase
spring.liquibase.enabled=true
spring.liquibase.change-log=classpath:db/liquibase-change-log.xml

#Kafka producer
spring.kafka.producer.bootstrap-servers=localhost:9092
spring.kafka.producer.key-
serializer=org.apache.kafka.common.serialization.LongSerializer
spring.kafka.producer.value-
serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.admin.properties.bootstrap.servers=localhost:9092

#Kafka consumer
spring.kafka.consumer.bootstrap-servers=localhost:9092
spring.kafka.consumer.key-
deserializer=org.apache.kafka.common.serialization.LongDeserializer
spring.kafka.consumer.value-
deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.group-id=chat-listener-group

#Elasticsearch
chat.elasticsearch.url=localhost:9200
```

CD melléklet tartalma

A dolgozat PDF változatát a *diplomamunka.pdf* fájlban található. A dolgozat szerkeszthető formátumban is megtalálható *diplomamunka.docx* néven.

A CD tartalmaz

- két db docker compose fájlt:
    - docker-compose_chat.yml
    - docker-compose.yml
- Dockerfile
- README.md
- /war/chat-1.0.0-SNAPSHOT.war
- /backend/chat/*

Találunk egy *README.md* fájlt, ami egy részletes leírást tartalmaz az alkalmazás és a docker container-ek indításáról. A */backend/chat/* mappában találjuk az alkalmazás teljes forráskódját. A */war* mappában találjuk *chat-1.0.0-SNAPSHOT.war*-t, ami a legutolsó build-elt war fájl. Ennek a segítségével tud a *Dockerfile* egy docker image-et készíteni a Chat alkalmazásból. A *docker-compose_chat.yml* elindítja az alkalmazást az összes függőségével docker container-ben, A *docker-compose.yml* csak az alkalmazás függőségeit indítja el docker container-ben.