 Miskolci Egyetem

Gépészmérnöki és Informatikai Kar

Mérnökinformatikus mesterképzési szak

Alkalmazásfejlesztői szakirány

Oktatást segítő alkalmazás kibővítése felhasználók közötti csevegéssel és részletes kereséssel

Diplomamunka

Vécsi Ádám

IZBTF9

2020

|  |  |  |  |
| --- | --- | --- | --- |
| **MISKOLCI EGYETEM**  **GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR** | | | |
| **Alkalmazott Informatikai Intézeti Tanszék**  **H-3515 Miskolc-Egyetemváros** | | |
|  | Mérnökinformatikus mesterképzési szak  Alkalmazásfejlesztői szakirány  **Diplomamunka azonosító: IAL/IZBTF9/MSc/2021**  **Intézmény azonosító: FI 87515** |  |

**DIPLOMAMUNKA FELADAT**

|  |  |
| --- | --- |
| **VÉCSI ÁDÁM** | |
| MSc mérnökinformatikus jelölt részére | |
| A tervezés tárgyköre: | | **Alkalmazás készítés** | |
| A diplomaterv címe: | | **Oktatást segítő alkalmazás kibővítése felhasználók közötti csevegésel és részletes kereséssel** | |

**A feladat részletezése:**

* TODO

|  |  |  |
| --- | --- | --- |
| Tervezésvezető: | **Dr. Krizsán Zoltán, egyetemi docens** | **Alkalmazott Informatikai Intézeti Tanszék** |
| Konzulens: | **Dr. Krizsán Zoltán, egyetemi docens** | **Alkalmazott Informatikai Intézeti Tanszék** |

|  |  |
| --- | --- |
| Diplomaterv kiadásának időpontja: | **2020.09.18.** |
| Diplomaterv beadásának határideje: | **2020.11.20.** |

Miskolc, 2020.09.18.

**Dr. Nehéz Károly**

tanszékvezető, egyetemi docens

1. A diplomamunka beadható:

............................... ................................. ...........................................  
 dátum tervezésvezető konzulens

2. A diplomamunka ...............................szövegoldalt,  
 ...............................db nyomtatott mellékletet,  
 ...............................db CD / DVD lemezt,  
 ...............................db egyéb mellékletet  
 tartalmaz.

................................. ....................................................  
 dátum tervezésvezető

3. A diplomamunka bírálatra bocsátható

nem bocsátható  
  
A bíráló neve, beosztása, munkahelye:.……….........................................................................................

…………………….................................……….......................................................................................

…….................... ..........................................  
 dátum tanszékvezető

4. A diplomamunka osztályzata:  
  
A bíráló javaslata: ...............................................  
  
A tanszék javaslata: ...............................................  
  
**A ZVB döntése:** ...............................................

miskolc, ......................................................

..................................................

**ZVB** elnök

**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, 2020.11.11.

Hallgató aláírása

Tartalomjegyzék

[1. Bevezetés 5](#_Toc55852080)

[2. Chat alkalmazások vizsgálata?? 6](#_Toc55852081)

[3. Chat alkalmazáshoz használt technológiák bemutatása 7](#_Toc55852082)

[3.1 Spring boot keretrendszer 7](#_Toc55852083)

[3.2 Docker TODO: Ez még nincs kész 7](#_Toc55852084)

[3.3 Apache Kafka elosztott streaming platform 9](#_Toc55852085)

[3.3.1 Apache Kafka Architektúra 9](#_Toc55852086)

[3.3.2 Kafka Broker 11](#_Toc55852087)

[3.3.2.1 Topic és partíciók 11](#_Toc55852088)

[3.3.3 Consumer csoportok 13](#_Toc55852089)

[3.4 Elasticsearch 14](#_Toc55852090)

[3.4.1 Elasticsearch műveletek 15](#_Toc55852091)

[3.4.1.1 Index létrehozás 15](#_Toc55852092)

[3.4.1.2 Egyszerű lekérdezések 17](#_Toc55852093)

[3.4.2 Kibana 18](#_Toc55852094)

[4. Chat alkalmazás implementáció 20](#_Toc55852095)

[4.1 Felhasználók, szobák létrehozása, módosítása, lekérdezése, tárolása 20](#_Toc55852096)

[4.2 Apache Kafka integrálódás 26](#_Toc55852097)

[4.2.1 Topic létrehozása és csatlakozás hozzá 26](#_Toc55852098)

[4.2.2 Producer 26](#_Toc55852099)

[4.2.3 Consumer 28](#_Toc55852100)

[4.3 Elasticsearch integrálódás 29](#_Toc55852101)

[4.3.1 Index létrehozása 30](#_Toc55852102)

[4.3.2 Üzenetek mentése és lekérdezése 31](#_Toc55852103)

[4.4 Websocket és JQuery használata 32](#_Toc55852104)

[4.5 Keresés, szűrés implementálása 38](#_Toc55852105)

[5. Összegzés 41](#_Toc55852106)

[Summary 42](#_Toc55852107)

[Irodalomjegyzék 43](#_Toc55852108)

[Mellékletek 44](#_Toc55852109)

# Bevezetés

A mai világban az egyik legjobban használatos 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, 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. De 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.

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

# Chat alkalmazások vizsgálata??

TODO: Nem tudom mit kellene ide írni?

# Chat alkalmazáshoz használt technológiák bemutatása

## Spring boot keretrendszer

A Spring-et széles körben használják skálázható alkalmazások létrehozására. Spring MVC, a Spring egyik legjobban használt modulja. De a Spring projektek fő hátránya, hogy a konfiguráció valóban időigényes és bonyolult az új fejlesztők számára. Az alkalmazás fejlesztése időt vesz igénybe, ha még a fejlesztő nem ismeri jól a Spring-et. Ennek megoldása a Spring Boot.

A Spring boot a Spring-re épül, tartalmazza az összes tulajdonságát. Manapság a fejlesztők kedvence ez a keretrendszer mert, gyors gyártásra kész környezet biztosít, amely lehetővé teszi a fejlesztők számára, hogy közvetlenül a logikára koncentráljanak ahelyett, hogy a konfigurációval és a beállításokkal küszködnének. A Spring Boot egy Microservice alapú keretrendszer.

Az az osztály, ami tartalmazza a @SpringBootApplication annotációt és a main metódust 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);

}

}

## Docker TODO: Ez még nincs kész

A Docker egy olyan eszköz, amelyet az alkalmazások létrehozása, telepítése és futtatása konténerek segítségével könnyebbé tesz. A tárolók lehetővé teszik a fejlesztők számára, hogy csomagolják az alkalmazást az összes szükséges alkatrésszel, például könyvtárakkal és más függőségekkel, és egy csomagként telepítsék. Ezzel a tárolónak köszönhetően a fejlesztő megnyugodhat, hogy az alkalmazás bármely más Linux gépen futni fog, függetlenül attól, hogy a gépnek milyen testreszabott beállításai lehetnek, amelyek eltérhetnek a kód írásához és teszteléséhez használt géptől.

Bizonyos értelemben a Docker egy kicsit olyan, mint egy virtuális gép. De a virtuális géppel ellentétben, a teljes virtuális operációs rendszer létrehozása helyett, a Docker lehetővé teszi az alkalmazások számára, hogy ugyanazt a Linux-kernelt használják, mint az általuk futtatott rendszer, és csak olyan alkalmazásokat igényel, amelyek még nem futnak a gazdagépen. Ez jelentős teljesítménynövekedést eredményez és csökkenti az alkalmazás méretét.

Ami fontos, a Docker nyílt forráskódú. Ez azt jelenti, hogy bárki hozzájárulhat a Docker fejlesztéséhez és kibővítheti saját igényeinek kielégítéséhez, ha további funkciókra van szüksége, amelyek már nem kaphatók.

Kinek szól Docker?

A Docker egy olyan eszköz, amelyet mind a fejlesztők, mind a rendszergazdák számára előnyössé alakítottak ki, így számos DevOps (fejlesztő + művelet) eszköztár részévé teszi. A fejlesztők számára ez azt jelenti, hogy a kód írására összpontosíthatnak, anélkül, hogy aggódnának a rendszer mellett, amelyen végső soron futni fog. Ez azt is lehetővé teszi számukra, hogy előnyt szerezzenek, ha alkalmazásuk részeként a Docker konténerben futtatásra tervezett több ezer program egyikét használják. Az operatív személyzet számára a Docker rugalmasságot biztosít és potenciálisan csökkenti a szükséges rendszerek számát a kis lábnyoma és az alacsonyabb rezsi miatt.

A konténerek megértése

A konténerek három szoftverkategóriát igényelhetnek:

• Építő: a konténer építéséhez használt technológia.

• Motor: a konténer futtatásához használt technológia.

• Orchestration: sok konténer kezeléséhez használt technológia.

A konténerek használatának egyik vonzereje az, hogy képesek kecsesen meghalni, és igény szerint újjáéledni. Akár a konténer megszűnését összeomlás okozza, akár azért, mert egyszerűen nincs rá szükség, ha a kiszolgáló forgalma alacsony, a konténerek olcsók elindítani, és úgy vannak kialakítva, hogy zökkenőmentesen jelennek meg és tűnjenek el. Mivel a konténerek elmúlóak, és új példányokat hoznak létre, ahányszor csak szükséges, várható, hogy a megfigyelésüket és kezelésüket nem az ember valós időben végzi el, hanem automatizált módon.

A Linux-konténerek elősegítették a nagy rendelkezésre állású számítástechnika hatalmas változását, és számos eszközkészlet van a szolgáltatások (vagy akár az egész operációs rendszer) konténerekben történő futtatásához. A Docker az egyik lehetőség a sok közül, amelyet az Open Container Initiative (OCI) meghatároz, az ipari szabványügyi szervezet, amely az innovációt ösztönzi, miközben elkerüli a gyártók bezáródásának veszélyét. Az OCI-nak köszönhetően választása van a tárolóeszköz-lánc kiválasztásakor, beleértve a Docker, az OKD, a Podman, az rkt, az OpenShift és másokat.

Ha úgy dönt, hogy a szolgáltatásokat konténerekben futtatja, akkor valószínűleg szüksége van a tárolók tárolására és kezelésére tervezett szoftverre. Ezt széles körben konteiner hangszerelésnek nevezik. A Kubernetes biztosítja a konténerek hangszerelését a konténerek különféle futási idejéhez.

## Apache Kafka elosztott streaming platform

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. És ezért 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.

### Apache Kafka Architektúra

Egy Kafka architektúra legalább egy Kafka szerverből (bróker) áll, ami a konfigurációját kötelezően a Zookeeper nevű elosztott konfigurációs management rendszerben tárolja. A Kafka Broker-hez csatlakoznak a Producer-ek és Consumer-ek. A Kafka Cluster-ben úgynevezett Topic-ok találhatók.



*3.1. ábra Kafka architektúra*

A producer mindig egy dedikált Topic-ra ír, és a Consumer(ek) mindig egy dedikált Topic-ról olvasnak, tehát a Topic az a logikai egység, ami egy producer-consumer páros számára az üzeneteket tárolja és továbbítja. Mikor elindítunk egy Kafka példányt, akkor valójában egy Kafka Broker-t indítunk el. Ha Producer-ek mindig egy Broker-hez csatlakoznak. A teljes konfiguráció zookeeper-ben van tárolva. A Zookeeper tudja értesíteni a klienseket, ha a konfiguráció változik, ezért hamar elterjed a hálózaton a változás.

Nem csak Producers API-n és Consumers API-n lehet adatokat ki és berakni egy Kafka Topic-ba. A Connect API segítségével, külső fájlrendszerekről, adatbázisokból, Elasticsearch-ből tudunk kiszedni és betenni adatot egy-egy topic-ba.

**Connect API:**

*Source Connector:* adatbázisból, fájlrendszerből, elasticsearchból kiszedik a szükséges adatot és beküldik a Kafka topic-ba.

*Sink Connector:* Ellentéte a Source Connectornak, egy Kafka Topic-ra feliratkozik és menti le az adatokat.

**Kafka streams (Streams API):** Fogja az adatot kiszedi a kafkából átalakítja, valamilyen formába majd visszarakja.

### Kafka Broker

Egy Kafka Cluster több Kafka broker-ből áll. Fentebb említettem Zookeeper, aki azért felelős, hogy a cluster-ban lévő brokerek megfelelően működjenek. A Kafka broker-ben találhatók a Topic-ok már fentebb volt róla pár szó, egy broker-ben több Topic is lehet. A következő szekcióban a Kafka Topic-ról fogok írni.

#### Topic és partíciók

A Topic-ot úgy kell elképzelni, mint ha egy táblát egy adatbázisban. A működés a *3.2* 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 kiszedi a Topic-ból.



*3.2. ábra Üzenet küldése Kafka Topic-ba*

A Partíció az, ahol egy adott üzenet él a Topic-ba, minden Topic-nak egy vagy több Partíciója lehet. Ezek a partíciók rendezett rekordok listája. A Producer tudja, hogy melyik üzenet melyik Partícióban van. A Partíciók függetlenek egymástól és minden rekordnak van egy szekvencia száma, amit Offset-nek nevezzük. Az összes rekordot egy központi logfájlban is tárol. 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 brokereken fognak létrejönni. Minden partíciónak lehet egy vagy több replikája, amik biztonsági másolatok.

A képen szöveg látható

Automatikusan generált leírás

*3.3. ábra Topic partíciókra bontása*

Minden partíció új üzenete mindig a partíció végére íródik. A partíció elejétől számoljuk az üzenetek sorszámát, ezt hívjuk offset-nek. Mikor egy consumer kiolvas egy üzentet, attól az még ott marad a partícióba egészen addig, amíg len nem jár, alapértelmezetten ez egy nap. Tehát ez eltér a hagyományos sor kezeléstől. A Kafka nyilvántartja, hogy melyik consumer egy adott partícióban melyik offset-nél tartott. Ezt egy speciális topic-ban tartja nyilván: Ha újra is indul a világ, akkor is tudni fogják a consumer-ek, hogy hol tartottak, és onnan folytatják.

Ez Kafka üzenetnek két tagja van, egy kulcs és egy érték. A kulcs opcionális, de ha nem küld a Producer kulcsot nem garantálható az üzenetek sorrendje. A Producer és a Topic között van egy Partition réteg (Partitioner), ami megkapja az üzenetet és megnézi van-e 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 *3.4*.-es ábrán.



*3.4. ábra Üzenet küldése kulcs nélkül*

Ha talál kulcsot az üzenet mellett a kulcsokat 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 *3.5.*-ös ábrán látható erre egy példa.

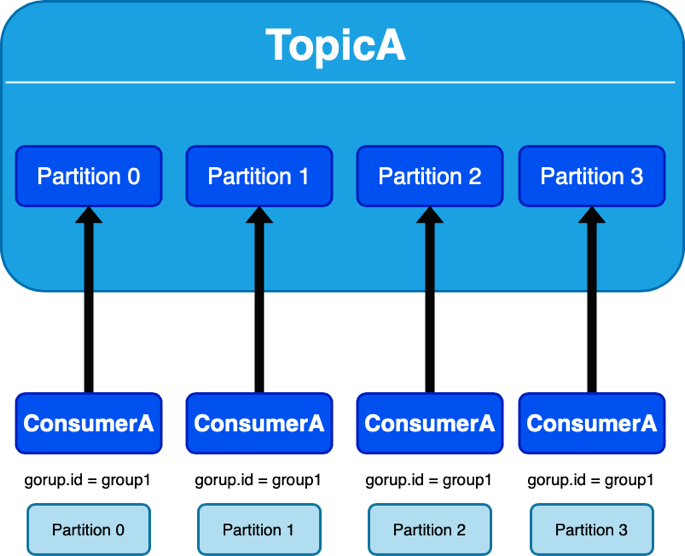


*3.5. ábra Üzenet küldése kulccsal*

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

### Consumer csoportok

A comsumer-eket úgynevezett consumer-group-okba szervezzük az azonosítójuk szerint. Egy csoport mindig ugyan azon topic üzeneteit olvassa, de minden egyes consumer a csoportban más és más partícióból. Minden partíció csak egy consumer-hez rendelhető hozzá egy csoporton belül.



*3.6. ábra Consumer csoportok*

De ha nincs annyi consumer a csoportban, mint ahány partíció, akkor egy consumer több partíciót is fog olvasni. Viszont, ha több consumer van, mint partíció egy csoportban, akkor bizonyos consumer-ek mindig idle állapotban lesznek. Minden csoporton belül van egy vezető consumer, általában az, aki először csatlakozott. Ő teríti a többieknek a cluster információkat.

## 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.



*3.7. ábra Elasticsearch architektúra*

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

### Elasticsearch műveletek

Az adatok Elasticsearch-be juttatása REST-en keresztül történik. A Kibana egy adatmegjelenítő és kezelő eszköz hozzá erről részletesebben a 3.4.2 szekcióban írok. Mivel REST-en lehet létrehozni, törölni és adatokat juttatni az elasticsearch-be, így egyszerű CURL paranccsal command line segítségével is kezelhet. A következő fejezetekben egy pár szóban erről szeretnék írni.

#### Index létrehozás

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

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

curl -XPUT -H "Content-Type: application/json" localhost:9200/message\_10 -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 index 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.

#### 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

}

}

]

}

}

### Kibana

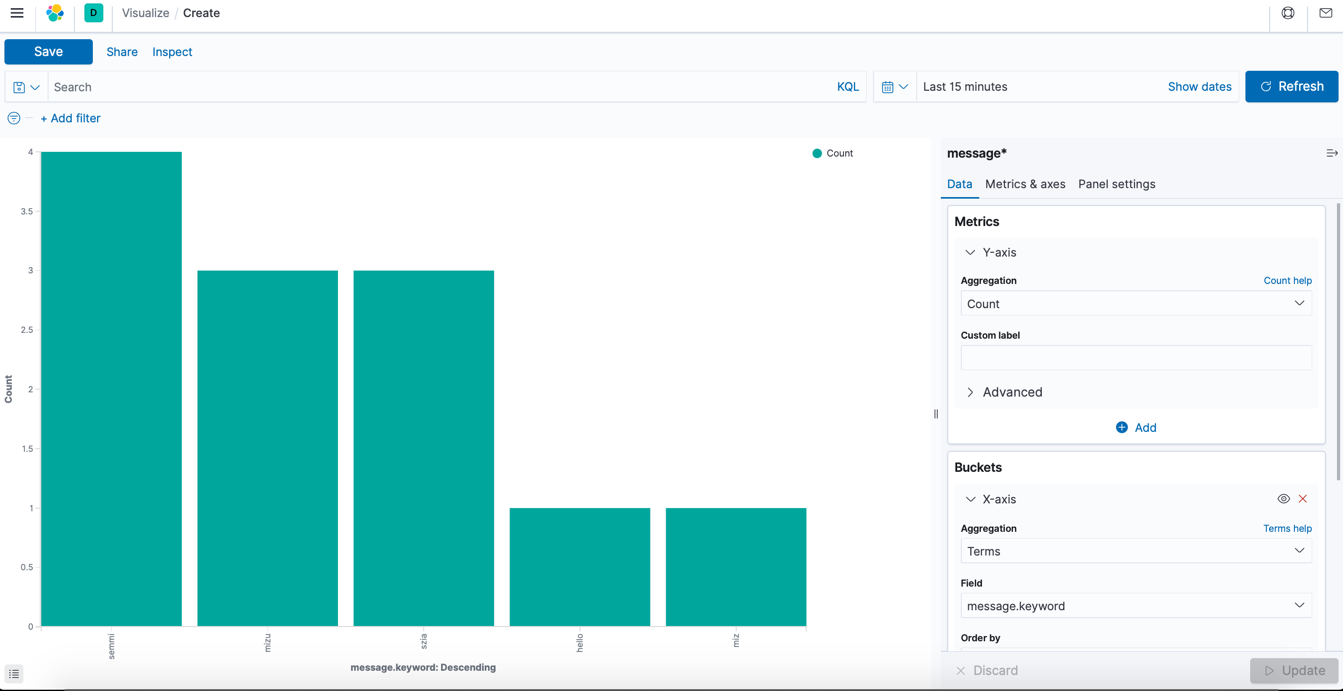
A Kibana egy adatmegjelenítő és -kezelő eszköz az Elasticsearch számára, amely valós idejű hisztogramokat, vonaldiagramokat, kördiagramokat biztosít. Ez lehetővé teszi az Elasticsearch adatok egyszerű megjelenítését. A Kibana remek módja az index-en belüli keresésére és megjelenítésére egy hatékony és rugalmas felhasználói felülettel. Fontos hátránya azonban, hogy minden megjelenítés csak egyetlen index/index mintázat alapján működhet. Tehát ha szigorúan eltérő adatokkal rendelkező indexekkel rendelkezik, akkor mindegyikhez külön megjelenítést kell létrehoznia.

A képen szöveg látható

Automatikusan generált leírás

*3.8. ábra Kibana - Dev Tools*

A Visualize menüpont alatt könnyedén készíthetők diagrammok, hisztogramok, kördiagramok a *3.9*-es 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.



*3.8. ábra Kibana - Visualize*

# Chat alkalmazás implementáció

Ahhoz, hogy az alkalmazás működőképes legyen szükség van arra, hogy megteremtésük az előfeltételeket. Szükség van Elasticsearch-re, Apache Kafka broker-re, Zookeeper-re és végül nem feltétlenül szükséges, de a könnyű tesztelhetőség kedvéért egy Kibana-ra is. Mivel bonyolult lenne ezt mind telepíteni, konfigurálni úgy döntöttem, hogy készítek hozzá egy Docker compose fájlt. Ami ezt mind összerakja a fejlesztéshez. A docker-compose up -d paranccsal futtatható a compose fájl. Először lehúzza azokat a Docker image-ket, amikre szükség van, majd indítja őket.

## Felhasználók, szobák létrehozása, módosítása, lekérdezése, tárolása

Ahhoz, hogy a szobákat, felhasználókat tudjuk tárolni szükség van több függőségre is.

|  |  |
| --- | --- |
|  | <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 adatbázist használok az adatok tárolására, ez egy olyan adatbázis, 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 init adatok. A Liquibase egy nagyon hasznos függőség, xml, yml vagy json segítségével írhatók le az adatbázis műveletek pl: tábla létrehozás, oszlop törlés, módosítás stb.Azért is jó ez mert nem nekünk kell létrehozni, módosítani törölni, hanem az alkalmazás induláskor ez megtörténik és minden ilyen műveletet egy úgynevezett changeSet-be kell felvenni, amik naplózva tarolódnak az adatbázisban.

A Példában itt kettő changeSet látható, a user tábla létrehozása, és egy sql file beolvasása. Ebben az data.sql-ben 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.

#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 érhető el az adatbázis, milyen felhasználóval, jelszóval, érhető el. Liquibase-t is engedélyezni kell, és meg kell adni, hogy hol található az xml 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.

Felhasználókhoz kapcsolódó végpontok:

*4.1. 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:

*4.2. 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."

}

]

## Apache Kafka integrálódás

Ahogy fentebb is említettem, ahhoz, hogy az alkalmazás elinduljon szükséges egy Zookeeper és egy Kafka Broker ezt a Docker segítségével nagyon egyszerűen indítható. De ahhoz, hogy az alkalmazás tudjon kapcsolódni a Kafka broker-hez 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 dependency-re van szükség:

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

</dependency>

### 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, 20 partícióval és 1 replikával.

**@Configuration**

**public** **class** **KafkaTopicConfig** {

**@Bean**

**public** NewTopic **createTopic**() {

**return** TopicBuilder.name("chat-rooms")  
.partitions(**20**).replicas(**1**).build();

}

}

### Producer

Kafka producer konfigurációjában meg kell adnunk, a Kafka Broker 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**

*4.3. 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**);

}

### Consumer

Ahhoz, hogy ezeket az üzeneteket fogadjuk, szükség van egy Consumer-re is. A következő beállítások szükségesek az application.properties-be. Itt is meg kell adni, a Kafka broker 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 nagyon egyszerű a **@KafkaListener** annotációnak megadjuk, hogy milyen topic az, poll-ozunk és várjuk, hogy üzenet kerüljön bele. Ha talál üzenetet kiveszi és tovább küldi a messageSaveService-nek, ami sendAndSaveMessage metódus 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());

}

## Elasticsearch integrálódás

Az alkalmazás indulásához és helye működéséhez elengedhetetlen az elasticsearch, mivel itt is tárolódik az üzeneteket, igaz tárolódik H2 adatbázisban is, de a keresés és szűrési lehetőségek az elasticsearch-öt használják. Az alkalmazás konfigurációban, azaz 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 dependency-t 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 RestHighLevelClientet 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 segítségéve.

**@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());

}

}

### Index létrehozása

Az index létrehozása annotációkkal történik, egy entititást hoztam létre hozzá, ami az üzenetek tárolásáért felelős. A **@Document** annotációnál szükséges megadni, hogy milyen indexet szeretnénk létrehozni az elasticsearchben, itt lehet megadni, hogy hány darab primary shards és hány darab replika legyen. Itt ebben az esetben egy shard és egy replica elég lesz. Nincs egzakt mondás arra, hogy mennyi az ideális shard és replika, függ attól, hogy hány milyen terhelése van az elastic-nak, 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. Itt látható, hogy teljes egészében 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;

}

### Üzenetek mentése és lekérdezése

Ami nagyon egyszerűvé teszi az elasticsearch-el való integrálódást, hogy teljesen úgy működik, mint bármelyik repository hívás jpa segítségével. Egy repository interface szükséges a mentéshez és lekérdezéshez, ami extendál egy ElasticsearchRepository-t és megadni, hogy milyen entitásokat használ és milyen id-ja van. Először a lekérdezéseket nézzük, három darab query látható lentebb. 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 boot és 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\"}},\"filter\":{\"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 save metódust, amivel letárolódik az adott üzenet az elasticsearch-ben. A következőkben bemutatom a saveAndMessage 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);

}

## 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>

Először a websocket szerver oldali konfigurálásáról és implementálásáról, 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 broker 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 ellátott metódusokra. Ezzel az előtaggal fogják meghatározni az összes üzenet-hozzárendelést.

A **registerStompEndpoints** metódus regisztrálja a "/chat-websocket" végpontot, lehetővé téve a SockJS tartalék opcióit, hogy alternatív szállítások is használhatók legyenek, ha a WebSocket nem érhető el. A SockJS kliens megpróbál csatlakozni a "/chat-websocket"-hez.

**@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-jú 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.

*4.4. ábra Felhasználói felület*

Ehhez a 4.3-as szekcióban említett /chat/rooms végpontot hívjuk, ami a getRooms() metódust hívja és vissza adja az összes H2 adatbázisban létező szobát.



*4.5. 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 5 üzenet betöltődik az adatbázisból.

*4.6. 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 4.3-as szekcióban említett /chat/user-by-neptun/{neptun} végpontról.

*****4.7. 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 field 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.

## Keresés, szűrés implementálása

Fentebb a 4.5-ös 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 4.4.2-es fejezetben bemutattam, hogyan lehet lekérdezni az Elasticsearch-ből. Ezeket a lekérdezéseket, kivezettem egy-egy végpontra. A következő kódsorakban 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(roomId,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);

}

A SearchController osztály sawagger-ből is hívható.

*4.8. 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"

}

]

# Összegzés

# Summary

# Irodalomjegyzék

TODO: szerző és cím után nézni! linkek ellenőrzése

[1] https://opensource.com/resources/what-docker

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

[3] https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html

[4] https://wiki.berki.org/index.php/Apache\_Kafka#Kafka\_bemutat.C3.A1sa

[5] https://programmertoday.com/apache-kafka-architecture-and-components/

[6] https://www.elastic.co/what-is/elasticsearch

[7] https://programmertoday.com/apache-kafka-architecture-and-components/

[10] <http://hilite.me/>

[11] <https://spring.io/guides/gs/messaging-stomp-websocket/>

[12] <https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm>

[13] https://www.geeksforgeeks.org/introduction-to-spring-boot/

# Mellékletek

CD melléklet tartalma