Multidiszciplináris tudományok, 10. kötet. (2020) 2 sz. pp. 437-441 **https://doi.org/10.35925/j.multi.2020.2.49**

MONOLITIKUS ALKALMAZÁS KIBŐVÍTÉSE ÚJABB FUNKCIÓKKAL

Vécsi Ádám
MSc hallgató, Miskolci Egyetem, Informatikai Intézet, Általános Informatikai Tanszék
3515 Miskolc, Miskolc-Egyetemváros, e-mail: vecsi@iit.uni-miskolc.hu

Krizsán Zoltán
docens, Miskolci Egyetem, Informatikai Intézet, Általános Informatikai Tanszék
3515 Miskolc, Miskolc-Egyetemváros, e-mail: krizsan@iit.uni-miskolc.hu

Absztrakt

A cikk egy általános monolitikus alkalmazás kibővítésének egyik optimálisabb módját mutatja be. Célunk, hogy egy régi monolitikus rendszert minimális változtatással bővítsünk ki új képességekkel. Javaslatunk, hogy az alap rendszerbe csak HTTP kliens végpontokat alakítsunk ki, ami az új részek (lehetőleg microszervizek) elérését biztosítsák. Nagyobb flexibilitás érdekében az üzenetsorok alkalmazása javasolt. A cikkben bemutatjuk a fent említett minták alkalmazását, csevegés és keresés funkciókkal bővítünk ki egy már meglévő alkalmazást.

Kulcsszavak: *alkalmazás fejlesztés, mikroszerviz, Java, Web szolgáltatások*

Abstract

This article suggests a method for extension of monolithic web application. Our goal is to improve the legacy system with new feature but with minimal modification of base system. Out suggestion is creating some HTTP client in the base system to achieve the new system (micro services in most cases). In order to high flexibility the application of message queue is suggested. Moreover this article presents the application of this theoretical proposes the extension with chat and search feature of existing application.

Keywords: *application development, micro services, Java, Web services*

1. Bevezetés

Alkalmazások fejlesztésénél örökös probléma, hogy hogyan tudjuk kibővíteni a régebben megírt alkalmazásunkat, hogyan tudjuk a legacy, technológiailag elavult részeket kiterjeszteni, frissíteni. Azon alkalmazások esetén, amelyek már microszervíz architektúra [1], [3], [4], [9], [10], [11] alapján készültek könnyebb a helyzet, mert ott lehet szervizenként átalakítani, újakat bevezetni. A monolitikus alkalmazások esetében azonban csak egy nagy kódbázis van, ahol nem tudunk lényegi részeket átírni kis erőbefektetéssel. A monolitikus alkalmazás egyetlen egységként épül fel. Az vállalati alkalmazások általában három részből állnak:

- Adatbázis réteg: sok táblából állhatnak, általában egy relációs adatbázis-kezelő rendszerben.
- Szervíz réteg**:** végrehajt bizonyos üzleti logikákat, pl adatokat fog lekérni és frissíteni az adatbázisból, és elküldi a felhasználói felületnek.
- Felhasználói felület**:** HTML-oldalakból és/vagy egy böngészőben futó JavaScript-ből áll. A kliens oldali alkalmazás kezeli a HTTP-kéréseket, és továbbítja az üzleti Szervíz felé.

Előnye a monolitikus alkalmazásoknak [2]:
- Egyszerű telepíteni. Csak át kell másolnia a csomagot alkalmazásszerverre.
- Egyszerű vízszintes skálázás több példány futtatásával a load-balancer mögött (bár ilyenkor az állapotot szinkronizálni kell).

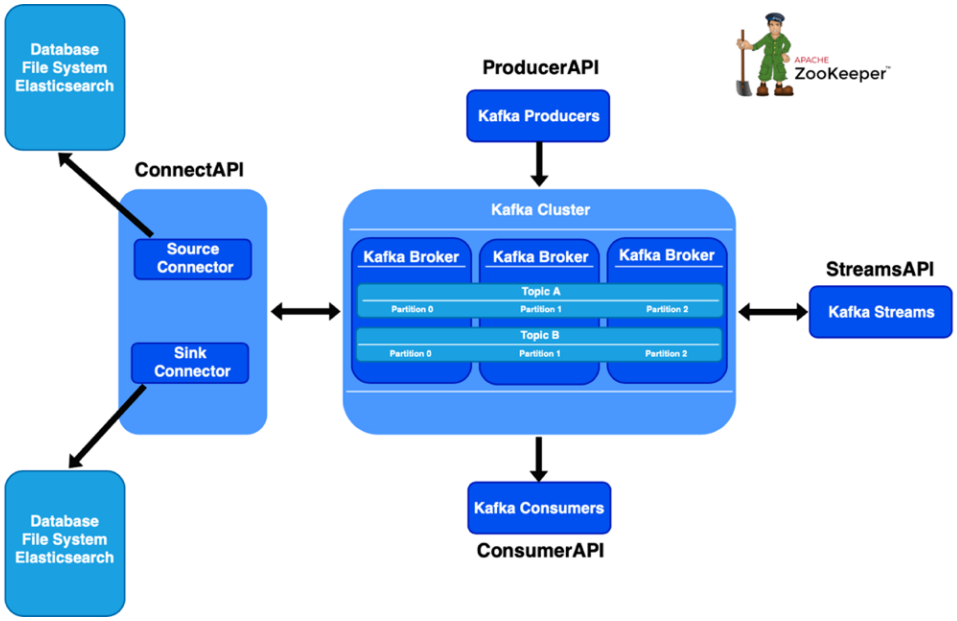
Hátrányai:
- Az alkalmazás túl nagy és összetett ahhoz, hogy teljes mértékben megértsük és gyorsan és helyesen változtassunk rajta.
- Az alkalmazás mérete lassíthatja az indítási időt.
- Minden frissítésnél újra kell építeni és telepíteni a teljes alkalmazást.
- A monolit alkalmazások nehezen skálázhatók abban az esetben, ha a különböző moduloknak ellentmondó erőforrásigényük van.
- A monolit alkalmazások másik problémája a megbízhatóság. Bármely modul hibája (pl. memóriahiány) potenciálisan leállíthatja az egész alkalmazást.
- A monolit alkalmazások akadályozzák az új technológiák alkalmazását. Mivel a keretrendszer vagy a nyelv változásai az egész alkalmazást érintik, ez nagyon költséges időben és anyagilag.

A régi rendszerünk adott, szeretnénk tehát új alrendszereket integrálni, ezzel bővíteni a rendszerünk funkcionalitását. Nem szeretnénk a régi monolitikus megoldást tovább fejleszteni, és a régi rendszer átírása microszervíz architektúrára is túl erőforrás igényes lenne. Ezért azt javasoljuk, hogy a régi rendszerbe egy illesztőt kell írni, ami integrálja az új részeket. A minimális, minél lazább függőség miatt a HTTP REST végpontok alkalmazását javasoljuk, ha a kérésnek szinkronnak kell lennie, és az esemény alapú megközelítést, ha a válaszidő nem kritikus. Az esemény sorok alkalmazása lehetővé teszi, hogy a feldolgozó logika tetszőleges ütembe dolgozza fel a kéréseket, és ez a feldolgozó rész implementációs nyelve különböző is lehet. Manapság az egyik legnépszerűbb, üzembiztos eseménykezelő rendszer a Kafka, melyet a következő alfejezetben ismertetünk. Javasoljuk továbbá a Java Spring boot használatát a kiegészítő funkciók implementálására, mert a Kafka integráció minimális kódolást igényel, hiszen annotációk segítségével megvalósítható.

1.1. Kafka

A Kafka elosztott rendszer szerverekből és kliensekből áll (bővebben [5], []), amelyek nagy teljesítményű TCP hálózati protokollon keresztül kommunikálnak és alapvetően üzenet sorokat használnak.
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.

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. Ez a szerviz folyamatosan ellenőrzi a brókerek állapotát, és 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 topic-okra.



1. ábra. *Kafka eseménykezelő rendszer általános felépítése*

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 [3].

Nem csak Producers API-n és Consumers API-n lehet üzeneteket beküldeni és kiszedni egy topic-ból, hanem a Connect API segítségével [6], külső fájlrendszerekről, adatbázisokból, Elasticsearch-ből is 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. Érdemes ezeket használni, mert csak be kell állítani, és nem kell illesztő programokat írni. A Kafka Connect egy keretrendszer a Kafka és külső rendszerek, például adatbázisok, kulcsérték-tárolók, keresési indexek és fájlrendszerek összekapcsolására.

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.

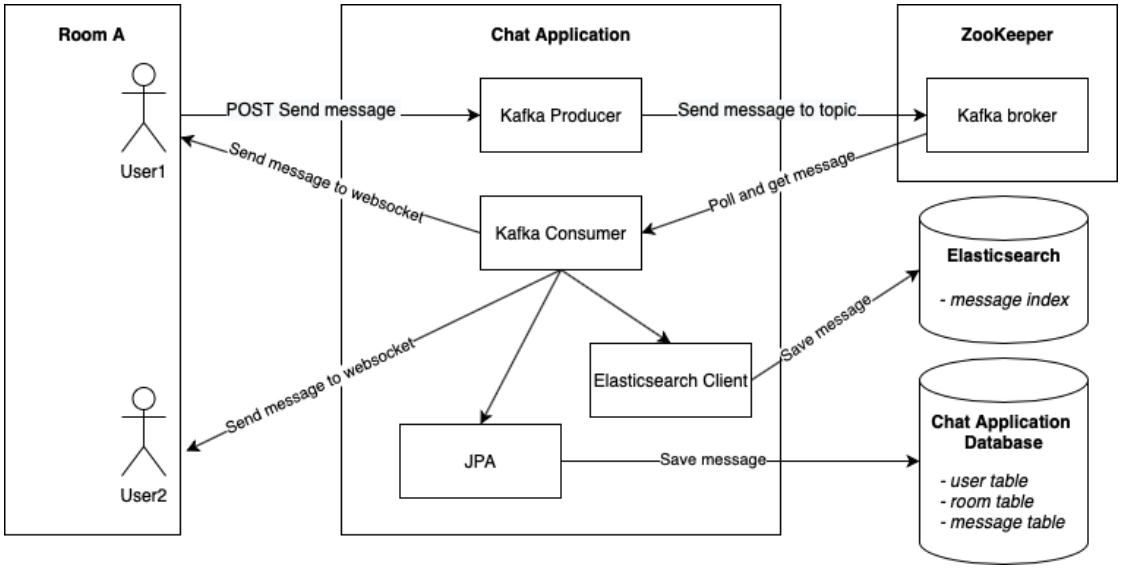
1.2. Spring boot

Spring keretrendszer [8] egy Java programozói keretrendszer, amely kész elemeket ad, segítségével gyorsan lehet nagyvállalati rendszereket fejleszteni. Spring boot egy Spring-re épülő keretrendszer, ami automatizmusokat tartalmaz a Spring rendszer beállítására. 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 foglalkoznának. A rendszer paramétereinek beállítása történhet annotációk segítségével vagy hagyományos java properties fájlokkal. A Spring Boot alkalmazásokat nem szükséges web archív (war) fájlként telepíteni, hiszen a Spring boot segítségével egy jar állomány építhető, ami önállóan

futó alkalmazás hozható létre beépített web alkalmazás motorral. Nem igényel egyáltalán XML konfigurációt, így a testre szabása és kezelése is könnyebb, mint a Spring-nek [8].

2. Oktatást segítő alkalmazás kibővítése chat és keresés funkcióval

Az Egyetemi óra keretein belül diákok dolgoznak egy olyan oktatást segítő alkalmazáson, ami megkönnyíti a diákok órai munkáját. Ehhez készítettünk egy olyan alkalmazás modult, amivel valós időben tudnak csevegni a diákok egymással és tanáraikkal. Tantárgyanként lehetne létrehozni szobákat, a diákok és tanárok oda csatlakozhatnak be és cseveghetnek. A chat üzenetekben lehet keresni, rész szövegek alapján. A kereséshez az Elasticsearch rendszert integráltuk, stabilitása, és gyors működése miatt. Az alrendszer logikai felépítése és a rendszer elemek közötti üzeneteket a következő ábra szemlélteti.



2. ábra. *Chat alkalmazás üzenet küldés-fogadás folyamata*

A rendszer legfontosabb új funkciója a *chat üzenet küldése*, és az ehhez kapcsolódó *beérkező chat üzenet megjelenítése* funkciók.

A *chat üzenet küldése* funkció a következőképpen valósult meg. Ha 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"). Az általunk írt 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 *beérkező chat üzenet megjelenítése* funkció az előző funkció hatására lép működésbe aszinkron módon. Az általunk implementált Kafka Consumer folyamatosan kiolvassa a Kafka bróker üzenet sorából a soron következő üzeneteket. Az új üzenetet három irányba továbbítja. Egyrészt elküldi az Elasticsearch felé, ami letárolja az üzenetet egy indexben. Az Elasticsearch integráció nagyon egyszerű, csak egy repository-t kell használni, teljesen úgy működik, mint bármilyen más repository, így a kommunikációs protokoll teljesen transzparens. Másrészt a Kafka Consumer továbbítja az üzenetet a

kliensek felé websocket protokollt használva. Minden a szobában bent levő felhasználó megkapja az üzenetet (az üzenet küldője is), és megjeleníti azt sorrend helyesen. Harmadrészt letárolja egy relációs adatbázis is az üzenet részleteit. A hagyományos adatbázis alkalmazásával (Mysql) az üzenetek tárolódnak, így lehetőség van azok későbbi elemzésére is.

Ezzel a topológiával az általunk készített alkalmazás, mint gateway működik, illeszti az elasticsearch, Kafka, és Mysql alkalmazásokat a régi alkalmazáshoz. A már meglévő alkalmazásnak minimális módosítási igénye van, hiszen csak a kliens oldali Javascript felületbe kell beilleszteni pár vezérlőt, gombot az üzenet küldésre, beviteli mezőt a küldendő üzenethez, és egy text area komponenst a szoba üzenetek megjelenítésére. Így csak egyirányú kommunikáció van, a legacy rendszer frontendje hívja az új funkciókat implementáló új http REST végpontokat. A válaszokat a websocketen kapja meg a kliens böngésző, így a lehető legkisebb kommunikáció zajlik a kliens és a szerver között.

3. Összefoglalás

Az általuk javasolt topológiával, a hagyományos monolitikus és a modernebb mikroszerviz topológiák ötvözésével ötvöztük a két koncepció előnyeit. Az eredeti rendszer forrás kódjában csak minimális változtatásra volt szükség (csak kliens oldali fejlesztés), és az új gateway jellegű alkalmazással tudtuk illeszteni az elasticsearch lehetőségeit. Az illesztést az aszinkron üzenet sorokkal tettük lehetővé, amely a lehető legnagyobb flexibilitást tette lehetővé. Továbbfejlesztés lépésként javasoljuk, hogy a régi (technológiailag elavult) rendszert fel kell darabolni több mikroszervizre, és az egyes szervizeknek kell direkt hozzáférést adni a Kafka üzenet soraihoz.

Irodalom

[1] Fowler, M., Lewis, J.: *Microservices*, 2014, https://martinfowler.com/articles/microservices.html

[2] Richardson, C.: *Monolithic Architecture Pattern*, https://microservices.io/patterns/monolithic.html

[3] MuleSoft: *Monolitikus és microservice koncepció összehasonlítása*, https://www.mulesoft.com/resources/api/microservices-vs-monolithic

[4] Kharenko, A.: *Monolitikus és microservice koncepció összehasonlítása*, https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

[5] Berki, Á.: *Apache Kafka leírás*, https://wiki.berki.org/index.php/Apache_Kafka#Kafka_bemutat.C3.A1sa

[6] Gulden, M.: *Introduction to Kafka Connectors*, https://www.baeldung.com/kafka-connectors-guide

[7] Gulden, M.: *Introduction to Kafka Connectors*, https://kafka.apache.org

[8] Spring framework fejlesztő csapata: *Hivatalos Spring framework dokumentáció*, https://spring.io/projects/spring-framework

[9] Namiot, D., Sneps-Sneppe, M.: *On micro-services architecture*, International Journal of Open Information Technologies 2.9 (2014) pp. 24-27.

[10] Dragoni, N., et al.*: Microservices: yesterday, today, and tomorrow*, Present and ulterior software engineering. Springer, Cham, 2017. pp. 195-216. **https://doi.org/10.1007/978-3-319-67425-4_12**

[11] Dragoni, N., et al.: *Microservices: How to make your application scale*, International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Springer, Cham, 2017. **https://doi.org/10.1007/978-3-319-74313-4_8**