Multidiszciplináris tudományok, 10. kötet. (2020) 2 sz. pp. 430-436 **https://doi.org/10.35925/j.multi.2020.2.48**

OKTATÁS SEGÍTŐ ALKALMAZÁS KÉSZÍTÉSE MIKROSZERVIZ ALAPOKON

Szőke Attila
MSc hallgató, Miskolci Egyetem, Informatikai Intézet, Általános Informatikai Tanszék
3515 Miskolc, Miskolc-Egyetemváros, e-mail: szoke.attila@student.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 modern alkalmazás architektúrát mutat be, annak alkalmazását. Egy éppen zajló fejlesztést, ami a gyakorlati kurzusok vezetését segíti. A javasolt architektúra elsődleges célja a lehetőleg nagy flexibilitás, skálázhatóság és egyszerű későbbi fejlesztés biztosítása. Szeretnénk továbbá biztosítani, hogy minimális változtatással (lehetőleg csak konfiguráció) lehessen üzemeltetni egy hoszton, vagy privát felhőben, vagy publikus felhőben egyaránt. Mindezekre a Spring boot alkalmazás fejlesztői keretrendszert és a mikroszervíz architektúrát javasoljuk.

Kulcsszavak: *alkalmazás fejlesztés, micro service, Java, Web szolgáltatások*

Abstract

This article presents a modern application development architecture and its application. This paper describes an improvement of product which will help to lead practical courses. The primary goal of the suggested topology is providing high flexibility, scalability, easy improvement (later development). We would like to provide an architecture and pattern which can be easy to deploy and can be operate in local host or private cloud or public cloud. We suggest the Spring Boot Java application framework and micro service architecture.

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

1. Bevezetés

Alkalmazás fejlesztése során törekednünk kell a minimális minél lazább függőség kialakítására, és olyan technológiák, eszközök, megoldások használatára, amelyek nem kötik meg a kezünket, így egy koncepció váltás esetén nem kell lényegesen átdolgozni a már meglévő kódjainkat, az alkalmazást. Amikor a projektet elkezdünk csak azokat a tényezőket tudjuk figyelembe venni, amik akkor már ismeretesek. Azonban az informatikában hirtelen, és nagyokat változnak a trendek, a koncepciók. Az iparban is azt tapasztaltam, hogy nagyok sok legacy (megtűrt régi) rendszert kell használnunk. Ennek egyszerű az oka. Amikor az elavult alkalmazás elkészült, akkor modern és korszerű volt, újabb és újabb kérésekre újabb és újabb implementációk jelentek meg, de nincs annyi erőforrás és idő, hogy minden alkalommal teljesen átírjuk a rendszerünket. Ezen rendszerek javítása, fejlesztése egy idő után reménytelen, mert már nem találunk hozzá fejlesztőt, és a keretrendszereket szállító fejlesztő csapatok nem szállítanak újabb javításokat, frissítéseket. Mindezek miatt a hagyományos monolitikus alkalmazás fejlesztése helyett a mikroszerviz architektúra (mikroserviz arhitektúra koncepció, leírás bővebben

a [1], [2], [3], [4], [5], [6], [9], [10], [11] helyeken) terjedt el. Ez az új architektúra viszont más fajta fejlesztést, más szemléletet követel meg. Erre szeretnénk ebben a cikkben rávilágítani, segíteni a jövőbeli alkalmazások elkezdését, készítését egy minta alkalmazás segítségével. Először ismertetjük röviden az alkalmazás követelményeit, azután a mikroszerviz architektúrát, végül javasolunk konkrét programozói keretrendszert és technológiákat is.

Az alkalmazás mikroszerviz struktúrájának megvalósításához a Spring keretrendszert és a Spring Cloud modulja által nyújtott technológiákat javasoljuk. A lokális fejlesztés a felhőbe telepítéshez a Docker alkalmazás konténert javasoljuk, és ismertetjük röviden.

1.1. Alkalmazás követelményeinek rövid leírása

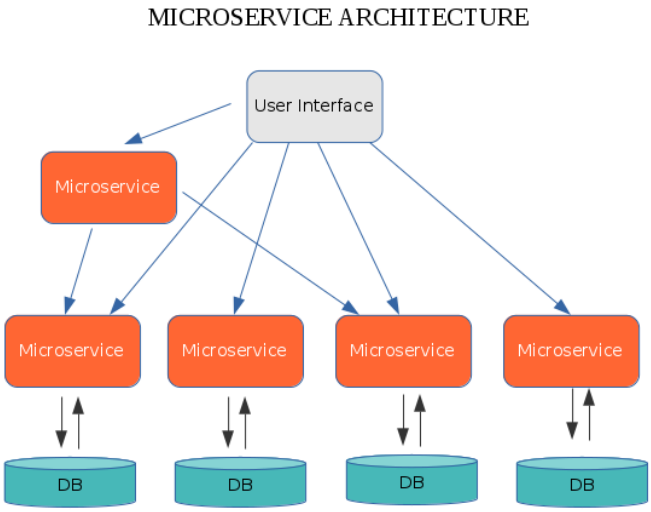
Az alkalmazást főképp gyakorlati kurzusokon való használatra terveztük és minden interakció böngészőben webes felületen történik, ezáltal platform független, minimális előfeltétel van a felhasználókra. Egy bejelentkezett tanár létrehozhat kurzusokat és meghívhat rá diákokat. A diákok felvett kurzusaiknál láthatják a kurzusokhoz kapcsolódó információkat és feltöltött anyagokat. A tanórák keretein belül a tanár kiadhat feladatokat a diákoknak, amiknek a megoldásait a diákok feltölthetik, vagy segítséget kérhetnek a tanártól. A feladat adatlapján, a tanár nyomon tudja követni a haladást és lezárhatja a feladatot, ha szeretné.

Fontos, hogy ez valós időben kell, hogy történjen, lássa az oktató, hogy hány hallgatónak van már kész? Melyik hallgató hol akadt le. A Covid-19 miatt ez nagyon hasznos funkció, hiszen az elő chat-es és beszéd órákkal (Teams, Skype), ez nehezen követhető. Ki hol jár kérdésre, ha beírják a válasz, akkor vissza kell menni a beszélgetés történetében, ha meg szóban mondják be, akkor elnémítják egymást, mert ezek a rendszerek elnémítják az éppen kevésbé aktív résztvevőt. Őt főig ez még követhető, felette már nem, és nem lehet az a megoldás, hogy képezzük öt fős csoportokat, mert akkor drasztikusan nőne az oktatók terhelése. Egy felületen láthatja majd, hány hallgató van kész, a hallgatói statisztikákat.

A feladatokhoz komment is rendelhető, ami további interakciót tesz lehetővé. Tehát a program lényege, hogy növelje a tanár és diákok közti interakciót a tanórákon, a diákok pedig el tudják érni az egyes gyakorlatok feladatait és feltöltött állományait, ami segítheti a felkészülést a vizsgákra, zárthelyikre.

1.2. Mikroszerviz architektúra

A mikroszervizek a monolitikus architektúra hátrányait hivatottak megoldani, mégpedig oly módon, hogy egy ilyen architektúrát megvalósító szoftver olyan moduláris, amennyire csak lehetséges. Mikroszerviz alapú szoftver tervezésének az egyik legfontosabb lépése az, hogy a szoftvert felbontjuk szolgáltatásokra (innentől szervizekre). A legegyszerűbb úgy elképzelni, hogy minden ilyen szerviz egy külön processz egyértelmű felelősséggel. Minden egyes mikroszerviz egy jól meghatározott feladattal és felelősség körrel rendelkezik, könnyen karbantartható és egység tesztelhető, nagyon lazán kötődik más szervizekhez, függetlenül futtatható/kitelepíthető, egyenként könnyen fejleszthető akár kis fejlesztő csapatok által is. Ahogy a következő ábrán látható, ezen szervizek kollaborációja adja a kliensek kéréseire a válaszokat. A részek közötti kommunikáció http kérés, így a különböző szervizek implementációs nyelve eltérhet, könnyen monitorozható, és a biztonság is megoldható, a részek cseréje is egyszerű.



1. ábra. *Mikroszerviz architektúra*

1.3. Spring boot

A Spring Framework (dokumentáció a [7] helyen) átfogó programozási és konfigurációs modellt biztosít a modern Java alapú vállalati alkalmazásokhoz. A Spring egyik kulcsfontosságú eleme az infrastrukturális támogatás alkalmazás szinten: a Spring a vállalati alkalmazások technikai részeire összpontosít, így a fejlesztők az alkalmazás szintű üzleti logikára koncentrálhatnak. A Spring több modulból áll, amelyek szívében a konfigurációs modell és a dependency injection áll. A további modulok különböző alkalmazás architektúráknak nyújtanak támogatást, mint például üzenetküldés, perzisztencia és Web MVC. Ezek a modulok természetesen opcionálisan választhatók attól függően, hogy a felhasználónak mire van szüksége. Nagy hangsúlyt fektet arra, hogy minden szinten választást kínáljon a felhasználónak, próbál minden architektúrát lefedni, mindenhol támogatást nyújtani. A Spring nyílt forráskódú és teljesen ingyenes, ezért széles körben használt és könnyen hozzáférhető. A Springboot segítségével teljesen egyedülálló alkalmazásokat készíthetünk, minimális konfigurációval. Az alkalmazás indulásakor a Spring boot automatizmusa futás időben dönt egy funkció ki/be kapcsolásáról, és a szükséges paraméterek biztosításáról. Ezen döntést az elérhető osztályok alapján és/vagy meglévő objektumok és azok tulajdonságai alapján hozza meg. Az így elkészített programokat nem kell telepíteni hagyományos java web alkalmazás motorba, mivel a Spring beágyaz már egyet. Szinte minden konfigurációs lehetőségnek van egy default értéke, amit felül lehet definiálni annotált osztályokkal és properties vagy yml fájlokkal, nem igényel XML konfigurációt.

1.4. Spring cloud

A Spring boot, ha elérhetőek a Spring cloud osztályai, akkor segít az egyes mikroszervizeket gyorsan és egyszerűen beállítani, azonban az együttes elosztott működést, ezáltal a munka oroszlánrészét a *Spring Cloud* implementáció fogja futás időben végezni (dokumentáció a [8] helyen). A *Spring Cloud* olyan eszközöket biztosít a fejlesztők számára, amelyek segítségével gyorsan fel tudnak építeni egy elosztott rendszert, néhány gyakran használt minta alapján. Az alap mikroszerviz struktúra felállításához nekünk főleg szerviz konfiguráció menedzsment, a szerviz felfedezés (discovery), a szervizek

közti kommunikációs és valamilyen proxy funkciókra lesz szükség. Segítségükkel több szerviz is bevezethető, lecserélhető a használók átállítása nélkül.

A *Spring Cloud Config*, ami lényegében egy konfigurációs szerver implementálását teszi lehetővé, ami a fejlesztéstől, a tesztelésen át az éles használatig végig biztosítja a szervizeknek a konfigurációs állományaikat, amire szükségük van. Pár annotációval egyszerű módon tudunk definiálni egy szolgáltatást, melyet az összes mikroszerviz tud használni. A mikroszervizek lekérdezhetik a beállításaikat ettől a központi szolgáltatástól. Csak ezt a központi szolgáltatást kell csak ismerniük, minden mást attól kapnak. Alapértelmezett beállításokat itt is felülírhatjuk számos módon, mi a Git repository-ban tárolt *properties,* illetve *yaml* fájlokat használjuk és javasoljuk, amelyek nevei az egyes szervizek nevei.

A második nagyon fontos komponens az a discovery (felderitő) szerviz. A nevéből adódóan ezen komponens feladata az, hogy nyilván tartsa az egyes mikroszervizeket és megossza az elérhetőségeiket a többi szervizzel. Erre a feladatra mi a Netflix Eureka discovery szervizét választottuk. Az Eureka a Netflix által fejlesztett nyílt forráskódú szerviz, kiváló Spring integrációval. A kliensek induláskor, miután megszerezték a config fájlokat regisztrálnak az Eurekába és meta adatokat osztanak meg önmagukról, mint például: health (egészség) indikátor, port, host stb. A regisztrációk ezután úgy maradnak frissek, hogy a kliensek szívdobbanásokat (heartbeat) küldenek, ha ezek elmaradnak, akkor a szerviz UP-ból DOWN állapotba kerül. A szívdobbanások ellenőrzésének frekventáltsága szabadon konfigurálható. A szervizek le tudják kérni a discovery szerviz „katalógust" és így el tudják érni a többi szervizt.

1.5. Docker alkalmazás konténer

A *Docker* egy olyan eszköz, ami megkönnyíti az alkalmazások fejlesztését és telepítését. A *Docker* konténereken alapszik. Ezek a konténerek operációs rendszer szintű virtualizációval létrehozott virtuális alkalmazás konténerek. A konténereken általában a *Linux* egy csonkított verziója fut és teljesen hozzáigazíthatjuk az alkalmazásunkhoz. Innentől kezdve a benne futó alkalmazás ugyanazt a lehetőségeket kapják bármely operációs rendszer esetén, és el is szigetelik a gazda géptől. Majd megmondhatjuk, hogy ezen a gépen az alkalmazásunk mely része fusson, hogyan kommunikálhat a többi konténerrel vagy a külvilággal. Amikor az alkalmazásunkból létrehozunk egy konténert, akkor előbb egy image (rendszerkép) készül, ami nem változik, mindegy milyen környezetben indítjuk a konténert, így biztosítva az egységes működést. Mikroszervizek fejlesztésénél elengedhetetlen a *Docker* alkalmazása, hiszen lokálisan úgy futtatjuk a szervizeinket mintha már több külön álló szervergépen üzemelnének és ugyanúgy fognak együttműködni, mint ahogy azt élesben tennék. A legnagyobb leggyakoribb felhő szolgáltatók támogatják a docker image-et telepítését, futtatását. Privát felhőben is tudunk docker konténereket futtatni, így a docker alkalmazásával minden üzemeltetése feltételt teljesítünk.

2. Oktatást segítő alkalmazás megvalósítása

Ebben a fejezetben ismertetjük a már implementált alkalmazásunk szereplőit, feladatait, iránymutatást adva a későbbi alkalmazások fejlesztésére.

2.1. Domain modell tervezése

Összesen 8 entitást különböztettünk meg, amelyek között 16 kapcsolat azonosítható. 3 entitás (*User, Teacher, Student*) között szülő-gyerek viszony alakítható ki. Az elsődleges feladat a mickroszervíz határok kialakítása.

A probléma abban áll, hogy egy hagyományos, monolitikus adatbázis használata nem egyeztethető össze a mikroszerviz architektúra koncepciójával. Ha ezt nem vesszük figyelembe, akkor a következő problémákkal kell szembenéznünk:

- Szoros kapcsolatot hozunk létre a service-k között, ezáltal az adatbázis séma változásait csak egyszerre, a teljes alkalmazáson tudjuk érvényesíteni. Ehhez a különböző szolgáltatások közötti koordináció szükséges, ami lassítja a fejlesztést.
- A különböző mikroszervizek skálázása a teljes monolit adatbázis skálázását vonja maga után. Emiatt csökkenne az erőforrások kihasználásának hatékonysága.
- A szolgáltatások fejlesztői nem dönthetnek arról, hogy milyen típusú adatbázist szeretnének használni. Pedig vannak olyan szituációk, ahol különböző NoSQL (dokumentum, esetleg gráf alapú) adatbázisok alkalmazása előnyösebb lenne.

Ezek miatt a problémák miatt, lényegében felesleges lenne bármilyen plusz munka, amit a mikroszervizek kialakítására fordítanánk, hiszen ez nem járna valódi előnyökkel. Ezért érdemes inkább minden szolgáltatáshoz egy teljesen különálló adatbázist rendelni. Majd ezeket úgy felhasználni, hogy egy service kizárólag a saját adatbázisát érhesse el közvetlenül. Ha egy másik adatbázist szeretne olvasni vagy manipulálni, azt kizárólag az azt használó másik service által nyújtott API-n keresztül teheti meg.

2.2. Alkalmazás szervizei, szerkezete

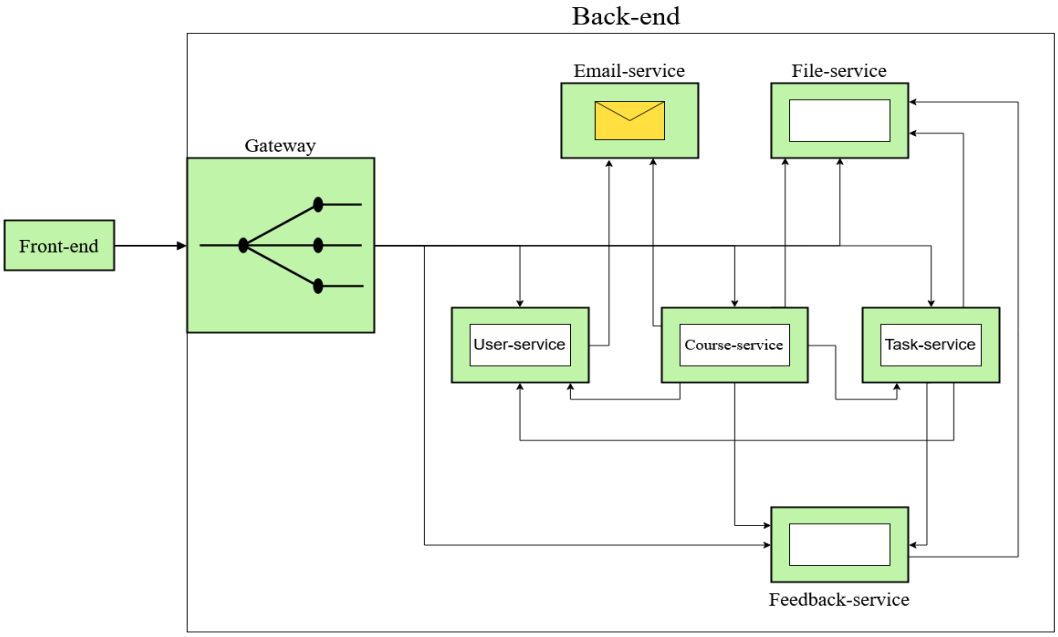
Az alkalmazásunkban két féle szerviz van jelen, a strukturális és a funkcionális szervizek. A strukturális szervizek a *config, discovery* és *gateway* szerviz, amik az egyes szervizek közti kommunikációt és, ezáltal a mikroszerviz architektúra alapjait valósítják meg. Ami a funkcionális szervizeket illeti, további 6 szerviz valósítja meg a projekt feladatait. Ezek részben főnévi alapon, részben gyakorta használt tevékenységek alapján definiáltak. Előbbibe tartozik a *course-service, task-service, feedback-service, user-service.* Ezek mindegyike kezeli a nevükben szereplő objektumok minden műveletét. Utóbbiba pedig az *email-service* és a *filemanagement-service* tartozik. Az alkalmazásunk szerkezetét, a mikroszervizek futás idejű függéségét a következő ábra szemlélteti.

A fent felsorolt szervizek mindegyike szinkron módon kommunikál egymással, tehát a kommunikációt megvalósító részük megegyezik és csak abban térnek el, hogy milyen műveleteket végeznek az entitásaikkal. A szervizek mindegyike web-MVC modell szerint épül fel és ahol szükség van perzisztenciára, ott *Spring Data JPA*-val van a megvalósítva az adattárolás, *MySQL* adatbázisban.

Az alkalmazás funkcióit biztosító mikroszervizek a következők:

- Gateway: *Eurekába* regisztrált kliensek között ossza el a forgalmat automatikusan az előre megadott útvonalak alapján.
- A *course-service* az alkalmazás központi szervize. Metódusai szinte minden szervizt használnak, implementálják az alapvető funkciókat, mint a kurzus létrehozás, a kurzus lekérdezés, diák meghívása. Ez a legfontosabb szerviz. Egy végpont hívása sokszor más mikroszerviz hívással járhat.
- User-service, ami a felhasználókat menedzseli. Felelős a felhasználó (User) entitáshoz kapcsolódó többnyire CRUD műveletek végrehajtásáért. Lebonyolítja a felhasználók regisztrációját és beléptetését. Nyújt egy úgynevezett „Who Am I" végpontot, ami megfelelő authentikáció esetén, minden felhasználóhoz kapcsolódó információt biztosít.
- *Task-service*: az órán felmerülő feladatokat tartja nyilván. A feladatok hierarchiába is rendezhetőek.

- *Feedback-service*: a feladatokhoz kötődő visszajelzéseket tárolja. Jelenleg ez külön szolgáltatás, a jövőben ezt össze lehet vonni a feladattal.
- *Email-sevice*: alacsony szintű (szolga) szerviz. Bármely fő funkciót biztosító szerviz hívhatja, ha az adott lépéshez email értesítés szükséges.
- *File-service*: szintén alacsony szintű szerviz. Bármely entitáshoz, funkcióhoz kötődő fájlok menedzseléséért felelős. File kötődhet jelenleg a feladathoz, mint probléma leírás, illusztráció vagy mint megoldás. Későbbiekben fel lehet majd tölteni a felhasználó profilképét is.



2. ábra. *A funkcionális szervizek kommunikációja*

3. Összefoglalás

A fent említett szerkezettel, ajánlásokkal (Spring *boot* + *Docker* + *mikroszerviz*) biztosítható a skálázhatóság, a minimális laza függőség a rendszer részei között, az egységes fejleszthetőség és a minimális konfiguráció.

Az alkalmazás teljes mértékben skálázható, konfigurálható forráskód változás nélkül, lehet újabb funkcionális szervizeket kitelepíteni, amik aztán az alkalmazás részei lesznek és fogadják a kéréseket. A terhelés elosztás automatikusan működik az egyes szervizek között és a *Gateway*-re beérkező kérés esetében is. A *Docker* lehetővé teszi, hogy egységes környezetben fejleszthessük és tesztelhessük a szervizeket, és ha esetleg szerverre szeretnénk kitelepíteni, az sem okozna különösebb problémát a konténerek természete miatt.

Továbbfejlesztés mind technológia, mind funkciók terén lehetséges. Jelenleg egyszerű szinkron kérések formájában érkeznek be a mikroszervizekhez a feladatok. A Spring segítségével le lehetne cserélni „*circuit breaker pattern”* mintára, hogy a rendszerünk hibatűrő legyen (főleg a mentésnél releváns). Funkciók terén a keresést kellene integrálni (pl.: elastic search), automatizmusokat beépíteni, valamint értesítések küldését és automatikus feladat kiértékelést.

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] Richardson, C.: *Microservice Architecture Pattern*, https://microservices.io/patterns/microservices.html
[4] Huston, T.: *The Six Characteristics Of Microservices*, https://smartbear.com/solutions/microservices
[5] Hombergs, T.: *Microservice Communication Patterns*, https://reflectoring.io/microservice-communication-patterns
[6] Novoseltseva, E.: *Microservices architecture benefits*, https://apiumhub.com/tech-blog-barcelona/microservices-architecture-implementation
[7] Spring framework fejlesztő csapata: *Hivatalos Spring framework dokumentáció*, https://spring.io/projects/spring-framework
[8] Spring cloud fejlesztő csapata: *Hivatalos Spring cloud dokumentáció*, https://spring.io/projects/spring-cloud
[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**