



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Microservice architektúra fejlesztésének és tesztelésének támogatása

DIPLOMATERV

Készítette
Borlay Dániel

Konzulens
Szatmári Zoltán

2016. december 7.

Tartalomjegyzék

Kivonat	5
Abstract	6
1. Bevezetés	7
2. Háttérismeretek	9
2.1. Mikroszolgáltatások	9
2.1.1. Szolgáltatás elválasztás tervezése	9
2.1.2. Architektúrális mintákhoz való viszonya	11
2.1.3. Eltérések a szolgáltatás orientált architektúrától	12
2.1.4. Példák mikroszolgáltatásokat használó alkalmazásokra	12
2.2. Mikroszolgáltatások előnyei és hátrányai	13
2.2.1. Előnyök	13
2.2.2. Hátrányok	14
2.2.3. Összehasonlítva a monolitikus architektúrával	15
2.3. Technológiai áttekintés	15
2.3.1. Futtatókörnyezetek	16
2.3.2. Környezet felderítési technológiák	17
2.3.3. Konfiguráció management	17
2.3.4. Terheléselosztás	18
2.3.5. Skálázási technológiák	19
2.3.6. Virtualizációs technológiák	20
2.3.7. Szolgáltatás jegyzékek (service registry)	20
2.3.8. Monitorozás, loggolás	21
2.4. Kommunikációs módszerek	22
2.4.1. Technológiák	22
2.4.2. Interfészek	23
3. Feladat megtervezése	25
3.1. Minta alkalmazás tervezése	25
3.1.1. Minta alkalmazás	25
3.1.2. Szolgáltatások	25
3.1.3. Kommunikáció	26

3.2.	Folytonos Integráció	27
3.2.1.	Használati módok mikroszolgáltatások esetén	28
3.2.2.	Keretrendszer előnyei a fejlesztésre nézve	29
3.2.3.	Lépések bemutatása	30
4.	Feladat implementációja	31
4.1.	Felhasznált technológiák	31
4.2.	Szolgáltatások implementálás	32
4.2.1.	Docker konténerek	32
4.2.2.	Alkalmazás részletek	33
4.3.	Kommunikáció, avagy hogy működik a Consul	35
4.4.	Működés és alkalmazás	37
4.5.	Folytonos integráció elkészítése	38
4.5.1.	Jenkins	38
4.5.2.	Pipeline Job	39
4.5.3.	Jenkins Job-ok a keretrendszerhez	40
4.5.4.	Job Konfigurációk	41
5.	Értékelés	44
5.1.	Lehetőségek	44
5.2.	Problémák, Kritika	44
5.3.	Hol használható	44
A.	Függelék	50
A.1.	Dockerfile-ok	50
A.1.1.	Authentikáció	50
A.1.2.	Proxy	51
A.1.3.	Adatbázis	51
A.1.4.	Vásárlás	52
A.1.5.	Webkiszolgáló (böngészés)	53
A.2.	Szkriptek	54
A.2.1.	Futtatáshoz	54
A.2.2.	Szolgáltatásokhoz	56
A.2.3.	Pipeline job szkript	60
A.2.4.	Proxy	60

HALLGATÓI NYILATKOZAT

Alulírott *Borlay Dániel*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2016. december 7.

Borlay Dániel
hallgató

Kivonat

Napjainkban komoly gondot okoz, hogy hogyan lehet hatékonyan elosztott, jó rendelkezésre állású, könnyen skálázható alkalmazást építeni. Sok architektúrális megközelítés van, amit alapul véve hatékonyan tervezhetjük meg a rendszerünket, és könnyen elkészíthetjük az alkalmazásunkat. Egy ilyen architektúrális megközelítés a mikroszolgáltatásokon alapuló architektúra, amivel apró részletekre bontva a feladatot, könnyen kezünkben tarthatjuk az elosztott alkalmazásunkat.

A mikroszolgáltatásokra épülő architektúra (microservices), egy olyan alkalmazás fejlesztési módszertan, ami a programot alkotó elemekre szedi, és minden funkcionalitást teljesen különálló egységként kezel. Egy ilyen alkalmazás fejlesztése közben oda kell figyelni az összes szolgáltatással való együtteműködésre, a visszamenőleges kompatibilitásra, és meg kell tartani az alkotóelemek kapcsolatának a konzisztenciáját. Ennek a fenntartása egy nehéz feladat, amit kézileg szinte lehetetlen hosszú távon fenntartani.

A diplomaterv keretében az volt a feladatom, hogy megismerjem az architektúra lényegét és működését, illetve kiderítsem, hogy milyen eszközökkel tudom automatizálás segítségével támogatni a fejlesztés, és működtetés folyamatát.

A diplomaterv célkitűzése, hogy egy olyan mikroszolgáltatásokra épülő alkalmazást készítssek, amellyel be tudom mutatni az architektúra előnyeit, végig tudom vezetni rajta a tesztelés folyamatát, tudom automatizálni a tesztelését, és működtetését, és betekintést tudok adni az architektúrához használatos technológiákba.

Abstract

Nowadays it's a very big problem, that how to build a distributed, highly available, easily scalable application efficiently. There are many ways to design our system and application which we could base on our plans. One of these design patterns is the microservice based architecture, which creates small services from the big application, by separating the functionality, and we can handle more efficiently our distributed application.

Microservice architecture is a software development methodology, which separates the parts of the application to the smallest functionality, which could be run from a separated environment. It is important to take care about the cooperation between the services, the backward compatibility and the consistency of the service connectivity through the development flow. It is hard to keep these attributes, and almost impossible to keep it on long term by manual verification.

The goal of this thesis is to gather knowledge about the architecture, how it works or how it could be designed, or which continuous integration tool could be used for helping development and maintenance.

The goal of this thesis is to create an example application based on microservice architecture, which could be used for showing the advantages of the technique, and I can show the full process of testing and I can create a framework for helping the development of the application by automated testing and integration.

1. fejezet

Bevezetés

Fontos, hogy az alkalmazásaink megbízhatóan, karbantarthatóan, és nagy rendelkezésre állással legyenek elérhetőek. Napjaink egyik feltörekvő architektúra építési elve a mikroszolgáltatások architektúrája. Ezt az architektúra típust az elosztott működése, a szolgáltatásonkénti könnyen fejleszthetősége, és a jó skálázhatósága teszi népszerűvé. Sok cég választja ezt az új elvet, mivel az egyes komponensek fejlesztése egyszerűbb és gyorsabb, a végeredmény pedig könnyebben karbantartható, és egyszerűen számítási felhőbe integrálható.

Jelen labor keretében megismerkedem a mikroszolgáltatások felépítésével, működésével, és egy automatizált megoldást adok a használatukra. Bemutatom, hogy hogyan lehet azt a technológiát automatizáltan tesztelni, és a fejlesztési folyamatot egyszerűen és felügyelve vezetni.

Az második fejezetben beleástam magam a technológiai áttekintésbe, ahol az architektúra lényegét próbáltam megérteni, és összeszedtem, hogy milyen tervezési kérdések merülnek fel egy mikroszolgáltatásokra épülő alkalmazás elkészítésénél.

A harmadik fejezetben megnéztem, hogy milyen előnyei illetve hátrányai lehetnek ennek a módszernek, illetve megnéztem egy példát (Archivematica), ami segíthet az átfogó képalkotásában, és saját alkalmazás fejlesztésében.

A negyedik fejezetben a kapcsolódó technológiákról készítettem egy összefoglalást, ami tartalmazza a jelenleg használt technológiákat, amikkel mikroszolgáltatásokra épülő architektúrát lehet építeni, illetve olyan technológiákat, amikkel kiegészítve teljesen felügyelhető a szolgáltatások működése.

Az ötödik fejezetben a különböző kommunikációs lehetőségekkel foglalkoztam, amikkel össze lehet kötni a szolgáltatásokat, illetve a kommunikáció tervezése közben felmerülő nehézségeket néztem át.

A hatodik fejezetben megterveztem a példa alkalmazást, illetve az architektúrát, amit meg fogok alkotni a diplomaterv során.

A hetedik fejezetben az alkalmazás implementációját járom körbe.

A nyolcadik fejezetben az elkészítés közben tapasztalt nehézségeket, és az alkalmazás értékelését fejtem ki bővebben.

A kilencedik fejezetben az automatizálást járom körbe, kitérek arra, hogy mit hogyan célszerű csinálni egy mikroszolgáltatásokra épülő automatizált rendszerben, és mit lehet

véghez vinni az általam elkészített struktúrában.

A tizedik fejezetben a valós felhasználási módokat mutatom be, és kifejtem, hogy mikor és milyen körülmények között van értelme az automatizált támogatásnak és a mikroszolgáltatások használatának.

A tizenegyedik fejezet tartalmazza az összefoglalást, ami egy értékelést ad az elvégzett munkáról. Ebben a fejezetben térek ki a diploma munka lehetséges folytatási lehetőségeire.

2. fejezet

Háttérismeretek

2.1. Mikroszolgáltatások

A mikroszolgáltatás[26] [9] [25] egy olyan architektúrális tervezési módszer, amikor a tervezett rendszert/alkalmazást kisebb funkciókra bontjuk, és önálló szolgáltatásokként, önálló erőforrásokkal, valamilyen jól definiált interfészen keresztül tesszük elérhetővé.

Ezt az architektúrális mintát az teszi erőssé, hogy nem függenek egymástól a különálló komponensek, és csak egy kommunikációs interfészt ismerve is karbantartható a rendszer. Egy szoftver fejlesztési projektben előnyös lehet, hogy az egyes csapatok fókuszálhatnak a saját szolgáltatásukra, és nincs szükség a folyamatos kompatibilitás tesztelésére.

Egy mikroszolgáltatást használó architektúra kiépítéséhez sokféle funkcionális elkülönítési módot használnak, amivel a szolgáltatásokat kialakíthatjuk. Egy ilyen elválasztási módszer a rendszer specifikációjában lévő főnevek vagy igék kiválasztása, és az így kapott halmaz felbontása. Egy felbontás akkor minősül ideálisnak, ha nem tudjuk tovább bontani az adott funkciót. A valóságban soha nem lesz az ideálisnak megfelelő felbontás, mivel erőforrás pazalró, és túlzottan elosztott rendszert kapnánk.

2.1.1. Szolgáltatás elválasztás tervezése

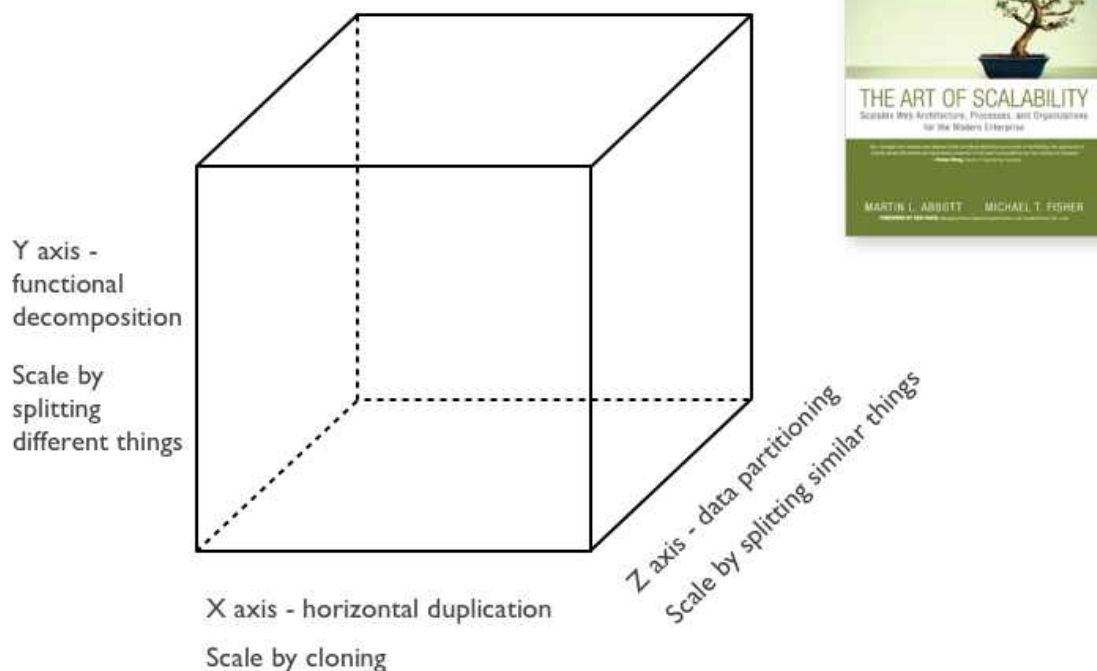
A tervezési folyamatnál a következő szempontokat szokták figyelembe venni:

- Szolgáltatások felsorolása valamilyen szempont szerint
 - Lehetséges műveletek felsorolása (igék amik a rendszerrel kapcsolatosak)
 - Lehetséges erőforrások vagy entitások felsorolása (főnevek alapján szétválasztás)
 - Lehetséges use-case-ek szétválasztása (felhasználási módszerek elválasztása)
- A felbontott rendszert hogyan kapcsoljuk össze
 - Pipeline-ként egy hosszú folyamatot összeépítve és az információt áramoltatva
 - Elosztottan, igény szerint meghívva az egyes szolgáltatásokat
 - Egyes funkciókat összekapcsolva nagyobb szolgáltatások kialakítása (kötegelés)
- Külső elérés megszervezése

- Egy központi szolgáltatáson keresztül, ami a többivel kommunikál, és csak ennyi a feladata
- Add-hoc minden szolgáltatás külön hívható

Ezekkel a lépésekkel meg lehet alapozni, hogy az általunk készítendő rendszer hogyan is lesz kialakítva, és milyen paraméterek mentén lesz felvágva. A választást segíti a témában elterjedt fogalom, a scaling cube[38], ami azt mutatja, hogy az architektúrális terveket milyen szempontok mentén lehet felosztani.

3 dimensions to scaling



2.1. ábra. Scaling Cube

Ahogy a 2.1. ábrán képen is látható a meghatározó felbontási fogalmak, az adat menti felbontás, a tetszőleges fogalom menti felbontás, illetve a klónozás.

2.1.1.1. Adat menti felbontás

Az adat menti felbontás annyit tesz, hogy a szolgáltatásokat annak megfelelően bontjuk fel, hogy milyen erőforrással dolgoznak, vagy konkrétan egy adattal kapcsolatos összes funkciót egy helyen készítünk el.

Példa: Erőforrás szerinti felbontás ha külön található szolgáltatás, amivel az adatbázis műveleteket hajtjuk végre, és külön van olyan is, ami csak a HTTP kéréseket szolgálja ki. Az egy adatra épülő módszernél pedig alapul vehetünk egy olyan példát, ahol mondjuk egy

szolgáltatás az összes adminisztrátori funkciót látja el, míg más szolgáltatások a más-más kategóriába eső felhasználók műveleteit hajtják végre.

Mivel a mikroszolgáltatások elve a hardvert is megosztja nem csak a szoftvert, ezért az erőforrás szerinti szétválasztás kissé értelmetlennek tűnhet, azonban a különböző platformok különböző erőforrásait megéri külön szolgáltatásként kezelni. Ha egy mikroszolgáltatást tartunk arra, hogy az adatbázis kéréseket kiszolgálja, akkor az adatbázis nem oszlik meg a szolgáltatások között. Ennek ellenére pazarló lehet minden szolgáltatásnak saját adatbázist fenntartani.

2.1.1.2. Fogalmi felbontás

A tetszőleges fogalom menti felbontás annyit tesz hogy elosztott rendszert hozunk létre tetszőleges funkcionalitás szerint. Erre épít a mikroszolgáltatás architektúra is, mivel a lényege pont az egyes funkciók atomi felbontása.

Példa: Adott egy könyvtár nyilvántartó rendszere, és ezt akarjuk fogalmanként szétvágni. Külön-külön lehet szolgáltatást csinálni a keresésnek, indexelésnek, foglalásnak, kivett könyvek nyilvántartásának, böngészésre, könyvek adatainak tárolására, és kiolvasására, és ehhez hasonló funkciókra. Ezekkel a szétválasztásokkal a könyvtár működését kis részekre bontottuk, és ezek egy-egy kis szolgáltatásként könnyen elérhetők.

2.1.1.3. Klónozás

A harmadik módszer arra tér ki, hogy hogyan lehet egy architektúrát felosztani, hogy skálázható legyen. Itt a klónozhatóság, avagy az egymás melletti kiszolgálás motivál. Ez a mikroszolgáltatásoknál kell, hogy teljesüljön, mivel adott esetben egy terheléelosztó alatt tudnunk kell definiálni több példányt is egy szolgáltatásból. Azért szükséges a skálázhatóság a mikroszolgáltatások esetén, mivel kevés hardver mellett is hatékonyan kialakítható az architektúra, de könnyen lehet szűk keresztmetszetet létrehozni, amit skálázással könnyen megkerülhetünk.

2.1.2. Architektúrális mintákhoz való viszonya

Mint korábban láthattuk vannak bizonyos telepítési módszerek, amik mentén szokás a mikroszolgáltatásokat felépíteni. Van aki az architektúrális tervezési minták közé sorolja a mikroszolgáltatás architektúrát, de nem könnyű meghatározni, hogy hogyan is alkot önnáló mintát. Nagyon sok lehetőség van a mikroszolgáltatásokban, és leginkább más architektúrákkal együtt használva lehet hatékonyan és jól használni.

Nézzünk meg három felhasználható architektúrális mintát:

2.1.2.1. Pipes and Filters

A Pipes and filter architektúrális minta[31] lényege, hogy a funkciókra bontott architektúrát az elérni kívánt végeredmény érdekében különböző módokon összekötjük. Ebben a módszerben az adat folyamatosan áramlik az egyes alkotó elemek között, és lépésről lépésre alakul ki a végeredmény. Elég olcsón kivitelezhető architektúrális minta, mivel csupán

sorba kell kötni hozzá az egyes szolgáltatásokat, azonban nehezen lehet optimalizálni, és könnyen lehet, hogy olyan részek lesznek a feldolgozás közben, amik hátráltatják a teljes folyamatot.

2.1.2.2. Publisher/Subscriber

Egy másik, elosztott rendszerekhez kitallált minta a publisher/subscriber[32], amely azon alapszik, hogy egy szolgáltatásnak szüksége van valamilyen adatra vagy funkcióra, és ezért feliratkozik egy másik szolgáltatásra. Ennek az lesz az eredménye, hogy bizonyos szolgáltatások, bizonyos más szolgáltatásokhoz fognak kötődni, és annak megfelelően fognak egymással kommunikálni, hogy milyen feladatot kell végrehajtaniuk.

2.1.2.3. Esemény alapú architektúra

Az esemény alapú architektúrákat[6] könnyen kalakíthatjuk, ha egy mikroszolgáltatásokból álló rendszerben olyan alkalmazásokat és komponenseket fejlesztünk ahol eseményeken keresztül kommunikálnak az egyes elemek. Ezzel a nézettel olyan struktúrát lehet összeépíteni, ahol a kis egységek szükség szerint kommunikálnak, és a kommunikáció egy jól definiált interfészen keresztül történik.

2.1.3. Eltérések a szolgáltatás orientált architektúrától

A mikroszolgáltatások a szolgáltatás orientált architektúrális minta finomítása, mivel elsősorban szeparált egységeket, önműködő szolgáltatásokat hoz létre, amik életképesek önmagukban is, és amennyire lehet oszthatatlanok. A szolgáltatás orientált esetben viszont a meglévő szolgáltatásainkat kapcsoljuk össze, ami akár egy helyen is futhat és egyáltalán nem az atomicitás a lényege.

2.1.4. Példák mikroszolgáltatásokat használó alkalmazásokra

- **Amazon:** minden Amazon-nal kommunikáló eszköz illetve az egyes funkciók implementációja is szolgáltatásokra van szedve, és ezeket hívják az egyes funkciók (vm indítás, törlés, mozgatás, stb.)
- **eBay:** Különböző műveletek szerint van felbonva a funkcionalitás, és ennek megfelelően külön szolgáltatásként érhető el a fizetés, megrendelés, szállítási információk, stb.
- **NetFlix:** A nagy terhelést elkerülendő bizonyos streaming szolgáltatásokat átalakítottak, hogy a mikroszolgáltatás architektúra szerint működjön.
- **Archivematica:** Az Archivematica[16] egy nyílt forráskódú elektronikus tartalom kezelő, ami tud kezelni különböző fájlokat, multimédiás adatokat, illetve akármi-lyen szöveges tartalmat. Ez az alkalmazás alapvetően monolitikus architektúrára épül, azonban elkezdtek átalakítani a struktúráját mikroszolgáltatásokat használó architektúrára. Ezt úgy kiviteleztek, hogy a különböző plusz funkciókat az eredeti

alkalmazás plugin szerűen mikroszolgáltatásokból nyeri ki, és ennek megfelelően a tovább fejlesztés is megalapozott[5].

2.2. Mikroszolgáltatások előnyei és hátrányai

Ahogy minden architektúrális mintának, a mikroszolgáltatásoknak is vannak előnyei[26], amik indokoltá teszik a minta használatát, és vannak hátrányai[47], amiket mérlegelnünk kell a tervezés folyamán.

2.2.1. Előnyök

2.2.1.1. Könnyű fejleszteni

Mivel kis részekre van szedve az alkalmazásunk, a fejlesztést akár több csapatnak is ki lehet osztani, hogy az alkalmazás részeit alkossák meg, hiszen önállóan is életképesek a szolgáltatások. Az egyes szolgáltatások nem rendelkeznek túl sok logikával, így kis méretű könnyeb kezelhető feladatokkal kell a csapatoknak foglalkozni.

2.2.1.2. Egyszerűen megérthető

Egy szolgáltatás nagyon kis egysége a teljes alkalmazásnak, így könnyen megérthető. Kevés technológia, és kevés kód áll rendelkezésre egy szolgáltatásnál, így gyorsan beletanulhat egy új fejlesztő a munkába. A dokumentáció, átláthatóság, illetve a hibák analizálása közben is jól jön, hogy élesen elvállnak az egyes egységek.

2.2.1.3. Könnyen kicserélhető, módosítható, telepíthető

A szolgáltatások önnálón is működnek, így az azonos interfésszel rendelkező szolgáltatásra bármikor kicserélhető, illetve módosítható ha megmaradnak a korábbi funkciók. A szolgáltatás telepítése is egyszerű, mivel csak kevés környezeti feltétele van annak, hogy egy ilyen kis méretű program működni tudjon. A fejlesztést nagyban segíti, hogy egy korábbi verziójú programba plugin-szerűen be lehet integrálni az újonnan fejlesztett részeket, mivel ez gyors visszajelzést ad a fejlesztőknek. Ez a tulajdonsága a folytonos integrációt támogató eszközöknél is előnyös, mivel könnyen lehet vele automatizált metodológiákat készíteni.

2.2.1.4. Jól skálázható

Mivel sok kis részletből áll az alkalmazásunk, nem szükséges minden funkciónkhoz növelni az erőforrások allokációját, hanem kis komponensekhez is lehet rendelni több erőforrást. Például egy számítási felhőben, a teljesítményben látható változásokat könnyen és gyorsan lehet kezelni, a problémát okozó funkció felskálázásával.

2.2.1.5. Támogatja a kevert technológiákat

Az egyik legnagyobb ereje ennek az architektúrának, hogy képes egy alkalmazáson belül kevert technológiákat is használni. Mivel egy jól definiált interfészen keresztül kommu-

nikálnak a szolgáltatások, ezért mindegy milyen technológia van mögötte, amíg ki tudja szolgálni a feladatát. Ennek megfelelően el tudunk helyezni egy Linux-os környezetben használt LDAP-ot, és egy Windows-os környezetben használt Active Directory-t is, és minden gond nélkül használni is tudjuk őket az interfészek segítségével.

2.2.2. Hátrányok

2.2.2.1. Komplex alkalmazás alakul ki

Mivel minden funkcióra saját szolgáltatást csinálunk, nagyon sok lesz az elkülönülő elem, és a teljes alkalmazás egyben tartása nagyon nehéz feladattá válik. Mivel fontos a szolgáltatások együttműködése, a sok interfésznek ismernie kell egymást, és fenn kell tartani a konzisztenciát minden szolgáltatással.

2.2.2.2. Nehezen kezelhető az elosztott rendszer

A mikroszolgáltatások architektúra egy elosztott rendszert ír le, és mint minden elosztott rendszer ez is bonyolultabb lesz a monolitikus változatánál. Elosztott rendszereknél figyelni kell az adatok konzisztenciáját, a kommunikáció plusz feladatot ad minden szolgáltatás fejlesztőjének, és folyamatosan együtt kell működni a többi szolgáltatás fejlesztőjével.

2.2.2.3. Plusz munkát jelenthet az aszinkron üzenet fogadás

Mivel egy szolgáltatás egyszerre több kérést is ki kell hogy szolgáljon egyszerűbb ha aszinkron módon működik. Ezt azonban mindig le kell implementálni, és az aszinkron üzenetek bonyolítják az adatok kezelését. Az egyes szolgáltatások között könnyen lehetnek adatbázisbeli inkonzisztenciák, mivel aszinkron működés esetén nem minden kiszolgált kérésnek ugyan az a ritmusa. Ugyan nem megoldhatatlan feladat ezeket az időbeli problémákat lekezelni, de plusz komplexitást hozhat be, amit egy közös környezetben lock-olással könnyedén megoldhatnánk.

2.2.2.4. Kód duplikátumok kialakulása

Amikor nagyon hasonló (kis részletben eltérő) szolgáltatásokat csinálunk, megesik, hogy ugyan azt a kódot többször fel kell használnunk, és ezzel kód, és adat duplikátumok keletkeznek, amiket le kell kezelnünk. Nem nehéz találni olyan példát, ahol a létrehozás és szerkesztés művelete megvalósítható ugyan külön szolgáltatásként, viszont nehezíti a feladatot, hogy 2 külön adatbázist kéne módosítani az ideális megvalósításban, és ezek konzisztenciáját fenn kéne tartani.

2.2.2.5. Interfészek fixálódnak

A fejlesztés folyamán a szolgáltatásokhoz rendelt interfészek fixálódnak, és ha módosítani akarunk rajta, akkor több szolgáltatásban is meg kell változtatni az interfészt. Ennek a problémának a megoldása, alapos tervezés, és sokszintű, bonyolult interfész struktúra használatával megoldható.

2.2.2.6. Nehezen tesztelhető egészben

Mivel sok kis részletből rakódik össze a nagy egész alkalmazás, a tesztelési fázisban kell olyan teszteket is végezni, ami a rendszer egészét, és a kész alkalmazást teszteli. Egy ilyen teszt elksézítése bonyolult lehet, és plusz feladatot ad a sok szolgáltatás külön-külön fordítása, és telepítése is.

2.2.3. Összehasonlítva a monolitikus architektúrával

A mikroszolgáltatás architektúra a monolitikus architektúra ellentetje, melyben az erőforrások központilag vannak kezelve, és minden funkció egy nagy interfészen keresztül érhető el. A monolitikus architektúra egyszerűen kiépíthető, könnyű tervezni és fejleszteni, azonban nehezen lehet kicserélni, nem elég robusztus, és nehezen skálázható, mivel az erőforrásokat közösen kezelik a funkciók.

Ezzel ellentétben a mikroszolgáltatás architektúrát ugyan nehezen lehet megtervezni, hiszen egy elosztott rendszert kell megtervezni, ahol az adatátviteltől kezdve az erőforrás megosztáson keresztül semmi sem egyértelmű. A kezdeti nehézségek után viszont a későbbi továbbfejlesztés sokkal egyszerűbb, mivel külön csapatokat lehet rendelni az egyes szolgáltatásokhoz, és könnyen integrálhatók, kicserélhetőek az alkotó elemek. Mivel sok kis egységből áll, könnyebben lehet úgy skálázni a rendszert, hogy ne pazaroljuk el az erőforrásainkat, és ugyanakkor a kis szolgáltatások erőforrásokban is el vannak különítve, így nem okoz gondot, hogy fel vagy le skálázzunk egy szolgáltatást. Ennek az a hátránya, hogy le kell kezelni a skálázáskor a közös erőforrásokat. (Például ha veszünk egy autentikációs szolgáltatást, akkor ha azt fel skálázzuk, meg kell tartanunk a felhasználók listáját, így duplikálni kell az adatbázist, és fenntartani a konzisztenciát) Ugyan csak előnye a mikroszolgáltatás architektúrának, hogy különböző technológiákat lehet keverni vele, mivel az egyes szolgáltatások különböző technológiákkal különböző platformon is futhatnak.

2.3. Technológiai áttekintés

Az integrációhoz olyan technológiákat[35] lehet használni, melyek lehetővé teszik az egyes szolgáltatások elkülönült működését. Ahhoz, hogy jó technológiákat válasszunk, mindeképpen ismernünk kell az igényeket, mivel a technológiák széles köre áll rendelkezésünkre. Fontos szem előtt tartani pár általános érvényű szabályt is[33], ami a mikroszolgáltatások helyes működéséhez kell. Ezek pedig a következők:

- Modulárisan szétválasztani a szolgáltatásokat
- Legyenek egymástól teljesen elkülönítve
- Legyen jól definiált a szolgáltatások kapcsolata

A következő feladatokra kellenek technológiák:

- Hogyan lehet feltelepíteni egy önálló szolgáltatást? (telepítés)
- Hogyan lehet összekötni ezeket a szolgáltatásokat? (automatikus környezet felderítés)

- Hogyan lehet fenntartani, változtatni a szolgáltatások környezetét? (konfiguráció management)
- Hogyan lehet skálázni a szolgáltatást? (skálázás)
- Hogyan lehet egységesen használni a skálázott szolgáltatásokat? (load balance, konzisztencia fenntartás)
- Hogyan lehet virtualizáltan ezt kivitelezni? (virtualizálás)
- A meglévő szolgáltatásokat hogyan tartjuk nyilván? (service registry)
- Hogyan figyeljük meg az alkalmazást működés közben (monitorozás, loggolás)

2.3.1. Futtatókörnyezetek

A mikroszolgáltatásokat valamilyen módon létre kell hozni, egy hosthoz kell rendelni, és az egyes elemeket össze kell kötni. A szolgáltatások telepítéséhez olyan technológiára van szükség amivel könnyen elérhetünk egy távoli gépet, és könnyen kezelhetjük az ottani erőforrásokat. Ehhez a legkézenfekvőbb megoldás a Linux rendszerek esetén az SSH kapcsolaton keresztül végrehajtott Bash parancs, de vannak eszközök, amikkel ezt egyszerűbben és elosztottabban is megtehetjük.

- **Jenkins**[\[24\]](#): A Jenkins egy olyan folytonos integráláshoz kifejlesztett eszköz, mellyel képesek vagyunk különböző funkciókat automatizálni, vagy időzítetten futtani. A Jenkins egy Java alapú webes felülettel rendelkező alkalmazás, amely képes bash parancsokat futtatni, Docker konténereket kezelni, build-eket futtatni, illetve a hozzá fejlesztett plugin-eken keresztül, szinte bármire képes. Támogatja a fürtözést is, így képesek vagyunk Jenkins slave-eket létrehozni, amik a mester szerverrel kommunikálva végzik el a dolgukat. A mikroszolgáltatás architektúrák esetén alkalmas a szolgáltatások telepítésére, és tesztelésére.
- **ElasticBox**[\[11\]](#): Egy olyan alkalmazás, melyben nyilvántarthatjuk az alkalmazásainkat, és könnyen egyszerűen telepíthetjük őket. Támogatja a konfigurációk változását, illetve számos technológiát, amivel karbantarthatjuk a környezetünket (Docker, Puppet, Ansible, Chef, stb). Együtt működik különböző számítási felhő megoldásokkal, mint az AWS, vSphere, Azure, és más környezetek. Hasonlít a Jenkins-re, csupán ki van élezve a mikroszolgáltatás alapú architektúrák vezérlésére (Illetve fizetős a Jenkins-el ellentétben). Mindent végre tud hajtani ami egy mikroszolgáltatás alapú alkalmazáshoz szükséges, teljes körű felügyeletet biztosít. [\[23\]](#)
- **Kubernetes**[\[12\]](#): A Kubernetes az ElasticBox egy opensource változata, ami lényegesen kevesebbet tud, azonban ingyenesen elérhető. Ez a projekt még nagyon gyerekcipőben jár, így nem tudom felhasználni a félév során.

Egyéb lehetőség, hogy a fejlesztő készít magának egy olyan szkriptet, ami elkészíti számára a mikroszolgáltatás alapú architektúrát, és lehetővé teszi az elemek dinamikus kicserélését (ad-hoc megoldás). Ennek a megoldásnak a hátránya hogy nincs támogatva, és minden funkciót külön kell implementálni. Sokkal nagyobb erőforrásokat emészt fel mint egy ingyenes, vagy nyílt forrású megoldást választani.

2.3.2. Környezet felderítési technológiák

Az egyes szolgáltatásoknak meg kell találniuk egymást, hogy megfelelően működhessen a rendszer, azonban ez nem mindig triviális, így szükség van egy olyan alkalmazásra, amivel felderíthetjük az aktív szolgáltatásokat.

- **Consul**[19]: A Hashicorp szolgáltatásfelderítő alkalmazása, amely egy kliens-szerver architektúrának megfelelően megtalálja a környezetében lévő szolgáltatásokat, és figyeli az állapotukat (ha inaktívvá válik egy szolgáltatás a Consul észre veszi). Ez az alkalmazás egy folyamatosan választott mester állomásból és a többi slave állomásból áll. A mester figyeli az alárendelteket, és kezeli a kommunikációt. Egy új slave-et úgy tudunk felvenni, hogy a consul klienssel kapcsolódunk a mesterre. Ha automatizáltan tudjuk vezényelni a feliratkozást, egy nagyon erős eszköz kerül a kezünkbe, mivel eseményeket küldhetünk a szervereknek, és ezekre különböző feladatokat hajthatunk végre.
- **Serf**[?]: A Hashicorp egy másik hálózati feldeítő eszköze, ami elosztottan nagy rendelkezésre állással képes a fürtözött elemek között fenntartani a kapcsolatot. Ugyan elsősorban fürtözött elemek közé tervezték, de tetszőleges kapcsolat feldeítésre, és nyilvántartására alkalmas. A Consul-hoz hasonló módon egy saját protokollon keresztül kommunikál a többi serf ágenssel, de nem szükséges neki szerver-kliens kapcsolat. Képes egyszerű események küldésére, azonban a szolgáltatásokat nem képes külön kezelni, inkább csak a végpont állapotát figyeli.

A Consul-t leszámítva nem nagyon találtam olyan eszközt ami a nekem kellő funkciókat tudta volna, főleg csak bizonyos szolgáltatásokhoz találtam felderítő eszközt. A kézi megoldás itt is lehetséges, mivel saját névfeloldás esetén a névfeloldó szervert használhatjuk az egyes állomások felderítésére, vagy Docker-t használva a Docker hálózatok elérhetővé teszik a szolgáltatásokat a futtató konténer hosztjával.

2.3.3. Konfiguráció management

A telepítéshez és a rendszer állapotának a fenntartásához egy olyan eszköz kell, amivel gyorsan egyszerűen végrehajthatjuk a változtatásainkat, és ha valamit változtatunk egy szolgáltatásban, akkor az összes hozzá hasonló szolgáltatás értesüljön a változtatásról, vagy hajtsa végre ő maga is változtatást.

- **Puppet**[37]: Olyan nyílt forrású megoldás, amellyel leírhatjuk objektum orientáltan, hogy milyen változtatásokat akarunk elérni, és a Puppet elvégzi a változtatásokat. Automatizálja a szolgáltatás változtatásának minden lépését, és egyszerű, gyors megoldást szolgáltat a komplex rendszerbe integráláshoz.
- **Chef**[21]: A Chef egy olyan konfiguráció menedzsment eszköz ami nagy mennyiségű szerver számítógépet képes kezelni, fürtözhető, és megfigyeli az alá szervezett szerverek állapotát. Tartja a kapcsolatot a gépekkel, és ha valamelyik konfiguráció

nem felel meg a definiált receptkönyvnek, (amiben definiálhatjuk az elvárt környezeti paramétereket) akkor változtatásokat indít be, és eléri, hogy a szerver a megfelelő konfigurációval rendelkezzen. Népszerű konfiguráció menedzsment eszköz, amit könnyedén használhatunk integrációhoz, illetve a szolgáltatások cseréjéhez, és karbantartásához.

- **Ansible**[2]: A Chef-hez hasonlóan képes változtatásokat eszközölni a szerver gépeken egy SSH kapcsolaton keresztül, viszont a Chef-el ellentétben nem tartja a folyamatos kapcsolatot. Az Ansible egy tipikusan integrációs célokra kifejlesztett eszköz, amelyhez felvehetjük a gépeket, amiken valamilyen konfigurációs változtatást akarunk végezni, és egy „playbook” segítségével leírhatjuk milyen változásokat kell végrehajtani melyik szerverre. Könnyen irányíthatjuk vele a szolgáltatásokat, és definiálhatunk szolgáltatásonként egy playbook-ot ami mondjuk egy fürtnyi szolgáltatást vezérel. Ez az eszköz hasznos lehet, ha egy szolgáltatásnak elő akarjuk készíteni a környezetet.
- **SaltStack**[22]: A SaltStack nagyon hasonlít a Chef-re, mivel ez a termék is széleskörű felügyeletet, és konfiguráció menedzsmentet kínál számunkra, amit folyamatos kapcsolat fenntartással, és gyors kommunikációval ér el. Az Ansible-höz nagyon hasonlóan konfigurálható, szintén ágens nélküli kapcsolatot tud létesíteni, és a Chef-hez hasonlóan több 10 ezer gépet tud egyszerre karbantartani.

Minden konfigurációs menedzsment eszköznek megvan a saját nyelve, amivel deklaratívan le lehet írni, hogy mit szeretnénk változtatni, és azokat a program beállítja. Erre a feladatra nem nagyon érdemes saját eszközt készíteni, mivel számos megoldás elérhető, és a megvalósítás komoly tervezést, és fejlesztést igényel. Érdemes megemlíteni a Docker konténerek adta lehetőséget, mivel a Docker konténerek gyorsan konfigurálhatók, fejleszthetők, és a konténer gépeken keresztül jól karbantarthatók, így a konfiguráció menedzsment is megoldható velük. Ami hiányzik ebből a megoldásból az a többi szolgáltatás értesítése a változtatásról.

2.3.4. Terheléselosztás

A mikroszolgáltatás alapú architektúrának egyik fontos eleme a terhelés elosztó, vagy valamilyen fürtözést lehetővé tevő eszköz. Ez azért fontos, mert egy egységes interfészt tudunk kialakítani a szolgáltatásaink elérésére, és könnyíti a skálázódást a szolgáltatások mentén.

- **HAProxy**[18] [30]: Egy magas rendelkezésre állást biztosító, és megbízhatóságot növelő terheléselosztó eszköz. Konfigurációs fájlokon keresztül megszervezhetjük, hogy mely gépet hogyan érjük el, milyen IP címek mely szolgáltatásokhoz tartoznak, illetve választhatóan round robin, legkisebb terhelés, session alapú, vagy egyéb módon osztja szét a kéréseket az egyes szerverek között. Ez az eszköz csak és kizárólag a HTTP TCP kéréseket tudja elosztani, de egyszerű, könnyen telepíthető, és könnyen kezelhető (ha nem dinamikusán változnak a fürtben lévő gépek, mert ha igen akkor szükséges egy mellékes frissítő logika is).

- **Nginx**[34]: Az Nginx egy nyílt forráskódú web kiszolgáló és reverse proxy szerver, amivel nagy méretű rendszereket kezelhetünk, és segít az alkalmazás biztonságának megőrzésében. A kiterjesztett változatával (Nginx Plus) képesek lehetünk a terheléselosztásra, és alkalmazás telepítésre. Nem teljesen a proxy szerver szerepét váltja ki, de képes elvégezni azt.

A kézi megvalósítás gyakorlatilag egy kézíleg implementált terheléselosztó eszköz lenne, amihez viszont hálózati megfigyelés, és routing szükséges, így nem javallott ilyen eszköz készítése.

2.3.5. Skálázási technológiák

A mikroszolgáltatás alapú architektúrák egyik nagy előnye, hogy az egyes funkciókra épülő szolgáltatásokat könnyedén lehet skálázni, mivel egy load balancert használva csupán egy újabb gépet kell beszervezni, és máris nagyobb terhelést is elbír a rendszer. Ahhoz hogy ezt kivitelezni tudjuk, szükségünk van egy terheléselosztóra, és egy olyan logikára, ami képes megsokszorozni az erőforrásainkat. Számítási felhő alapú környezetben ez könnyen kivitelezhető, egyébként hideg tartalékban tartott gépek behozatalával elérhető. Sajnálatos módon általános célú skálázó eszköz nincsen a piacon, viszont gyakran készítenek maguknak saját logikát a nagyobb gyártók.

- **Elastic Load Balancer**[1]: Az Amazon AWS-ben az ELB avagy rugalmas terheléselosztó az, ami ezt a célt szolgálja. Ennek a szolgáltatásnak az lenne a lényege, hogy segítse az Amazon Cloud-ban futó virtuális gépek hibatűrését, illetve egységbe szervezi a különböző elérhetőségi zónákban lévő gépeket, amivel gyorsabb elérést tudunk elérni. Mivel ez a szolgáltatás csupán az Amazon AWS-t felhasználva tud működni, nem megfelelő általános célra, azonban ha az Amazon Cloud-ban építjük fel a mikroszolgáltatás alapú architektúránkat, akkor erős eszköz lehet számunkra.
- **Terheléselosztó használata:** Vannak olyan terheléselosztók, melyek bizonyos szolgáltatásokat nyújtó gépeket képesek automatikusan skálázni. Ilyen a HAProxy, és az Nginx is. Például a HAProxy a TCP üzeneteket képes fogadni és elosztani a hozzá beregisztrált gépek között, és beállításától függően képes kezelni automatikusan gépeket elvenni, vagy hozzáadni a hálózathoz.

A skálázás egyik legegyszerűbb megvalósítása, hogy egy proxy szerverrel felhasználva, valamilyen módon egységesen elosztjuk a kéréseket, és egy saját monitorozó eszközzel figyeljük a terhelést (processzor terheltség, memória, hálózati terhelés). Ha valamelyik érték megnő, egy ágenses vagy ágens nélküli technológiával a virtualizált környezetben egy új példányt készítünk a terhelt szolgáltatásból, és a proxy automatikusan megoldja a többi. Nem tökéletes megoldást kapunk, azonban ez a legtöbb felhasználási esetben megfelelőnek bizonyul.

2.3.6. Virtualizációs technológiák

A mikroszolgáltatás alapú architektúrák kialakításánál nagy előnyt jelenthet, ha valamilyen virtualizációt használunk fel a környezet kialakításához. Virtualizált környezetben könnyebb a telepítés, skálázás, és a monitorozás is egyszerűbb lehet.

- **Docker**[10]: Egy konténer virtualizációs eszköz, amelynek segítségével egy adott kernel alatt több különböző környezettel rendelkező, alkalmazásokat futtató környezetet hozhatunk létre. A Docker egy szeparált fájlrendszert hoz létre a gazda gépen, és abban hajt végre műveleteket. Készíthetünk vele előre elkészített alkalmazás környezeteket, és szolgáltatásokat, ami ideálissá teszi mikroszolgáltatás alapú architektúrák létrehozásánál. A Docker konténerek segítségével egyszerűen telepíthetjük, skálázhatjuk, és fejleszthetjük a rendszert.
- **libvirt**[28]: Többféle virtualizációs technológiával együtt működő eszköz, amivel könnyedén irányíthatjuk a virtuális gépeket, és a virtualizálás komolyabb részét el absztrahálja. Támogat KVM-em, XEN-t, VirtualBox-ot, LXC-t, és sok más virtualizáló eszközt. Ezzel az eszközzel a környezet kialakítását szabhatjuk meg, tehát a hardveres erőforrások megosztásában nyújt nagy segítséget.
- **KVM**[27]: A KVM egy kernel szintű virtualizációs eszköz, amivel virtuális gépeket tudunk készíteni. Processzor szintjén képes szétválasztani az erőforrásokat, és ezzel szeparált környezeteket létrehozni. Virtualizál a processzoron kívül hálózati kártyát, háttértárat, grafikus meghajtót, és sok más. A KVM egy nyílt forráskódú projekt és létrehozhatunk vele Linux és Windows gépeket is egyaránt.
- **Kereskedelmi cloud szolgáltatás**: Ha virtualizációról beszélünk, akkor adja magát hogy a számítási felhőket is ide értsük. Egy mikroszolgáltatás architektúrájú programot a legcélszerűbb valamilyen számítási felhőben létrehozni, mivel egy ilyen környezetnek definíciója szerint tartalmaznia kell egy virtualizációs szintet, megosztott erőforrásokat, monitorozást, és egyfajta leltárat a futó példányokról. Ennek megfelelően a mikroszolgáltatás alapú architektúra minden környezeti feltételét lefedi, csupán a szolgáltatásokat, business logikát, és az interfészeket kell elkészítenünk. Jellemzően a Cloud-os környezetek tartalmaznak terheléelosztást, és skálázási megoldást is, amivel szintén erősítik a szolgáltatás alapú architektúrákat. Ilyen környezet lehet az Amazon, Microsoft Azure, Google App Engine, OpenStack, és sokan mások.

Amennyiben nincs a kezünkben egy saját virtualizáló eszköz, a virtualizálás kézi megvalósítása értelmetlen plusz komplexitást ad az alkalmazáshoz.

2.3.7. Szolgáltatás jegyzékek (service registry)

Számon kell tartani, hogy milyen szolgáltatások elérhetők, milyen címen és hány példányban az architektúránkban, és ehhez valamilyen szolgáltatás nyilvántartási eszközt[39] [36] kell használnunk.

- **Eureka**[29]: Az Eureka a Netflix fejlesztése, egy AWS környezetben működő terheléselosztó alkalmazás, ami figyeli a felvett szolgáltatásokat, és így mint nyilvántartás is megfelelő. A kommunikációt és a kapcsolatot egy Java nyelven írt szerver és kliens biztosítja, ami a teljes logikát megvalósítja. Együtt működik a Netflix által fejlesztett Asgard nevezetű alkalmazással, ami az AWS szolgáltatásokhoz való hozzáférést segíti. Ugyan ez az eszköz erősen optimalizált az Amazon Cloud szolgáltatásaihoz, de a leírás alapján megállja a helyét önállóan is. Mivel nyílt forráskódú, mintát szolgáltat egyéb alkalmazásoknak is.
- **Consul**: Korábban már említettem ezt az eszközt, mivel abban segít, hogy felismerjék egymást a szolgáltatások. A kapcsolatot vizsgáló és felderítő logikán kívül tartalmaz egy nyilvántartást is a beregisztrált szolgáltatásokról, amiknek az állapotát is vizsgálhatjuk.
- **Apache Zookeeper**[3]: A Zookeeper egy központosított szolgáltatás konfigurációs adatok és hálózati adatok karbantartására, ami támogatja az elosztott működést, és a szerverek csoportosítását. Az alkalmazást elosztott alkalmazás fejlesztésre, és komplex rendszer felügyeletére és telepítés segítésére tervezték. A consulhoz hasonlóan működik, és a feladata is ugyan az.

Kézi megoldás erre nem nagyon van, csupán egy központi adatbázisban, vagy leltár alkalmazásban elmentet adatokból tudunk valamilyen jegyzéket csinálni, amihez viszont a szolgáltatások mindegyikének hozzá kell férni. Könnyen konfigurálható megoldást kapunk, és tetszőleges adatot menthetünk a szolgáltatásokról, de egyéb funkciók, mint az esemény küldés és fogadás, csak bonyolult implementációval lehetséges.

2.3.8. Monitorozás, loggolás

Ha már megépítettük a mikroszolgáltatás alapú architektúrát, akkor meg kell bizonyosodnunk róla, hogy minden megfelelően működik, és minden rendben zajlik a szolgáltatásokkal. Ezekhez az adatokhoz többféle módon és többféle eszközzel is hozzáférhetünk, mivel az alkalmazás hibákat egy log szerver, a környezeti problémákat egy monitorozó szerver tudja megfelelően megmutatni számunkra[4] [17].

- **Zabbix**[48]: A Zabbix egy sok területen felhasznált, több 10 ezer szervert párhuzamosan megfigyelni képes, akármilyen adatot tárolni képes monitorozó alkalmazás, ami képes elosztott működésre, és virtuális környezetekben jól használható. Ágens nélküli és ágenses adatgyűjtésre is képes, és az adatokat különböző módokon képes megjeleníteni (földrajzi elhelyezkedés, gráfos megjelenítés, stb.). Nem egészen a mikroszolgáltatás alapú architektúrákhoz lett kialakítva, de egy elég általános eszköz, hogy felhasználható legyen ilyen célra is.
- **Elasticsearch + Kibana**[13] + **LogStash**[14]: A Kibana egy ingyenes adatmegjelenítő és adatfeldolgozó eszköz, amit az Elasticsearch fejlesztett ki, és a Logstash

pedig egy log server, amivel tárolhatjuk a loggolási adatainkat, és egyszerűen kereshe-
tünk benne. Kifejezetten adatfeldolgozásra szolgál mind a két eszköz, és közvetlenül
együttműködnek az Elasticsearch alkalmazással.

- **Sensu**[42]: A Sensu egy egyszerű monitorozó eszköz, amivel megfigyelhetjük a szer-
vereinket. Támogatja Ansible Chef, Puppet használatát, és támogatja a plugin-szerű
bővíthetőséget. A felülete letisztult, és elég jó áttekintést ad a szerverek állapotáról.
Figyel a dinamikus változásokra, és gyorsan lekezel a változásokkal járó riasztá-
sokat. Ezek a tulajdonságai teszik a számítási felhőkben könnyen és hatékonyan
felhasználhatóvá.
- **Cronitor**[8] [7]: Ez a monitorozó eszköz mikroszolgáltatások és cron job-ok megfigye-
lésére lett kifejlesztve, HTTP-n keresztül kommunikál, és a szolgáltatások állapotát
figyeli. Nem túl széleskörű eszköz, azonban ha csak a szolgáltatások állapota érdekel
hasznos lehet, és segíthet a szolgáltatás jegyzék képzésében is.
- **Ruxit**[?] [41]: Egy számítási felhőben működő monitorozó eszköz, amivel teljesítmény
monitorozást, elérhetőség monitorozást, és figyelmeztetés küldést végezhetünk. Az
benne a különleges, hogy mesterséges intelligencia figyeli a szervereket, és kianalizálja
a szerver állapotát, és a figyelmeztetéseket is követi. Könnyen skálázható, és használat
alapú bérezése van. Ez a választás akkor jön jól, ha olyan feladatot szánunk az
alkalmazásunknak, ami esetleg időben nagyon változó terhelést mutat, és az itt kapot
riasztások szerint akarunk skálázni.

A monitorozás kézi megvalósítása egyszerűen kivitelezhető, ha van egy központi adat-
bázisunk, amit minden szolgáltatás elér, és ebben az adatbázisban a szolgáltatásokba
ültetett egyszerű logika küldhet adatokat, amit valamilyen egyszerű módszerrel megjele-
nítve, valamilyen monitorozást érhetünk el. Ennek egyik előnye, hogy nem kell komplex
eszközt telepíteni mindenhova, és nem kell karban tartani, hátránya viszont, hogy nehezen
karbantartható, minden szolgáltatásra külön kell elkészíteni, és a fenti megoldásokkal
ellentétben a semmiből kell kiindulni.

2.4. Kommunikációs módszerek

A szolgáltatások közötti kommunikáció nincs lekötve de jellemző a REST-es API, vagy a
webservice-re jellemző XML alapú kommunikáció[43].

2.4.1. Technológiák

A kommunikáció megtervezéséhez egy jó leírást olvashatunk az Nginx egyik cikkében[40].
Ez a cikk leírja, hogy fontos előre eltervezni, hogy a szolgáltatások egyszerre több másik
szolgáltatással is kommunikálnak vagy sem, illetve szinkron vagy aszinkron módon akarunk-
e kommunikálni. A cikk kifejti, hogy az egyes technológiák hogyan használhatók jól, és
hogyan lehet várakoztatási sorral javítani a kiszolgáláson.

2.4.1.1. REST (HTTP/JSON):

A RESTful[25] kommunikáció egy HTTP feletti kommunikációs fajta, aminek az alapja az erőforrások megjelölése egyedi azonosítókkal, és hálózaton keresztül műveletek végzése a HTTP funkciókat felhasználva. Ez a módszer napjainkban nagyon népszerű, mivel egyszerű kivitelezni, gyakorlatilag minden programozási nyelv támogatja, és nagyon egyszerűen építhetünk vele interfészeket. Az üzenetek törzsét a JSON tartalom adja, ami egy kulcs érték párokból álló adatstruktúra, és sok nyelv támogatja az objektumokkal való kommunikációt JSON adatokon keresztül (Sorosítással megoldható). Mikroszolgáltatások esetén az aszinkron változat használata az előnyösebb[46], mivel ekkor több kérést is ki tudunk szolgálni egyszerre, és a szolgáltatásoknak nem kell várniuk a szinkron üzenetek kiszolgálására.

2.4.1.2. SOAP (HTTP/XML):

A szolgáltatás alapú architektúrákban nagyon népszerű a SOAP[45], mivel tetszőleges interfészt definiálhatunk, és le lehet vele képezni objektumokat is. Kötött üzenetei vannak, amiket egy XML formátumu üzenet ír le. Ebben az esetben nem erőforrásokat jelölnek az URL-ek, hanem végpontokat, amik mögött implementálva van a funkcionalitás. Ennek a kommunikációs módszernek az az előnye, hogy jól bejáratot, széleskörben használt technológiáról van szó, amit jól lehet használni objektum orientált nyelvek közötti adatátvitelre. Hátránya, hogy nagyobb a sávszélesség igénye, és lassabb a REST-es megoldásnál. Létezik szinkron és aszinkron megvalósítása is és mivel ez a kommunikációs fajta is HTTP felett történik, a REST-hez hasonló okokból az aszinkron változat a célszerűbb.

2.4.1.3. Socket (TCP):

A socket[44] kapcsolat egy TCP feletti kapcsolat, ami egy folytonos kommunikációs csatornát jelent az egyes szolgáltatások között. Ez azért lehet előnyös, mert a folytonos kapcsolat fix útvonalat és fix kiszolgálást jelent, amivel gyors és egyszerű kommunikációt lehet végrehajtani. A három technológia közül ez a leggyorsabb és a legkisebb sávszélességet igénylő, azonban nincs definiálva az üzenetek formátuma (protokollt kell hozzá készíteni), és az aszinkron elv nem összeegyeztethető vele, így üzenetsorokat kell létrehozni a párhuzamos kiszolgáláshoz. Indokolt esetben sok előnye lehet egy mikroszolgáltatásokra épülő struktúrában is, azonban általános esetben nem igazán használható kommunikációs eszköz.

2.4.2. Interfészek

A korábban említett Nginx-es cikk kitért arra is, hogy az interfészek megalkotása milyen gondokkal járhat, és mi az előnye, hogyan lehet úgy megtervezni őket, hogy ne legyen sok gondunk velük.

Az interfészeket úgy kell megtervezni, hogy könnyen alkalmazhatók legyenek, képesek legyünk minden funkciót teljes mértékben használni, és később bővíthető legyen (visszafelé kompatibilis). Erre egy megoldás a RESTful technológiáknál a verziózott URL-ek

használata, amivel implicit módon megmondhatjuk, hogy melyik verziójú interfészre van szükségünk éppen. Ha rosszul tervezzük meg az interfészek struktúráját, és nem készítjük fel a szolgáltatásokat egy lehetséges interfész változtatásra, akkor könnyen lehet, hogy nagy mennyiségű plusz munkát adunk a fejlesztőknek, akiknek minden hívást karban kell tartaniuk.

3. fejezet

Feladat megtervezése

3.1. Minta alkalmazás tervezése

A feladatom egyik része az volt, hogy egy olyan minta alkalmazást készítssek, amelyiken keresztül be lehet mutatni a mikroszolgáltatások fejlesztésének, és tesztelésének a lépéseit, és jól reprezentálja az architektúrális minta jellegzeteségeit. Minta feladatnak egy webes könyvtárház webes kiszolgáló felületét választottam, mivel ez nem túl bonyolult, és könnyen meghatározhatók benne az elkülönülő szolgáltatások.

3.1.1. Minta alkalmazás

Egy webes könyvesbolt esetén szükség van arra, hogy képesek legyünk megtalálni azt a könyvet, amit meg akarunk venni, képesek legyünk bejelentkezni, hogy hozzánk rendelhesse a rendszer az adott könyvet, és képesnek kell lennie a vásárlás lebonyolítására.

Ahhoz hogy ezeket képes legyen véghez vinni, a következőkre van szükség: * Egy olyan felület, ahol a felhasználó könnyen és egyszerűen tájékozódhat a bolt kínálatáról, * egy mögöttes adatbázis, amiben megtalálhatók az adatok * és felhasználó bejelentkezéséhez egy autentikációs szolgáltatás

A minta alkalmazás tetszőleges webes felületen elérhető szolgáltatás lehetett volna, mivel működésükből fakadóan követik a mikroszolgáltatásokat. A webes kiszolgálás esetén különböző URL-eken keresztül érhetjük el az egyes funkciókat, amik leggyakrabban elemien kicsire vannak tervezve.

3.1.2. Szolgáltatások

A minta alkalmazást először is a korábban már felvázolt módon (2.1.1. fejezet) fel kell bontanom funkciókra, amikre a szolgáltatásaim épülni fognak. Egy webes könyvesbolttal kapcsolatban olyan szavakkal találkozhatunk, mint a böngészés, vásárlás, vagy bejelentkezés. Ezekre a funkciókra határoztam meg szolgáltatásokat, illetve az ehhez kellő más erőforrás vezérlő szolgáltatásokat.

Alkalmazás szolgáltatásai:

- Bejelentkezés: Ennek a szolgáltatásnak az a célja, hogy a beregisztrált felhasználók

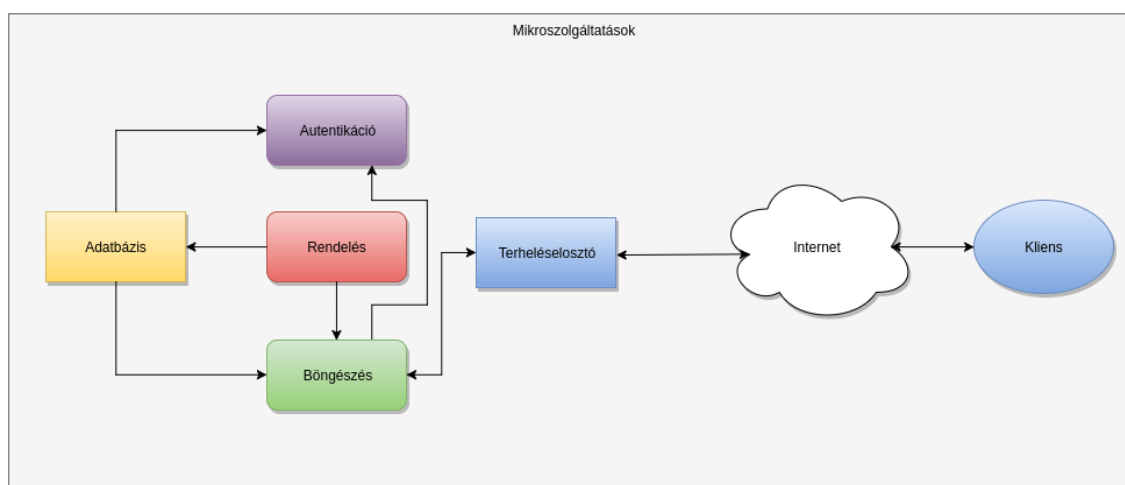
képesek legyenek belépni, és magukhoz rendelni a megrendeléseket.

- Vásárlás: A vásárlók megrendeléseit, és a készlet csökkentését végzi.
- Böngészés: Megjeleníti az boltban lévő könyveket, amiket meg lehet venni, illetve információt szolgáltat a mennyiségről is, hogy lehessen tudni, ha valamelyik könyv már nem kapható.

Környezethez tartozó egyéb szolgáltatások:

- Adatbáziskezelő: Mivel minden szolgáltatásnak valamilyen módon közös adathalmazon kell dolgoznia, kényelmesebb lehet egy külső szolgáltatás formájában elérhetővé tenni az adatbázist, amit több funkció is módosíthat.
- Terheléelosztó: A mikroszolgáltatások egyik legnagyobb előnye, hogy szabadon és egyszerűen skálázható. Ezt a tulajdonságot egy terheléelosztón keresztül könnyen meg tudom oldani.

Egy webes áruháznak lehet sokkal több alkotó eleme is, hiszen keresés, és korás funkciók nem lettek felsorolva, viszont az alap funkciókat tartalmazza, és képes kiszolgálni a felhasználókat így elegendő a feladat szempontjából.



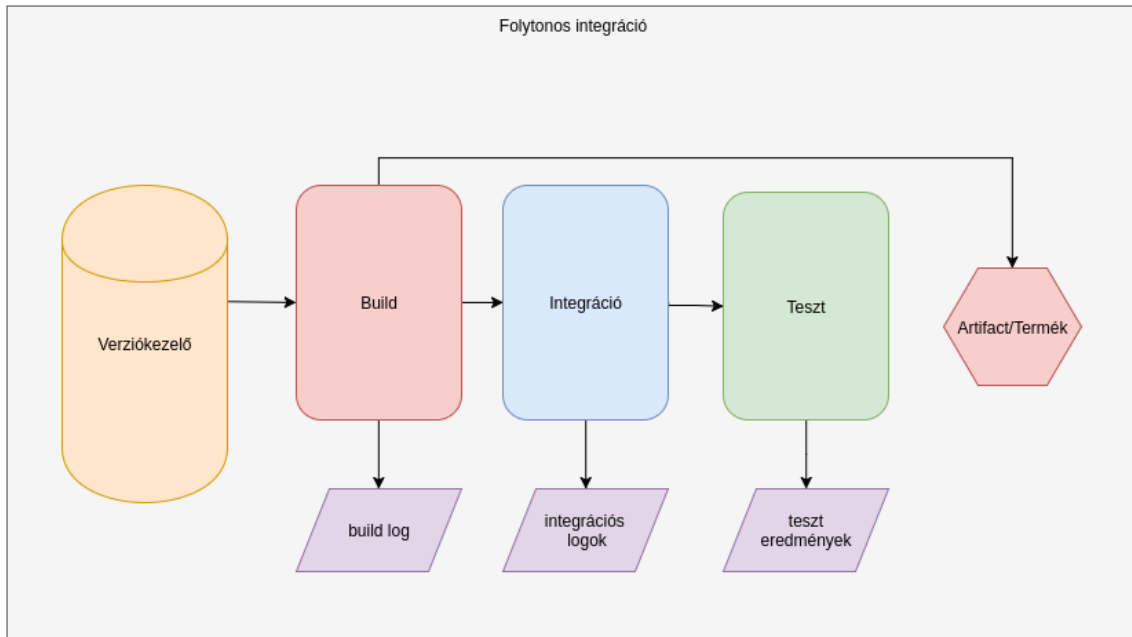
3.1. ábra. Mikroszolgáltatások terve

3.1.3. Kommunikáció

A szolgáltatások közötti kommunikáció alapja egy olyan széles körben használt protokollon fog alapulni, amivel könnyen lehet tervezni, könnyű implementálni, és hatékonyan képes a kéréseket kezelni. Ahhoz hogy a szolgáltatások egymás között kommunikálni tudjanak, szükség lesz egy nyilvántartó eszközre, hogy az egyes szeparált szolgáltatások valahogy egymásra találjanak.

3.2. Folytonos Integráció

A folytonos integráció[15] (continuous integration) egy jól bevált szoftver fejlesztési gyakorlat, ami azt a célt szolgálja, hogy automatizáltan képesek legyünk a fejlesztett alkalmazásról megmondani, hogy jól funkcionál-e. Az elmélet különböző fázisokat különböztet meg, amiket azért kell véghezvinni, hogy könnyen és egyszerűen tudjuk integrálni a tesztelendő alkalmazást, illetve gyorsítja a visszajelzés folyamatát. Ezeket a fázisokat mutatja az ?? ábra.



3.2. ábra. Folytonos integráció fázisai

- *Verziókezelő*: Ahhoz, hogy követni lehessen a változásokat, és a forrásokat meg tudjuk szerezni szervezett, következő módon, egy verziókezelőre van szükség.
- *Automatizált build rendszer*: A kinyert változtatások alapján el kell készíteni az alkalmazás futtatható és végleges formáját, amit build formájában nyerhetünk ki. Ez sok esetben egy konkrét csomag elkészítése, ami Linux rendszerek esetén egy Debian, vagy RPM csomag, vagy egy olyan termék az eredménye, amit egy az egyben fel lehet telepíteni tetszőleges rendszerre. Ennek a fázisnak a kimenete legalább egy log fájl, ami tartalmazza a build folyamat minden lépését, és a hibákat tartalmazza, ha valami rosszul sikerül, illetve egy olyan artifact, ami sikeres build esetén felhasználható mint maga a termék. A folyamat során teszteket is végrehajthatunk, mivel a leggyakrabban a build folyamat részeként futnak le az egység tesztek, amik az alap funkciók működő képességéről tanúskodnak.
- *Automatikus integráció*: Ha már van egy sikeres build-ünk, akkor azt a bizonyos eredmény artifact-ot valamilyen környezetbe integrálni kell, hogy a valós körülményeknek megfelelően tesztelhessük őket. A környezet maga lehet virtuális vagy valós, és lehetséges, hogy szükség van a telepítés előtt felkészítő folyamatokra is, melyek

részét képezik ennek a fázisnak. Egy ilyen előzetes felkészítés lehet például a tűzfal helyes beállítása, ha az alkalmazásunk külső hálózati kapcsolatot is használ.

- *Alkalmazás valós környezetben való tesztelése:* Ebben a fázisban történik az alkalmazás széleskörű funkcionális és stressz tesztelése, ami azt jelenti, hogy az éles környezetben futó alkalmazást olyan bemeneteknek, és eseményeknek tesszük ki, hogy a valóságot lehetőleg jobban megközelítsük.
- *Kiértékelhető eredmények mentése:* Minden fázisnak van valamilyen információval bíró kimenetele, amiket el kell menteni egy olyan helyre, ahol bármikor visszakövethető, és kikereshetők az eredmények. A build folyamatnak, az integrációnak, és a teszt eredményeknek olyan kimenetei is vannak, melyek alapján a konkrét környezet elérése nélkül is képesek lehetünk megmondani, hogy mi is volt a hiba forrása.

A folytonos integrációt nem csak ilyen módon lehet felhasználni, hanem majdnem tetszőleges folyamat kidolgozható hozzá, ha a valamilyen egyszerű módon képesek vagyunk a változtatásokat időről időre integrálni és tesztelni. Nem szükséges hozzá, hogy a teljes folyamat automatizált legyen, azonban megkönnyíti a fejlesztők dolgát, ha minden fázis automatikusra van készítve.

A folytonos integrációhoz hasonlóan létezik egy olyan fejlesztési gyakorlat, ami a termék kiadására vonatkozóan írja le a folyamatot. Ez a folytonos szállítás[20] (continuous delivery), ami kicsit szabadabban van megfogalmazva, és akár részének tekinthető a folytonos integráció. A folytonos szállítást, a már megismert fázisokon kívül egy fejlesztési, és egy kiadási fázissal toldották meg, ami a megjelenő termék kiadását, és a kód fejlesztésének meghatározását takarja.

3.2.1. Használati módok mikroszolgáltatások esetén

Mikroszolgáltatások esetén a folytonos integrációt támogató rendszereket hatékonyan lehet fölhasználni, mivel minden szolgáltatás külön termékként képzelhető el, és a közös integráció is egy fontos tesztelendő elem, amit a fejlesztő csapatok csak nagyon nehézkesen tudnának megoldani kézzel.

Egy módszer a folytonos integráció felhasználására, ha a mikroszolgáltatásokat automatizáltan elkészítjük, és az egyes szolgáltatásokat magukban kipróbáljuk, majd a nagy egységbe foglalt alkalmazást hozzáértő emberek kezében hagyjuk, és nem törődünk vele.

Másik megközelítés lehet, ha az egyes szolgáltatásokat nem kezeljük külön, hanem egyben mindent elkészítünk, és az egész alkalmazást teszteljük automatizáltan. Ekkor persze felmerülhet az a gond, hogy nehezebben tudjuk megmondani, melyik szolgáltatás hiábja okozta a végleges termék hibáját, azonban megfelelő információk kinyerésével ez sem okozhat gondot.

Végül a teljesen automatizált változatnál külön elkészítjük a szolgáltatásokat, és futtatjuk a hozzájuk tartozó teszteket, és ha minden rendben ment az összes többi szolgáltatással együtt még robosztusabb, és alkalmazás szempontjából kritikus teszteket futtatunk automatizáltan.

Azt, hogy melyik a leghatékonyabb megoldás, azt csak konkrét felhasználók tudják

megmondani, mivel lehet hogy az erőforrások, lehet hogy az igény, de az is lehet, hogy a praktikusság fogja megszabni melyiket is válasszuk.

A teljesen automatizált folytonos integrációs keretrendszer esetén nagy mennyiségű erőforrásra van szükség, mivel az egyes szolgáltatások tesztelése, és a teljes alkalmazás tesztelése is külön erőforrásokat igényel, viszont ha van elég erőforrás, ez lehet a legjobb döntés.

A kis elemi szolgáltatásokat figyelő keretrendszer esetén sokkal kevesebb erőforrással is elég, és sokkal gyorsabb visszajelzést ad a fejlesztő csapatnak, mivel nem kell kivárniuk a komplex alkalmazás tesztjeinek az eredményeit. Ennek lehet az a hátulütője, hogy később kapunk információt egy hiba jelenlétéről, ami sok időt elvehet a fejlesztésből, így vigyázni kell ebben az esetben.

A teljes alkalmazást figyelő integrációs keretrendszerrel megkapjuk a folyamat végén az eredményt, ami valós eredményt ad a szolgáltatások működéséről, azonban nagyon lassú visszajelzési forma lehet ez. Ha minden szolgáltatást együtt nézünk, a tesztelés hasonlóan működik, mint egy monolitikus program esetén, így nem célszerű így tesztelni mikroszolgáltatások esetén, azonban kevesebb erőforrást vihet el ez a megoldás, mint a korábbi kettő.

A feladat elvégzéséhez próbáltam egy mindent automatizáló keretrendszert létrehozni, hogy maximalizálni tudjam a mikroszolgáltatásokhoz adott előnyöket.

3.2.2. Keretrendszer előnyei a fejlesztésre nézve

A feladat részeként el kell készítenem egy folytonos integrációt támogató keretrendszert, ami a korábban felsorolt tulajdonságokat, és feladatokat képes végrehajtani. Egy mikroszolgáltatás alapú alkalmazás fejlesztése közben egy ilyen keretrendszer segít a gyors visszajelzésben, mivel egy szolgáltatás fejlesztése közben nem lehetünk biztosak benne, hogy minden esetben a teljes alkalmazás működő képes, illetve segít felderíteni a szolgáltatás interfészek közötti kritikus eltéréseket.

A gyors visszajelzés mindig fontos, hiszen lassú, és erőforrás igényes feladat, ha a fejlesztő csapatnak kell kipróbálnia a szolgáltatást mind magában egyedileg, mind a komplex alkalmazás részeként. Ezt a költséget megspóroljuk, ha központileg fut az ellenőrzés, és a közös erőforrásokat is könnyebben lehet optimalizálni. Az összes csapatnak adható olyan folytonos integrációs struktúra, amely a szolgáltatás változtatása esetén egy build folyamatot futtatva megpróbálja integrálni az eredményként kapott alkalmazás részletet, és teszteli az együttműködő képességet. Mivel közös erőforrásokon fut nem kell várni, hogy szabad időszávot kapjon a csapat, és az automatizáltság segít, hogy a csapat másra fordíthassa a figyelmet, amíg nem kap eredményt.

Az interfészek figyelését ugyan az az infrastruktúra figyelheti ami az integrációt, és a szolgáltatás helyességét figyeli, de ebben a tesztelési logikában szerepelnie kell egy olyan interfész tesztelésnek, ami képes detektálni azt, hogy a jelenlegi interfészekkel kompatibilis a szolgáltatás, illetve azt is, hogy visszafelé, korábbi verziókkal kompatibilis-e az új kialakítás.

Fejlesztés szempontjából a folytonos integrációs eszköz tartalmazhat olyan logikát, ami

a közvetlen változásokra fut le, és intelligens módon határozza meg a változás hatásait. Az egyik legnépszerűbb folytonos integrációt támogató rendszerben a Jenkins-ben például van plugin minden verziókezelőhöz, amely képes a változtatások felküldésére olyan feladatokat futtatni, amik a kód minőségét (kódolási technika, formázottság, dokumentáció generálás, stb.) figyelik és javítják. Ennek a funkciónak a használata ugyan úgy hasznos lehet mikroszolgáltatások esetén, mint bármilyen fejlesztési módszer esetén.

3.2.3. Lépések bemutatása

A feladat tervezése közben próbáltam minden szempontot szem előtt tartani, és a következő részeket határoztam meg a minta alkalmazásomhoz.

- Buildelés minden szolgáltatásra: Mivel minden szolgáltatás egyedi, és a szolgáltatások külön termékként kezelésével növelhetjük a modularitást, és az újrafelhasználhatóságot, így minden szolgáltatásnak külön build folyamatot terveztem, amik eredményeként az önműködő alkalmazás részleteket kapjuk meg.
- Alkalmazás indítása, minden szolgáltatással: Ha minden részlet elkészült, akkor az összes szolgáltatás indításával és a környezet felkészítésével egy minta környezetet készítünk amiben az alkalmazás fut.
- Tesztek futtatása: A minta környezetben teszteket futtatok, amivel megbizonyosodhatom az alkalmazás működőképességéről.
- Utómunkálatok elvégzése: Mivel a környezet a saját gépem lesz, ezért a nem használt elemek törlése, és a környezet kitisztítása, illetve az eredmények lementése kerül ide.

4. fejezet

Feladat implementációja

A minta alkalmazás implementálása során próbáltam a ma legnépszerűbb és leggyakrabban használt technológiákat használni, és a terveknek megfelelő legjobb megvalósítást.

4.1. Felhasznált technológiák

A minta alkalmazás szolgáltatásaiban több kevert technológiát használtam, hogy be tudjam mutatni a mikroszolgáltatásokon alapuló alkalmazások legnagyobb erősségét, a kerver technológiás megvalósítást. A kiszolgálási logikát Python, Java és PHP nyelveken írtam, és használtam Apache webkiszolgáló alkalmazást, és Nginx webkiszolgálót, és reverse proxy szerveret. A szolgáltatások futtatásához bash szkripteket készítettem, és Docker konténereket használtam fel a környezetfüggetlenség eléréséhez. Az adatbázis kezelésére MySQL adatbázis kezelő szerveret használtam, mivel ez a legelterjedtebb ingyenesen használható adatbázis kezelő.

Python nyelven nagyon egyszerű implementálni egy webkiszolgálót, amin keresztül a kiszolgáló interfészt elkészíthetem, illetve a program logika és adatbázis kapcsolat is könnyen megvalósítható, mivel rengeteg elkészített könyvtár áll rendelkezésemre. A Python egy széles körben felhasznált technológia, így egy valós mikroszolgáltatás alapú alkalmazásba is nagy valószínűséggel belekerülne.

A Java egy platform független nyelv, amit mind kliens oldali alkalmazásokhoz, mind szerver alkalmazások elkészítéséhez is használják, és viszonylag öregebb nyelv amihez rengeteg típusú hálózati kommunikációs protokollt implementáltak, így könnyen és széles körben használható nyelv. Egy másik előnye a feladat szempontjából, hogy fordítás szükséges hozzá, és így a build folyamatban egy jar állomány elkészítését is be tudom mutatni.

A PHP egy olyan dinamikus webkiszolgáló nyelv, amit már hosszú ideje használnak, és elég a feladathoz tartozó egyszerű webes böngésző felület elkészítéséhez. Ezzel a nyelvvel gyorsan tudok dolgozni, mivel korábban is találkoztam a nyelvvel.

Az Nginx a Java-s alkalmazások kiszolgálásához használt leggyakoribb webkiszolgáló szerver, az Apache webszerver pedig a statikus és dinamikus tartalmak kiszolgálásban gyakran használt eszköz.

A Docker a mikroszolgáltatások gyakran használt virtualizáló eszköze, amit gyors,

és egyszerű használata teszi alkalmassá. A Docker által létrehozott virtuális konténer környezetben pontosan annyi van, amennyit a szolgáltatáshoz fel akarunk használni, de könnyen bővíthető, karbantartható, és kicserélhetők a konténerek.

A kommunikációhoz, és a szolgáltatások egymásra találásához a Consul szolgáltatás-felderítő eszközt használtam, amivel már volt korábbi tapasztalatom. A Consul könnyen telepíthető, és egyszerűen használható, és a piacon található más termékekkel ellentétben ingyenes, és gyorsan fejlődő eszköz. Minden funkciót tartalmaz amire szükségem van, és sokkal könnyebben bekonfigurálható, mint a mikroszolgáltatásoknál gyakran használt Apache Zookeeper.

4.2. Szolgáltatások implementálás

Minden szolgáltatáshoz külön el kellett készítenem a környezetet, és a szolgáltatást futtató logikát, amit jól elkülöníthető módon tudtam véghez vinni.

4.2.1. Docker konténerek

Szolgáltatásonként készítettem egy Docker image-et, amiből elindíthatók az alkalmazás részei. Minden konténer egyedi, de vannak olyan részei amik minden szolgáltatáshoz szükségesek. Egy ilyen rész a Consul alkalmazás telepítése, és a konfiguráció bemásolása:

```
FROM ...

# Install consul
COPY consul consul-template /usr/bin/
RUN chmod +x /usr/bin/consul && \
    chmod +x /usr/bin/consul-template && \
    mkdir -p /etc/consul.d
COPY <service>.json /etc/consul.d/<service>.json
...
```

Az eszköz telepítése egy egyszerű fájl másolásból áll, és a jogosultságok beállítása után, már használható is. Minden szolgáltatáshoz tartozik egy JSON fájl, ami tartalmazza a szolgáltatás Consul-hoz kapcsolódó adatait, mint a felhasznált kommunikációs port, és a szolgáltatás megnevezése, kategorizálása.

Ami minden szolgáltatásnál különbözik, az a telepített alkalmazások listája, amik aszerint lettek meghatározva, hogy milyen függőségei vannak a szolgáltatást futtató programnak. Ez a rész a Docker image disztribúciójának megfelelő telepítő programmal történik. Az adatbáziskezelő esetében nem volt ennyire egyszerű a helyzet, ugyanis a **mysql-server** csomag telepítése, egy interaktív választ kér a felhasználótól, amit automatizáltan a következő módon tudunk megtenni:

```
FROM ...
```



```

RUN apt-get -y update && \
    /bin/bash -c "debconf-set-selections \
        <<< 'mysql-server mysql-server/root_password password root'" && \
    /bin/bash -c "debconf-set-selections \
        <<< 'mysql-server mysql-server/root_password_again password root'" && \
    apt-get -y install mysql-server
...

```

Minden Docker image készítésénél az első telepítéshez kellő parancs a repository frissítése, ami jelen esetben az **apt-get update**. Ha ezt nem tesszük meg akkor az alap image ismeretlenül régi csomaglistáját használjuk, ami azt eredményezi, hogy a telepítő nem talál csomagokat, illetve nem a jó verziót találja meg. A **debconf-set-selections** egy olyan program ami lehetővé teszi, hogy a debian csomagkezelő kérdéseire automatikusan tudjunk válaszolni.

Minden szolgáltatáshoz tartozik egy indító szkript, ami tartalmazza az alkalmazás többi elemének a megkeresését, és a szolgáltatás elindításához tartozó lépéseket. Az elkészült Dockerfile-ok amikkel az image-eket építettem, és az indító szkriptek megtalálhatók a függelékben (A.1. és A.2.2.1. fejezet).

4.2.2. Alkalmazás részletek

Minden szolgáltatáshoz tartozik egy vagy több forrásfájl, ami tartalmazza a kiszolgáló kódot. A kiszolgáló kód egy HTTP protokollal elérhető webkiszolgáló, mivel az általam választott Rest-es kommunikációhoz erre van szükség.

Az autentikációt lehetővé tevő szolgáltatás tartalmaz egy **Python Flask** implementációt, ami egy olyan webkiszolgáló Python könyvtár, amivel az egyes URL-eket metódusokhoz rendelhetjük. Ezen kívül egy adatbázis elérést tartalmaz, amit a **MySQLdb** modul segítségével implementáltam. Az egyszerűség kedvéért, az adatbázis már a kezdetektől fogva fel van töltve, és nem lehet bele új felhasználót felvenni. Ha valaki szerepel az adatbázisban, és a jelszava a neki megadott jelszó, akkor sikeresnek jelzem az autentikációt, egyébként „HTTP 401 Unauthorized” üzenettel jelzem a hibát.

Az adatbázishoz nem kellett külön szolgáltatáskódot írnom, mivel a MySQL adatbázis önmagában is képes kezelni a felé eső kéréseket. A szolgáltatást használat előtt töltöm fel adatokkal, amiken keresztül be tudom mutatni a működést. Az adatok betöltését kisebb SQL nyelven írt szkriptekkel írtam le. A könyvekhez tartozó adatbázis léterhozása például a következő képpen néz ki:

```

...
CREATE TABLE store
(
    store_id int NOT NULL AUTO_INCREMENT,
    book_name varchar(255) NOT NULL,
    count int NOT NULL,

```

```

        PRIMARY KEY (store_id)
    );
    ...

```

Ahogy látható a könyvekhez tartozik egy név és egy számosságot jelző count' mező, illetve egy kulcs érték, amin keresztül minden könyv egyedien megkülönböztethető. A többi adattábla ehhez hasonlóan van elkészítve, és a hozzájuk tartozó kód megtekinthető a függelékben (A.2.2.3. fejezet).

A böngészéshez tisztán PHP kódot írtam, amit az Apache HTTP szerver ajánl ki. Ezek a szkriptek teszik elérhetővé a felhasználó számára a szolgáltatásokat, egy nagyon egyszerű HTML oldalon keresztül. Az egyes funkciók magukban is elérhetők a saját kiszolgálójukon keresztül, de a PHP kódok egy közös felhasználói interfészt adnak, ami annyit tesz, hogy a PHP kódok tovább hívnak a szolgáltatások interfészeire. Ha kliens oldali alkalmazást készítettem volna, akkor a böngészéshez tartozó kód, egy grafikus felület kódja lett volna, ami pont ugyan így kommunikált volna a többi szolgáltatással. Ugyan így elmondható, hogy bármikor kicserélhető a megjelenítés a böngészés szolgáltatás kicserélésével. A PHP kódok megtalálhatók a függelékben (A.2.2.2. fejezet).

A legbonyolultabb szolgáltatás a megrendeléshez tartozó szolgáltatás, ami az alábbi kódrészletet tartalmazza. Ennek megfelelően a program megkeresi a megrendelendő könyvet a nyilvántartásban, megnézi lehetséges-e a feladott megrendelés, és ha igen bejegyzéseket tesz róla az adatbázisban. Levontja a kívánt mennyiséget a jelenlegi mennyiségből, és felvesz egy új rendelést a rendelések jegyzékébe.

```

...
selectStmt = this.conn.prepareStatement(
    "SELECT count FROM store WHERE book_name LIKE ?"
);
updateStmt = this.conn.prepareStatement(
    "UPDATE store SET count = ? WHERE book_name LIKE ?"
);
insertStmt = this.conn.prepareStatement(
    "INSERT INTO reservation (username, book_name, count, res_date)" +
    "VALUES (?, ?, ?, ?)"
);
this.logger.info("Get book list");
this.conn.setAutoCommit(true);
selectStmt.setString(1, nameOfBook);
if (selectStmt.execute()) {
    rs = selectStmt.getResultSet();
    rs.next();
    int count = rs.getInt("COUNT");
    count = count - number;
    if (count > 0){

```

```

        this.logger.info("Update books in database");
        updateStmt.setInt(1, count);
        updateStmt.setString(2, nameOfBook);
        updateStmt.executeUpdate();
        this.logger.info("Save the executed order");
        insertStmt.setString(1, "test");
        insertStmt.setString(2, nameOfBook);
        insertStmt.setInt(3, number);
        insertStmt.setString(4, new Date().toString());
        insertStmt.executeUpdate();
    } else {
        this.logger.info("Not enough book in the store!");
        status = 500;
    }
}
...

```

A Proxy szolgáltatás az adatbázishoz hasonlóan nem igényel önálló logikát, mivel a HAProxy, amit kiválasztottam mintt proxy vezérlő, nem igényel semmilyen mögöttes logikát, csupán egy konfiguráció beállítást. Ezt a beállítást Consul template segítségével értem el, amit a következő módon konfiguráltam:

```

...
frontend <servicename>
    bind *:<serviceport>
    default_backend <backendname>

backend <backendname>
    balance roundrobin{{range "app.<service>"}}
    service {{.ID}} {{.Address}}:{{.Port}}{{end}}
...

```

Minden szolgáltatáshoz tartozik egy ilyen bejegyzés, ami pontosan megmondja, hogy az adott szolgáltatáshoz melyik port tartozik, és hol található a backend szerverei. A Consul template pedig kitölti a backend szerverhez tartozó részeket az összes élő szolgáltatáspéldánnyal. Ha egy szolgáltatás konténer leáll, akkor a hozzá rendelt bejegyzés automatikusan eltűnik. A konkrét Consul template megtalálható a függelékben (A.2.4. fejezet).

4.3. Kommunikáció, avagy hogy működik a Consul

A szolgáltatások közötti kommunikáció Rest-es interfészeken keresztül történik, HTTP protokollal. Az interfészek minden szolgáltatásra egyediek, és csak egy olyan szolgáltatás van akinek mindegyik interfészt ismernie kell, ez pedig a böngészés, mivel ezen keresztül

érhető el az összes funkció. Az egyes szolgáltatások nem tudnak egymásról, így kellett egy mechanizmus, ami megtalálja az összes szolgáltatást, és elérhetővé teszi egymás számára.

Ez a technológia a Consul lett, amihez ha beregisztrálunk egy végpontot, akkor minden adatát elérhetővé teszi a consul által létrehozott hálózaton belül. Egészen pontosan úgy történik ez, hogy létrehozunk egy Consul szervert, ami képes megosztani az adatokat a hozzá beregisztrált kliensekhez, illetve csatlakozhat más hálózatokhoz is, amikben ő mint kliens jelenik meg. Ezt kihasználva egy olyan logikát csináltam, ami minden végpontot Consul szerverként kezel, és automatikusan a legkisebb IP című másik szervert megtalálja.

```
...
while true; do
    FOUND=false
    for ADDR in $(seq 1 255); do
        echo "${MASK}.${ADDR}  ${IP_ADDR}"
        [[ "${MASK}.${ADDR}" == "${IP_ADDR}" ]] && continue
        ping -c 1  "${MASK}.${ADDR}"
        [ $? -eq 0 ] || continue
        echo "Try consul with ${MASK}.${ADDR}"
        consul agent -server \
            -join "${MASK}.${ADDR}" \
            -datacenter "bookstore" \
            -data-dir "${CONSUL_DIR}" > /var/log/bookstore-consul.log &

        sleep 10
        cat /var/log/bookstore-consul.log
        if ps ax | grep -v grep | grep "consul" > /dev/null; then
            echo "Consul could run!!!"
            FOUND=true
            break
        fi
    done
    echo "${FOUND}"
    if [[ "${FOUND}" == "true" ]]; then
        break
    fi
done
...
```

Ha megtalálta egymást néhány szerver, akkor egyre több és több lehetőség lesz csatlakozni a hálózathoz, és bármely szerver kiesése esetén az össze szolgáltatás helyettesítheti a kiesőt. Ettől a consul hálózat hibatűró lesz, és az adat is meg lesz osztva a végpontok között.

A pontos cím és a szolgáltatás is kinyerhető a Consul adatbázisából, így bármelyik funkció elérhetővé válik.

4.4. Működés és alkalmazás

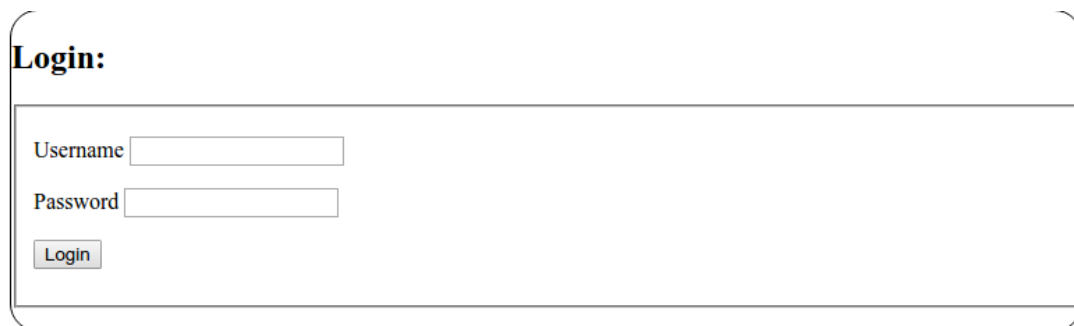
Az alkalmazás indításához elkészítettem egy elég általános indító szkriptet, ami a neki megadott Docker konténerekből indít el egy alkalmazást.

```
...
for service in ${services}
do
    echo "Start ${service} service ..."
    docker run -d --name "${service}" -h "${service}" --net=bookstore bookstore_${servi
done
...
```

Miután elindult az alkalmazás, a webes felület elérhető a proxy, vagy a böngésző Docker konténer IP címén keresztül. Ehhez az információhoz a következő paranccsal juthatunk:

```
docker inspect -f '{{ .NetworkSettings.Networks.bookstore.IPAddress }}' proxy
```

Ha valamilyen névfeloldás áll rendelkezésre, akkor a proxy szolgáltatás IP címét érdemes megadni neki. A felhasználó egy egyszerű bejelentkeztető felületet láthat az alkalmazás indítása után, amin keresztül a felhasználó nevet és jelszavat adhatja meg.



The image shows a web-based login interface. It has a title 'Login:' in bold. Below the title, there are two input fields: 'Username' and 'Password'. Each field has a small rectangular text box next to its label. Below these fields is a button labeled 'Login'.

4.1. ábra. Bejelentkező felület

Ha bejelentkeztünk a böngésző oldalra dob az alkalmazás, és lehetőségünk nyílik rendelesek feladására.

Name of Book:
Number of Books:

Books:

Name	Quantity
Harry Potter and the Goblet of fire	10
Harry Potter and the Philosopher's Stone	10
Harry Potter and the Chamber of Secret	10
Lord of the Rings: Fellowship of the ring	3
Lord of the Rings: The Two Towers	3
Lord of the Rings: The Return of the King	0

4.2. ábra. Böngésző felület

A háttérben minden kérésünkre a szolgáltatások között kommunikáció indul meg, és a különböző funkciók esetén más-más szolgáltatás kiszolgáló kódja indul el.

4.5. Folytonos integráció elkészítése

A folytonos integrációt támogató keretrendszerek közül a Jenkins-t választottam, mivel ez a legelterjedtebb nyílt forrású eszköz, amivel képes vagyok véghez vinni a feladatokat.

4.5.1. Jenkins

A Jenkins egy olyan folytonos integrációt támogató keretrendszer, aminek a Java implementációja lehetővé teszi, hogy bármely, az eszközhöz beregisztrált gépen futtassunk tetszőleges kódot. Ahhoz, hogy ezeket a kódokat futtassuk, egy jól struktúrált végrehajtási rendszert implementál, aminek a következők a részei:

- Jenkins: A Jenkins maga a legnagyobb egység, ami az összes végrehajtandó feladatot tartalmazza, struktúrálja, és konfigurálhatóvá teszi a felhasznált plugin-eket, autentikációt, és mindent ami a feladatokhoz tartozhat.
- View: A feladatok egy jól struktúrált egysége.
- Job: Ez a feladatok implementációja, minden Job tetszőleges mennyiségű végrehajtandó feladatot tartalmaz, képes más Job-ok hívására, és a plugin-ek használatával gyakorlatilag bármilyen feladatot képes elvégezni (build-et futtat, java forrásokat fordít maven-nel, vagy Docker konténereket vezérel, stb.).
- Build: Ez az egység egy Job egyszeri futását jelenti, ehhez tartozik egy azonosító, ami az adott Job-ra nézve megkülönbözteti, a futtatás paraméterei, környezeti változói, és egy olyan szeparált környezet (workspace), amiben a feladatokat végrehajtja.

Ahhoz hogy megcsinálhassam az alkalmazásom fejlesztését támogató keretrendszert, ahhoz Job-kat kellett létrehoznom, amik végrehajtották a szoftverrel kapcsolatos feladatokat.

4.5.2. Pipeline Job

A Pipeline job egy olyan Jenkins-ben elérhető Job fajta, ami képes egységbe szervezni a feladatokat, és levezényelni a közös futásukat. A legtöbb folytonos integrációt támogató rendszerben egy rövidebb, hosszabb munkafolyamatot kell lebonyolítani, aminek a pipeline nevet adta a Jenkins, mivel az egyes Job-ok az előzetes Job-ok build-jeitől függenek, ami annyit jelent, hogy például a telepítési fázis függ a build fázis artifact-jaitól.

A Jenkins 2.0-ban megjelenő Pipeline job-hoz tartozik egy új leíró nyelv is, amit Pipeline szkriptnek neveznek. Lehetőség van fájlban tárolni a konfigurációt, amit Jenkinsfile néven lehet menteni. A Dockerfile-hoz hasonlóan ez is a működést írja le, és Pipeline szkriptet tartalmaz.

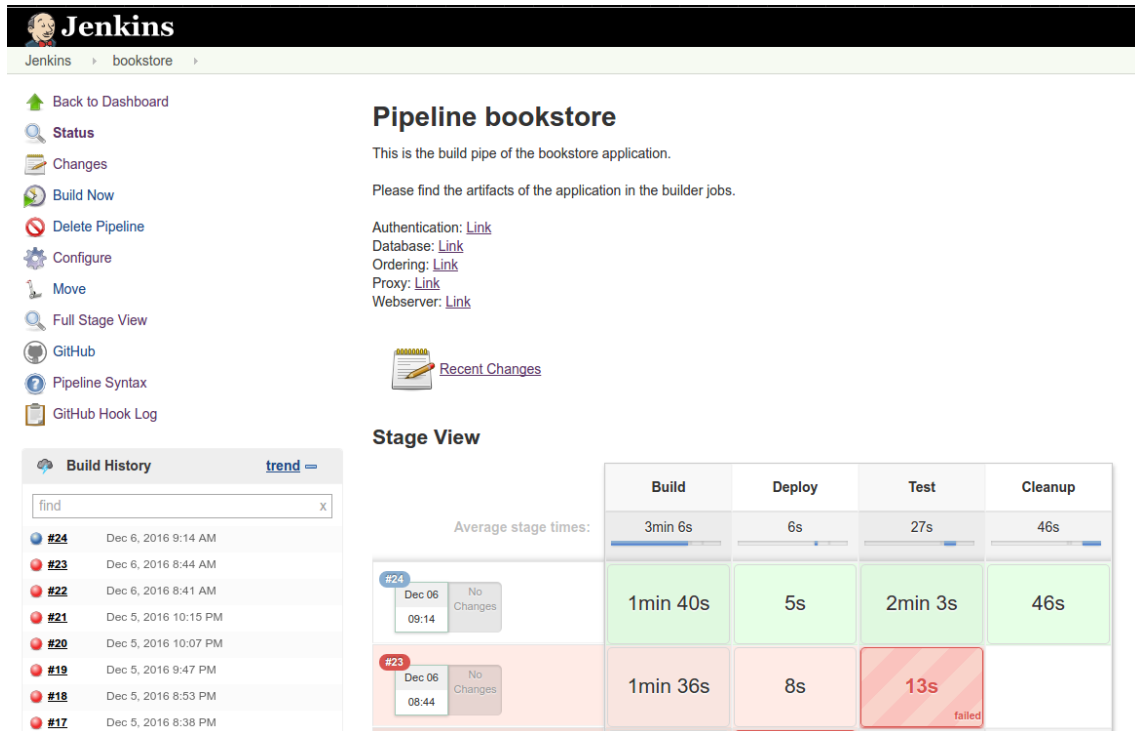
```
...
node {
    echo 'Running '+jobName+' job'
    build job: jobName
}
...
```

A fenti részlet egy olyan Pipeline szkriptet mutat, amiben egy bizonyos Job-ot akarunk futtatni. A `node` kulcsszó jelzi a gépet amin futtatni akarjuk a feladatot, aminek ha nem adunk meg semmit, akkor tetszőleges helyen fut le. A zárójelek között leírt utasítások pedig megmondják, mi történjen azon a végponton. Egy utasítás lehet bármi amit egy Jenkins által alapból támogatott Job esetén beállíthatnánk, de ezek a parancsok inkább arra vannak kiélezve, hogy a feladatok közötti kapcsolatot leírják. Az egyik ilyen parancs az **echo** amivel kiírhatunk a build során üzeneteket a console kimenetre. Másik utasítás, a **build job**, amivel elindíthatunk egy Job-ot. Az alábbi példában egy fázis definiálása látható.

```
stage 'Deploy'
echo 'Deploying services ...'
build job: 'deploy-services'
```

Egy fázis definiálása során egy képzeletbeli egységet alkotunk, ami job-ok futtatását, illetve parancsok futtatását tartalmazza. Új fázist a **stage** kulcsszóval definiálhatunk, és az utána írt utasítások mind a fázis részeként fognak megjelenni.

Az általam implementált Pipeline job, tartalmaz egy 'Build', egy 'Deploy', egy 'Test', és egy 'Cleanup' fázist. Az alábbi ábrán látható a Job-hoz tartozó összefoglaló nézet, ahol láthatók a Job futtatásai:



4.3. ábra. Pipeline Job a mikszolgáltatás támogatásához



















A képen látható, hogy hogyan is választja szép a fázisokat a Pipeline Job, és hogy hogyan lehet kategorizálni a feladatok megívását. Az első 'Build' fázis mögött például 5 darab build Job hívás van, ami azt jelenti, hogy minden szolgáltatást külön fordít le és készíti el a hozzá tartozó Docker image-et.

4.5.3. Jenkins Job-ok a keretrendszerhez

Az előző fejezetben már látható volt, hogy van egy egész folyamatot vezénylő Pipeline Job, amivel az összes folyamatot irányítom. Minden fázishoz tartozik legalább egy másik Job, amiben leírtam, hogy pontosan mit is kell csinálni abban a munkafolyamatban. A következő Job-okat hoztam létre:

- build-auth-service: Authentikációs szolgáltatás elkészítése, ami egy Docker image-et build-el.
- build-database-service: Adatbázis szolgáltatás elkészítése, ami egy Docker image-et build-el.
- build-order-service: Megrendelés szolgáltatás elkészítése, ami Maven build segítségével elkészíti a Java programot, és egy Docker image-et build-el.
- build-proxy-service: Proxy szolgáltatás elkészítése, ami egy Docker image-et build-el.
- build-webserver-service: Böngészés szolgáltatás elkészítése, ami egy Docker image-et build-el.
- cleanup-services: Letörli a futó konténereket, és eltünteti a régi image-eket a Docker-ből.

- `deploy-services`: Alkalmazás indítása, avagy a szolgáltatásokhoz tartozó konténerek indítása, hozzá tartozó hálózat elkészítésével.
- `test-services`: Tesztek futtatása, amely az alkalmazás funkcionalitását teszteli.

S	W	Name ↓
		bookstore
		build-auth-service
		build-database-service
		build-order-service
		build-proxy-service
		build-webserver-service
		cleanup-services
		deploy-services
		test-services

4.4. ábra. Jenkins Job-ok listája

4.5.4. Job Konfigurációk

Vannak bizonyos beállítások, amik minden létrehozott Job-ra egyformák, mivel egy közös verziókezelő eszközből a GitHub-ből vette a forrásfájlokat minden Job. Az ehhez tartozó beállításokat az 4.5.4. ábra mutatja.

☐ CVS
☐ CVS Projectset
☒ Git

Repositories
 Repository URL:
 Credentials:

Branches to build
 Branch Specifier (blank for 'any'):

Repository browser:

Additional Behaviours:

Ez a konfiguráció azt mondja meg, hogy honnan töltsse le a forrásokat a Jenkins, és milyen branch tartalmát akarom felhasználni. Van egy beállítás az autentikáció lebonyolítására is, ami a Jenkins-en belül egy felhasználónév jelszó pár, amit felhasználva a Jenkins tudja használni a GitHub-ot. Ezt a párost globálisan lehet megadni a Jenkins-nek, amit a *Jenkins/Credentials* fülön keresztül érhetünk el.

Másik mindenhol beállított tulajdonság a konkurens futtatás beállítása, ami azt jelenti, hogy több build is futhat egy időben. Feltételezve, hogy a Job-ok külön gépeken futnak, lehetséges, hogy több példány is fusson belőlük.

Amiben minden Job különbözik az a futtatott kód. Minden Job-hoz készítettem egy szkriptet amit meghívhatok a Jenkins-ből. Egy ilyen Jenkins beállítás a 4.5.4. ábrán látható.

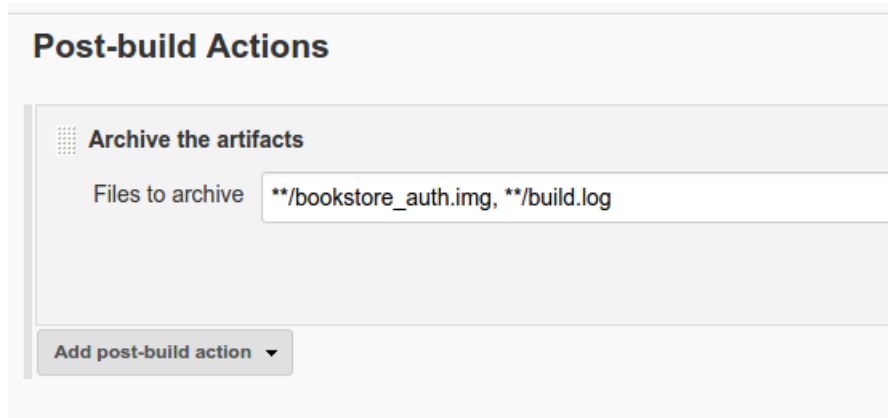
Build

☒ Execute shell

Command:

[See the list of available environment variables](#)

A build Job-ok esetén egy artifact is keletkezik, ami az adott build-hez kapcsolódóan lesz lementve. A beállítást a 4.5.4. ábra mutatja.



Az elementet Docker image-ek letölthetők a Jenkins-ből minden build után, így könnyen reprodukálható bármelyik futás, illetve könnyen kiadható bármelyik verzió.

5. fejezet

Értékelés

5.1. Lehetőségek

5.2. Problémák, Kritika

5.3. Hol használható

Táblázatok jegyzéke

Ábrák jegyzéke

2.1. Scaling Cube	10
3.1. Mikroszolgáltatások terve	26
3.2. Folytonos integráció fázisai	27
4.1. Bejelentkező felület	37
4.2. Böngésző felület	38
4.3. Pipeline Job a mikroszolgáltatás támogatásához	40
4.4. Jenkins Job-ok listája	41

Irodalomjegyzék

- [1] Amazon. Elastic load balancing. <https://aws.amazon.com/elasticloadbalancing/>.
- [2] Ansible. Ansible. <https://www.ansible.com/>.
- [3] Apache. Apache zookeeper. <http://zookeeper.apache.org/>.
- [4] Zohar Arad. Effectively monitor your micro-service architectures. <http://zohararad.github.io/presentations/micro-services-monitoring/>.
- [5] Archivemata. Archivemata 1.1 micro-services. https://wiki.archivemata.org/Archivemata_1.1_Micro-services.
- [6] David Chou. Using events in highly distributed architectures. *The Architecture Journal*, October 2008.
- [7] Cronitor. Monitoring microservices. <https://cronitor.io/help/micro-service-monitoring>.
- [8] Cronitor.io. Cronitor. <https://cronitor.io/>.
- [9] Manfréd Sneps-Sneppé Dimirty Namiot. On mirco-services architecture. <http://cyberleninka.ru/article/n/on-micro-services-architecture>.
- [10] Docker. Docker. <https://www.docker.com/>.
- [11] ElasticBox. Elasticbox. <https://elasticbox.com/how-it-works>.
- [12] ElasticBox. Kubernetes. <https://elasticbox.com/kubernetes>.
- [13] Elasticsearch. Kibana. <https://www.elastic.co/products/kibana>.
- [14] Elasticsearch. Logstash. <https://www.elastic.co/products/logstash>.
- [15] Martin Fowler. Continuous integration. <http://martinfowler.com/articles/continuousIntegration.html>, May 2006.
- [16] Peter Van Garderen. Archivemata: Using micro-services and open-source software to deliver a comprehensive digital curation solution. In *iPres 2010*, pages 145–150, Vienna, Austria, 19–24 2010.

- [17] Nemeth Gergely. Monitoring microservices. <https://www.loggly.com/blog/monitoring-microservices-three-ways-to-overcome-the-biggest-challenges/>.
- [18] HAProxy. The reliable, high performance tcp/http load balancer. <http://www.haproxy.org/>.
- [19] Hashicorp. Consul. <https://www.consul.io/>.
- [20] Jez Humble. Continuous delivery. <https://continuousdelivery.com/>.
- [21] Chef Software Inc. Chef. <https://www.chef.io/chef/>.
- [22] SaltStack inc. Saltstack. <http://saltstack.com/>.
- [23] Jenkins. Elasticbox ci. <https://wiki.jenkins-ci.org/display/JENKINS/ElasticBox+CI>.
- [24] Jenkins. Jenkins. <https://jenkins.io/index.html>.
- [25] Mercedes Garijo Jose Ignacio Fernández-Villamor, Carlos Á. Iglesias. Microservices: Lightweight services descriptors for REST architectural style. http://oa.upm.es/8128/1/INVE_MEM_2010_81293.pdf.
- [26] Chris Richardson (Kong). Pattern: Microservices Architecture. <http://microservices.io/patterns/microservices.html>.
- [27] KVM. Kernel virtual machine. http://www.linux-kvm.org/page/Main_Page.
- [28] Libvirt. libvirt: The virtualization api. <https://libvirt.org/>.
- [29] David Liu. Eureka at glance. <https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance>.
- [30] Fideloper LLC. Load balancing with haproxy. <https://serversforhackers.com/load-balancing-with-haproxy>.
- [31] Microsoft. Pipes and Filters Pattern. <https://msdn.microsoft.com/en-us/library/dn568100.aspx>.
- [32] Microsoft. Publish/Subscribe. <https://msdn.microsoft.com/en-us/library/ff649664.aspx>.
- [33] Mrina Natarajan. 3 golden rules of microservices deployments. <http://devops.com/2015/05/07/3-golden-rules-microservices-deployments/>.
- [34] Nginx. Nginx. <https://www.nginx.com/>.
- [35] Sebastián Peyrott. An introduction to microservices, part 1. <https://auth0.com/blog/2015/09/04/an-introduction-to-microservices-part-1/>.
- [36] Sebastián Peyrott. An introduction to microservices, part 3: The service registry. <https://auth0.com/blog/2015/10/02/an-introduction-to-microservices-part-3-the-service-registry/>.

- [37] Puppet. Puppet 4.4 reference manual. <https://docs.puppet.com/puppet/latest/reference/>, 2016.
- [38] Chris Richardson. The Scale Cube. <http://microservices.io/articles/scalecube.html>.
- [39] Chris Richardson. Service registry pattern. <http://microservices.io/patterns/service-registry.html>.
- [40] Chris Richardson. Building microservices: Inter-process communication in a microservices architecture. <https://www.nginx.com/blog/building-microservices-inter-process-communication/>, July 2015.
- [41] Ruxit. Microservice monitoring. https://ruxit.com/microservices/#microservices_start.
- [42] Sensu. What is sensu? <https://sensuapp.org/docs/latest/overview>.
- [43] Puja Padiya Snehal Mumbaikar. Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3, May 2013.
- [44] tutorialspoint. What is a socket? http://www.tutorialspoint.com/unix_sockets/what_is_socket.htm.
- [45] w3schools. Xml soap. http://www.w3schools.com/xml/xml_soap.asp.
- [46] Daniel Westheide. Why restful communication between microservices can be perfectly fine. <https://www.innoq.com/en/blog/why-restful-communication-between-microservices-can-be-perfectly-fine/>, march 2016.
- [47] Benjamin Wootton. Microservices - not a free lunch! <http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html>, apr 2014.
- [48] Zabbix. What is zabbix. <http://www.zabbix.com/product.php>.

A. függelék

Függelék

A.1. Dockerfile-ok

A.1.1. Authentikáció

Dockerfile.auth.service

```
FROM ubuntu
MAINTAINER Borlay Dániel <borlay.daniel@gmail.com>

ENV CONSUL_DIR /usr/share/consul

# Install Service
COPY auth-service.py /usr/sbin/auth-service.py
RUN apt-get -y update && \
    apt-get -y install \
        bash \
        iputils-ping \
        python-oauth \
        python-mysqldb \
        python \
        python-flask && \
    chmod +x /usr/sbin/auth-service.py

# Install consul
COPY consul consul-template /usr/bin/
RUN chmod +x /usr/bin/consul && \
    chmod +x /usr/bin/consul-template && \
    mkdir -p /etc/consul.d
COPY auth.json /etc/consul.d/auth.json

# Install entry point
COPY init /usr/sbin/init-auth
```

```
RUN chmod +x /usr/sbin/init-auth
```

```
ENTRYPOINT /bin/bash /usr/sbin/init-auth
```

```
EXPOSE 8081 8301 8302 8500 8400
```

A.1.2. Proxy

Dockerfile.proxy.service

```
FROM haproxy
```

```
MAINTAINER Borlay Dániel <borlay.daniel@gmail.com>
```

```
ENV CONSUL_DIR /usr/share/consul
```

```
# Install Service
```

```
COPY proxy.sh /usr/sbin/proxy.sh
```

```
RUN apt-get -y update && \
```

```
    apt-get -y --force-yes install haproxy iputils-ping && \
```

```
    chmod +x /usr/sbin/proxy.sh
```

```
COPY haproxy.cfg /etc/haproxy/haproxy.cfg
```

```
# Install consul
```

```
COPY consul consul-template /usr/bin/
```

```
RUN mkdir -p /etc/consul.d && \
```

```
    chmod +x /usr/bin/consul && \
```

```
    chmod +x /usr/bin/consul-template
```

```
COPY proxy.json /etc/consul.d/proxy.json
```

```
# Install entry point
```

```
COPY init /usr/sbin/init-proxy
```

```
RUN chmod +x /usr/sbin/init-proxy
```

```
ENTRYPOINT /bin/bash /usr/sbin/init-proxy
```

```
EXPOSE 8080 8301 8302 8500 8400
```

A.1.3. Adatbázis

Dockerfile.database.service

```
FROM ubuntu
```

```
MAINTAINER Borlay Dániel <borlay.daniel@gmail.com>
```

```

USER root
ENV CONSUL_DIR /usr/share/consul

# Install Service
COPY database.sh /usr/sbin/database.sh
COPY auth_init.sql bookstore_init.sql /tmp/
RUN apt-get -y update && \
    /bin/bash -c "debconf-set-selections <<< 'mysql-server mysql-server/root_password password' \
    /bin/bash -c "debconf-set-selections <<< 'mysql-server mysql-server/root_password_again password' \
    apt-get -y install mysql-server \
        iputils-ping \
        mysql-client && \
    chmod +x /usr/sbin/database.sh

# Install consul
COPY consul consul-template /usr/bin/
RUN mkdir -p /etc/consul.d && \
    chmod +x /usr/bin/consul && \
    chmod +x /usr/bin/consul-template
COPY database.json /etc/consul.d/database.json

# Install entry point
COPY init /usr/sbin/init-db
RUN chmod +x /usr/sbin/init-db && \
    sed -i 's/bind-address=.*./bind-address = 0.0.0.0/g' /etc/mysql/mysql.conf.d/mysqld.cnf

ENTRYPOINT /bin/bash /usr/sbin/init-db

EXPOSE 3306 8301 8302 8500 8400

```

A.1.4. Vásárlás

Dockerfile.order.service

```

FROM tomcat
MAINTAINER Borlay Dániel <borlay.daniel@gmail.com>

ENV CONSUL_DIR /usr/share/consul

# Install Service
RUN sed -i 's/8080/8888/g' /usr/local/tomcat/conf/server.xml && \
    sed -i 's/<Connector /<Connector address="0.0.0.0" /g' /usr/local/tomcat/conf/server.xml
COPY target/ReserveRESTJerseyExample-0.0.2-SNAPSHOT.war /usr/local/tomcat/webapps/order.war

```

```
# Install consul
COPY consul consul-template /usr/bin/
RUN mkdir -p /etc/consul.d && \
    chmod +x /usr/bin/consul && \
    chmod +x /usr/bin/consul-template
COPY order.json /etc/consul.d/order.json

# Install entry point
COPY init /usr/local/sbin/init-order
RUN chmod +x /usr/local/sbin/init-order

ENTRYPOINT /bin/bash /usr/local/sbin/init-order

EXPOSE 8888 8301 8302 8500 8400
```

A.1.5. Webkiszolgáló (böngészés)

Dockerfile.web.service

```
FROM ubuntu
MAINTAINER Borlay Dániel <borlay.daniel@gmail.com>

ENV CONSUL_DIR /usr/share/consul

# Install Service
COPY webserver.sh /usr/sbin/webserver.sh
RUN apt-get -y update && \
    apt-get -y install apache2 php \
        libapache2-mod-php \
        php-mysql \
        curl \
        iputils-ping \
        php-curl && \
    chmod +x /usr/sbin/webserver.sh
COPY index.html main.css login.php store.php order.php /var/www/html/

# Install consul
COPY consul consul-template /usr/bin/
RUN mkdir -p /etc/consul.d && \
    chmod +x /usr/bin/consul && \
    chmod +x /usr/bin/consul-template
COPY web.json /etc/consul.d/web.json
```

```
# Install entry point
COPY init /usr/sbin/init-web
RUN chmod +x /usr/sbin/init-web

ENTRYPOINT /bin/bash /usr/sbin/init-web

EXPOSE 80 443 8301 8302 8500 8400
```

A.2. Szkriptek

A.2.1. Futtatáshoz

A.2.1.1. Build

```
build_{service}.sh
```

```
#!/bin/bash
```

```
<SERVICE>_SERVICE_HOME=services/<SERVICE>
<SERVICE>_SERVICE_DOCKERFILE=Dockerfiles/Dockerfile.<SERVICE>.service
<SERVICE>_SCRIPT_DIR=scripts/<SERVICE>
<SERVICE>_CONF_DIR=conf/<SERVICE>
<SERVICE>_IMAGE_NAME=bookstore_<SERVICE>
```

```
pushd ..
```

```
if [[ ! -e consul ]]; then
```

```
    echo "Get Consul script from Internet"
```

```
    wget https://releases.hashicorp.com/consul/0.7.0/consul_0.7.0_linux_386.zip && unzip con
```

```
fi
```

```
if [[ ! -e consul-template ]]; then
```

```
    echo "Get consul-template script from Internet"
```

```
    wget https://releases.hashicorp.com/consul-template/0.16.0/consul-template_0.16.0_linux_
```

```
fi
```

```
echo "Create <SERVICE> service for bookstore ..."
```

```
echo " - Create directory for Docker data"
```

```
mkdir -p ${<SERVICE>_SERVICE_HOME}
```

```
echo " - Move Dockerfile to data directory"
```

```
cp ${<SERVICE>_SERVICE_DOCKERFILE} ${<SERVICE>_SERVICE_HOME}/Dockerfile
```

```
echo " - Move script files to data directory"
```

```
cp -R ${<SERVICE>_SCRIPT_DIR}/* ${<SERVICE>_SERVICE_HOME}/
```

```
echo " - Move config files to data directory"
```

```

cp -R ${<SERVICE>_CONF_DIR}/* ${<SERVICE>_SERVICE_HOME}/
echo " - Move consul to data directory"
cp consul ${<SERVICE>_SERVICE_HOME}/
cp consul-template ${<SERVICE>_SERVICE_HOME}/
echo " - Building Docker image"
docker build -t ${<SERVICE>_IMAGE_NAME} ${<SERVICE>_SERVICE_HOME} &> ${<SERVICE>_SERVICE_LOG}
echo " - Save image"
docker save --output ${<SERVICE>_SERVICE_HOME}/${<SERVICE>_IMAGE_NAME}.img ${DATABASE_IMAGE_NAME}

echo "<SERVICE> service has been created!"
popd

```

A.2.1.2. Futtatás

```

run_containers.sh

#!/bin/bash

services="database webserver order auth proxy"

docker network create bookstore

for service in ${services}
do
    echo "Start ${service} service ..."
    docker run -d --name "${service}" -h "${service}" --net=bookstore bookstore_${service}
done

```

A.2.1.3. Tisztogatás

```

clean_docker.sh

#!/bin/bash

services="database webserver proxy order auth"

docker stop $(docker ps -a | awk '/bookstore/ {print $1}')
docker rm $(docker ps -a | awk '/bookstore/ {print $1}')

for service in ${services}
do
    echo "Delete ${service} image"
    docker rmi bookstore_${service}
done

```

```

if [ -d services ]; then
    rm -rf services
fi
docker network rm bookstore

if [ -e consul ]; then
    rm -rf consul*
fi

```

A.2.2. Szolgáltatásokhoz

A.2.2.1. Init szkript

```

#!/bin/bash

IP_ADDR=$(hostname -I)
MASK=${IP_ADDR%.*}

while true; do
    FOUND=false
    for ADDR in $(seq 1 255); do
        echo "${MASK}.${ADDR} ${IP_ADDR}"
        [[ "${MASK}.${ADDR}" == "${IP_ADDR}" ]] && continue
        ping -c 1 "${MASK}.${ADDR}"
        [ $? -eq 0 ] || continue
        echo "Try consul with ${MASK}.${ADDR}"
        consul agent -server \
            -join "${MASK}.${ADDR}" \
            -datacenter "bookstore" \
            -data-dir "${CONSUL_DIR}" > /var/log/bookstore-consul.log &

        sleep 10
        cat /var/log/bookstore-consul.log
        if ps ax | grep -v grep | grep "consul" > /dev/null; then
            echo "Consul could run!!!"
            FOUND=true
            break
        fi
    done
    echo "${FOUND}"
    if [[ "${FOUND}" == "true" ]]; then
        break
    fi
fi

```



```
done
<service>.sh
```

A.2.2.2. Böngészés kódjai

Login oldal:

```
<?php

if(!isset( $_POST['username'], $_POST['password']))
{
    echo 'Please enter a valid username and password';
}
else
{
    $username = filter_var($_POST['username'], FILTER_SANITIZE_STRING);
    $password = filter_var($_POST['password'], FILTER_SANITIZE_STRING);

    $ch = curl_init();
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_URL,
        "http://auth:8081/auth/{$_username}/{$_password}"
    );
    $output = curl_exec($ch);
    $info = curl_getinfo($ch);
    if ($output === false || $info['http_code'] != 200) {
        header("Location: /login.php");
        die();
    }
    else {
        header("Location: /store.php");
        die();
    }
    curl_close($ch);
}
?>
```

Könyveket megjelenítő oldal:

```
<html>
<head>
<title>Bookstore Microservice</title>
<link rel="stylesheet" type="text/css" href="main.css">
</head>
```

```

<body>

<div id="storeBox">
<h2>Books:</h2>

<table>
  <tbody>
    <tr><th>Name</th><th>Quantity</th></tr>

    <?php
      $servername = "database";
      $username = "store";
      $password = "store";
      $dbname = "bookstore";

      // Create connection
      $conn = new mysqli($servername, $username, $password, $dbname);
      // Check connection
      if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error);
      }

      $sql = "SELECT * FROM store";
      $result = $conn->query($sql);

      if ($result->num_rows > 0) {
        // output data of each row
        while($row = $result->fetch_assoc()) {
          echo "<tr><td>" . $row["book_name"] . \
            "</td><td> " . $row["count"] . "</td></tr>";
        }
      } else {
        echo "0 results";
      }
      $conn->close();
    ?>

  </tbody>
</table>
</div>
<div id="orderBox">
<form action="order.php" method="post">

```

```

    <span>Name of Book: </span>\
        <input type="text" name="nameOfBook" /><br/>
    <span>Number of Books: </span>\
        <input type="text" name="numberOfBooks"/><br/>
    <input type="submit" name="send" value="Send"/>
</form>
</div>

</body>
</html>

```

A.2.2.3. Adatbázis inicializálás

Authentikáció:

```

# Add permission to databases
GRANT ALL PRIVILEGES ON authenticate.* TO 'root'@'%';
GRANT ALL PRIVILEGES ON authenticate.* TO 'root'@'localhost';
# Create Tables
CREATE TABLE user_auth
(
    user_id int NOT NULL AUTO_INCREMENT,
    username varchar(255) NOT NULL,
    password varchar(255) NOT NULL,
    credential varchar(255),
    PRIMARY KEY (user_id)
);
# Fill Tables
INSERT INTO user_auth (username, password) VALUES ("test", "testpassword");

```

Bookstore raktár:

```

# Add permission to databases
GRANT ALL PRIVILEGES ON bookstore.* TO 'root'@'%';
GRANT ALL PRIVILEGES ON bookstore.* TO 'root'@'localhost';
# Create Tables
CREATE TABLE store
(
    store_id int NOT NULL AUTO_INCREMENT,
    book_name varchar(255) NOT NULL,
    count int NOT NULL,
    PRIMARY KEY (store_id)
);
CREATE TABLE reservation

```

```
(
    reservation_id int NOT NULL AUTO_INCREMENT,
    username varchar(255) NOT NULL,
    book_name varchar(255) NOT NULL,
    count int NOT NULL,
    res_date varchar(255),
    PRIMARY KEY (reservation_id)
);
# Fill Tables
INSERT INTO store (book_name, count)
VALUES ("Harry Potter and the Goblet of fire", 10);
INSERT INTO store (book_name, count)
VALUES ("Harry Potter and the Philosopher's Stone", 10);
INSERT INTO store (book_name, count)
VALUES ("Harry Potter and the Chamber of Secret", 10);
INSERT INTO store (book_name, count)
VALUES ("Lord of the Rings: Fellowship of the ring", 3);
INSERT INTO store (book_name, count)
VALUES ("Lord of the Rings: The Two Towers", 3);
INSERT INTO store (book_name, count)
VALUES ("Lord of the Rings: The Return of the King", 0);
```

A.2.3. Pipeline job szkript

Pipeline job full szkript:

```
„{Pipeline script} buildNames = [ 'build-auth-service', 'build-database-service', 'build-
order-service', 'build-proxy-service', 'build-webserver-service']
def buildJobs = [:]
for (int i=0; i<buildNames.size(); ++i) { def buildName = buildNames[i] build-
Jobs[buildNames[i]] = { node { echo 'Running'+buildName+'build' build job: buildName }
} }
stage 'Build' echo 'Building services ...' parallel buildJobs stage Deploy' echo Deploying
services ...' build job: deploy-services' stage Test' echo Testing services ...' build job:
test-services' stage Cleanup' echo Cleaning up services ...' build job: cleanup-services' „
```

A.2.4. Proxy

Proxy config template:

```
global
    log 127.0.0.1 local1 notice
    chroot /var/lib/haproxy
    stats socket /run/haproxy/admin.sock mode 660 level admin
    stats timeout 30s
```

```

user haproxy
group haproxy
daemon

defaults
    log      global
    mode     http
    option    httplog
    option    dontlognull
    timeout  connect 5000
    timeout  client  50000
    timeout  server  50000
    errorfile 500 /etc/haproxy/errors/500.http

frontend web
    bind *:80
    mode http
    default_backend nodes

backend nodes
    mode http
    balance roundrobin{{range "app.web"}}
    service {{.ID}} {{.Address}}:{{.Port}}{{end}}

frontend database
    bind *:3306
    default_backend dbnodes

backend dbnodes
    balance roundrobin{{range "app.database"}}
    service {{.ID}} {{.Address}}:{{.Port}}{{end}}

frontend order
    bind *:8888
    default_backend onodes

backend onodes
    balance roundrobin{{range "app.order"}}
    service {{.ID}} {{.Address}}:{{.Port}}{{end}}

frontend auth
    bind *:8081

```

```
default_backend anodes

backend anodes
  balance roundrobin{{range "app.auth"}}
  service {{.ID}} {{.Address}}:{{.Port}}{{end}}
```