Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Mikroszolgáltatásokra épülő architektúra fejlesztésének és tesztelésének támogatása

Diplomaterv

Készítette
Borlay Dániel

Konzulens
Szatmári Zoltán

2016. november 18.

Tartalomjegyzék

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. november 18.

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 hatákonyan tervezhetjük meg a rendszerünket, és könnyen elkészíthetjük az alkalmazásunkat. Egy ilyen architektúrális megközelítés a mikro szolgáltatásokon alapuló architektúra, amivel apró részletekre bontva a feladatot, könnyen kezünkben tarthatjuk az elosztott alkalmazásunkat.

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

A diplomaterv célkitűzése, hogy egy olyan mikro szolgáltatásokra épülő alkalmazást készítsek, 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

1. fejezet

Bevezetés

Fontos, hogy az alkalmazásaink megbízhatóan, karbantarthatóan, és nagy rendelkezésre állással legyenek elérhetőek. Napjaink egyik feltörekvő architektúra építési elve a mikroszolgáltatások archtektúrája. 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 kép alkotásában, 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[25] [9] [24] egy olyan architektúrális modellezési mód, 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 kapot 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.

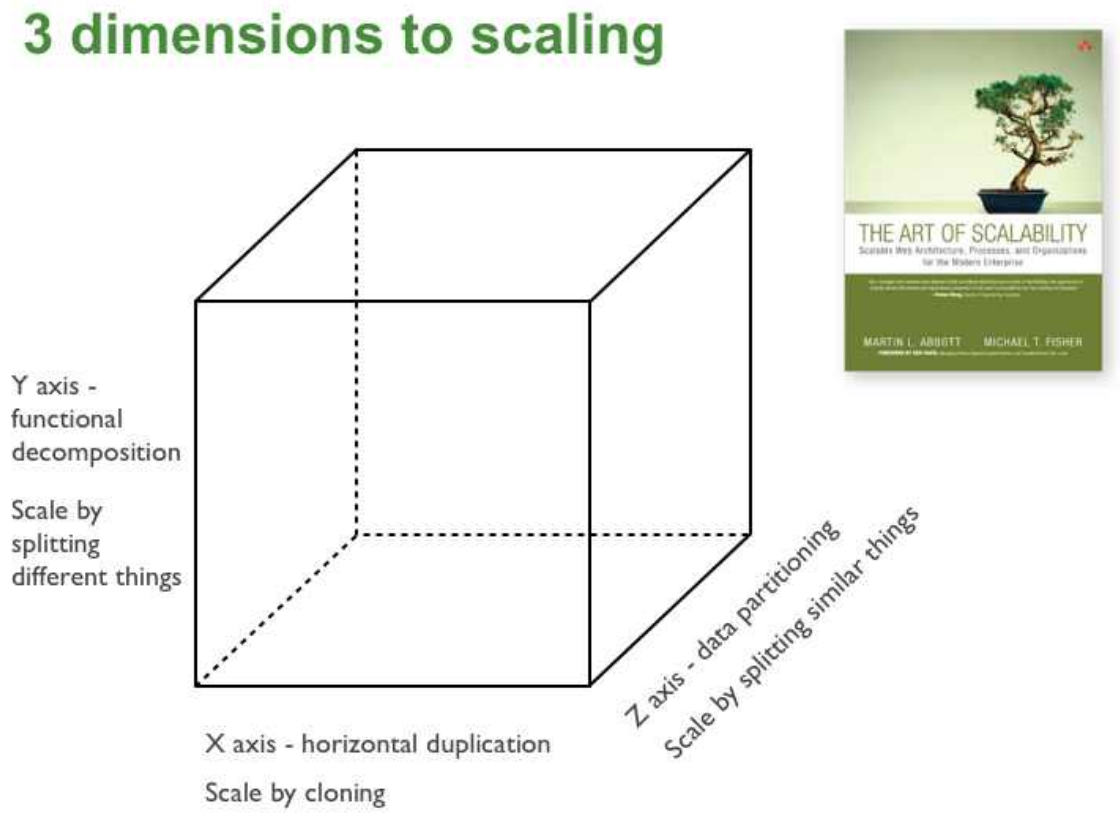
2.1.1. Szolgáltatás elválasztás tervezése

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 lépéssekkel 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[37], ami azt mutatja, hogy az architektúrális terveket milyen szempontok mentén lehet felosztani.



2.1. ábra. Scaling Cube

Ahogy a 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 adatbézist 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 terheléselosztó 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[30] 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[31], 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[15] - Egy Fájlkezelő rendszer, amiben mikroszolgáltatásoknak megfelelően alakították ki a plugin-ként használható funkciókat.

2.2. Mikroszolgáltatások előnyei és hátrányai

Ahogy minden architektúrális mintának, a mikroszolgáltatásoknak is vannak előnyei[25], amik indokoltá teszik a minta használatát, és vannak hátrányai[46], 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 önnálóan 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 progam 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 felskálázásval.

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 kommunikálnak 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 interfésziek 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 jelénthet 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 ellentetjei, 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 robosztus, 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, kicserélhetők 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.

2.2.4. Archivematica

Az Archivematica[15] egy nyílt forráskódú elektronikus tartalom kezelő, ami tud kezelni különböző fájlokat, multimédiás adatokat, illetve akármilyen szöveges tartalmat. Ez az alakmazás alapvetően monolitikus architektúrára épül, azonban elkezdték átalakítani a struktúráját mikroszolgáltatásokat használó architektúrára. Ezt úgy kivitelezték, hogy a különböző plusz funkciókat az eredeti alkalmazás plugin szerűen mikroszolgáltatásokból nyeri ki, és ennek megfelelően a tovább fejlesztés is megalapozott[5].

2.3. Technológiai áttekintés

Az integrációhoz olyan technológiákat[34] 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[32], 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 tartsuk nyilván? (service registy)
- Hogyan figyeljük meg az alkalmazást működés közben (monitorozás, loggolás)

2.3.1. Telepítési technológiák

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**[23]: 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 karban tarthatjuk 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. [22]

- **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 emészthet 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.

A Consult 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 hoszt nevével.

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 hajtson végre ő maga is változtatást.

- **Puppet**[36]: 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 szolgáltatat a komplex rendszerbe integráláshoz.

- **Chef**[20]: 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 repectkönynek, (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**[21]: 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 képeken 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. 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 terhelés elosztó az, ami ezt a célt szolgálja. Ennek a szolgáltatásnak az lenne a lényege, hogy segítse az Amazon Cloud-ban futó virtuális gépek hibatűrését, illtve 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.

A skálázás egyik legegyszerűbb megvalósítása, hogy egy proxy szervert 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 többit. Nem tökéletes megoldát kapunk, azonban ez a legtöbb felhasználási esetben megfelelőnek bizonyul.

2.3.5. 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] [29]: Egy magas rendelkezésre állást biztosító, és megbízhatóságot növelő terheléselosztó eszköz. Konfigurációs fájlokon keresztül megszervezhetjük, hogy mely gépet hogyan érjünk 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 kizárólak a HTTP TCP kéréseket tudja elosztani, de egyszerű, könnyen telepíthető, és könnyen kezelhető (ha nem dinamikusan változnak a fürtben lévő gépek, mert ha igen akkor szükséges egy mellékes frissítő logika is).
- **ngnix**[33]: Az Nginx egy nyílt forráskódú web kiszolgáló és reverse proxy szerver, amivel nagy méretű rendszereket kezelhetünk, és segít az alkalmazás biztonságának megörzésében. 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 kézileg implementált terheléselosztó eszköz lenne, amihez viszont hálózati megfigyelés, és routing szökséges, így nem javalott ilyen eszköz készítése.

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**[27]: 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 eszköt. Ezzel az eszközzel a környezet kialakítását szabhatjuk meg, tehát a hardveres erőforrások megosztásában nyújt nagy segítséget.

- **kvm**[26]: 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 mást. A KVM egy nyílt forráskódú projekt és létrehozhatunk vele Linux és Windows gépeket is egyaránt.

- **Akármilyen cloud**: 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 definiciója 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 terheléselosztást, 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[38] [35] kell használnunk.

- **Eureka**[28]: 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 álltal 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] [16].

- **Zabbix**[47]: 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 kereshetünk benne. Kifejezetten adatfeldolgozásra szolgál mind a két eszköz, és közvetlenül együttműködnek az Elasticsearch alkalmazással.

- **Sensu**[41]: A Sensu egy egyszerű monitorozó eszköz, amivel megfigyelhetjük a szervereinket. 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 lekezeli a változásokkal járó riasztásokat. Ezek a tulajdonságai teszik a számítási felhőkben könnyen és hatékonyan felhasználhatóvá.

- **Cronitor**[8] [7]: Ez a monitorozó eszköz mikroszolgáltatások és cron job-ok megfigyelésére lett kifejlesztve, HTTP-n keresztül kommunikál, és a szologáltatások állapotát figyeli. Nem túl széleskörű eszköz, azonban ha csak a szolgáltatások állapota érdekel hasznos lehet, és segíthet a szolgáltatás jegyzék képzésében is.

- **Ruxit**[**?**] [40]: 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 adatbázisunk, amit minden szolgáltatás elér, és ebben az adatbázisban a szolgáltatásokba ültetett egyszerű logika küldhet adatokat, amit valamilyen egyszerű módszerrel megjelenítve, 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[42].

2.4.1. Technológiák

A kommunikáció megtervezéséhez egy jó leírást olvashatunk az Nginx egyik cikkében[39]. 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[24] 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[45], 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[44], 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[43] 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 kommunkácót 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.

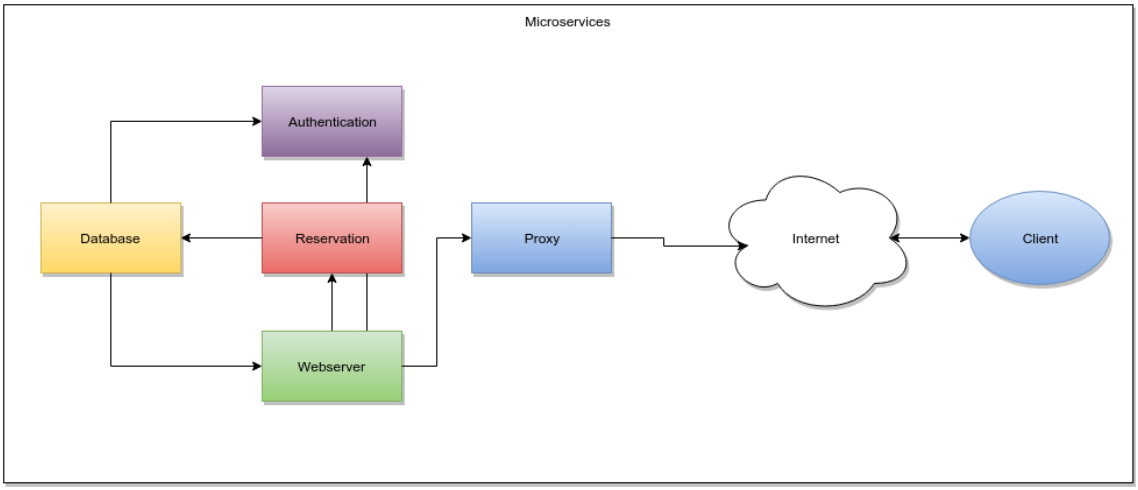
2.4.2. Interfészek

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

Minta alkalmazás



3.1. ábra. Microservices

3.1. Alkalmazás leírás:

Az alkalmazás, amin keresztül a mikroszolgáltatások működését bemutatom, egy könyvesbolt webárúháza lesz, ami rendelkezik egy webes felülettel. A felhasználó be tud jelentkezni a felületre, és tud könyveket vásárolni magának a webes nyilvántartásból.

3.2. Szolgáltatások:

A szolgáltatások meghatározásánál elsőnek azt vettem alapul, hogy milyen feladatokat kell teljesítenie a rendszernek, majd az erőforrásokat vettem alapul.

A könyvesbolthoz tartozóan a következő tevékenységeket határoztam meg:

- **Bejelentkezés**: Felhasználó felületen történő authentikálása.
- **Böngészés**: Felhasználó láthatja mi van a raktáron.
- **Vásárlás**: Felhasználó valamit a saját nevére ír, megvásárol.

A könyveket egy adatbázisban tárolom a megrendelésekkel együtt, és egy proxy-t is készítettem, hogy könnyen skálázható legyen bármely funkció.

- **Adabtbázis**: Tartalmazza a tárolt könyveket, autentikációs adatokat, és vezeti a vásárlások naplóját.
- **Proxy**: Minden szolgáltatást elérhetővé tesz egy közös interfészen keresztül, és skálázhatóvá teszi a szolgáltatásokat.

Ezekből a feladatokból a következő szolgáltatásokat lehet elkészíteni:

- **Felület kiszolgálása**: Egy web kiszolgáló alkalmazása, amin keresztül elvégezhetők a különböző műveletek, mint a bejeletkezés, vagy vásárlás. Ez a felület magába foglalja a böngészést lehetővé tevő szolgáltatást is, mivel szorosan ehhez a funkcióhoz tartozik. Egy másik megvalósítás lehetne, hogy a különböző vetületekhez tartozó információt egy-egy szolgáltatás adja vissza, azonban ez túl nagy komplexitást eredményezne, így az elöbbi megoldásnál maradtam.
- **Authentikációs szolgáltatás**: A bejelentkezni szándékozó felhasználó adatait ellenőrzi, és hibás bejelentkezés esetén hibát dob. Az adatbázissal kommunikál, és ellenőrzi, hogy a felhasználónak van-e bejegyzése az adatbázisban.
- **Vásárlási szolgáltatás**: A böngészés közben kiválasztott könyveket lefoglalja a raktári készletből. Csökkenti az adatbázisban a készlet tartalmát, és létrehoz egy új bejegyzést a vásárlások adattáblájában, ami tartalmazza a vásálás adatait. Ha túl sokat akar venni a felhasználó, vagy nem létező könyvből akar vásárolni, akkor hibajelzést küldök.
- **Adatbázis szolgáltatás**: Ez a szolgáltatás tartalmazza a raktár tartalmát, a vásárlási naplót, és a bejelentkezési adatokat. Mivel a legtöbb adatbázis kezelő távolról elérhető, és fürtözési lehetőségeket is nyújt, ezért ehhez a szolgáltatáshoz nem szükséges mögöttes logika. Az interfésze az adatbázist kezelő eszköz interfésze (pl.: MySQL szerver, és SQL nyelvű interfész.).
- **Terhelés elosztó szolgáltatás**: Ez a szolgálatatás a skálázhatóságot segíti, és egy egységes interfész kialakításában segít. Tartalmaz egy proxy kiszolgálót, és egy konfigurációs fájlt, amit az infrastruktúra változásával bővítünk (pl.: HAProxy, és a hozzá tartozó konfigurációs fájl). Nehéz kérdés lehet egy új szolgáltatás beiktatása, mivel a proxy szerver interfésze változna meg tőle, így ennek a szolgáltatásnak a többivel együtt kell változnia.

3.3. Értékelés

Az alkalmazás kellően egyszerű, hogy rövid idő alatt implementálni lehessen, és alkalmas rá, hogy a fejlesztés során használt folytonos integrációra épülő feladatokat bemutassam rajta.

A tervben azért szerepelnek külön az adatbázis és a proxy szoláltatások, mivel túl bonyolulttá, és feleslegesen erőforrás igényessé tenné az alkalmazást, ha minden szolgáltatás

saját adatbázissal rendelkezne és a saját skálázhatóságát is menedzselné. Van néhány bottleneck szolgáltatás, mint a web kiszolgáló, és az adatbázis szerver, viszont egy olyan változat, melyben ez nem áll fenn, nagyságrendekkel több idő lett volna elkészíteni, így a diplomaterv során ezzel nem foglalkoztam.

4. fejezet

Implementáció és kapcsolódó nehézségek

4.1. Minta alkalmazás implementációja

A minta alkalmazás megvalósításához *Docker* konténereket használtam, hogy a saját laptopomon, vagy egy dedikált gépen is meg tudjam oldani az elosztott alkalmazást, mivel így kevés erőforrással is meg tudtam csinálni amit akartam. A kommunikációra kiválasztott módszer a *REST*-es kommunikáció lett, mivel egy széles körben használt megoldás, és rengeteg nyelv támogatja. A szolgáltatások jegyzéséhez *Consult* használtam, ami az utóbbi években sokat fejlődött, és elég átfogó funkcionalitást nyújt számomra.

4.1.1. Szolgáltatások megvalósítása Docker konténerekben

A mikroszolgáltatások egyik legnagyobb előnye, hogy különböző platformokat és programozási nyelveket használhatunk az architektúrában különösebb probléma nélkül. Ezt a Docker-el úgy oldottam meg, hogy Centos és Ubuntu disztribúciójú környezeteket, és PHP, Python, Java, illetve Bash szkripteket használtam.

A szolgáltatásokhoz tartozó Docker konténerek:

- **Adatbázis**: Az alapja egy *'mysql'* nevezetű konténer, ami tartalmaz egy lightweight Ubuntu-t és benne telepítve egy mysql szervert. Ezt a konténert egy inicializáló szkripttel egészítettem ki, ami elkészítette az alap adatbázist.
- **Terheléselosztó**: A terheléselosztást *HAProxy*-val oldottam meg, amit egy Ubuntu konténerre alapoztam. Létezik egy olyan Docker konténer, ami kifejezetten HAProxy mikroszolgáltatásnak van nevezve, azonban ez a konténer nehezen használható, és a szolgáltatás újraindítása is el lett rontva benne, így egyszerűbbnek láttam egy saját megvalósítást használni.
- **Webkiszolgáló**: A weboldal kiszolgálását egy *'httpd'* nevű lightweight konténer szolgálja ki amiben egy apache webkiszlgáló van. Ezt kiegészítettem néhány *PHP* szkripttel, ami kiszolgálja a kéréseket. Ezt a konténert terveztem úgy, hogy a kéréseket ő szolgálja ki és ezen a szolgáltatáson keresztül érhető el a többi funkcionalitása.

- **Authentikáció**: Egyszerű Ubuntu konténer, ami fel van szerelve *Python*-nal, és a *MySQLdb* Python könyvtárral. Ezen felül tartalmaz egy REST-es kiszolgálót, amin keresztül elérhető a szolgáltatás. Az authentikáció egyszerű adatbázis alapú azonosítás, amit kódolatlanul tárolunk a MySQL adatbázisunkban. Éles változatban természetesen ez nem elegendő.
- **Vásárlás**: Centos konténer alapú környezet, amiben *Java* lett telepítve, és egy webes REST API-n keresztül érhetjük el a szolgáltatását. A konténer webes szolgáltatását *Ngnix* segítségével tettem elérhetővé. Sajnos a félév során változott a Centos alap image repository-ja, és használhatatlanná tette a szolgáltatást, mivel az Ngnix szolgáltatást többé nem tudtam elindítani. Erről a problémáról bövebben a Nehézségek, megoldás értékelése' fejezetben írok.

4.2. Szolgáltatás felismerés megoldásai

A szolgáltatások magukban nem sokat érnek, viszont nem tudnak együtt működni ha nem ismerik egymás elérési útvonalát. Ahhoz, hogy a megfelelő helyre érkezzenek be az üzenetek, valamilyen névfeloldásra van szükség, amely képes irányítani a kéréseket a komplex architektúrában. Egyik lehetőség, hogy a Docker álltal szolgáltatott személyre szabott válózatot használjuk, és jól elnevezett konténereket hozunk létre. Egy másik lehetőség a Consul használata, ami nyilvántartja a beregisztrált végpontokat, és elérhetővé teszi egymás számára. A névfeloldáshoz Consul esetében kis trükkre van szükség, de nem olyan nagy feladat. Ugyan a diplomaterv keretei között csak ezt a két módszert vizsgáltam meg, azonban lehetséges saját névfeloldó szervert karbantartani, vagy egy jól felügyelt környezetben konténereket indítani, mint az Amazon AWS Cloud-ban, ami támogatja ezeket a funkciókat.

4.2.1. Kapcsolatok építése Docker-el:

Ahogy korábban már említettem lehetőség van a Docker legújabb verzióiban megadni, hogy ez egyes konténerek milyen néven és milyen hálózaton keresztül érhető el a többi konténer. A név beállításához a docker run parancs `--hostname` paraméterét használhatjuk, míg a hálózat definiálásához előbb létre kell hozni egy új Docker hálózatot

```
docker network create bookstore
```

amire a konténerek tudnak csatlakozni a `--net` kulcsszóval. Ennek segítségével elérhető, hogy nagyon egyszerűen és egy eszköz felhasználásával képesek legyenek látni egymást a szolgáltatások, viszont egy nagy hátulütője van a megoldásnak, mégpedig az, hogy egy gépen kell futnia az összes alkalmazásnak. Másik hátránya ennek a megoldásnak, hogy a konténerek közül csak egynek lehet olyan nevet adni, ami a szolgáltatásra utal (egyszerre csak egy konténernek lehet mondjuk webserver neve), és éppen ezért a skálázáshoz egy saját logikát kell építeni. Mivel ez egy mikroszolgáltatásokra épülő architektúránál közel sem ideális, így ez csupán fejlesztési, és reprezentatív jelleggel használható.

4.2.2. Kapcsolatok építése Consul-al:

A Consul alkalmazást korábbi félév folyamán használtam már, teljesítmény mérések futtatására Docker konténerben, így megpróbáltam átültetni a logikát a jelenlegi mikroszolgáltatásokat biztosító architektúrába. A gondot az okozta, hogy a Consul alkalmazásnak szükséges egy fix pont, így alkottam egy logikát, melyben minden szolgáltatás önálló szerverként funkcionál, és az összes szolgáltatás próbál csatlakozni az alháózatban lévő végpontokra. Ennek az az eredménye, hogy minden csatlakozó szerver egy új lehetőséget biztosít, és lassan de biztosan kiépül a Consul hálózat.

Ez a megoldás magában nem elegendő, hogy felismerjék egymást a szolgáltatások, így fel kellett használnom a Consul Template nevezetű technikát, amiben a proxy beállításokban az egyes szolgáltatásokhoz tartozó elemeket egy bejegyzés alá rendelem. Ennek megfelelően a proxy-n keresztül minden szolgáltatás minden más szolgáltatást elér, ás a skálázás esetán a proxy megoldja az új elem lekezelését. Mivel ez a megoldás sokkal robosztusabb a Docker alapúnál, ezért ezt használtam a végső programban.

4.3. Nehézségek, megoldás értékelése

Sajnálatos módon egy ilyen összetett, és elágazó rendszer esetén gyakran ütközünk problémákba, és ezeket a gondokat leggyakrabban nem is a saját fejlesztésünk, hanem a használt eszküzük okozzák. Erre a célra hasznos lehet fenntartani a folytonos integrációt támogató eszközben egy eszköz konzisztenciát, és visszafelé kompatibilitást tesztelő részt fejleszteni, illetve lassan, és megfontoltan váltani az új eszközverzióknál.

4.3.1. Nehézségek a Docker használatával

A diplomaterv elkészítése során számtalanszor futottam olyan hibába, amit a Docker konténerek használata okozott. Volt olyan, hogy az alapul szolgáló image készítésekor nem volt megfelelő a repository, és eltűntek csomagok amikre szükségem lett volna, hol a futó konténerek álltak le minden ok nélkül. A Docker fejlesztése közben sajnálatos módon nem gondoltak a visszafele kompatibilitásra, és minden új verzióban van valamilyen meglepetés, ami nem úgy működik, mint a korábbiban. Találtam egy cikket[17], ami leírja a legtöbb problémát, amivel Docker konténerek használata közben találkozhatunk, viszont nem igazán van alternatív lehetőség.

Az egyik legnagyobb probléma amivel találkoztam, az a Centos alapú konténerek esetében jelentkezett. Ez a probléma az volt, hogy a Centos alap image megváltozott, és onnantól kezdve nem lehetett használni benne az Nginx szolgáltatást, vagy akármyilen szolgáltatást. A Docker konténerekben használt programok megváltoztak, és a build során történt frissítés lehetetlenné tette a szolgáltatás működését. Megoldásként átálltam egy Ubuntu alapú image-re, mivel abban nem jelentkezett ilyen jellegű hiba.

4.3.2. Nehézségek a Consul használatával

A Consul használatához hozzászoktam már a korábbi félévek során, de az újítások miatt kicsit át kellett gondolnom a használatot. Eleinte megpróbáltam használni az önálló laboratórium során használt szkriptemet, de valamilyen szintaktikai hibára panaszkodott, és nem tudtam használni azt a változatot. Az én megérzésem az, hogy az események küldésében történt valami, mivel látszólag a Docker-el ellentétben a Consul visszafelé kompatibilis, és a szerver építés nem változott. Sajnálatos módon elég sok időt emésztett fel egy új változat elkészítése, ami végül sokkal hibatűrőbb lett, mint elődje, és nem csak egy fix ponton keresztül lehet elérni a többi Consul klienst. Egy másik változtatás lett, hogy minden végpont szerverként funkcionál, így ha kiesik valamelyik szerver, a Consul hálózat nem szakad meg, és nem kell újra építeni.

A Consul template egy új fejlesztése a Hashicorp-nak, és a honlapon ad is pár mintát a fejlesztő cég, azonban nincs kiélezve a valóságra, így kénytelen voltam más forrásokból megtudni, hogy hogyan célszerű használni a Consul Template-et HAProxy-val. Az előző félévben események küldésével oldottam ezt meg, viszont ez az új funkció sokkal ígéretesebbnek tűnt.

4.3.3. Váratlan meglepetések

Ugyan a legnagyobb meglepetést a Docker okozta az image inkonzisztenciájával, de ettől függetlenül volt néhány apróbb meglepetés, ami a fejlesztést visszavetette. Egy ilyen meglepetés például, hogy a Docker konténerekben beállított hosts fájl tartalma már nem az mint korábban, és ezen keresztül nem látják egymást a konténerekek. A HAProxy-val kapcsolatban ez előző félévhez képest nem történt semmi, azonban a felhasznált verzió változhatott, mivel sem az indító szkript, sem a konfigurációs fájl nem ott volt megtalálható, mint az előző félévben.

Ami a MySQL-t illeti a Docker konténerekben való használata nem igazán támogatott dolog, mivel nem kéne kritikus adatokat használni a konténerekben, de az előző félévhez képest változott a csomag, mivel a konténer buildelésénél egy olyan hiba merült fel, hogy a csomag kiakadt, amikor a felhasználó nevet és jelszót kérte a telepítő. Ezt egy egyszerű módosítás megoldotta, de nagyon sok idő volt rájönni, mivel is van a baj. Ami az adatbázist illeti, nem csak a csomag de a működés és a hozzá tartozó authentikáció is megváltozott, mivel az előző félévben képes voltam jelszó nálkül használni a felhasználót, addig ebben a félévben kötelező volt megadnom valamit, ami az összes szolgáltatás változtatásával járt.

4.3.4. Értékelés

4.3.4.1. Komponensek

A minta alkalmazás főbb komponensei a mikroszolgáltatások elve szerint lettek szétválasztva, azonban az adatbázis külön szolgáltatásként kezelése nem teljesen ezt a metodológiát követi. A praktikusság és erőforrás kezelés szempontjából sokkal jobb megoldást kaptam, mint ha az össze szolgáltatás önálló adatbázissal rendelkezne, de lehet, hogy még jobb lett

volna egy szolgáltatásokon kívül elhelyezkedő adatbázis szerver. Ami az authentikációt illeti, lehetséges, hogy jobban jártam volna egy szolgáltatásonkénti ellenörzéssel, mivel ha kiveszem az authentikációt vezérlő szolgáltatást, az egész alkalmazás veszélynek lesz kitéve. Természetesen más megközelítésből ez a tervezés éppen jó, hiszen az authentikálás funkcióját csak egy szolgáltatás végzi, és így plugin-szerűen kivehető, vagy berakható.

4.3.4.2. Kommunikáció

Ami a Rest kommunikációt illeti, teljesen megfelelt a célnak, és könnyen egyszerűen tudtam vele változtatni az interfészeket, és nagyon egyszerű volt a kiszolgáló alkalmazások elkészítése is. Ha cél lett volna más kommunikációs technológiák bemutatása is, akkor sokkal nehezebb feladatom lett volna, mivel a kevert Rest-es és XML alapú, vagy socket kommunikáció, komoly többlet fejlesztés okozott volna, illetve a struktúrát is átrajzolta volna.

4.3.4.3. Fejlesztési élmény

A minta alkalmazás fejlesztése során nem használtam semmilyen folytonos integrációt támogató eszközt, se semmilyen automatizált módszert, ami ellenőrizte volna a munkámat. Sajnos ez nagyon megnehezítette a fejlesztést, mivel minden alkalommal külön meg kellett győződnöm arról, hogy az alkalmazás minden eleme külön elkészíthető, és a létrehozott alkalmazás is jól használható, illetve az interfészek is jól működnek. Ha lett volna valamilyen automatizált módszer, ami a kód szintaktikáját folyamatosan figyelte volna minden változtatáskor, vagy lett volna valamilyen logika, ami megpróbálta volna összerakni az alkalmazást, és funkció tesztek alapján megmondta volna, hogy jó-e amit csinálok, akkor sokkal gyorsabban is haladhattam volna. A félév során több Docker-es problémába is beleütköztem, és volt, hogy csak órák alatt jöttem rá, hogy a Docker konténer a hibás. Ezt egy egyszerű konzisztencia ellenőrzés elkaphatta volna és percek alatt meglett volna a hibaok.

4.4. Automatizálás

Mint azt az előző fejezetben kifejtettem, sok haszna lehet, ha van valamilyen automatizált tesztelő, és integráló eszköz a mikroszolgáltatás alapú infrastruktúránk támogatására. Ahhoz, hogy teljes mértékben támogatni tudjuk a fejlesztés minden aspektusát, különböző módszereket, és alkalmazásokat tudunk használni.

4.4.1. Kódolás tesztelése

Mivel folyamatosan készül új kód, és ennek a kódnak minőségileg és tartalmilag is helyesnek kell lennie, teszteket készíthetünk, amik még kezdeti fázisban elkapják a stillisztikai és kódolási hibákat. Ilyen lehet például a Git verzió kezelő esetén egy olya hook, ami leellenőrzi egy kód ellenőrzővel a szintaktikát (mint például Python kód esetén pylint alkalmazást futtatva), majd megvizonyosodik a szemantikáról is egy gyors unit teszt futtatással. Ez

a megoldás egy azonnali választ ad a fejlesztőnek, hogy jól csinálja amit csinál, és vissza tartja a hibás kódrészleteket.

4.4.2. Funkcionalitás tesztelése

Mivel minden konténer önálló alkalmazás, így az egyes alkalmazásokat magukban is lehet tesztelni, és lefedve az összes lehetséges felhasználási módot funkció teszteket futtathatunk. Ez a megoldás szintén lehet valamilyen verzió kezelőhöz kötve, vagy egy automatozáló eszközt felhasználva, tudunk eseményekre teszteket futtatni. Ez főleg abban segít, hogy ne kelljen értelmetlenül időt pazarolni a mikroszolgáltatás buildelésére, illetve az integrációra, ha meg tudjuk mondani, hogy nem megfelelő a szolgáltatás alapja.

4.4.3. Interfész tesztek

Ez egy nehezebb kérdés, de hasznos lehet, ha valamilyen módon le tudjuk ellenőrizni, hogy a szolgáltatások közötti interfészek konzisztensek-e a régiekkel, és használhatók-e rendesen. Ez leginkább csak akkor kivitelezhető, ha jó leíró fájlokkal tudjuk megmondani, hogy milyen intefészek vannak az alkalmazásban. Ezzel a teszttel a cél az lenne, hogy ne tudja a fejlesztő csapatok közül egy se elrontani a többi szolgáltatás interfészével az konzisztenciát. Az automatizált megoldás a kódolás tesztek mellett futhat, hiszen erről is egy gyors visszajelzés a legcélravezetőbb.

4.4.4. Teljes működés tesztje

Ez a legbonyolultabb művelet, amivel meg tudjuk mondani, hogy mindaz amit fejlesztettünk működik, együttműködik a többi szolgáltatással, és pontosan úgy működik ahogy szeretnénk. Sokan a pipeline terminológiát használják az ehhez hasonló funkcióhoz, ami annyit jelent, hogy a beadott adatokra egy kimeneti artifactot kapunk. A legáltalánosabb módszer, amit szinte mimnden felhasználásra rá lehet húzni, az egy buildelési folyamattal indul (építés), majd összeállít egy tesztelhető környezetet (telepítés), és végül a funkció teszteket, vagy integrációs teszteket (teszt) futtata. Ehhez a tesztelési stílushoz már kell egy olyan eszköz, amivel összehangoltan fel tudjuk építeni a teljes folyamot, és meg tudjuk oldani, hogy a végeredmény és a köztes információk elérhetők legyenek. Egy ilyen eszköz a folytonos integrációt támogató Jenkins, amiben könnyedén össze lehet állítani a megfelelő folyamot, hívási struktúrát. Természetesen ha szeretnénk csinálhatunk saját megoldást is, ami szkripttekkel és időzítő eszközökkel (például Bash és Cron) szintén megoldható.

4.5. Folytonos integrációs lépések tervezése

Mielőtt nekikezdtem volna a konkrét implementációnak, megnéztem, hogy miket is kéne végrehajtanom a saját pipeline-om futtatása során.

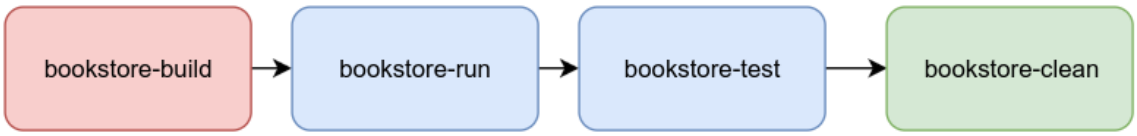
1. **Teszt alkalmazás build-elése**: Gyakran van szükség a szolgáltatást futtató fájlok és egyéb tartalmak fordítására (C, Java, bináris kép fájlok frodítása), és ezeket a

forrásokat könnyedén elkészíthetjük automatizáltan is, mielőtt a környezetet összeépítenénk.

2. **Teszt architektúra telepítése**: Az egyes szolgáltatásokat egy felügyelt környezetbe helyezve valamilyen környezeti konfigurációval együtt telepíthetjük (esetünkben Docker konténerekbe csomagolhatjuk), és az így kialakuló architektúrát használhatjuk fel a céljainkra. (Esetünkben kialakítunk egy könyvesboltot)
3. **Teszt architektúra tesztelése**: Az éles futó architektúrán futtathatunk teszteket, amikkel megbizonyosodhatunk, hogy a rendszer megfelelően működik, és minden rendben van, átadható a megrendelőnek, vagy átengedhető a felhasználóknak. Ilyen teszt lehet az alkalmazás elemeinek a unit tesztelése, szolgáltatásonként funkció tesztek futtatása, a szolgáltatások kapcsolaihoz integrációs és rendszer tesztek futtatása, illetve a skálázás és egyéb teljesítményt befojásoló tényezőkhöz teljesítmény tesztek futtatása.

A korábbi félévek során találkoztam már a Jenkins eszközzel, és ennek megfelelően volt némmi tapasztalatom a használatával, azonban kijött belőle egy új verzió, ami sok új funkciót, és sok lehetőséget adott a hatékonyabb tervezésre.

Először is megpróbáltam megtervezni a feladatokat és végrehajtási sorrendet, az általam ismert Jenkins verzióban:

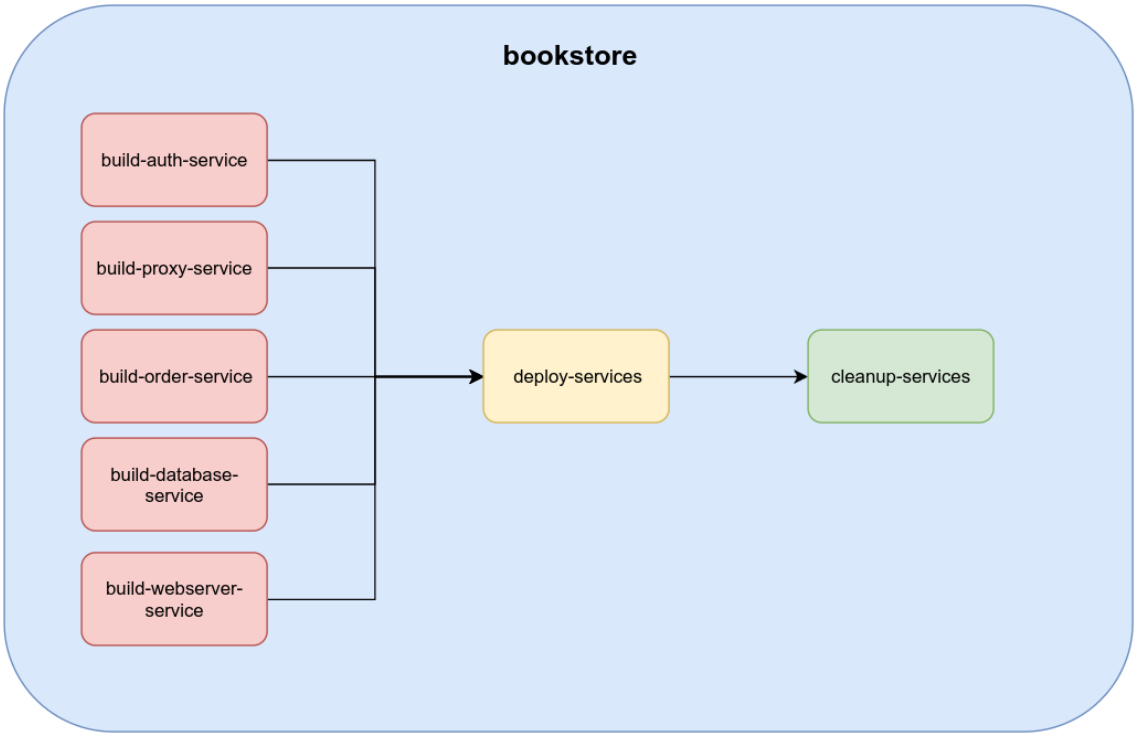


4.1. ábra. Régi Jenkins terve

- **bookstore-build**: A forrásfájlok és a Docker konténerek felkészítése. Miután a job végzett, a teljes infrastruktúra elkészíthető Docker konténerekből.
- **bookstore-run**: Feladata a Docker konténerek indítása, a szolgáltatások iniciaálizálása.
- **bookstore-test**: Unit tesztek futtatása a feladata, de ide tartoznának a funkció és integrációs tesztek is, illetve a teljesítmény tesztek.
- **bookstore-clean**: Ennek a job-nak a feladata, hogy a környezet ki legyen tisztítva, és ne maradjon a tesztek után semilyen Docker konténer, vagy fordított fájl a munkaterületen (workspace).

Egymásra épülő feladatokat (job-okat) alakítottam ki, amik szépen sorban lettek végrehajtva. Ahogy az látható egy build készítette el az összes szolgáltatást, és a hozzájuk tartozó Docker konténert. Ha minden elkészült rendben, akkor egy alap infrastruktúra jött létre a konténerekből, majd teszteket futtathattunk, és végül felszámolva a minta környezetet törlődött minden. Amit ezzel szemben az új verzió nyújtott, az egy átfogó nézet a futó folyamatokról, és egy párhozamosan futtatható build szakasz, ami annyit jelent, hogy külön-külön egyesével készíthettem el a szolgáltatásokat, de mégis csak 1 build

idejét kellett kivárnom. Mivel nem tudtam hogyan is lehetne jól tesztelni a folyamatot, és a tesztelés fázisa bármiből állhatna, ezért ezt a szakaszt kihagytam, de éles rendszeren ajánlott alkalmazni. Az új feladat rendszer tehát a következőképp nézett ki:

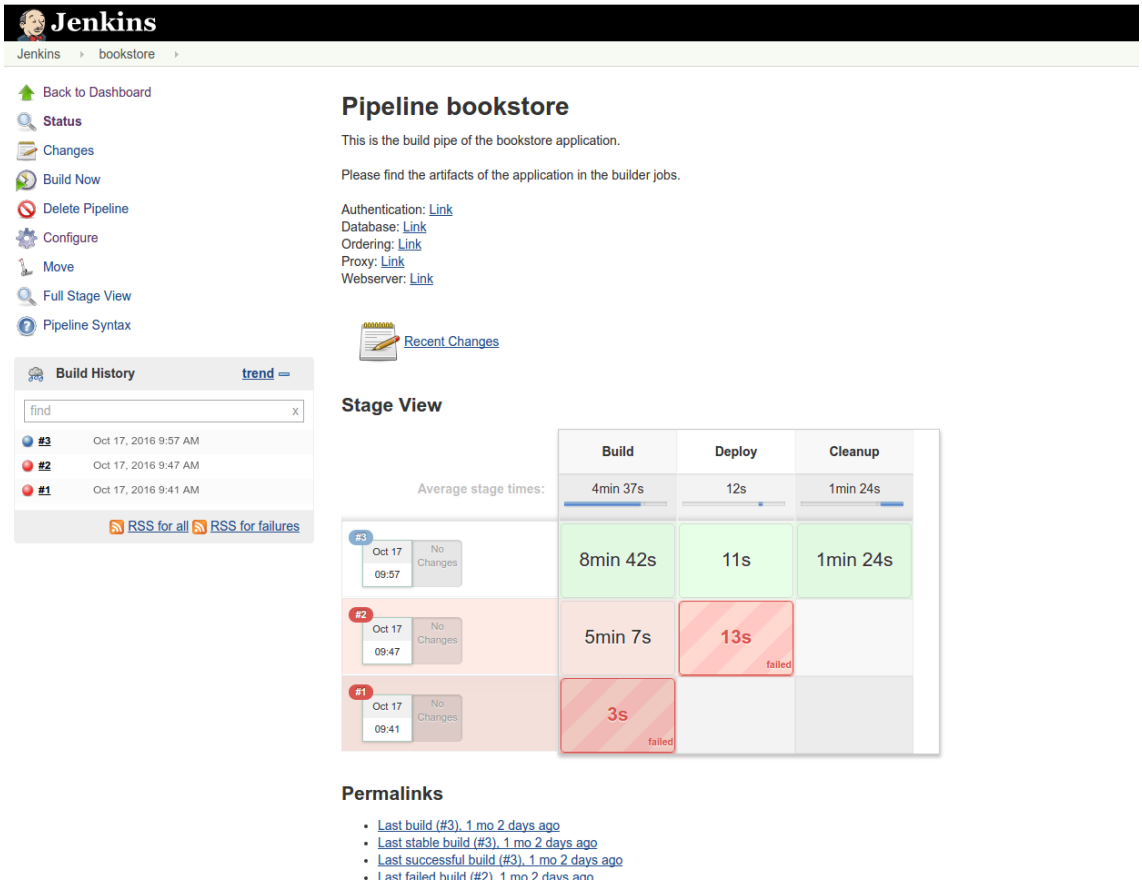


4.2. ábra. Új Jenkins terve

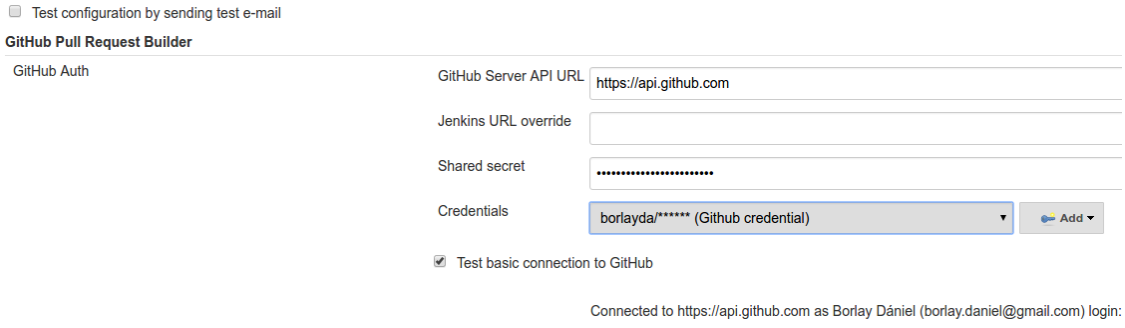
Az alkalmazáshoz több féle végeredmény is tartozik, mivel minden szolgáltatás önálló termékként funkcionál. Ennek megfelelően minden build eredménye mentődik a Jenkins-ben, és jól beazonosítható, minden futtatáshoz. Nagyon is hasznos lehet ez egy debug-olás, vagy hibakeresés során, mivel a környezetre szükség van a folyamatos ellenörzés közben, de a fejlesztő saját gépén kipróbálhatja pontosan ugyan azt az alkalmazást, amiben a hiba előfordult.

Az összefogó pipeline job-nak az az előnye, hogy látható mely szakasz mennyi ideig tart, és hogy melyik szakaszban történt hiba. A Jenkins lehetőséget ad egy saját Pipeline alkotó nyelv, ami a Groovy-n alapszik, és meg lehet vele hastározni, hogy pontosan mit és hogyan csináljon meg a folyam. A diplomatervben elkészített pipeline job-nak a szkriptje megtalálható a Függelékben.

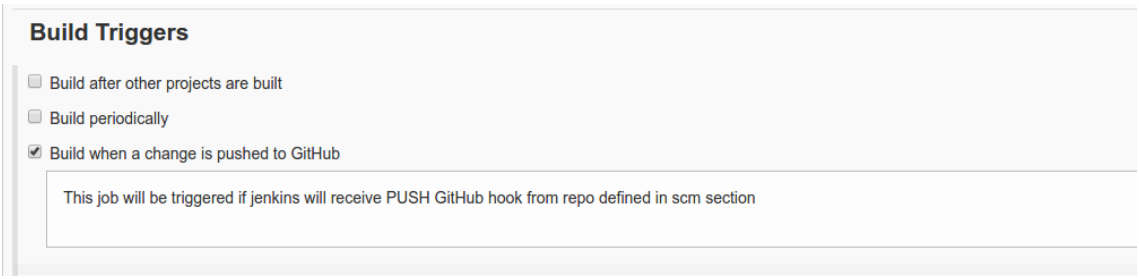
Amit még ajánlott alkalmazni egy ilyen rendszer kiépítése során, az egy verzió kezelő alkalmazás, amit össze lehet kapcsolni a Jenkins-el, mivel sok verziókezelőhöz van támogatott feladat indító feltétel (trigger). Egy ilyen feltétel beállításai láthatók az 4.4 és 4.5 képeken.



4.3. ábra. Pipeline job



4.4. ábra. Github beállítások a Jenkinsben



4.5. ábra. Github trigger beállítása egy job-on

5. fejezet

Valós alkalmazás értékelése

6. fejezet

Összefoglaló

A diplomaterv összefoglaló fejezete.

Táblázatok jegyzéke

Ábrák jegyzéke

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] Archivematica. Archivematica 1.1 micro-services. https://wiki.archivematica.org/Archivematica_1.1_Micro-services.

[6] David Chou. Using events in highly distributed architectures. *The Architecture Jounal*, October 2008.

[7] Cronitor. Monitoring microservices. https://cronitor.io/help/micro-service-monitoring.

[8] Cronitor.io. Cronitor. https://cronitor.io/.

[9] Manfréd Sneps-Sneppe 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] Peter Van Garderen. Archivematica: Using micro-services and open-source software to deliver a conprehensive digital curation solution. In *iPres 2010*, pages 145–150, Vienna, Austria, 19–24 2010.

[16] Nemeth Gergely. Monitoring microservices. https://www.loggly.com/blog/monitoring-microservices-three-ways-to-overcome-the-biggest-challenges/.

[17] The HFT Guy. Docker in production: A history of failure. https://thehftguy.wordpress.com/2016/11/01/docker-in-production-an-history-of-failure/, November 2016.

[18] HAProxy. The reliable, high performance tcp/http load balancer. http://www.haproxy.org/.

[19] Hashicorp. Consul. https://www.consul.io/.

[20] Chef Software Inc. Chef. https://www.chef.io/chef/.

[21] SaltStack inc. Saltstack. http://saltstack.com/.

[22] Jenkins. Elasticbox ci. https://wiki.jenkins-ci.org/display/JENKINS/ElasticBox+CI.

[23] Jenkins. Jenkins. https://jenkins.io/index.html.

[24] Mercedes Garijo Jose Ignacio Fernández-Villamor, Carlos Á. Iglesias. Microservices: Lightwieght services descriptors for REST architectural style. http://oa.upm.es/8128/1/INVE_MEM_2010_81293.pdf.

[25] Chris Richardson (Kong). Pattern: Microservices Architecture. http://microservices.io/patterns/microservices.html.

[26] KVM. Kernel virtual machine. http://www.linux-kvm.org/page/Main_Page.

[27] Libvirt. libvirt: The virtualization api. https://libvirt.org/.

[28] David Liu. Euraka at glance. https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance.

[29] Fideloper LLC. Load balancing with haproxy. https://serversforhackers.com/load-balancing-with-haproxy.

[30] Microsoft. Pipes and Filters Pattern. https://msdn.microsoft.com/en-us/library/dn568100.aspx.

[31] Microsoft. Publish/Subscribe. https://msdn.microsoft.com/en-us/library/ff649664.aspx.

[32] Mrina Natarajan. 3 golden rules of microservices deployments. http://devops.com/2015/05/07/3-golden-rules-microservices-deployments/.

[33] Nginx. Nginx. https://www.nginx.com/.

[34] Sebastián Peyrott. An introduction to microservices, part 1. https://auth0.com/blog/2015/09/04/an-introduction-to-microservices-part-1/.

[35] 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/.

[36] Puppet. Puppet 4.4 reference manual. https://docs.puppet.com/puppet/latest/reference/, 2016.

[37] Chris Richardson. The Scale Cube. http://microservices.io/articles/scalecube.html.

[38] Chris Richardson. Service registry pattern. http://microservices.io/patterns/service-registry.html.

[39] Chris Richardson. Building microservices: Inter-process communication in a microservices architecture. https://www.nginx.com/blog/building-microservices-inter-process-communication/, July 2015.

[40] Ruxit. Microservice monitoring. https://ruxit.com/microservices/#microservices_start.

[41] Sensu. What is sensu? https://sensuapp.org/docs/latest/overview.

[42] Puja Padiya Snehal Mumbaikar. Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3, May 2013.

[43] tutorialspoint. What is a socket? http://www.tutorialspoint.com/unix_sockets/what_is_socket.htm.

[44] w3schools. Xml soap. http://www.w3schools.com/xml/xml_soap.asp.

[45] 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.

[46] Benjamin Wootton. Microservices - not a free lunch! http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html, apr 2014.

[47] 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>

COPY auth.sh /usr/sbin/auth.sh
COPY auth-service.py /usr/sbin/auth-service.py

RUN apt-get -y update
RUN apt-get -y install vim bash python-oauth python-mysqldb python \
    python-flask
RUN chmod +x /usr/sbin/auth.sh
RUN chmod +x /usr/sbin/auth-service.py

EXPOSE 8081
```

A.1.2. Proxy

Dockerfile.proxy.service

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

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

RUN apt-get -y update
RUN apt-get -y install vim haproxy
RUN chmod +x /usr/sbin/proxy.sh
COPY haproxy.cfg /etc/haproxy/haproxy.cfg
```

```
ENTRYPOINT proxy.sh

EXPOSE 8080
```

A.1.3. Adatbázis

Dockerfile.database.service

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

COPY database.sh /usr/sbin/database.sh
COPY auth_init.sql /tmp/auth_init.sql
COPY bookstore_init.sql /tmp/bookstore_init.sql

RUN apt-get -y update
RUN apt-get -y install mysql-server mysql-client vim
RUN chmod +x /usr/sbin/database.sh
RUN sed -i 's/bind-address.*=.*/bind-address = 0.0.0.0/g' /etc/mysql/my.cnf

EXPOSE 3306
```

A.1.4. Vásárlás

Dockerfile.reserve.service

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

COPY reserve.sh /usr/sbin/reserve.sh

RUN yum -y update
RUN yum -y install vim java-1.8.0-openjdk-devel tomcat7
RUN chmod +x /usr/sbin/reserve.sh

EXPOSE 8888
```

A.1.5. Webkiszolgáló (böngészés)

Dockerfile.webserver.service

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

```
COPY webserver.sh /usr/sbin/webserver.sh
COPY index.html /var/www/html/index.html
COPY login.php /var/www/html/login.php
COPY store.php /var/www/html/store.php

RUN apt-get -y update
RUN apt-get -y install vim php5 php5-mysql curl php5-curl
RUN chmod +x /usr/sbin/webserver.sh

EXPOSE 80 443
```

A.2. Szkriptek

A.2.1. Futtatáshoz

A.2.1.1. Build

build_docker.sh

```
#!/bin/bash

services="database webserver proxy reserve auth"

for service in ${services}
do
    echo "Create ${service} for bookstore ..."
    mkdir -p services/${service}
    cp Dockerfiles/Dockerfile.${service}.service services/${service}/Dockerfile
    cp -R scripts/${service}/* services/${service}/
    docker build -t bookstore_${service} services/${service} \
      &> services/${service}/build.log
done

echo "Microservices has been created!"
```

A.2.1.2. Futtatás

run_containers.sh

```
#!/bin/bash

services="database webserver reserve 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} ${service}.sh "${DOCKER_IP_HAPROXY}"
done
```

A.2.1.3. Tisztogatás

clean_docker.sh

```
#!/bin/bash

services="database webserver proxy reserve 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

rm -rf services
docker network rm bookstore
```

A.2.2. Szolgáltatásokhoz

A.2.2.1. 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.2.2.2. Bejelentkezéshez

login.php:

```
<?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}"
    );
    $content = curl_exec($ch);
    echo $content;
}
?>
```

auth-service.py:

```
#!/usr/bin/env python
from flask import Flask, abort
import MySQLdb as mdb
app = Flask(__name__)

@app.route("/auth/<username>/<password>")
def hello(username, password):
    try:
        con = mdb.connect('database', 'root', '', 'authenticate');
        cur = con.cursor()
        cur.execute("SELECT user_id FROM user_auth \
                     WHERE username='%s' AND password='%s'" %
                     (username, password))
        user_id = cur.fetchone()
        print user_id
        if not user_id:
            abort(401)
    except mdb.Error, e:
        print "Error %d: %s" % (e.args[0],e.args[1])
        abort(401)
    finally:
```

```
        if con:
            con.close()
    return "Successfully authenticated!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8081)
```

A.2.2.3. Böngészés

index.html:

```
<html>
<head>
<title>Bookstore Microservice</title>
</head>
<body>

<h2>Login:</h2>
<form action="login.php" method="post">
  <fieldset>
    <p>
      <label for="username">Username</label>
      <input type="text" id="username" name="username" value=""/>
    </p>
    <p>
      <label for="password">Password</label>
      <input type="text" id="password" name="password" value=""/>
    </p>
    <p>
      <input type="submit" value="Login" />
    </p>
  </fieldset>
</form>

</body>
</html>
```

store.php:

```
<html>
<head>
<title>Bookstore Microservice</title>
</head>
<body>
```

```
<h2>Books:</h2>

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

    <?php
      $servername = "database";
      $username = "root";
      $password = "";
      $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>

</body>
</html>
```

Proxy config:

```
frontend web
    bind *:80
    mode http
    default_backend nodes

backend nodes
    mode http
    balance roundrobin
    server webserver webserver:80 cookie check
```