



M Ű E G Y E T E M 1 7 8 2

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. május 15.

Tartalomjegyzék

| | |
|--|-----------|
| Kivonat | 4 |
| Abstract | 5 |
| 1. Mikro szolgáltatások[5] [2] [4] | 6 |
| 1.1. Szolgáltatás elválasztás tervezése | 6 |
| 1.1.1. Adat menti felbontás | 7 |
| 1.1.2. Fogalmi felbontás | 8 |
| 1.1.3. Klónozás | 8 |
| 1.2. Architektúrális mintákhoz való viszonya | 8 |
| 1.2.1. Pipes and Filters | 8 |
| 1.2.2. Publish/Subscribe | 8 |
| 1.2.3. Esemény alapú architektúra | 9 |
| 1.3. Eltérések a szolgáltatás orientált architektúrától | 9 |
| 1.4. Példák mikro szolgáltatásokat használó alkalmazásokra | 9 |
| 2. Mikro szolgáltatások előnyei és hátrányai | 10 |
| 2.1. Előnyök [5] | 10 |
| 2.1.1. Könnyű fejleszteni | 10 |
| 2.1.2. Egyszerűen megérthető | 10 |
| 2.1.3. Könnyen kicserélhető, módosítható, telepíthető | 10 |
| 2.1.4. Jól skálázható | 10 |
| 2.1.5. Támogatja a kevert technológiákat | 11 |
| 2.2. Hátrányok [14] | 11 |
| 2.2.1. Komplex rendszer alakul ki | 11 |
| 2.2.2. Nehezen kezelhető az elosztott rendszer | 11 |
| 2.2.3. Plusz munkát jelent az aszinkron üzenet fogadás | 11 |
| 2.2.4. Kód duplikátumok kialakulása | 11 |
| 2.2.5. Interfészek fixálódnak | 11 |
| 2.2.6. Nehezen tesztelhető egészben | 11 |
| 2.3. Összehasonlítva a monolitikus architektúrával | 12 |
| 3. Technológiai áttekintés | 13 |
| 3.1. Telepítési technológiák | 13 |

| | | |
|-----------|--|-----------|
| 3.2. | Környezet felderítési technológiák | 14 |
| 3.3. | Integrációs keretrendszerek | 14 |
| 3.4. | Skálázási technológiák | 15 |
| 3.5. | Terhelés elosztás | 15 |
| 3.6. | Virtualizációs technológiák | 16 |
| 3.7. | Service registry-k | 17 |
| 3.8. | Monitorozás, loggolás | 17 |
| 4. | Kommunikációs módszerek | 19 |
| 4.1. | Technológiák | 19 |
| 4.1.1. | REST (HTTP/JSON)[4]: | 19 |
| 4.1.2. | SOAP (HTTP/XML)[12]: | 19 |
| 4.1.3. | Socket (TCP)[11]: | 20 |
| 4.2. | Interfészek | 20 |
| 5. | Minta alkalmazás terve | 21 |
| 5.1. | Alkalmazás leírás: | 21 |
| 5.2. | Szolgáltatások: | 21 |
| 6. | Minta alkalmazás elkészítése | 23 |
| 6.1. | Megvalósítás Docker konténerekkel: | 23 |
| 6.2. | Kapcsolatok építése Consul-al: | 24 |
| 6.3. | Kapcsolatok építése Docker-el: | 24 |
| 6.4. | Kapcsolatok építése Zookeeper-el: | 24 |
| 6.5. | Automatizálás: | 24 |
| 6.6. | Jenkins Job-ok fejlesztése: | 25 |
| 6.7. | Egyéb minta alkalmazások: | 26 |
| 7. | Összefoglaló | 28 |
| A. | Függelék | 33 |
| A.1. | Dockerfile-ok | 33 |
| A.1.1. | Authentikáció | 33 |
| A.1.2. | Proxy | 33 |
| A.1.3. | Adatbázis | 34 |
| A.1.4. | Vásárlás | 34 |
| A.1.5. | Webkiszolgáló (böngészés) | 34 |
| A.2. | Szkriptek | 35 |
| A.2.1. | Futtatáshoz | 35 |
| A.2.2. | Szolgáltatásokhoz | 36 |

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. május 15.

Borlay Dániel
hallgató

Kivonat

Napjainkban komoly gondot okoz, hogy hogyan lehet hatékonyan elosztott, jó rendelkezésre állású, könnyen skálázható rendszert építeni. Sok architektúrális megközelítés van, amit alapul véve hatékonyan 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 működését, 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

Mikro szolgáltatások[5] [2] [4]

A mikro szolgáltatás 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 mikro szolgáltatást 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 ideális, ha nem tudjuk tovább bontani az adott funkciót.

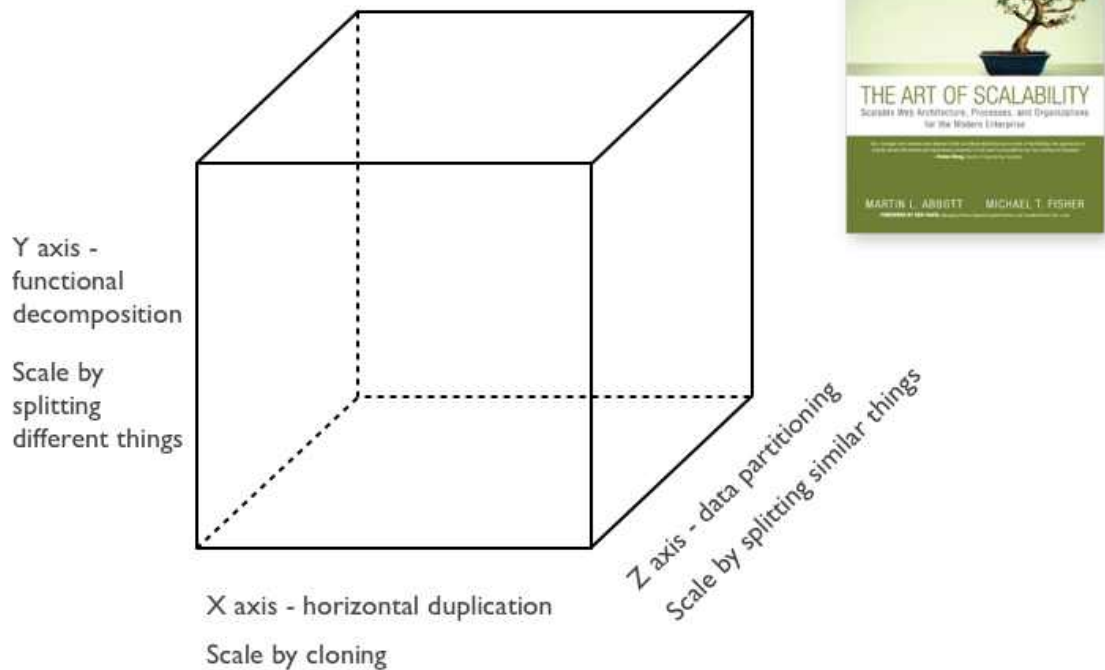
1.1. Szolgáltatás elválasztás tervezése

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

3 dimensions to scaling



1.1. ábra. Scaling Cube

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

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 mikro szolgáltatások 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.

1.1.2. Fogalmi felbontás

A tetszőleges fogalom menti felbontás annyit tesz hogy elosztott rendszert hozunk létre tetszőleges funkcionalitás szerint. Erre épít a mikro szolgáltatás 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.

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 mikro szolgáltatásoknál kell, hogy teljesüljön, mivel adott esetben egy terhelés elosztó alatt tudnunk kell definiálni több példányt is egy szolgáltatásból. Azért szükséges a skálázhatóság a mikro szolgáltatások esetén, mivel kevés hardver mellett is hatékonyan kialakítható az architektúra, de könnyen lehet szűk keresztmetszetet létrehozni, amit skálázással könnyen megkerülhetünk.

1.2. Architektúrális mintákhoz való viszonya

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

Nézzünk meg két felhasználható architektúrális mintát:

1.2.1. Pipes and Filters

A Pipes and filter architektúrális minta[6] 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.

1.2.2. Publish/Subscribe

Egy másik elosztott rendszerekhez kitallált minta a subscriber/publisher[7], amely arra 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.

1.2.3. Esemény alapú architektúra

Az esemény alapú architektúrákat[1] könnyen kalakíthatjuk, ha egy mikro szolgáltatásokból álló rendszerben olyan alkalmazásokat és kompoenenseket fejlesztünk ahol eseményeken keresztül kommunikálnak az egyes elemek. Ezzel a nézettel olyan mikro szolgáltatásokból kialakuló struktúrát lehet összeépíteni, ahol a kis egységek szükség szerint kommunikálnak, és a kommunikáció egy jól definiált interfészen keresztül történik.

1.3. Eltérések a szolgáltatás orientált architektúrától

A szolgáltatás orientált architektúra nagyon hasonló a mikro szolgáltatásokhoz, azonban utóbbi esetén a lényeg az, hogy minnél kisebb alkotó elemekre bontsuk szét a rendszert és akár alkalmazásokat is, míg előbbi több alkalmazást köt össze hálózati kapcsolaton keresztül, és ezeket szolgáltatja ki. Bizonyos értelemben a mikro szolgáltatás architektúra a szolgáltatás orientált architektúra finomítása.

1.4. Példák mikro szolgáltatásokat 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 felbontva a a funkcionalitás, és ennek megfelelően külön szolgáltatásként érhető el a fizetés, megrendelés, szállítási információk, stb.

NetFlix - A nagy terhelést elkerülendő bizonyos streaming szolgáltatásokat átlalakítottak, hogy a mikro szolgáltatás architektúra szerint működjön.

Archivematica[3] - Egy Fájlfelkezelő rendszer, amiben mikro szolgáltatásoknak megfelelően alakították ki a plugin-ként használható funkciókat.

2. fejezet

Mikro szolgáltatások előnyei és hátrányai

Ahogy minden architektúrális mintának, a mikro szolgáltatásoknak is vannak előnyei, amik indokoltá teszik a minta használatát, és vannak hátrányai, amiket mérlegelnünk kell a tervezés folyamán.

2.1. Előnyök [5]

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 ez kis feladatokat csinál, és könnyebben kezelhető.

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.

2.1.3. Könnyen kicserélhető, módosítható, telepíthető

A szolgáltatások önállóak, így az azonos interfésszel rendelkező szolgáltatásra bármikor kicserélhető, illetve módosítható ha megmaradnak a korábbi funkciók. A mikro szolgáltatás telepítése is egyszerű, mivel csak kevés környezeti feltétele van, hogy egy ilyen kis méretű program működni tudjon.

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.

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, és minden gond nélkül használni is tudjuk az interfész segítségével.

2.2. Hátrányok [14]

2.2.1. Komplex rendszer alakul ki

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

2.2.2. Nehezen kezelhető az elosztott rendszer

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

2.2.3. Plusz munkát jelent 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.

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.

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.

2.2.6. Nehezen tesztelhető egészben

Mivel sok kis részletből rakódik össze a nagy egész alkalmazás, a tesztelési fázisban kell olyan tesztek 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.

2.3. Összehasonlítva a monolitikus architektúrával

A mirco-service architektúrák a monolitikus architektúra ellentetjai, 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 ellenzétben a mikro szolgáltatás architektúrát ugyan nehezen lehet megtervezni, hiszen egy elosztott rendszert kell megtervezni, ahol az adatátviteltől kezdve az erőforrás megosztáson keresztül semmi sem egyértelmű, viszont a későbbi tovább fejlesztés 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 mirco-service architektúrának, hogy különböző technológiákat lehet keverni vele, mivel az egyes szolgáltatások különböző technológiákkal különböző platformon is futhatnak.

3. fejezet

Technológiai áttekintés

Az integrációhoz olyan technológiákat lehet használni, melyek lehetővé teszik az egyes szolgáltatások elkülönült működését.

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? (integrációs keretrendszer) * 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 registry) * Hogyan figyeljük meg a rendszert működés közben (monitorozás, loggolás)

3.1. Telepítési technológiák

A microservice-eket 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 kezelhetsük 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:** 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 microservice architektúrák esetén alkalmas a szolgáltatások telepítésére, és tesztelésére.
- **ElasticBox:** 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 cloud megoldásokkal, mint az AWS, vSphere, Azure, és más környezetek. Hasonlít a Jenkins-re, csupán ki van élezve a microservice architektúrák vezérlésére. (Illetve fizetős a Jenkins-el ellentétben) Mindent végre tud hajtani ami egy microservice alkalmazáshoz szükséges, teljes körű felügyeletet biztosít.

Egyéb lehetőség, hogy a fejlesztő készít magának egy olyan szkriptet, ami elkészíti számára a micro-service architektúrát, és lehetővé teszi az elemek dinamikus kicserélését. (ad-hoc megoldás)

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:** A Hashicorp szolgáltatás felderítő alkalmazása, amely egy kliens-szerver architektúrának megfelelően megtalálja a környezetében lévő szolgáltatásokat, és figyeli az állapotukat (ha inaktívvá válik egy szolgáltatás a Consul észreveszi). Ez az alkalmazás egy folyamatosan választott mester node-ból és a többi slave node-ból 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.

3.3. Integrációs keretrendszerek

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:** 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:** 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, amiz könnyedén használhatunk integrációhoz, illetve a szolgáltatások cseréjéhez, és karbantartásához.

- **Ansible:** 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:** A SaltStack nagyon hasonlít a Chef-re, mivel ez a termék is széleskörű felügyeletet, és konfiguráció menedzsment-et kínál számunkra, amit folyamatos kapcsolat fenntartással, és gyors kommunikációval ér el. Az Ansible-höz nagyon hasonlóan konfigurálható (nem lennék meglepve ha azt használná a háttérben), szintén ágens nélküli kapcsolatot tud létesíteni, és a Chef-hez hasonlóan több 10 ezer gépet tud egyszerre karbantartani.

3.4. Skálázási technológiák

A microservice architektúrák egyik nagy előnye, hogy az egyes funkciókra épülő szolgáltatásokat könnyedén lehet skálázni, mivel egy load balancert használva csupán egy újabb gépet kell beszerezni, és máris nagyobb terhelést is elbír a rendszer. Ahhoz hogy ezt kivitelezni tudjuk, szükségünk van egy terhelés elosztóra, és egy olyan logikára, ami képes megsokszorozni az erőforrásainkat. Cloud-os 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:** 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, 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 microservice architektúránkat, akkor erős eszköz lehet számunkra.

3.5. Terhelés elosztás

A microservice 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:** Egy magas rendelkezésre állást biztosító, és megbízhatóságot növelő terhelés elosztó eszköz. Konfigurációs fájlokon keresztül megszervezhetjük, hogy mely gépet hogyan érjük el, milyen IP címek mely szolgáltatásokhoz tartoznak, illetve

round robin 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 dinamikusan változnak a fürtben lévő gépek, mert ha igen akkor szükséges egy mellékes frissítő logika is)

- **nginx:** 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 terhelés elosztásra, és alkalmazás telepítésre. Nem teljesen a proxy szerver szerepét váltja ki, de képes elvégezni azt.

3.6. Virtualizációs technológiák

A microservice 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:** 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 host gépen, és abban hajt végre műveleteket. Készíthetünk vele előre elkészített alkalmazás környezeteket, és szolgáltatásokat, ami ideálissá teszi microservice architektúrák létrehozásánál. A Docker konténerek segítségével egyszerűen telepíthetjük, skálázhatjuk, és fejleszthetjük a rendszert.
- **libvirt:** 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, és sok más virtualizáló eszközt. Ezzel az eszközzel a környezet kialakítását szabhatjuk meg, tehát a hardware-eserőforrások megosztásában nyújt nagy segítséget.
- **kvm:** 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.
- **Akármilyen cloud:** Ha virtualizációról beszélünk, akkor adja magát hogy a Cloud-os környezeteket is ide értsük. Egy microservice architektúrájú programot a legcélsebb valamilyen Cloud-os környezetben létrehozni, mivel egy ilyen környezetnek definíciója szerint tartalmaznia kell egy virtualizációs szintet, megosztott erőforrásokat, monitorozást, és egyfajta leltárat a futó példányokról. Ennek megfelelően a microservice architektúra minden környezeti feltételét lefedi, csupán a szolgáltatásokat, business logikát, és az interfészeket kell elkészítenünk. Jellemzően a Cloud-os

környezetek tartalmazznak terhelés elosztást, 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.

3.7. Service registry-k

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

- **Eureka:** Az Eureka a Netflix fejlesztése, egy AWS környezetben működő terhelés elosztó 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:** 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.

3.8. Monitorozás, loggolás

Ha már megépítettük a microservice architektúrát, akkor meg kell bizonyosodnunk róla, hogy minden megfelelően működik, és minden rendben zajlik a szolgáltatásokkal. Ehhez többféle módon és többféle eszközzel is hozzáférhetünk, mivel az alkalmazás hibákat egy log szerver, a környezeti problémákat egy monitorozó szerver tudja megfelelően megmutatni számunkra.

- **Zabbix:** 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 microservice architektúrákhoz lett kialakítva, de egy elég általános eszköz, hogy felhasználható legyen ilyen célra is.

- **Kibana + LogStash:** 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:** 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 Cloud-okban könnyen és hatékonyan felhasználhatóvá.
- **Cronitor:** Ez a monitorozó eszköz mikro-szolgáltatások és cron job-ok megfigyelésére 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 Service Registry képzésében is.
- **Ruxit:** Egy Cloud-osított 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 kapott riasztások szerint akarunk skálázni.

4. fejezet

Kommunikációs módszerek

A szolgáltatások közötti kommunikáció nincs lekötve de jellemző a REST-es API, vagy a webservice-re jellemző XML alapú kommunikáció[10].

4.1. Technológiák

A kommunikáció megtervezéséhez egy jó leírást olvashatunk az Nginx egyik cikkében[9]. Ez a cikk leírja, fontos előre eltervezni, hogy a szolgáltatások egyszerre több másik szolgáltatással is kommunikálnak vagy nem, 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.

4.1.1. REST (HTTP/JSON)[4]:

A RESTful 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. Mikro szolgáltatások esetén az aszinkron változat használata az előnyösebb[13], mivel ekkor több kérést is ki tudunk szolgálni egyszerre, és a szolgáltatásoknak nem kell várniuk a szinkron üzenetek kiszolgálására.

4.1.2. SOAP (HTTP/XML)[12]:

A szolgáltatás alapú architektúrákban nagyon népszerű, 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.

4.1.3. Socket (TCP)[11]:

A Socket 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 mikro szolgáltatásokra épülő struktúrában is, azonban általános esetben nem igazán használható kommunikációs eszköz.

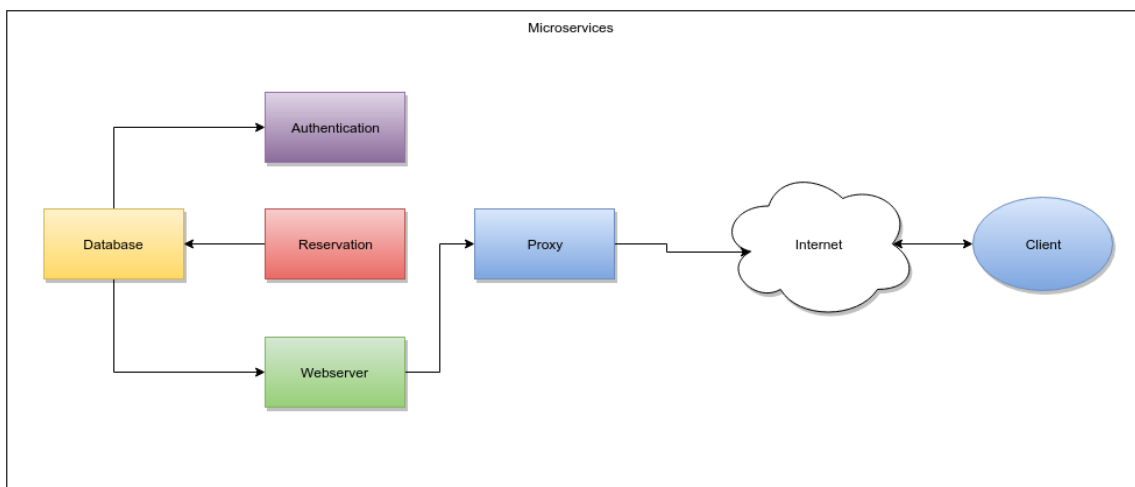
4.2. Interfészek

A korábban említett Nginx-es cikk[9] 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 (visszafele 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.

5. fejezet

Minta alkalmazás terve



5.1. ábra. Microservices

5.1. Alkalmazás leírás:

Az alkalmazás, amin ketesztül a mikró szolgáltatások működését bemutatom, egy könyvesbolt webáruháza lesz, ami rendelkezik egy webes felülettel. A felhasználó be tud jelentkezni a felületre, és tud könyveket vásárolni magának.

5.2. 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 autentikálása
- **Böngészés:** Felhasználó láthatja mi van a raktáron
- **Vásárlás:** Felhasználó valamit a saját nevére ír

Ezekből a feladatokból a következő szolgáltatásokat lehet elkészíteni:

- **Felület kiszolgálása:** Egy web kiszolgáló alkalmazása, amin keresztül elvégezhetők a különböző műveletek, mint a bejelentkezés, vagy vásárlás. Ez a felület magába foglalja a böngészést lehetővé tevő szolgáltatást is.
- **Authentikációs szolgáltatás:** A bejelentkezni szándékozó felhasználó adatait ellenőrzi, és hibás bejelentkezés esetén hibát dob.
- **Vásárlási szolgáltatás:** A böngészés közben kiválasztott könyveket lefoglalja a raktári készletből.
- **Adatbázis szolgáltatás:** Ez a szolgáltatás tartalmazza a raktár tartalmát, a vásárlási naplót, és a bejelentkezési adatokat.
- **Terhelés elosztó szolgáltatás:** Ez a szolgáltatás a skálázhatóságot segíti, és egy egységes interfész kialakításában segít.

6. fejezet

Minta alkalmazás elkészítése

A megvalósításhoz felhasznált technológiák a szolgáltatások felismerésében különböztek. Kipróbáltam a korábbi félévek során használt *Consul*-t, amivel dinamikusan esemény vezérelten képesek kommunikálni a szolgáltatások. Másodszorra a *Docker* konténerekbe beépített módszert használtam fel, amivel könnyen, már indítás közben felismerik egymást a szolgáltatások. Harmadszorra pedig egy gyakran használt service registry-t használtam, az *Apache Zookeepert*.

6.1. Megvalósítás Docker konténerekkel:

Az egyszerűség kedvéért, és a koncepció kipróbálásához Docker konténereket használtam, mivel ezek könnyedén elindíthatók, kkonfigurálhatók, és helyi gépen is lehetővé teszik egy komplex architektúra kipróbálását.

A mikro szolgáltatások 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 szerveret. Ezt a konténert egy inicializáló szkripttel egészítettem ki, ami elkészítette az alap adatbázist.
- **Terhelés elosztó:** A terhelés elosztást *HAProxy*-val oldottam meg, amit egy Ubuntu konténerre alapoztam. Létezik egy olyan Docker konténer, ami kifejezetten HAProxy mikro szolgáltatásnak 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 webkiszolgáló van. Ezt kiegészítettem *PHP*-val, és néhány szkripttel, ami kiszolgálja a kéréseket.
- **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.

- **Vásárlás:** Centos konténer alapú környezet, amiben *Java* lett telepítve, és egy webes REST API-n keresztül érhetjük el a szolgáltatását.

6.2. Kapcsolatok építése Consul-al:

A Consul alkalmazást korábbi félév folyamán használtam már, teljesítmény mérések futtatására, így megpróbáltam átültetni a logikát a jelenlegi mikro szolgáltatásokat biztosító architektúrába. A gondot az okozta, hogy a Consul alkalmazásnak szükséges egy fix pont, és ehhez találnom kellett egy olyan elemet, ami mindenképpen elsőnek indul el. Ez az elem lett a proxy szerver, ami összefogja az elemeket. A korábbi félévben használt kód megfelelő volt számomra, mivel nagyon hasonló minta alkalmazást használtam a teljesítmény mérésekhez is.

Ez a megoldás nem elég elosztott a mikro szolgáltatások tekintetében, azonban egy elég hatékony, és könnyen implementálható megoldás. A mikro szolgáltatásokra épülő architektúrában jellemzően van egy Service Registry elem, ami lehetővé teszi a szolgáltatások nyilvántartását, és ez biztosíthatja a kapcsolatot is. A Consul ebben a kialakításban pontosan így is működött, viszont található olyan eszköz amit kifejezetten a szolgáltatásokhoz találtak ki. Ez lenne például az Apache Zookeeper.

6.3. 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 elértem, 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. Mivel ez egy mikro szolgáltatásokra épülő architektúránál közel sem ideális, így ez csupán fejlesztési, és reprezentatív jelleggel használható. (Mivel a labor célja, hogy bemutassam az architektúra működését, ezért ez megfelelő számomra)

6.4. Kapcsolatok építése Zookeeper-el:

TODO: Kipróbálni a Zookepert

6.5. Automatizálás:

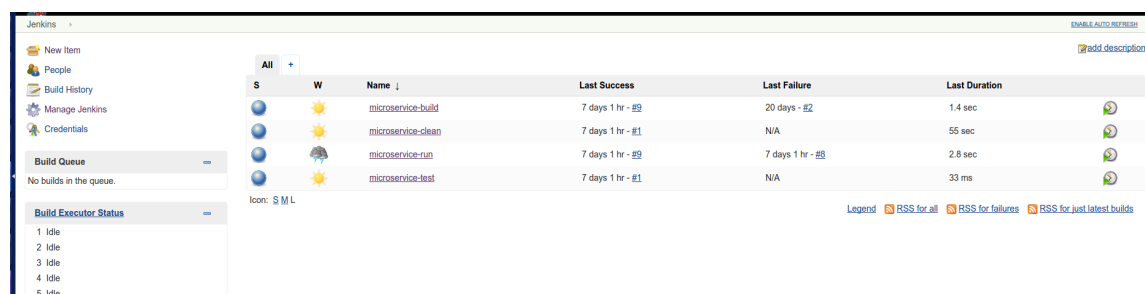
A mikro szolgáltatások architektúrájában a következő feladatokat lehet automatizálni:

1. **Teszt alkalmazás build-elése:** Gyakran van szükség a szolgáltatást futtató fájlok és egyéb tartalmak fordítására (C, Java, bináris kép fájlok fordítása) , 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 céljaunkra. (Esetünkben kialakítunk egy könyvesboltot)
3. **Teszt architektúra konfigurálása:** Van, hogy telepítés után nem elég magára hagyni a rendszert, és használni a szolgáltatásokat, de szükséges különböző beállításokat végrehajtani, hogy a megfelelő módon működjön az alkalmazás. Ilyen feladat lehet a szolgáltatásokhoz tartozó registry frissítése, vagy a futtató gépeken a rendelkezésre állás javítása, és egyéb biztonsági mechanizmusok alkalmazása. (Esetemben a Zookeeper felkonfigurálása lesz a feladat.)
4. **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. (Esetemben unit teszteket fogok futtatni)

6.6. Jenkins Job-ok fejlesztése:

Az architektúra összeállításának automatizálását a Jenkins folytonos integrációt támogató eszközt használtam, aminek segítségével egyszerű feladatok létrehozásával, és bash parancsok futtatásával képes voltam fellőni egy teszt környezetet.

A létrehozott feladatok (job-ok):



| S | W | Name | Last Success | Last Failure | Last Duration |
|---|---|--------------------|------------------|------------------|---------------|
| | | microservice-build | 7 days 1 hr - #9 | 20 days - #2 | 1.4 sec |
| | | microservice-clean | 7 days 1 hr - #1 | N/A | 56 sec |
| | | microservice-run | 7 days 1 hr - #9 | 7 days 1 hr - #9 | 2.8 sec |
| | | microservice-test | 7 days 1 hr - #1 | N/A | 33 ms |

6.1. ábra. Jenkins job-ok

- **bookstore-build:** Ennek a feladata a forrásfájlok és a Docker konténerek felkészítése. Miután a job végzett, a teljes infrastruktúra elkészíthető Docker konténerekből.

Console Output

```
Started by user anonymous
Building in workspace /var/lib/jenkins/jobs/microservice-build/workspace
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://github.com/borlayda/dipterv2016-microservice.git # timeout=10
Fetching upstream changes from https://github.com/borlayda/dipterv2016-microservice.git
> git --version # timeout=10
using .gitcredentials to set credentials
> git config --local credential.username borlayda # timeout=10
> git config --local credential.helper store --file=/tmp/git6588968186272459217.credentials # timeout=10
> git -c core.askpass=true fetch --tags --progress https://github.com/borlayda/dipterv2016-microservice.git +refs/heads/*:refs/remotes/origin/*
> git config --local --remove-section credential # timeout=10
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
> git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision 8b0902f49c17ca6b49a5537411e53712b9a87754 (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 8b0902f49c17ca6b49a5537411e53712b9a87754
> git rev-list 8b0902f49c17ca6b49a5537411e53712b9a87754 # timeout=10
[workspace] $ /bin/sh -xe /tmp/hudson5338336264308612708.sh
+ /var/lib/jenkins/jobs/microservice-build/workspace/build_docker.sh
Create database for bookstore ...
Create webserver for bookstore ...
Create proxy for bookstore ...
Create reserve for bookstore ...
Create auth for bookstore ...
Microservices has been created!
Triggering projects: microservice-run
Finished: SUCCESS
```

6.2. ábra. Microservice build

- **bookstore-run:** Ennek a job-nak a feladata a Docker konténerek indítása, a szolgáltatások inicializálása.
- **bookstore-clean:** Ennek a job-nak a feladata, hogy a környezet ki legyen tisztítva, és ne maradjon a tesztek után semmilyen Docker konténer, vagy fordított fájl a munkaterületen (workspace).
- **bookstore-test:** Unit tesztek futtatása a feladata, de ide tartoznának a funkció és integrációs tesztek is, illetve a teljesítmény tesztek.

A Jenkins lehetővé teszi, hogy az egyes feladatok alfeladatokat hívjanak, és egy komplex hierarchiát hozzanak létre. Ha bonyolultabb vagy részletesebb felbontást szeretnének, csak fel kell vennem pár újabb feladatot, és meg kell hívnom egy feladatból a többi.

6.7. Egyéb minta alkalmazások:

KanBan board minta:

<https://github.com/eventuate-examples/es-kanban-board>

Archivematica minta:

<https://www.archivematica.org/en/>

Console Output

```
Started by upstream project "microservice-build" build number 9
originally caused by:
  Started by user anonymous
Building in workspace /var/lib/jenkins/jobs/microservice-run/workspace
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://github.com/borlayda/dipterv2016-microservice.git # timeout=10
Fetching upstream changes from https://github.com/borlayda/dipterv2016-microservice.git
> git --version # timeout=10
using gitcredentials to set credentials
> git config --local credential.username borlayda # timeout=10
> git config --local credential.helper store --file=/tmp/git3575049518591043388.credentials # timeout=10
> git -c core.askpass=true fetch --tags --progress https://github.com/borlayda/dipterv2016-microservice.git +refs/heads/*:refs/remotes/origin/*
> git config --local --remove-section credential # timeout=10
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision 8b0902f49c17ca6b49a5537411e53712b9a87754 (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 8b0902f49c17ca6b49a5537411e53712b9a87754
> git rev-list 8b0902f49c17ca6b49a5537411e53712b9a87754 # timeout=10
[workspace] $ /bin/sh -xe /tmp/hudson6453331697224313406.sh
+ /var/lib/jenkins/jobs/microservice-run/workspace/run_containers.sh
283c4e5e16ee648d9cdb445d2aa47f7798763f96ea2558aaabde25226d00b01e
Start database service ...
ebd626be8fc2b85152ff2efe625de426e686a9e15e94e8019df775831c8c4316
Start webserver service ...
d0456425fc8add51e632d1970085911b3ccf9b6730ce52309aa8ff67323253
Start reserve service ...
64cd3e1afbb54f2da72480574a307c43219259da1386985e95df7a38a6e4f97b
Start auth service ...
49712dee7a28bc5df1c421ca10d06c52d0f002f9091d78371a57302531284672
Start proxy service ...
cfd6cc51805d437c17449e4cdc636325e0db3ce7e1b3b3d4700311116caa559
Triggering projects: microservice-test
Finished: SUCCESS
```

6.3. ábra. Microservice run

Console Output

```
Started by upstream project "microservice-test" build number 1
originally caused by:
  Started by upstream project "microservice-run" build number 9
originally caused by:
  Started by upstream project "microservice-build" build number 9
originally caused by:
  Started by user anonymous
Building in workspace /var/lib/jenkins/jobs/microservice-clean/workspace
Cloning the remote Git repository
Cloning repository https://github.com/borlayda/dipterv2016-microservice.git
> git init /var/lib/jenkins/jobs/microservice-clean/workspace # timeout=10
Fetching upstream changes from https://github.com/borlayda/dipterv2016-microservice.git
> git --version # timeout=10
using gitcredentials to set credentials
> git config --local credential.username borlayda # timeout=10
> git config --local credential.helper store --file=/tmp/git4178405747046778953.credentials # timeout=10
> git -c core.askpass=true fetch --tags --progress https://github.com/borlayda/dipterv2016-microservice.git +refs/heads/*:refs/remotes/origin/*
> git config --local --remove-section credential # timeout=10
> git config remote.origin.url https://github.com/borlayda/dipterv2016-microservice.git # timeout=10
> git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
> git config remote.origin.url https://github.com/borlayda/dipterv2016-microservice.git # timeout=10
Fetching upstream changes from https://github.com/borlayda/dipterv2016-microservice.git
using gitcredentials to set credentials
> git config --local credential.username borlayda # timeout=10
> git config --local credential.helper store --file=/tmp/git5621869497934903534.credentials # timeout=10
> git -c core.askpass=true fetch --tags --progress https://github.com/borlayda/dipterv2016-microservice.git +refs/heads/*:refs/remotes/origin/*
> git config --local --remove-section credential # timeout=10
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
Checking out Revision 8b0902f49c17ca6b49a5537411e53712b9a87754 (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 8b0902f49c17ca6b49a5537411e53712b9a87754
First time build. Skipping changelog.
[workspace] $ /bin/sh -xe /tmp/hudson4298315446159827323.sh
+ /var/lib/jenkins/jobs/microservice-clean/workspace/clean_docker.sh
cfd6cc51805d
49712dee7a28
64cd3e1afbb5
d0456425fc8a
ebd626be8fc2
cfd6cc51805d
49712dee7a28
```

6.4. ábra. Microservice clean

7. fejezet

Összefoglaló

A diplomaterv összefoglaló fejezete.

Táblázatok jegyzéke

Ábrák jegyzéke

| | |
|-----------------------------------|----|
| 1.1. Scaling Cube | 7 |
| 5.1. Microservices | 21 |
| 6.1. Jenkins job-ok | 25 |
| 6.2. Microservice build | 26 |
| 6.3. Microservice run | 27 |
| 6.4. Microservice clean | 27 |

Irodalomjegyzék

- [1] David Chou. Using events in highly distributed architectures. *The Architecture Journal*, October 2008.
- [2] Manfréd Sneps-Sneppé Dimirty Namiot. On mirco-services architecture. <http://cyberleninka.ru/article/n/on-micro-services-architecture>.
- [3] Peter Van Garderen. Archivematica: 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.
- [4] 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.
- [5] Chris Richardson (Kong). Pattern: Microservices Architecture. <http://microservices.io/patterns/microservices.html>.
- [6] Microsoft. Pipes and Filters Pattern. <https://msdn.microsoft.com/en-us/library/dn568100.aspx>.
- [7] Microsoft. Publish/Subscribe. <https://msdn.microsoft.com/en-us/library/ff649664.aspx>.
- [8] Chris Richardson. The Scale Cube. <http://microservices.io/articles/scalecube.html>.
- [9] Chris Richardson. Building microservices: Inter-process communication in a microservices architecture. <https://www.nginx.com/blog/building-microservices-inter-process-communication/>, July 2015.
- [10] Puja Padiya Snehal Mumbaikar. Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3, May 2013.
- [11] tutorialspoint. What is a socket? http://www.tutorialspoint.com/unix_sockets/what_is_socket.htm.
- [12] w3schools. Xml soap. http://www.w3schools.com/xml/xml_soap.asp.
- [13] 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.

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

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
```