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 10.

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. május 10.

Borlay Dániel
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon Markdown leírónyelven készült, Pandoc rendszerrel fordítható le TEX Live vagy MiKTEX LATEX disztribúciókkal.

Abstract

This document is a skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. The skeleton was implemented in Markdown and can be compiled with Pandoc, using the TEX Live and or the MiKTEX LATEX compiler.

1. fejezet

Bevezetés

A bevezető tartalmazza a diplomaterv-kiírás elemzését, történelmi előzményeit, a feladat indokoltságát (a motiváció leírását), az eddigi megoldásokat, és ennek tükrében a hallgató megoldásának összefoglalását.

A bevezető szokás szerint a diplomaterv felépítésével záródik, azaz annak rövid leírásával, hogy melyik fejezet mivel foglalkozik.

2. fejezet

Mikro szolgáltatások

2.0.1. Definíció:

Nem találtam konkrét definicót, de 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.

2.0.2. Technológiáról:

A mikro szolgáltatás architektúra kiépítéséhez sokféle szétválasztási módot használnak, amik közül van olyan amit a tervezési folyamat közben felmerülő főneveket, vagy igéket használják fel, de abban megegyeznek, hogy a funkcionlaitást bontják fel. Ezzekkel az integrációval foglalkozó részben olvashatunk bővebben.

A mikro szolgáltatások tervezése során a következő szempontok szerint szokták megtervezni a rendszert:

- Milyen szolgáltatásokat tud nyújtani a rendszer
- 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)
- A kommunikáció a felhasználóval
- Egy központi szolgáltatáson keresztül, ami a többivel kommunikál
- Add-hoc minden szolgáltatás külön hívható

Ezekkel a lépéssekkel meg lehet alapozni, hogy az álltalunk készítendő rendszer hogyan is lesz kialakítva, és milyen paraméterek mentén lesz felvágva. A választást segíti a témában elterjedt fogalom, a scaling cude[1], 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.

Az adat menti felbontás annyit tesz, hogy a szolgáltatásokat annak megfelelően bontjuk fel, hogy az egyes szolgáltatások csak adatbázissal, vagy csak web kiszolgálással foglalkozzanak, vagy csak a felhasználói adatok esetleg a tanulók jegyeit felügyelik. Ez a mérce a mikro szolgáltatás architektúrák esetén nem annyira fontos, mivel a szolgáltatásoknak erőforrásaikat tekintve is el kell különülniük, így nem éri meg erőforrások vagy adat mentén vágni.

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.

A harmadik módszer arra tér ki, hogy hogyan lehet egy architektúrát felosztani, hogy skálázható legyen. Itt a klónozhatóság, avagy az egymás melleti kiszálgálás motivál. Ez a mircro-service-eknél kell, hogy teljesüljön, mivel adott esetben a load balancer alatt tudnunk kell definiálni több példányt is egy szolgáltatásból.

2.1. Architektúrális minták

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, azonban nem lehet élesen elkülöníteni, mivel valamilyen csatolási módszerre szükség van, ami nem specifikus a mikro szolgáltatás-ek esetén, viszont más architektúrális mintákra jellemző.

Ilyen a Pipes and fileter architektúrális minta [2], aminek a lényege, hogy a funkciókra bontott architektúrát az elérni kívánt végeredmény érdekében különböző módokon összekötünk. 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.

Egy másik elosztott rendszerekhez kitallált minta a subscriber/publisher[3], amely arra alapszik, hogy egy szolgáltatásnak szüksége van valamilyan 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.

2.1.1. Előnyök-hátrányok:

A mirco-service architektúrák a monolitikus architektúra ellentetjei, melyben az erőforrások központilag vannak jezelve, 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.

2.1.2. Kommunikáció:

A szolgáltatások közötti kommunikáció nincs lekötve de jellemző a REST-es API, vagy a webservice-re jellemző XML alapú kommunikáció. Minden szolgáltatához tartozik egy önálló interfész, amin keresztül a többi szolgáltatás kommunikálhat vele, és minden funkcióját el lehet érni. Ennek az interfésznek olyannak kell lennie, hogy az implementáció szabadon változtatható legyen, és ne kelljen más szolgáltatásokat megváltoztatni, ha a saját szolgáltatásunkat változtatjuk. Ez segíti a több csapattal való munkát, és lehetővé teszi hogy teljesen függetlenül létezzenek a szolgáltatások.

2.1.3. Példák:

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 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.

Mintapéldák: http://eventuate.io/exampleapps.html

3. fejezet

Technológiai áttekintés

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

A következő feladatokre 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 registy) * Hogyan figyeljük meg a rendszert működés közben (monitorozás, loggolás)

3.0.4. Telepítés:

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)

3.0.5. Környezet felderítés:

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.

3.0.6. Integrációs keretrendszer:

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 változtatáról, vagy hajtson végre ő maga is változtatást.

- **Puppet**: Olyan nyilt 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**: 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, 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.

3.0.7. Skálázás:

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 beszervezni, 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, 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 microservice architektúránkat, akkor erős eszköz lehet számunkra.

3.0.8. Load balancing:

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 érjünk el, milyen IP címek mely szolgáltatásokhoz tartoznak, illetve

round robin 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**: Az Nginx egy nyilt 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 terhelés elosztásra, és alkalmazás telepítésre. Nem teljesen a proxy szerver szerepét váltja ki, de képes elvégezni azt.

3.0.9. Virtualizálás:

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 eszköt. 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 mást. A KVM egy nyilt forrűskódú 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 legcélszerűbb valamilyen Cloud-os környezetben létrehozni, mivel egy ilyen környezetnek definiciója 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 tartalmaznak 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.

3.0.10. Service registy:

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

- **Euraka**: 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 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 nyilt 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 conzulhoz hasonlóan működik, és a feladata is ugyan az.

3.0.11. 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 mikró-szolgáltatások 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 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 kapot riasztások szerint akarunk skálázni.

4. fejezet

Előnyök

4.1. Skálázhatóság

5. fejezet

Hátrányok

5.1. Bonyolult fejlesztés

6. fejezet

Összehasonlítva a monolitikus architektúrával

A mirco-service architektúrák a monolitikus architektúra ellentetjei, melyben az erőforrások központilag vannak jezelve, 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 micro-service 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.

7. fejezet

Összefoglaló

A diplomaterv összefoglaló fejezete.

Táblázatok jegyzéke

Ábrák jegyzéke

A. függelék

Függelék

A.1. Válasz az „Élet, a világmindenség, meg minden” kérdésére

A Pitagorasz-tételből levezetve

$$c^2 = a^2 + b^2 = 42.$$

A Faraday-indukciós törvényből levezetve

$$\mathrm{rot}E = -\frac{dB}{dt} \quad \longrightarrow \quad U_i = \oint\limits_{\mathbf{L}} \mathbf{Edl} = -\frac{d}{dt}\int\limits_{A} \mathbf{Bda} = 42.$$