|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | **NEUMANN JÁNOS INOFRMATIKAI KAR** | |  |
| **SZAKDOLGOZAT** | | | | |
| **OE-NIK  2017** | Hallgató neve:  Hallgató törzskönyvi száma: | | **Kostenszky Kálmán Ákos T/004328/FI12904/N** | |

Tartalomjegyzék

[Tartalomjegyzék 1](#_Toc482471421)

[1 Bevezetés 2](#_Toc482471422)

[2 A szoftveripar változó igényei 4](#_Toc482471423)

[2.1 Conway törvénye 7](#_Toc482471424)

[2.1.1 Agilis szoftverfejlesztési módszertanok 7](#_Toc482471425)

[2.1.2 DevOps kultúra. 8](#_Toc482471426)

[2.2 A felhő technológiák térnyerése 10](#_Toc482471427)

[2.3 Összefoglalás 13](#_Toc482471428)

[3 A vizsgált architektúrák bemutatása 14](#_Toc482471429)

[3.1 A monolitikus szoftverarchitektúra bemutatása 14](#_Toc482471430)

[3.1.1 Monolitikus architektúra megvalósítása java ee platformon 16](#_Toc482471431)

[3.2 A microservice szoftverarchitektúra bemutatása 20](#_Toc482471432)

[3.2.1 A microservice szoftverarchitektúra „elődei” 20](#_Toc482471433)

[3.2.2 A microservice szoftverarchitektúra jellemzői 22](#_Toc482471434)

[3.3 A két vizsgált architektúra összehasonlítása 27](#_Toc482471435)

[4 Java EE alkalmazás a microservice világában, felhasználható technológiák 30](#_Toc482471436)

[4.1 Pehelysúlyú alkalmazásszerverek 31](#_Toc482471437)

[4.2 Keretrendszerek 31](#_Toc482471438)

[4.3 Srping Keretrendszer 32](#_Toc482471439)

[5 Irodalomjegyzék 34](#_Toc482471440)

# Bevezetés

A modern szoftver-életciklus módszertanok mellett a szoftverkrízisből kivezető út egyik fontos tényezője volt a szoftvertervezési minták elterjedése és széleskörű alkalmazása. Ezek a minták az implementáció különböző szintjein nyújtottak általános és könnyen újra hasznosítható „legjobb megoldásokat” a fejlesztőknek. Ezek a „legjobb megoldások” olyan példákat takarnak, melyek betartják az adott paradigmára (például az objektum orientált paradigmára) jellemző irányelveket, könnyen értelmezhetőek és széleskörben ismertek. A tervezési minták a szoftvertervezés magasabb, rendszerszintű kontextusában is jelen vannak. Ezek a minták természetükben eltérnek a korábban említett tervezési mintáktól: míg az előbbiek sok esetben pszeudokódok segítségével leírhatóak és egy adott nyelvben könnyen megvalósíthatóak, az utóbbiak inkább tekinthetőek tervezési szempontok jól definiált halmazának. Ezen szempontok mentén szerveződnek a tervezés során a szoftver komponensei, moduljai, illetve ezek egymás közötti viszonyai. Ezek az architekturális minták tehát azon tervezési szempontok és megközelítések összessége, melyek alapjaiban határozzák meg a leendő szoftver főbb karakterisztikáit (agilitás, bővíthetőség, skálázhatóság, hibatűrés, komplexitás) és viselkedését [21]

A Java Enterprise Edition platform 1998-as megjelenésével és elterjedésével a monolitikus architektúrájú – vagy a Java világában n-rétegűnek nevezett – szoftverrendszerek de-facto sztenderdé váltak az enterprise (nagyvállalati, üzleti) alkalmazás fejlesztés világában. Ez a tervezési megközelítés az elmúlt bő másfél évtizedben nagy sikernek örvendett és számos más enterprise fejlesztői platform ezt a szoftver architektúrát támogatja. Az utóbbi években ez a hegemónia megtörni látszik és egy másik tervezési megközelítés alkalmazása is megjelent a szoftverfejlesztők körében: ezt a megközelítést microservice architekturának (MSA) hívja a szakirodalom. [12]

Döntően két tényező járult hozzá ahhoz, hogy ez a megközelítés elterjedt. Egyrészt az üzleti igényeknek folyamatos változása, amely egyre nagyobb versenyre készteti a piaci szereplőket, másrészt széleskörben elérhetővé váltak olyan technológiák, melyek döntően megváltoztathatják egy szervezet IT infrastrukturáját. Ezekre a jelenségekre a dolgozatom következő fejezetében részletesen kitérek, mivel ezek hatása nemcsak az IT szervezetek felépítésére és működésére, hanem az általuk előállított szoftverek felépítésére is visszahatnak. Így a későbbi fejezetekben könnyebben rá tudok világítani, hogy mik azok az elvárások, amelyeknek az egyik vagy másik architektúra jobban megfelel.

A szakirodalomi kutatásom a két architekturális minta összehasonlító feltárása fog irányulni.

Fontosnak tartom megjegyezni, hogy egyik architekturális minta sem a Java EE sajátja, sőt mint látni fogjuk a fejlődésük történetében a Java EE platform teljesen eltérő szerepet játszik. Így azt is célom világossá tenni az szakirodalomban végzett kutatás végén, hogy mik a Java platform azon hiányosságai és jellemzői, amelyek kihívás elé állíthatják a fejlesztőket, ha MSA jellegű rendszert implementálnak a Java EE platformon. Ez utóbbi probléma miatt célom a két architektúrát a Java EE ökoszisztéma kontextusán keresztül vizsgálni. Tehát az irodalomkutatás során olyan technológiákat fogok vizsgálni, majd az implementáció során felhasználnálni, melyek vagy részei a platformnak vagy hasonlóságuk miatt könnenyen implementálhatóak.

A dolgozat rendszerterv és implementációs fejezeteinek alapjául egy saját fejlesztésű irodai ügyviteli alkalmazás szolgál, melyet Java EE 7 platformon készítettem el a Szoftverfejlesztés Java EE platformon tantárgy keretében. Ezt az alkalmazást fogom felbontani és MSA alapokon felépíteni. Az kiindulásul szolgáló alkalmazást részletesen a Rendszerterv fejezetben kívánom bemutatni.

A rendszerterv készítése két lépésből fog állni: egyrészt dekomponálom a monolitikus alkalmazást, ezzel meghatározva a microserviceketm másrészt megtervezem a rendszer „külső” architektúráját [19], amely lényegében a szolgáltatások közötti réteget jelenti.

A dolgozat végén funcikonális összevetést kívánok adni a két megközelítés közötti különbségről, az egymáshoz viszonyított előnyeiökről és hátrányaokról, teljesítményben nyújtott különbségeikről, alkalmazhatóságuk körülményeiről.

# A szoftveripar változó igényei

Bár szakdolgozatom szakirodalmi kutatásának a fókusza a két architekturális minta funkcionális vizsgálata, ami során mérnöki megközelítést kívánok alkalmazni, fontosnak tartom, hogy előtte feltárjam azokat a motivációkat, amik új megközelítések kialakításra ösztönözték a szoftvertervező mérnököket.

Ezeket megisermerve könnyebben képet tudunk majd alkotni az MSA tervezési mintának az evolúciójáról és szerepéről, továbbá könnyebb lesz rávilágítani arra is, hogy a két tervezési megközelítés milyen eltérő jellegű kihívásokra próbál megoldást nyújtani az üzleti alkalmazás fejelsztés során.

Az üzleti vállakozások változásának egyik fő motivációja a kínált termékek és szolgáltatásaok iránt mutatott kereslet minőségi megváltozása. Az piaci világ digitalizációja pedig arra kényszeríti annak szereplőit, hogy a szervezetük IT tevékenységét újraértelmezzék és átstrukturálják.

Az olcsó telekommunikációs eszközök, az internet elterjedése a szolgáltató szektor és az ipar szinte minden területén egy új csatornát nyitott a keresleti oldal felé.

Ezt megelőzően csak néhány specifikus vállalat fejtette ki elsődleges üzleti tevékenyéségét az IT-n keresztül nyújtott szolgáltatásaival és termékeivel. Ebből kifolyólag a legtöbb vállalkozás esetében az IT részleg egy belső szervezet volt, mely elsődlegesen a vállalkozás üzleti és adminisztrációs tevékenységet támogatta. Mint a legtöbb ilyen támogató vállalati funkció az IT részleg is elsősorban költséghelyként jelenik meg. Tevékenységét elsősorban a vállalkozáson belül fejti ki, „ügyfeleit” és projektjeik „megrendelőit” a szervezet más osztályáinak igényei adták. Tevékenységükkel szemben támasztott elsődleges követelmény a költséghatékonyság, melyet folyamataik és technológiai platformjuk lehető legszélesebb körű sztenderdizálásával próbálnak meg elérni [10]. Az így megvalósított projektjek jellemzője, hogy általában hosszútávú üzemeltetésre tervezik a projekt produktumot, az üzemeltetési költségek minimalizálása mellett. Ezek a célok könnyen elérhetőnek tűnnek, hiszen mind a szoftvert előállító és üzemeltető szervezeti egység, mind az azt felhasználók köre ugyanazon szervezet alá tartozik, tehát jól szabályozható és kontrol alatt tartható folyamatokról van szó.

Ugyanakkor a nagyfokú sztenderdizálás hátulütője, hogy a szervezet beragadhat egy technológiai platformba, az idő előrehaladtával egyre nehezebbé válik az adatptálódás az új technológiákhoz. [10].

A digitalizáció nyomán kényszerűen fel kellett ismerni, hogy az IT által előállított szoftverek már nem csupán szervezeten belüli célokat szolgálnak. A termékek és szolgáltatások értékesítésének elsőrendű csatornájává váltak, ezáltal maguk is a termékek és szolgáltatások részét képezik.

Példának talán a hazai pénzügyi szektort tudnám említeni, ahol már az ezredforduló elején is nagyméretű, kiterjedt IT részlegek üzemeltek, elsősorban az üzleti tevékenységet támogató. Ez időtájt még az ügyfeleik nagy része a pénzügyeit még személyesen vagy telefonon keresztül – egy ügyintéző segítségével - intézte.

Mára ez a trend teljesen megfordult, a bankok nagy része kedvezményes kondiciókkal ösztönzi ügyfeleit, hogy a bankügyeik lehető legszélesebb körét internetbankon keresztül intézzék. Ezen túlmenően a piaci szereplők azt is felismerték, hogy az online csatornák további kiaknázható lehetőségeket hordoznak magukban. Az ügyfeleik felhasználói szokásai és visszajelzési azonnali visszacsatolást jelentenek számukra. Ezeken a csatornákon előtte nem tapasztalt mennyiségű adat kezdett visszaáramlani az ügyfelektől a szolgáltatók felé [20].

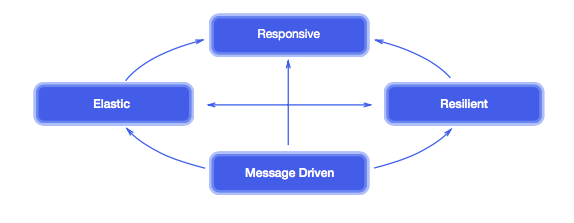
Itt is egy példát hozva, gondoljunk a webáruházakra, amelyek korábbi vásárlásaink alapján, illetve a hozzánk hasonló felhasználók szokásai alapján célzott, személyre szabott ajánlatokkal tudnak megkeresni minket. Az IT így egy újabb értékteremtő folyamat részvé válik.

Az IT által végzett tevékenység minősége tehát versenytényező lett. Az előállított szoftverek fogyasztói pedig új kihívások elé állítják a fejlesztőket. Az ügyfél felhasználói szokásai ugyanis kívül esnek a szervezet formalizált működésén. A felhasználói igények és azok volumene időben és térben gyorsan változik. Ugyanazt a szolgáltatást több különböző eszközön is azonos felhasználói-élmény mellett akarják igénybe venni. Így olyan megoldásokra van szükség, amelyek gyorsan le tudják követni ezeket a változásokat. Emellett törekedni kell arra, hogy lépést tartsanak vagy élen járjanak az újabb igények kielégítésében, az innovációban, illetve, ami a legfontosabb: ha a vásárló elégedetlen a szolgáltatás minőségével vagy túl „ósdinak” találja, akkor új szolgáltatót fog keresni. Az IT szervezeteknek tehát a következő kihívásoknak kell megfelelni:

* Innovatív, de minimum az üzelti igényekre gyorsan reagálni képes folyamatok kialakítása.
* Rövidebb fejlesztési ciklusok mellett állandó minőség biztosítása.
* Kevésbbé formalizált, gyorsan változó üzleti igények kielégítése szoftvereikkel.
* Olyan rendszerek tervezése, melyeknek inherens tulajdonsága a valósidejű, egyszerű skálázhatóság.
* Aktív részvétel a vállalkozás értékteremtő folyamataiban. Újitó megoldások gyors bevezetése és azonnali visszacsatolás.

A fenti követelmények természetesen a szoftverekkel szemben is új követelményeket támaszottak. A 2014-ben kiadott „The Reactive Manifesto”-ban [5] összegyűjtötték azokat a tulajdonságaokat, melyeket egy a fenti igényeket kielégítésére szánt szoftvernek tartalmaznia kell. Ezek a következőek [9]:

* **Reszponzivitás (Responsive):** A reszponzivitást az alkalmazás átlagos reakcióidejével (latency) mérjük. A reakcióidő a kérés-válasz interakció időintervalluma. A reszponzitivitás nem jelent valósidejűséget. A követelmény lényege, hogy az alkalmazás változó terhelés mellett is azonos reakcióidőt tudjon produkálni (quality of service) illetve, hogy a válaszidőt az alkalmazás egyes moduljainak tranziens hibái lényegesen ne befolyásolják (rendelkezésre állás).
* **Rugalmasság (Resilient):** Ahhoz, hogy a reszponzivitás követelményének eleget tudjunk tenni elengedhetelen az alkalmazás magas hibatűrése. Olyan rendszer megalkotására kell törekedni, ahol egy hibás komponens nem befolyásolja jelentősen a teljes rendszer működésést. Az objektum orientált világ „gyenge csatoltság, erős kohézió” elve itt a rendszer moduljainak szintjén jelenik meg. Fontos megjegyeznem, hogy itt nem az a cél, hogy olyan rendszert alkossunk, melynek moduljai fel vannak készítve minden lehetséges hibára, hiszen ez lehetetlen. Az viszont reális cél, hogy ha egy szolgáltatás leáll, akkor az azt felhasználó szolgáltatások erre fel legyenek készülve - pédálul ne várakozzanak feleslegesen a válaszra, hívjanak meg egy alternatív szolgáltatást, használjanak cachelt adatokat, stb.
* **Skálázhatóság (Elastic):** Ezt a kifejezés viszonylag nehezen fordítható magyarra, mivel ez szintén rugalmasságot jelent, azonban a két rugalmasság fogalma merőben eltérő. Előbbi a rendszer saját hibáira adott rugalmas válasz, addig utóbbi a rendszerre kivülről ható terhelésre adott rugalmas válasz. Ez azt jelenti, hogy a rendszert érő hirtelen terhelés növekedés eredményeként ne alakuljanak ki szűk keresztmetszetek, melyek globális teljesítmény romláshoz vezetnek. Az alkalmazásnak és moduljainak tehát dinamikusan skálázhatónak kell lenni. Mint látható ennek a feltételnek a teljesülése is elegedhetetlen a reszponzivitás telejsítéséhez.
* **Üzenet-vezérelt (Message Driven):** A fentieknek egy rendszer csak abban az esetben tud megfelelni, hogy az egyes moduljai közötti lehető leglazább csatolást valósítjuk meg. Ez úgy valósítható meg, hogy a rendszer moduljai egymás közt üzenetekkel kommunikálnak, lehetőség szerint aszinkron módon. Ennek feltételi:
  + A szolgáltatások definiálják explicit módon a kommunikációs protokolljaikat, amik legyenek implementáció-függetlenek.
  + Legyen biztosítva a helyátlászóság.
  + Az aszinkronitás miatt fontos, hogy a modulok vezérlése valóban üzenetalapú legyen. Egy modul csak akkor lesz aktív, ha üzenetet kap, vagy egy korábban feladott üzenetre válasz érkezik.



‑. Ábra: reaktív rendszerek tulajdonságai és viszonyuk, forrás: http://www.reactivemanifesto.org

Arra, hogy ezeknek a követelményeknek az MSA milyen módon tud megfelelni a következő fejezetben térek ki.

## Conway törvénye

Melvin Conway törvénye, mely szinte minden MSA-val foglalkozó irodalomban megtalálható, a következőt mondja ki:

*„Minden szervezet, mely rendszereket tervez elkerülhetlenül olyan rendszert fog alkotni, amelynek struktúrája megegyezik a szervezet kommunikációs struktúrájával.[[1]](#footnote-1)”* [6]

Nem elég tehát meghatározni, hogy milyen szoftvereket és szoftverrendszereket szeretnénk, olyan szervezeti struktúrát kell kialakítani, melyekkel lehetővé válik ezeknek az elkészítése.

A modern szoftverfejlesztési módszertanok megjelenése bőven megelőzi az általam vizsgált témakört, viszont fontos szerepet játszanak az általam tárgyalt témában.

Ezek közül kettőt szeretnék úgy bemutatni, hogy a későbbiekben láthatóvá váljon, milyen aspektusaihoz kapcsolódik az MSA tervezési megközelítés:

### Agilis szoftverfejlesztési módszertanok

Maga az agilis manifesto [4] is megfogalmaz olyan kívánalmakat, amelyek párhuzamba állíthatóak a fentebb megfogalmazott igényekkel. Jelen esetben az a tradicionális és agilis módszertanokat alkalmazó szervezetek felépítése közötti különbségre szeretnék rávilágítani, Conway törvényének megfontolásával.

A tradicionális IT szervezetek fejlesztései általában front-end, back-end és DBA funkciók szerint kerülnek felbontásra, azaz a technológiai platformok szerint. Mint látni fogjuk ez a technológiai funkciók szerinti felosztás jelenik meg a monolitikus rendszereken belül. Ennek két fő velejárója a következő:

* A rétegek teljes vertikumában végighúzódó üzleti funkciókat, minden rétegnek a saját kontextusában kell értelmezni.
* Az így széttagolt csapatok hiába fejlesztik ugyanazt az üzleti igényet a rétegek határán definiálandó interfészek csapatok közötti egyeztetést igényel, így azok sokkal merevebbek lesznek. Tehát egy igény esetén is számos szerződést kell betartani egy módosítás során.

Ezzel szemben az agilis módszertanokra (kifejezetten a scrum) az a jelemző, hogy a fejlesztők a teljes vertikumban kis csapatokban dolgoznak egy üzleti igény megoldásán. Ez sokkal nagyobb rugalmasságot enged meg, hiszen példának okáért egy 3 rétegű architektúrában 4 határ helyett csupán 2 mentén kell a csapatok között egyeztetni. Ezenfelül a teljes vertikumban ugyannak a csapatnak a tagjai értelmezik és modellezik az üzleti igényt.

Bár ez utóbbi nagyobb szabadságfokot eredményez, továbbra is fennáll a probléma, hogy az alkalmazás monolitikus volta miatt számos megszorítást kell betartani: a különböző csapatok ugyanazon a technológiai platforomon és adatmodellen dolgoznak, noha az igények eltérőek lehetnek. Szükség lehet tehát egy olyan tervezési megközelítésre, amiben a csapatok egymás közötti megszorításai a lehető legszélesebb körben feloldhatóak.

### DevOps[[2]](#footnote-2) kultúra.

A DevOps kultúra a szoftverciklusban résztvevő szoftverfejlesztési, IT üzemeltetési (illetve minőségbiztosítási) szervezeti egységeket közösen érinti.

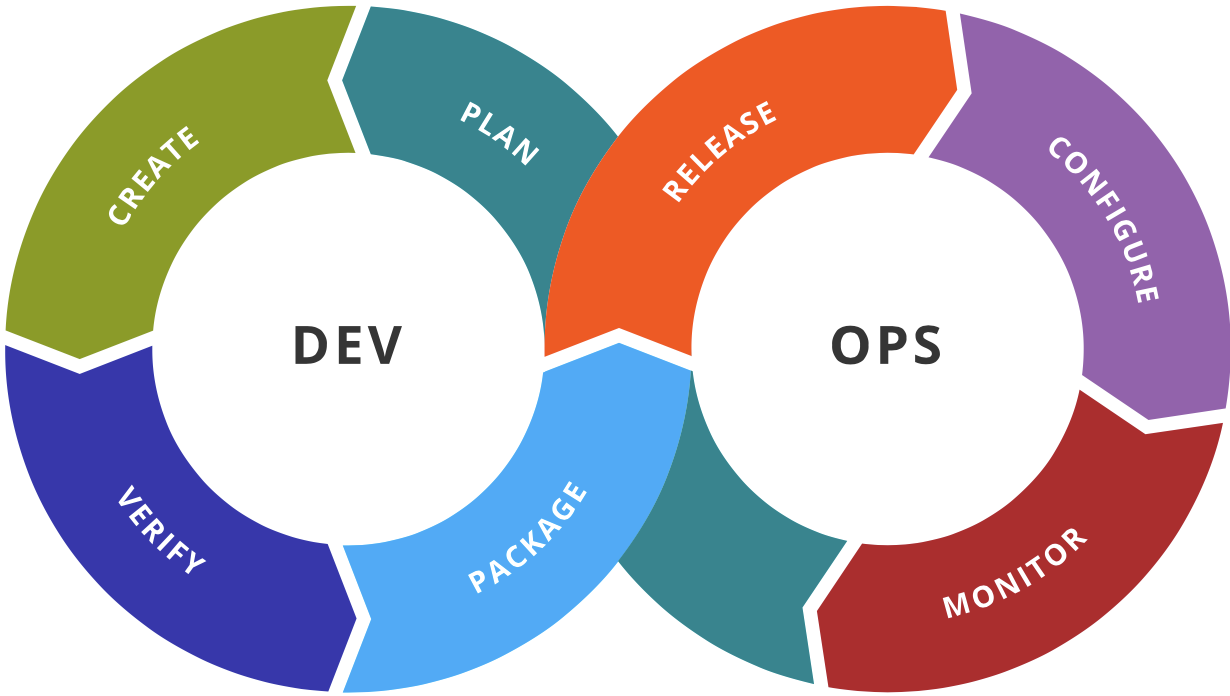
A fejlesztéseket megvalósító szervezeti egység feladatai közé tartozik az üzleti igények felmérése, a tervezés, a szoftverfejlesztés, a tesztelés és a csomagolás.

Az üzemeltetés feladatai közé tartozik a infrastruktúra igények felmérése, karbantartása és bővítése. A telepítésre átadott fejlesztések környezetek közötti szállítása, telepítése, konfigurálása és monitorozása.

Ebből a felosztásból következik, hogy a szoftverprojektek esetében az egyes fejlesztési ciklusok lezárást követően a produktum üzembehelyezése kritikus lépés, aminek kulcseleme a csapatok közötti hatékony együttműködés. Ez sok esetben arra kényszeríti a szervezetet, hogy egy adott projekt esetén akár több fejlesztés elkészültét bevárva, egy csomagban adja át az újabb verziókat. A másik hátránya ennek a megosztásnak az lehet, hogy az információ a csatornán visszafele sem megfelelően áramlik, így a fejlesztő csapat az üzemeltetés során felmerült hibákról nem tud megfelelően tájékozódni.

Ezekre a problémákra a DevOps kultúra kíván megoldást nyújtani. A DevOps olyan koncepciók, gyakorlatok összessége, melynek célja, hogy a szervezet gyorsan és hatékonyan tudjon megoldást szállítani az ügyfeleknek [8].

Míg az agilis módszertanok a szoftverciklusra általánosságban fogalmaznak meg irányelveket és helyes gyakorlatokat, addig a DevOps ezeknek már egy implementációja, ami lehetővé teszi a gyorsabb kibocsátását. [2] A fogalom maga sokkal tágabban értelmezendő, mint a fejlesztői és üzemeltetői gárda közötti határok elmosódása. A fókusz olyan csatornák kialakításán van, amelyekben az egyes fázisok (DevOps toolchain) közötti előrhaladás és visszacsatolás automatikussá tehető. A DevOps kultúra sikerességéhez tehát olyan szoftverarchitektúrára lesz szükségsünk, amivel az szoftver egyes kompononsei egymástól a lehető legnagyobb függetelenséggel tudnak haladni a csatornában. Az ilyen szoftverek moduljai egymástól függetlenül fejleszthetőek, tesztelhetőek, telepíthetőek, konfigurálhatóak és monitorozhatóak- vagyis teljesen önjáró komponensekre van szükség.



2‑2. Ábra: DevOps toolchain, forrás: http://www.wikipedia.org

## A felhő technológiák térnyerése

Az elosztott rendszerek világa már hosszú ideje az egyik fontos területe az informatikának. Az eloszott rendszerek olyan rendszerek, amelyek komponensei hálózatba szervezett erőforrásokon időben és térben elosztottan üzemelnek, és egymás közötti kommunikációjuk során üzenetetek használnak. [15].

Az ilyen rendszerek főbb karakterisztikái:

* konkurens komponensek,
* decentralizált menedzsment,
* lokális hibakezelés.

Ezeket a karakterisztikákat az elosztott rendszerek eltérő paradigmái különbözőképpen és mértékben valósították meg.

A rendszerek közös jellemzője ezidáig az volt, hogy mind az infrastruktúra, mind a szoftver szintjén jelentős többletköltséget jelentett az elosztottság megvalósítása, így nem vált mindenki számára gazdaságosan elérhető opcióvá.

Ebben a szakaszban az eloszott rendszerek infrastrukturális részével, azon belül is a felhő alapú technológiák jellemzőivel szeretnék foglalkozni. Röviden szeretném bemutatni a technológia karakterisztikáit, illetve azt, hogy milyen lehetőségeket adott a modern üzleti alkalmazás a fejlesztés világában.

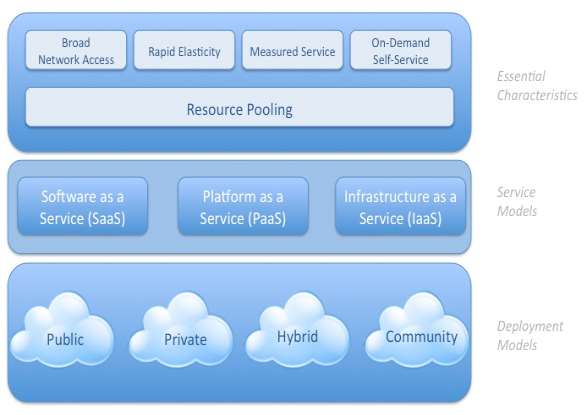
Az amerikai egyesült államokbeli Natoinal Institute of Standards and Technology a következőképpen definiálja a felhő alapú számítási modellt:

*„A felhő számítási modell bárhonnan (elhelyezkedéstől függetlenül) lehetővé teszi a megosztott, konfigurálható számítási erőforrások hálozaton keresztüli, igény szerinti hozzáférését. Az erőforrások gyorsan hozzáférhetőek és visszaadhatóak a szolgáltató minimális beavatkozása mellett. Ennek a modellnek öt kulcsfontosságú karakterisztikája, három szolgáltatási modellje és négy telepítési modellje van.”[[3]](#footnote-3)* [15]

Szabad értelmezésben egy olyan modellről van szó, ahol:

* A számítási erőforrások gyorsan hozzáférhetőek és elengedhetőek.
* Az erőforrások absztraktan értelmezhetőek és rugalmasan konfigurálhatóak.
* A hozzáférés nem igényel különösebb együttműködést a felhasználó és a szolgáltató között.
* A számítási erőforrások helyátlátszó módon bárhonnan elérhetőek a hálózaton.

A definicióban említett 5 karakterisztika, 3 szolgáltatás modell és 4 telepítési modell az alábbi ábrán látható.



‑. Ábra Cloud computing model aspektusai, forrás: https://www.nist.gov/

Dolgozatom szempontjából az öt karaktersztika a kiemelendő. A karakterisztikák azokat a jellemzőket foglalják össze, amellyel a felhő infrastruktúrának bírnia kell függetlenül attól, hogy milyen telepítési modellt alkalmaz, illetve, szolgáltatás modell mely szintjét vizsgáljuk.

* **Széleskörű hálózati hozzáférés (Broad Network Access)**: A felhasználó a számítási kapacitáshoz sztenderd mechanizmusokon keresztül, hálózaton át hozzá tud férni függetelnül attól, hogy milyen kliens eszközt használ (tablet, laptop, pc, ...).
* **Gyors skálázhatóság (Rapid Elasticity):** A számítási kapacitások rugalmasan hozzáférhetőek és elengedhetőek, lehetőség szerint automatizált módon, a számítási igény változásával összhangban.
* **Mérhető szolgáltatás (Measured Service):** Az erőforrás felhasználás felhasználónként mérhető, optimalizálható a felhasználó és a szolgáltató számára egyaránt.
* **Igény alapú önkiszolgálás (On-Demand Self-Service):** A felhasználó számára nem szükséges a szolgáltatóval való további megegyezés újabb erőforrások igénybevételéhez. Sok eseteben ezeknek az erőforrásoknak az igénylése, konfigurálása és elengedése teljesen programozottan megvalósítható API-n keresztül történik.
* **Erőforrás készletezés (Resource pooling):** A felhasználók számára a fizikai és virtuális erőforrások egy közös forrásból elérhetőek. Az erőforrások típusonként (tárolási kapacitás, számítási kapacitás) teljesen homongének a felhasználó számára sem a konkrét hardver, sem annak helye nem ismert.

Felhívnám a figyelmet, hogy ezeknek a jellemzőknek egyike sem fogalmaz meg semmilyen állítást a fizikai infrastruktúrára vonatkozóan. Az azt képező szerverek és tároló kapacitások absztrakt, particionálható virtuális erőforrások.

A saját szerverpark és infrastruktúra létrehozása, illetve bővítése képezi az IT szervezetek tőkeberuházásainak túlnyomó részét. Kisebb szervezetek, amelyek nem engedhetnek meg maguknak saját szerverparkot, ezért bérelni kényszerülnek, amely szintén beépül a hosszútávú üzemelési kötlségeikbe. Ennek velejárója, hogy az erőforrásbővítés beruházási kérdéssé válik, amelynél hosszútávú szempontok érvényesülnek, illetve az üresen álló kapacitások kieső nyereséget jelenthetnek. A már megvásárolt, illetve tartósan bérelt fizikai infrastruktúra jellemzői pedig döntően befolyásolhatja a későbbi fejlesztési döntéseket.

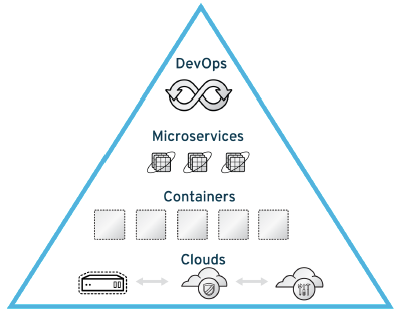
A felhő által nyújtott absztrakció, illetve a felhő szolgáltatók (Amazon AWS, Microsoft Azure, Google CloudPlatform) segítségével ezek a tőkeberuházások szinte teljes egészében üzemeltetési kötlséggé konvertálhatóak. Az újabb erőforrások bevonása vagy épp elengedése prompt igény alapon eldönthető. A nagy fokú virtualizációnak és konténerizációnak köszönhetően pedig technológiailag heterogén környezetek fenntartásának költsége is minimalizálható.

Ahhoz azonban, hogy a felhő infrastruktúra nyújtotta szolgáltatásokat ki tudjuk használni, a szoftvereknek a következő képességekkel kell rendelkezniük:

* gyorsan indíthatóak és leállíthatóak,
* teljes alkalmazások helyett a teljesítménykritikus modulok önállóan skálázhatóak,
* helyátlásztó módon kommunikálnak,
* kommunikációjuk nem jár jelentős többletköltséggel,
* futtathatóságuk a telepítési környezettől a lehető legkisebb mértékben függ.

## Összefoglalás

Azt gondolom, hogy az nyilvánvaló, hogy a fejezetben ismeretett változások nem érintenek egyenrangúan minden IT szervezetet. Mint minden üzleti területen, itt is megtaláljuk azokat az korai alkalmazókat, akik élen járnak ezeknek a technológiáknak a felhasználásában. Az ilyen vállalkozások fő jellemzője, hogy tevékenységüket globálisan, az interneten keresztül fejtik ki, elsősorban a „business to costumer” szegmensben, ahol leginkább jellemzőek a heterogén igények. Példaként említhetjük az Amazont, a Netflix-et, az Ebay-t és az Ubert. Ezek a vállakozások közös jellemzője, hogy a MSA tervezési megközelítés mellett a teljes fejlesztési vertikumukban a fejezetben ismertett módszereket és technológiákat adaptálták.



‑. Ábra Modern vállalati alkalmazás fejlesztés piramisa, forrás: Markus Eisle - Modern Java EE Design Patterns

# A vizsgált architektúrák bemutatása

Ebben a fejezeteben feltárom az általam vizsgált két architektúra közötti eltéréseket. A fejezet végére így világossá válik, hogy a munkám további részében milyen tényezőket kell figyelembe vennem a tervezés és az implementáció során. Az szakirodalomkutatás ívét a következők szerint alakítom.

Először röviden bemutatom a monolitikus architektúrát, ezt követően áttekintem, hogy a Java EE platform hogyan valósítja meg ezt a tervezési szemlélelet és milyen eszközöket kínál hozzá.

A fejezet második részében bemutatom a microservice architektúrát, illetve röviden ismertetem az alapjául szolgáló megközelítéséket.

Ezt követően a lényegi eltéréseket összefoglalva megvizsgálom a Java EE platform esetleges hiányosságait az MSA megközelítés alkalmazhatósága szempontjából, továbbá megvizsgálok néhány olyan technológiát, amelyeket a hiányosságok áthidalására később alkalmazni fogok.

## A monolitikus szoftverarchitektúra bemutatása

A monolitikus architektúrát a szakirodalom gyakran rétegzett, vagy ahogy a Java világában is, n-rétegű architektúrának hívja. Ez a legelterjedtebb architektúra az üzleti alkalmazások világában, mivel jól ismert és az előző fejezetben is leírt tradicionális IT szervezeti struktúrát képezi le.

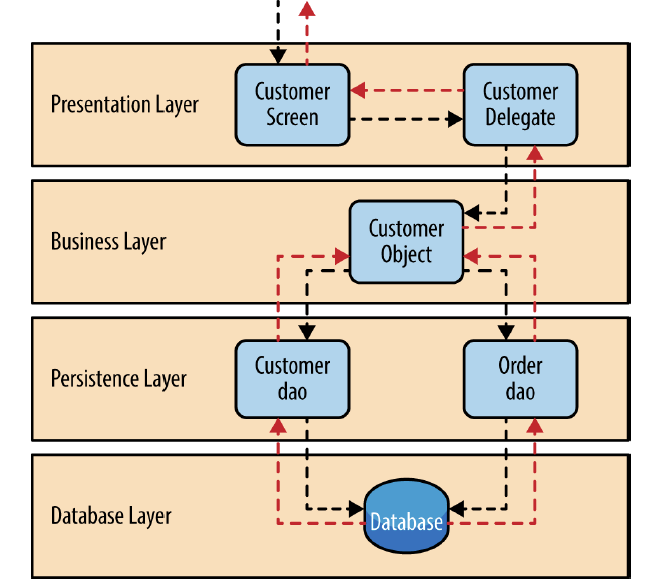
Ez a megközelítés az alkalmazást horizontális rétegekbe szervezi, a rétegeket funkcionálisan bontja szét. A rétegek száma tetszőleges lehet, de a legtöbb esetben négy szokott lenni:

* **megjelenítési**: elsődleges feladata a felhasználói interfész biztosítása,
* **üzleti logika**: az üzleti funkciók végrehajtása,
* **perzisztencia**: az üzleti logika entitásai és az adatbázis entitásai közötti megfeleltetés, tranzakciókezelés, integritás biztosítása,
* **adatbázis**: a perzisztens adatmodell tárolása.

Bizonyos alkalmazások tartalmaznak egy szolgáltatási réteget is, amelyek az alkalmazás egészében elérhető szerviz (authentikáció, autorizáció, loggolás, tranzakciókezelés, stb) szolgáltatásokat biztosítja.

Minden réteg tehát egy dedikált feladatot lát el, amelyért csak ő felel, és a többi réteg minden réteg számára fekete dobozként működik. Ez az elv az úgynevezett „separtaion of concerns” azaz a felelősségek szétválasztása [21]. Ez teszi lehetővé, hogy az egyes rétegek külön fejleszthetőek, tesztelhetőek és karbantarthatóak legyenek.

Az architektúra elmélteileg lehetővé teszi a rétegek közötti minimális csatoltság megvalósítását, így azok fizikailag is elkülönülhetnek és technológia választás szempontjából függetlenek lehetnek, de ez a gyakorlatban nem jellemző. Ilyen esetben a rétegek távoli eljárás hívással kommunikálnak.



3‑1. Ábra Monolitikus architektúra, forrás: Mark Richards - Software Architecture Patterns

A rétegek közötti adatáramlás tekintetében megkülönböztethetünk nyitott és zárt architektúrát [21]. Mint a fenti ábrán is látható, egy az alkalmazás által kiszolgált kérés-válasz folyamat felülről lefelé, majd a válasz alulról felfelé halad. A kérésfeldolgozás és a válaszgenerálás folyamatában azonban nem feltétlenül vesz részt minden réteg.

* **Zárt architektúra** esetén kommunikáció csak szomszédos rétegek között megengedett. Bármely kérés, ami megjelenítési réteg felől érzkezik (pl. egy megrendelés állapotának frissítése) szükségszerűen végig halad az összes köztes rétegen, mielőtt elérne az adatbázis rétegig.
* **Nyitott architektúra** esetén megengedett, hogy az egyes rétegek ne csak a közvetlen szomszédjaikon keresztül érhessék el a távolabbi rétegeket, hanem ezeket akár közvetlenül is megszólíthassák. Szolgáltatási réteg alkalmazása esetén például a prezentációs réteg közvetlenül lekérdezheti a felhasználóra vonatkozó autentikációhoz szükséges adatokat, így még azelőtt ellenőrizheti a jogsoultságokat, mielőtt megszólítaná az üzleti réteget.

A zárt megközelítés áttekinthetőbb feltételeket teremt a fejlesztők számára, mivel a rétegek közötti viszony végig egyértelmű marad. Azonban fejlesztési többletköltséget jelent, hogy minden egyes kérés-válasz folyamatot keresztül kell csatornázni minden rétegen még akkor is, ha lényegi logika nem kerül megvalósításra. A nyitott megközelítésnek ezzel szemben az a hátránya, hogy növeli a csatoltságot a rétegek között.

A monoltikus megközelítés esetén két további jellemzőt emelnék ki, melyek már az üzleti folyamatok leképezését is meghatározzák.



‑. Ábra A funkcionális rétegeket és üzleti domainek viszonya

A fenti ábrán az látszik, hogy az egyes alkalmazási területek, azok entitásai és folyamatai a teljes funkcionális vertikumon átívelnek. Ennek következményei a következők:

* Ha több alkalmazási terület is megvalósítja ugyanazt a funkciót, akkor azt az alkalmazáson belül össze lehet vonni. Megfelelő generikus tervezéssel tehát viszonylag könnyen bővíthető az alkalmazás.
* Ha egy üzleti folyamat több alkalmazási területet érint, akkor a közöttük lévő kapcsolat az alkalmazáson belül implementálható. Folyamaton (process) belüli kommunikáció valósul meg.
* A fejlesztés korábbi szakaszában kiválasztott technológiák döntően meghatározzák az új igényeknek kielegítésének módját és lehetőségét.
* Bár az üzleti szempontból egységes folyamatokat értelmezünk a technológiai platform egyes szintjein, a folyamat egészét nem látjuk. Az üzleti probléma egy meghatározott rétegbeli implementációja meg/lekötheti a többi réteg lehetőségeit.
* Az egyes szinteken lehetőség van az entitások bizonyos szintű „testreszabására” (data access object, data transfer object, viewmodel), azonban az alkalmazás szintjén általában egy adatmodell kerül definiálásra, így az eltérő alkalmazási területeket ehhez a modellhez kell kötni.

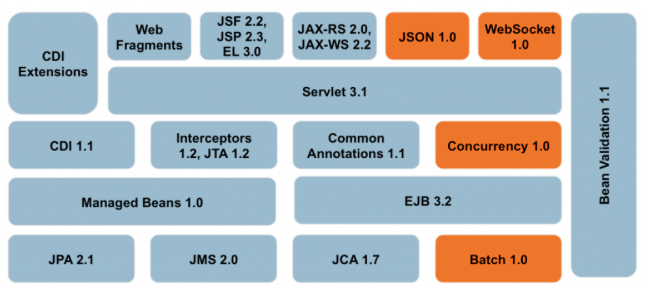
### Monolitikus architektúra megvalósítása java ee platformon

A monoltikus megközelítés a Java EE 1998-as megalkotása óta de-facto sztenderdje a platformoknak. A Java EE tulajdonképpen egy nyílt szabvány, ez a szabványvezéreltség végigkíséri a platform teljes történetét. A szabvány alapját a JSR-ek (Java Specification Request) képzik. Ezek követelményeket fogalmaznak meg a nyelvvel, könytáraival és komponsenseivel szemben. A Java EE platform verzió számai tulajdonképpen a JSR-eknek egy meghatározott halmazát jelenti.

A Java EE platform központi részei – a Java nyelv és komponensei mellett – az úgynevezett alkalmazásszerverek. Ezeknek az alkalmazásszervereknek a feladata, hogy megfeleljenek a Java EE egy meghatározott verziója által definiált elvárásásoknak. A korábban említett komponensek az alkalmazásszerveren futnak, pontosabban annak valamelyik konténerében. A konténer feladata, hogy biztosítsa azokat a middleware szolgáltatásokat a komponensek számára, melyeket a Java EE moduljai alkotnak. Ezeknek a middleware szolgáltatásokanak egy része megfeleltethető az előző részben említett szolgáltatás rétegnek.

A konténereknek három fajtája van. Szerveroldalon az úgynevezett webkonténerek és EJB-konténerek, illetve kliens oldalon beszélhetünk még appletkonténerről [3].

A Java EE 7-es verzióját az ábrán látható komponensek alkotják:



‑. Ábra Java EE 7 komponensei

Ezek a komponensek, illetve a Java SE néhány szabványa biztosítják az alábbi middle-ware szolgáltatásokat [3]:

* perzisztencia, ORM,
* tranzakciókezelés,
* biztonság,
* távoli elérés, többszálúság,
* monitorozás,
* életcilkus kezelés,
* névszolgáltatás.

Az előző részben bemutatott rétegrend a főbb Java EE komponensekkel kiegészítve a következő ábrán látható.



3‑4. Ábra Monolitikus architektúra Java EE alapokon

A webszerver és az alkalmazásszerver fizikailag lehet egy vagy több gépen. Egy gép esetén is futhat külön Java Virtuális Gépeben a kettő, ekkor processzek közötti hívásokkal tudnak kommuniálni a komponensek, mintha fizikailag külön helyezkednének el. Az egyes komponensek is futhatnak több alkalmazásszerveren elosztva, az EJB technológia és névszolgáltatás lehetővé teszi ezt a fajta felosztást is. Fontos megjegyezni, hogy az alkalmazás komponensei csak akkor tudnak folyamaton belüli hívásokkal kommunikálni, ha azok egy Java Virtuális Gépben futnak.

Az általános gyakorlat az, hogy a webszerver az alkalmazásszerver és az adatbázisszerver teljesen elkülönülnek. Az előbbi kettő között távoli eljárás hívással valósul meg a kommunikáció.

Az alkalamzás legmagasabb szintű logikai egységét az EAR (Enterprise Archive) állomány adja. Az állomány a modulokon kívül az alkalmazásszintű leíró adatokat tartalmazza. Az alkalamazás moduljait az EAR-on belül a JAR (Java Archive) és WAR (Web Archive) állományok tartalmazzák. Egy állomány egy modult reprezentál. A WAR állományok olyan modulokat (és a modulok leíró adatait) tartalmaznak, amelyek a webkontérnerben futnak. Így egy ilyen konténerrel rendelkező webszerver önállóan képes futtatni ezeket az állományokat, EAR nélkül is. A JAR-ok tartalmazzák az EJB konténeren belül futó modulokat (és leíró adataikat),

Az alkalmazásfejlesztés során a fejlesztő tehát elsősorban az üzleti logika fejlesztésével tud foglalkozni, míg az egyes middle-ware szolgáltatások implementációját figyelmen kívül hagyhatja: ezek telepítési időben kerülnek konfigurációra és feloldásra. Mivel az alkalmazás monolitikus, így a tranzakció- és jogosultságkezelés is könnyen megvalósítható a konténeren keresztül hívott middle-ware szolgáltatások által.

Ennek a megközelítésnek az a hátránya, hogy az alkalmazás komponensei nem önjáróak, csak a konténeren belül tudnak futni. Az alkalmazásszerverek nagy mérete és bonyolult konfigurálhatósága további skálázhatósági problémákhoz vezethetnek. Ezenfelül a hibák kezelése is igen nehézkessé válhat, mivel az alkalmazás egy pontján felmerülő hiba könnyen tovább gyűrűzhet az alkalmazás más rétegeibe, a futtató környezet hibái pedig könnyen a teljes alkalmazásszintű problémához vezethetnek.

## A microservice szoftverarchitektúra bemutatása

A microservice fogalmának megalkotásához egy 2011-ben megtartott konferencia vezetett. A konferencia résztvevőinek egy csoportja többek közt ezzel a névvel határozta meg azt az architekturális stílust, amivel újonnan kísérleteztek. A következő évben már több konferencián is ezzel a névvel prezentálták a csoport résztvevői a modellt [12].

A koncepció valamivel régebbi, hiszen a technológiai óriások már bő egy évtizede kísérleteznek ehhez hasonló modellekkel. Egy google keresés kb. 70 microservice-t hív meg, mielőtt eredménnyel tér vissza [17].

Az architektuális modell megalkotásához evolúciós út vezetett, így a következő szakaszban bemutatnék két olyan szoftver architektúrát, amelyek hatást gyakoroltak az MSA-ra.

### A microservice szoftverarchitektúra „elődei”

* **Esemény vezérelt architektúra:**

Az architektúrát eloszott modulok aszinkron együttműködése alkotja. A modulok jellemzője, hogy csak egy üzleti célt szolgálnak ki. A modell onnan kapta a nevét, hogy a végrehajtást események vezérlik, melyek üzenetek formájában áramolnak a modulok között (hívják üzenet vezérelt architektúrának is) [21].

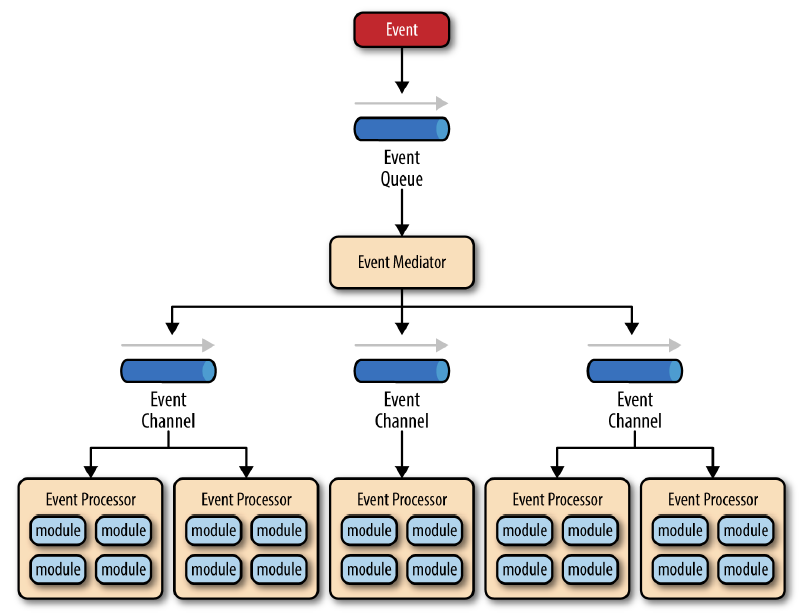
Az architektúra két lehetséges implementációs módja:

* közvetítő topológia,
* bróker topológia.

A közvetítő topológiát négy komponens alkotja:

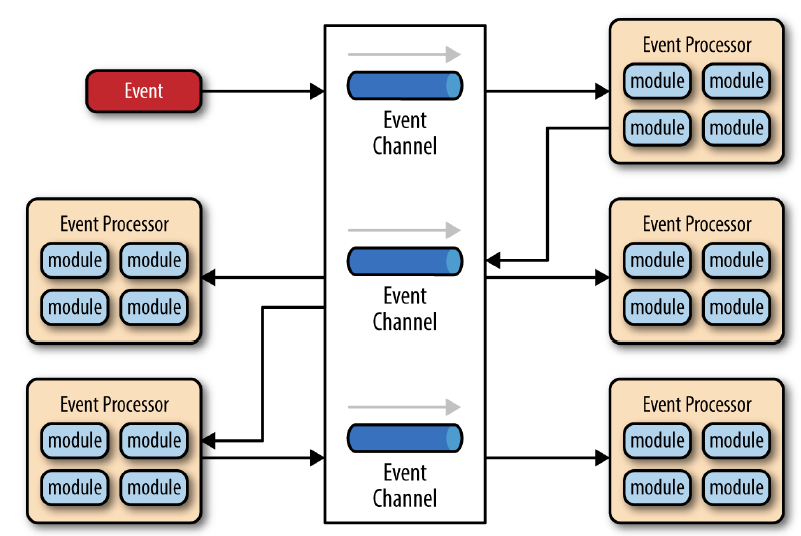
* esemény-sorok (üzenetsor, webszolgáltatás végpont, stb),
* esemény-közvetítők,
* esemény-csatornák,
* eseménykezelők.

A kliens egy kéréssel fordul az esemény-sorhoz. Ez a kérés lesz a kezdeti esemény. A feldolgozás első lépéseként az esemény-közvetítő kiolvassa a sorból az eseményt, majd eldönti, hogy milyen további feldolgozó eseményekre kell bontani. Ezután az „elemi” eseményeket elhelyezi a megfelelő esemény-csatornákban. Ebből a csatornából már a feldolgozást végző modulok olvassák ki és dolgozzák fel az eseményeket. Egy sort természetesen több modul is figyelhet (publish-subscribe). Az esemény-közvetítő feladata, hogy a feldolgozás eredményeit bevárja, összedolgozza és továbbítsa a kliens felé. Az eseménykezelőket úgy szokás megtervezni, hogy a feldolgozás során ne kelljen köztük kommunikációt lebonyolítani.



3‑5. Ábra Közvetítő topológia, forrás: Mark Richards - Software Architecture Patterns

Bróker topológia ennél egy fokkal decentralizáltabb megközelítést alkalmaz. Komponensei:

* bróker (esemény csatornák csoportja),
* eseménykezelők.

A bróker egy passzív elem, az eseménykezelőknek ismerni kell azokat az eseménycsatornákat, amikből eseményeket olvasnak és ahova a feldolgozás eredményeként előállt eseményeket el kell helyezniük. Mivel nincs központi vezérlés, ez a topológia lazább csatolást valósít meg, azonban az eseménykezelőkre rakódik a közvetítés terhe.

Ez a modell tekinthető a szolgáltatásorientált és a microservice architektúrák általános reprezentációnak. Azonban ez a modell nem fogalmaz meg implementációs megszorításokat az architektúra egyes komponenseivel szemben.

* **Szolgáltatás orientált architektúra (SOA):**

Az ezredforduló második felében jelent meg ez a tervezési megközelítés. Célja az volt, hogy komplex üzleti folyamatokat tudjanak integrálni szabványos protokollok segítségével. [3].

Az architektúra „forradalmiságát” az adta, hogy szoftver komponensek helyett a szolgáltatást helyezte előtérbe (absztakciós szintlépés). A kommunikációs protokolloknak nyílt, platformfüggetlen szabványokat választottak. Így viszonylag hamar vált de-facto sztenderdé az XML alapú webszolgáltatás, első sorban a SOAP (Simple Object Access Protocol).

Az SOA a szolgátatások jellemzőire, az interfészeikkel szemben támasztott követelményeken kívül nem fogalmaz meg semmilyen megszorítást.

3‑6. Ábra Bróker topológia, forrás: Mark Richards - Software Architecture Patterns

Az architektúra folyamat szemléletű, így a szolgáltatások együttműködését általában munkafolyamatba szervezik (workflow) és ezeken a folyamatokon definiálnak viselkedési szabályokat, házirendet (policy).

Az architektúra központi eleme az Enterprise Service Bus (ESB). A szolgáltatások nem közvetlenül egymással, hanem az ESB-n keresztül kommunikálnak. Az előző szakaszban bemutatott modell mediátor topológiáját valósítja meg az ESB. Az ESB-t implementáló gyártók nagy része törekszik valamilyen nyílt szabványon megvalósítani a bust. Az ESB fő feladatai:

* Üzenetirányítás, szolgáltások szabály-alapú összehangolása a folyamatban.
* Üzleti szabályok megvalósítása.
* Szolgáltatások felderítése.
* Protokollok közötti fordítás.
* Monitorozás.
* Biztonság, validáció.

Látható tehát, hogy az ESB nagy szerepet játszik a SOA működésében. A SOA-t implementáló cégek általában nem is fejlesztenek saját ESB-t, hanem gyártótól vásárolnak ilyen megoldást (pl IBM WebSphere ESB, SAP Process Integration, Oracle ESB).

### A microservice szoftverarchitektúra jellemzői

Ha röviden kellene definiálni a microserviceket, akkor az alábbi definíció viszonylag pontos leírást ad:

„*... a microservice architekturális stílus az szoftverfejlesztés olyan megközelítése, amelyben az alkalamzást kis méretű, önálló folyamatként futó szolgáltatások alkotják ...[[4]](#footnote-4)*” [12]

Ez a megfogalmazás a legtöbb irodalomban megtalálható, azoban ahogy azt a 3.2-es fejezet elején már említettem, a fogalmat megalkotók inkább a szoftver karakterisztikái mentén definiálják az MSA-t – így én is ezeket a karakterisztikákat tekintem át.

1. **Modularizálás szolgáltatások segítségével**

A modularizálás az alkalmazásfejlesztés különböző szintjein megjelenő visszatérő igény. Modul alatt olyan egységet értünk, amely képes önállóan megvalósítani egy vagy több feladatot, a modul önállóan fejleszthető és más moduloktúl függetlenül cserélhető, a külvilág számára zárt egységet képez. [12]

A világunk jobb megértése és modellezhetősége érdekében töreksztünk a komplex egészet részekre bontani, majd az egyszerűbb részekből megalkotni az egészet. Az objektum orientált paradigma irányelvei is a modularizáltság megalkotását célozzák. A monolitikus rendszerek is modularizáltságra törekednek, azonban az esetek nagy részében ezt leginkább az implementációs nyelv eszközeire bízzák (libraryk, packagek, interfészek, láthatósági szintek). Az MSA a modularizálást azzal próbálja meg az alkalmazás szintjén tovább erősíteni, hogy olyan követelményeket emel a modulokkal szemben, amelyek természetüknél fogva kikényszerítik az egységbe zárást, a explicit interfész definiciót, a módosítással szembeni zártságot. A szolgáltatások olyan egységek ugyanis, melyek a rájuk bízott feladatot a többi szolgáltatástól független, önálló folyamatként valósítják meg. Mivel folyamatok közötti a kommunikációról van szó, ezért az alkalmazásszintű interfészek egy szigorúbb, szemcsézetebb absztakcióját képezik a szoftver hierachiájának. Gyakori, hogy az ilyen interfésznek nemcsak a ki- és bemenő paramétereit definiálják, hanem Szolgáltatás Szintű Szerződést (Service Level Agreement) is [1], mint például az átlagos válaszidő, rendelkezésreállás, stb.

A szolgáltatás szintű interfészekre még „ígéretekként” (promises) is szokás hivatkozni. Ez az elnevezés arra utal, hogy a szolgáltatás megígéri a felhasználóinak, hogy minden körülmények között számíthatnak az megígért szolgáltatásra. Ez például jelentheti azt is, hogy ha a szogáltatásunk normális körülmények között nem tud teljesíteni egy ígéretet (mondjuk mert egy alsóbb szintű szolgáltatás nem elérhető), akkor is ad valamilyen, az ígéretet kielégítő választ [20].

Az ilyen szolgáltatások másik fő jellemzője a méret, amely egyben a névadó jellemzője is a tervezési stílusnak. A témában fellelhető publikációk széles skálán próbálnak erre választ adni, mind kvalitatív, mint kvantitatív jellemzőkkel. A kvantitatív megszorítások szerintem nehezen értelmezhetőek, mivel ezekre a megvalósítandó logika komplexitása, az implementáció nyelve és sok egyéb tényező hatással is lehet. Alább összeszedtem azokat a kvalitatív jellemzőket, amelyek magabiztos tervezést tesznek lehetővé:

* A szolgáltatás legyen akkora, hogy bármikor eldobható legyen. Az ajánlás abból fakad, hogy sok esetben az újabb technológiai és üzleti követelmények olyan mértékű változást indukálnak, hogy egyszerűbb lenne a teljes megoldást eldobni és újraírni, mint a meglévőt állandóan módosítgatni. „Rewrite over maintain” [16]. Az Ebay sikerrel alkalmazta ezt a megközelítést.
* A szolgáltatás legyen akkora, hogy az azt fejlesztő csapat számára annak egésze áttekinthető legyen. Gyakori probléma, hogy az idő előrehaladtával az alkalmazások akkorára nőnek, hogy a fejlesztők nem látják át a teljes rendszert, ezért annak csak egy-egy részével kapcsolatban rendelkeznek mély ismeretekkel. Ilyenkor az ismert részeken végrehajtott módosítások mellékhatásai az alkalmazás más pontjain előre nem látható következményekhez vezethetnek. Ezt az elvet szem előtt tartva az ilyen mellékhatások kiküszöbölhetővé válnak. További előnye ennek a megközelítésnek, hogy a csapatba frissen érkező fejlesztő hamarabb be tud kapcsolódni a munkába.
* UNIX filozófia: „Egy dolgot csinálj, de azt jól”. Az szolgáltatás legyen akkora, hogy a rábízott feladatot önállóan meg tudja valósítani a vele szemben támasztott minőségi követelményeknek megfelelően.

1. **Domain-Driven Design elvei szerint történő alkalmazás szervezés**

Az alkalmzás-fejlesztés tervezési szakaszában, az üzleti tervezés során el kell kezdeni az adatmodell kialakítását. Ekkor kerülnek definiálásra az üzleti logika szempontjából kulcsfontosságú fogalmak, ezek belső szerkezete és kapcsolataik, amelyből később elkészül a fizikai adatmodell. Ez az adatmodell később már nem, vagy csak nagyon nehezen változtatható. Így minden későbbi fejlesztést az adatmodell által adott keretek határoznak meg. Ez természetesen üdvözölendő is lehet, mivel az azonos alkalmazási területen dolgozó üzleti szereplők és fejlesztők biztosan támaszkodhatnak erre az adatmodellre. A gyakorlat azonban azt mutatja, hogy az alkalmazási terület bővülésével az egységes adatmodell kezelése hátrányokkal jár. Ugyanis különböző alkalmazások integrációja során, de akár még alkalmazáson belüli is előfordulhat, hogy ugyanaz a domain fogalom mást jelent. Gyakori az is, hogy egy újabb üzleti igény számára a korábban alkotott modell az entitást nem úgy ábrázolja, ahogy az a probléma szempontjából ideális volna.

A tervezők kényszerűen felismerték, hogy bizonyos komplexitás fölött nem univerzális adatmodellt kell alkotni, hanem üzleti területenként külön-külön, amelyeket aztán a kulcsfogalmak mentén össze lehet kapcsolni. Ez a felismerés adja Domain-Driven Design alapját.

Domain az a tárgykör, aminek keretében a felhasználó az alkalmazást használni kívánja. Itt tehát nem pusztán az adatmodellben értelmezett entitásokról van szó, hanem a kapcsolódó funkciókról, eseményekről, szolgáltatásokról is. A domaint két fő fogalom alkotja [11]:

* **Modell**: egy absztrakciós rendszer, ami a vizsgált domain főbb aspektusait és belső felépítését írja le,
* **Kontextus**: az a környezet, amiben az adott modell és elemei egyértelmű jelentéssel bírnak. Pl. a **fogyasztó** teljesen mást jelent egy gázszolgáltató ügyfélszolgálati osztályán, mint az infrastruktúra üzemeltetésért felelős osztályon.

Bounded context-nek nevezik azokat a határokat, amelyeken belül a modell még egy egységet képez. Ezek a határok például lehetnek a szervezeti funkciók között értelmezett határok vagy az alkalmazás adatbázis sémái között értelmezett határok.

A szolgáltatást tehát úgy kell meghatározni, hogy pontosan egy bounded contextbe tartozzon.

Ebben a megközelítésben fokozott körültekintést igényel az eltérő bounded-contextben elhelyezett szolgáltatások közötti együttműködés megvalósítása, a modellek közötti átjárhatóság kialakítása.

1. **Projektszemlélet helyett termékszemlélet**

Ez a jellemző nem annyira technikai jellegű. Sokkal inkább annak a tapasztalatnak akar érvényt szerezni az alkalamazásfejlesztés során, hogy a szoftverek nagy része evolúciós úton fejlődik. Az igény felmerülésétől a produktum átadásáig általában egy projektcsapat dolgozik az alkalmazáson, melyet az üzembehelyezése után feloszlatnak. Az elkészült alkalmazást viszont időről-időre szükséges lehet tovább fejleszteni, amihez gyakran egy újabb csapatott szerveznek. Számukra az alkalmazás feltárása az első lépés. Természetesen nagy méretű, monolitikus rendszerek világában elképzelhetetlen lenne, hogy a fejlesztői csapat feleljen a teljes üzemeltetésért, mivel ez a fejlesztői kapacitásuk nagy részét lekötné. Kis méretű, gyakran továbbfejlesztett szolgáltatások esetén azonban ez egy hatékony megközelítés, mivel a korábban említett feltárási folyamat szükségtelenné válik.

Ezt hívja Amazon „you build it, you run it”-nak.

1. **Okos végpontok, buta csatornák**

A SOA-ban tárgyalt ESB-khez képest az MSA az üzleti logika minden elemét a szolgáltatásokba igyekszik zárni. Az MSA kerüli a kommunikáció során a nehézsúlyú szabványok alkalmazását, mint pl a SOAP vagy BPEL. Olyan kommunikációs protokollokat részesít előnyben, melyekre igaz, hogy üzenet-alapúak, széleskörben elterjedtek, konvenció vezéreltek. [12]

A leggyakrabban alkalmazott protokollok:

* HTTP/REST
* Üzenetsor kezelők protokolljai: Advanced Message Queuing Protocol, Message Queue Telemetry Transport

Olyan végpontokat kell kialakítani, amelyek rugalmasan és hibatűrően tudják kezelni a kéréseket. Sok hibatűrést elősegítő tervezési minta (pl. consumer driven contract, tolerant reader) [20] nem, vagy csak nehezen lenne megvalósítható a séma validáció vezérlet SOAP világában. [23]

1. **Decentralizált menedzsment**

A decentralizált menedzsment alatt nemcsak azokat a törekvést értjük, amelyeknek a célja az, hogy a rendszer komponensei egymástól függetlenül, saját erőforrásokkal tudjanak működni, hanem minden olyan törekvést, aminek célja az, hogy ezek a komponensek technológiai, fejlesztési szempontból függetlenek legyenek egymástól.

A centralizált menedzsmentet megvalósító architektúrákban a tervezők hajlamosak arra törekedni, hogy az alkalmazás, vagy legalább nagyobb részegységei technológiailag homogének legyenek. Ez ösztönzi a közös kódbázis kialakítását, csökkenti a felesleges duplikációkat a rendszerben és segíti az újrafelhasználhatóságot.

Az MSA ezzel szemben támogatja a technológiailag heterogén rendszereket, mivel így lehetőség van az adott problémához megfelelő eszközt választani.

Ez természetesen nem jelenti azt, hogy az azonos problémákat újra és újra fel kell oldani, de nem feltétlenül törekszik a homogén kódbázis kialakítására.

A „Ne ismételd magad” irányelv tehát az MSA-ban nem érvényesül közvetlenül. Az alkalmazás infrastruktrújában ugyanis a közös részek minimalizálására kell törekedni. Természetesen egy adott problémára itt is szabadon felhasználhatóak a már kész library-k, azonban ezekből minden szolgáltatás sajáttal rendelkezik..

1. **Decentralizált adatmenedzsment**

A Domain Driven Design segítségével modellezés szintjén lehet dekomponálni és alkalmazási területenként hatékony adatmodellt kialakítani. Ennek implementációs párja a decentralizált adatmenedzsment.

A monolitikus alkalmazások általában egy központi (általában relációs) adatbázis fölött működnek. Ezzel szemben az MSA alkalmazás (vagy válallati szintű) adatmodell kezelés helyett szolgáltatásszintű adatmodell kezelést valósít meg. [12]

Így lehetőség nyílik arra, hogy adatmodellenként eltérő adatbázis típust használjunk, akár NoSQL adatbázisokat.

Ez a megközelítés implicit módon segíti a skálázhatóságot, mivel a monolitikus alkalmazások esetén a skálázás egyik korlátja a közös adatmodell elérésére szolgáló erőforrások kimerülése, melyet csak bonyolult adatbázis klaszterezéssel lehet megoldani. A decentralizált adatmenedzsmentnek köszönhetően az erőforrások elérése is elosztottan történik.

Az MSA fejlesztés során a Java EE világához szokott alkalmazásfejlesztőknek problémát okozhat ennek a szemléletnek a megvalósítása, mivel monolitikus, centralizált alkalmazás esetén a tranzakciókezelés, atomicitás middlre-ware szolgáltatáson keresztül garantálható.

A decentralizált rendszerekben ezzel szemben a tranzakció és atomicitás fogalma nehezen értelmezhető. A CAP tétel [22] szerint az elosztott rendszerekre az alábbi három képesség közül egy időben csak kettő érvényesíthető:

* Konzisztencia: A műveletek atomicitása biztosítja, hogy az adatbázis adatainak állapota mindig konzisztens legyen.
* Rendelkezésreállás: Bármely időben beérkező, valid kérés a legutolsó írás eredményét adja vissza.
* Partíció tolerancia: Az alkalmazás akkor is működőképes marad, ha a tetszőleges számú üzenet veszik el.

Az „eseti konzisztencia” lehetőségének bevezetésével biztosítható a rendelkezésre állás és a partíció tolerancia. Az eseti konzisztencia megengedi, hogy az alkalmazás egyes szolgáltatásai által kezelt adatmodellek között átementi inkozisztencia lépjen fel.

1. **Tervezz a hibákkal**

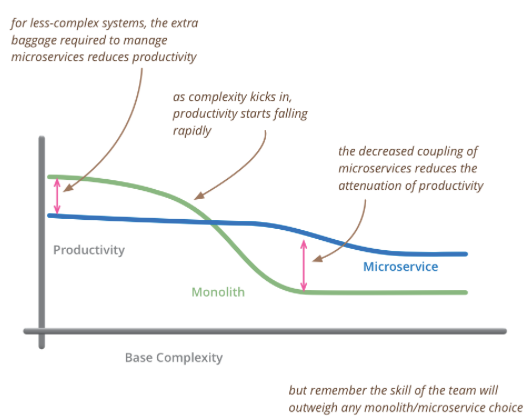
Az elosztott rendszerekre jellemző fokozott komplexitás nehezebbé teszi a hibafelismerést és -kezelést. A szolgáltatásokat és magát a rendszert ezért úgy kell megtervezni, hogy a hálózati kommunikációra jellemző sztochasztikus hibákra (pl a hívott szolgáltatás leszakad a hálózatról) fel legyen készítve. A hibák izolálása kulcsfontosságú, mivel a rendszer egy tesztőleges pontján bekövetkezett hiba továbbgyűrűzése esetén a teljes rendszer teljesítményének romlásához vezethet. Megfelelő izolációval azonban az ilyen sztochasztikus hibák a hiba forrásának detektálásáig elháríthatóak.

A végpontokat tehát fel kell készíteni arra, hogy az általuk hívott szolgáltatás nem képes válaszolni, és ebben az esetben is választ kell szolgáltatni az őt hívó felé, ezáltal megakadályozva a hiba továbbgyűrűzését.

A rendszer egészét tekintve is alkalmazhatóak olyan tervezési minták, melyek a hibás komponensek kikerülését célozzák, ilyen például a biztosíték (circuit-breaker) vagy a válaszfal (bulkhead) tervezési minta [20].

## A két vizsgált architektúra összehasonlítása

Azt gondolom, hogy az előző pontokban vázoltak alapján nem jelenthető ki, hogy egyik vagy másik architektúra egyértelműen jobb lenne. A két architektúra eltérő természetű problémákra kíván megoldást adni, így az egyik vagy másik sikeressége elsősorban a üzleti és technológai környezet helyes megértésén múlik. A monolitikus architektúra jól kiforott, közismert, széles körben támogatott. Amennyiben olyan, akár nagy méretű alkalmazást fejlesztünk, melynek felhasználói köre jól behátolható, a várható terhelési maximumok jól becsülehetőek és ésszerűen kielégíthetőek, akkor a monolitikus architektúra jól választás lehet. Sőt ilyen esetben az MSA megközelítés külső architektúrájának komplexitása irreleváns terhet ró a fejlesztőkre [13]. Számos iparági példa mutatja, hogy az MSA architektúra akkor jelenik meg, amikor a korábbi monolitikus rendszer eléri azt a kritikus méretet, terhelést vagy komplexitást, amely mellett már elfogadhatalanná válik a monolitikus megközelítés fejlesztési ciklusainak hossza vagy skálázhatóságának költsége.



‑. Ábra Rendszerkomplexitás és termelékenység viszonya, forrás: James Lewis - MicroservicePremium

Jó példa erre az eBay architektúrája, amelynek első verziója 1995-ben egy hétvége alatt készült el. Ezt 1997-ben egy 3 logikai és 2 fizikai rétegből álló monolitilkus architektúra követte, amit aztán hosszú éveken keresztül horizontális skálázással bővítettek, mígnem 1999-ben az adatbázis szerverek elérték a fizikai teljesítményük határát. A következő állomás az adatbázis logikai particionálása volt, majd pedig a 2000-es években az üzleti-alkalmazás réteg funkciók szerinti fokozatos felbontása. Ennek során fokozatosan elhagyták a Java EE architektúrát és heterogén platformot alakítottak ki. Az üzleti logikai réteget állapotmentessé tették, a tranziens állapotokat pedig adatbázisokkal vagy kliens oldali cookiekal biztosították. Az eBay architektúrájának jellegzetessége, hogy a szolgáltatásokat nemcsak üzleti funkciók szerint bontották fel, hanem bizonyos mértékben meghagyták a monolitikus alkalmazásokra jellemző horizontális tagoltságot, ezzel finomítotva tovább a skálázhatóságot.

Az Amazon kezdeti, 2 rétegű monolitikus architektúrája is hasonló utat járt be egészen addig, amíg nyilvánvalóvá nem vált, hogy a front-end az architektúrát megtartva már nem skálzható ésszerű keretek között. Ők is először az adatbázist bontották fel üzleti funkciók szerint, és minden adatbázis fölé dedikált interfészt fejlesztettek. A fő problémát az üzleti réteg felbontásában a csapatok közötti erős függőségi viszonyok korlátozták. Ezért a rendszert több iterációban bontották fel MSA alapokra a „szélekről” indulva (ahol a legkisebb volt a függőségi viszony foka) haladtak a rendszer magja felé, illetve ezzel párhuzamosan szervezték meg a szolgáltatásokat birtokló csapatokat is [24]. Az Amazon részéről ez a fajta váltás a menedzsment részéről erős elkötelezettséget követelt.

Az alábbiakban rövid, táblázatos formában foglalom összes a két architektúra közötti funkcionális különbségeket:

|  |  |  |
| --- | --- | --- |
| Értékelési szempont | Monolitikus | Microservice |
| Agilitás (a változó igények lekövetésének gyorsasága) | Az alkalmazás komplexitásának növekedésével az agilitás fokozatosan csökken, mivel a változásokat egyre bonyolultabb architektúrán kell átvezetni. | Az agilitásban az MSA jobban teljesít, mivel egy új igény akár egy teljesen új szolgáltatás kialakításával is kielégíthető. Amennyiben már meglévő szolgáltatásba implementálják, a komplexitás akkor is töredéke a monolitikus megfelelőnek. |
| Telepíthetőség | A széles körű támogattoságnak köszönhetően az alkalmazás telepítése könnyen megvalósítható, azonban minden apró változás a teljes alkalmazás újratelepítését kívánja meg. | A monolitikus rendszerekkel összehasonlítva a kezdeti infrastruktúra kialakítása jelentős többletköltséget jelent, de újabb szolgálatások beágyazása a működő infrastruktúrába, vagy a meglévők kicserélése akár üzemidőben is gyorsan megoldható. |
| Tesztelhetőség | Az egyes rétegek mockolása könnyen megoldható, így a tesztelés egészen elemi szinten is könnyen megvalósítható. | Mivel teljes szolgáltatások mockolhatóak, ha ismertek az interfészeik, az MSA szintén jól tesztelhető. Azonban az elosztott architektúrára jellemző tesztesetek megvalósítása többletmunkával jár. |
| Skálázhatóság | A rétegek szétválasztásával a skálázhatóság finomítható, de általában a teljes alkalmazás skálázására van szükség. Üzleti komponensek szerinti skálázás nem megoldható. | Az egyes szolgáltatások önálló skálázhatósága miatt ez az architektúra nagyon jól teljesít ebben a kategóriában. A skálázás itt két dimenzióban is megvalósítható. |
| Fejleszthetőség | Az architektúra egyszerűsége és támogatottsága miatt a monolitikus alkalmazások gyorsan és könnyen fejleszthetőek. | A szolgáltatások fejlesztése azok izolált jellege miatt könnyen kivitelezhető, a robusztus interfészek tervezésére viszont nagy gondot kell fordítani. |

# Java EE alkalmazás a microservice világában, felhasználható technológiák

A szakirodalmi kutatásom lezárásaként először megvizsgálom, hogy a monolitikus Java EE platform milyen mértékben illeszthető be a microservice világába, és megvizsgálom azokat a technológiákat, amik ezt a beillesztést segítik. A rendszertervezés és implementáció során ezeket a technológiákat kívánom alkalmazni.

Ahogy arról már korábban szó esett, a Java EE 1998-as megjelenése óta a monolitikus alkalmazásszervezést támogatta. A platform meghatározó két komponensét a szabványvezérelt API-k és azt azt implementáló alkalmazásszerverek adják. Az alkalmazásszerverek által menedzselt konténerek biztosítják a middleware szolgáltatásokat az üzleti komponensek számára. A Java EE fejlődésével a szolgálátatások palettája bővült, egyre magasabb absztrakciós szintek kialakításával könnyítették a fejlesztést. A Java EE 7 már 34 különböző szabvány megvalósítását követeli meg az alkalmazásszerverektől. Az alkalmazásszerverek mérete ennek köszönhetően egyre nagyobbra nőtt, illetve mivel a szolgáltatások esetén a szabvány a széleskörű testreszabhatóságot és hordózhatóságot is megköveteli, ezért a telepítési időben jelentős mértékű konfigurálásra van szükség. Ez utóbbi ellenére is sok nehézséget és nagy körültekintést igényel az alkalmazásszerverek verziói vagy implementáció közötti váltás.

Az alkalmazásszerverek komplexitás korábban olyan mértékű volt, hogy a Java EE 6-tal bevezették a Web Profile-t, melynek célja, hogy egyszerűbbé és kisebb méretűvé tegye az alkalmazásszervereken futó webalkalmazásokat.

A Java EE sztenderd telepítési modelljében a skálázhatóság általában az alkalmazásszerverek klaszterezését jelenti, amelyeken elosztottan futnak az alkalmazás moduljai, a modulok skálázását pedig resource poolok biztosítják. Ennek megfelelően a legtöbb API szinkronműködésre lett tervezve. Mindazonáltal a Java EE 7 számos aszinkron újítást vezetett be.

Fontos továbbá megemlíteni, hogy a Java EE világából számos olyan komponenes hiányzik, melyek az MSAban a külső architektúra részét képzik (pl. API Gateway, szolgáltatás felderítés, healtcheck, autentikáció elosztott környezetben, load balancing, stb.)l

Három szempontot kell tehát figyelembe venni, amikor Java EE alkalmazást szeretnék migrálni MSA alapokra:

1. Kisméretű, könnyen konfigurálható futtatási környezet biztosítása.
2. Aszinkron üzenetküldő API-k felhasználása.
3. MSA külső architektúráját megvalósító könyvtárak, szolgáltatások felhasználása.

A Java EE közösség a fenti kihívásokra már több választ próbált adni. Ezek közül mutatok be néhányat , hogy aztán közülük kiválasszam a dolgozatom további részében felhasználandó technológiát (technológiákat).

## Pehelysúlyú alkalmazásszerverek

A pehelysúlyú alkalmazásszerverekkel a Java EE alkalmazásszervereinek gyártói igyekeznek kielégíteni az MSA igényeit. A két leismertebb a Payara és a WildFly Swarm. Előbbit a Glassfisht is gyártó Oracle, utóbbit a Jbosst és WildFlyt is gyártó Red Hat készíti. Népszerűsége miatt én az utóbbit mutatom be röviden.

A WildFly Swarm alkalmazásszerver a WidFly alkalmazásszerver 10. verziója alapján készült. Az alkalamzásszervert több, kis méretű funkcionális egységre bontották fel és lehetővé tették, hogy ezeket az egységeket felhasználva az alkalmazás igényeinek megfelelő úgynevezett uberJar-t készítsünk. Az uberJar egy olyan Jar, mely egy teljesen önállóan futattható alkalmazást tartalmaz annak minden függőségével együtt.

Az alkalmazásszerver a Java EE API implementációkon felül még számos szolgáltatást biztosít a Netflix OSS-ből, amelyek a külső architektúra elemeinek megvalósítását támogatják. Mindezt Maven vagy Gradle build integráció segíti.

Az alkalmazás egy konténerben fut, mely a telepítés során a függőségeket figyelembe véve automatikusan felépül (bootstrap). A konfigurációs beállításokat az alkalmazásunk Main függvényében tudjuk módosítani.

A WildFly Swarm hiányossága azonban, hogy egyelőre a konfigurálhatósága az alkalmazás Main függvényéhez kötött, külső fájlból nem változtatható [20]. Ez fejlesztési környezetek közötti szállításánál hátrány lehet.

## Keretrendszerek

A szakirodalom számos keretrendszert ajánl, ha Java alapokon szeretnénk MSA rendszereket fejleszteni. Ezek egy része kifejezetten az eloszott rendszerek fejlesztésének támogatását segíti, mint például a Dropwizard. Egy másik részük a reaktív (aszinkron) működésű alkalmazások fejlesztését támogatja, mint például JavaRx, Vert.x.

Ezek közül kiemelekedik a KumulzEE, melyet 2015-ben a leígéretesebb új Java keretrendszernek minősítettek (Java Duke's Choice Award). Ezt a keretrendszert azért érdemes megemlíteni, mert fejlesztőinek kifejezett célja, hogy felhasználásával támogassák a monolitikus Java EE alkalmazások MSA alapokra migrálását.

A KumulzEE-vel szintén olyan alkalmazásokat tudunk fejleszteni, melyek kisméretű, a szükséges függőségeknél nem több könyvtárat tartalmazó JAR-okban futnak.

Az alkalmazásunk ez esetben egy beágyazott Jetty webszerveren fut.

A dokumentáció szerint ez az alkalamzás támogatja valamennyi Java EE könyvtárat és az ígérek szerint fokozatosan bővítik olyan könyvtárakkal, melyek támogatják az eloszott környezetben működő alkalmazások fejlesztését. Ilyenek például az Apache Kafka alapú elosztott eseményfolyam támogatása, eloszott környezetben történő authentikáció, monitorozás vagy naplózás.

A KumuluzEE jelenleg igen ígéretes keretrendszernek tűnik. Munkám során azonban nem fogom alklalmazni, mivel számos könyvtár még nem készült el vagy a fejlesztés korai fázisában van.

## Srping Keretrendszer

Mint ahogy a cím is mutatja, a Srping is valójában egy keretrendszer. Azért tárgyalom mégis külön pontban, mert a Spring napjainkban a Java EE platform legjelentősebb konkurenciája. A keretrendszert 2002-ben kezdte el fejleszteni a Pivotal szoftvercég.

A Spring is alkalmaz konténert, ennek mérete azonban lényegesen kisebb, mint a Java EE konténereké. Szerepe elsősorban a függőség injektálás és az alkalmazáskontextus kezelése. A Srping filozófiában a korábban említett uberJarok a kezdetektől fogva jelen voltak.

Ez jó alapot ad a Spring keretrendszer alkalmazásához az MSA világában.

A gyors alkalmazásfejlesztést a Spring Boot nevű modul teszi lehetővé, aminek segítségével könnyen készíthetünk önálló Spring alapú alkalmazásokat, amelyek beépülő Jetty vagy Tomcat szerveren futnak. A konfiguráció minden olyan esetben lehet automatikus, amikor konvenciókat alkalmazunk. Ezenfelül lehetőség van akár több konfigurációs profil közül is választani az alkalmazás indításakor. Ezeket a profilokat külső állományban is tárolhatjuk.

Rendelkezésre áll továbbá a Spring Cloud projekt. A felhő platform hasznos szolgáltatások és könyvtárak széles körét biztosítja, melyek jól hasznosíthatóak MSA alapú alkalmazásfejlesztés esetén. Ezek közül néhány fontosabb:

* Spring Cloud Bus: Szolgáltatások láncolását biztosítja eseményvezérelt alapon.
* Spring Cloud Connectors: Back-end erőforrások és felhő alapú szolgáltatások közötti kapcsolatok menedzselését támogatja.
* Spring Cloud Steam: Eseményvezérelt microservicek fejlesztését támogató mini keretrendszer.
* Spring Cloud Netflix: Netflix-OSS integrációt támogató keretrendszer. Segítségével a Netflix által megvalósított és nyíltá tett szolgáltatások alkalmazhatóak:
  + Eureka: Szolgáltatás felderítés.
  + Hystrix: Hiba detektálás, amellyel detektálhatóak és a hívási láncból kiiktathatóak a nem reagáló szolgáltatások.
  + Ribbon: Kliens oldali load balancer.
  + Zuul: Router filter.
* Spring Cloud Security: Segítségvel az authentikációs feladatokat eloszott környezetben is megvalósíthatjuk.
* Ezenfelül számos egyéb csomag áll rendelkezésre az ismertebb felhő alapú platfromok szolgáltatásainak integrálására (pl: Amzon AWS).

A Spring nem implemtálja az összes Java EE API-t, így például nincsen EJB implementációja, azonban a tapasztalatok azt mutatják, hogy nagyfokú az átjárhatóság a két keretrendszer között. Számos hibrid projekt létezik az iparban.

A Spring architektúra kiforrottsága és dokumentációjának gazdagsága miatt az általam készítendő microservice architektúrájú alkalmazást a Spring Boot és Spring Cloud keretrendszerekkel fogom megvalósítani.

# Irodalomjegyzék

|  |
| --- |
| [1] Alant, A.: Writing Microservice SLAs, (<https://team.goodeggs.com/writing-microservice-slas-eb7cfa387b31> ), utoljára megtekintve: 2017.05.06 |
| [2] Ambler W., S.: We Need More Agile IT Now!, (<http://www.drdobbs.com/architecture-and-design/we-need-more-agile-it-now/240169361?queryText=Release%2Bmanagement> ), utoljára megtekintve: 2017.05.06. |
| [3] Balogh, P., Berényi, Zs., Cévai, I., et al.: Szoftverfejlesztés Java EE platformon, SZAK Kiadó, 2007 |
| [4] Beck, K., Grenning, J., Martin, R., et al: Manifesto for Agile Software Development, (<http://agilemanifesto.org> ), utoljára megtekintve: 2017.05.06. |
| [5] Bonér, J., Farely, D., Kuhn, R., Thompson, M.: The Reactive Manifesto (<http://www.reactivemanifesto.org> ), utoljára megtekintve: 2017.05.06. |
| [6] Conway, M.: How Do Committees Invent?, Thompson Publications Inc, 1968 |
| [7] Coulouris, G., Dollimore, J., Kindberg, T., Blair. G.: Distributed System, Concepts and Design, Addison-Wesley, Ötödik kiadás |
| [8] Daya, S., Van Duy, N., Eati, K., et al.: Microservices from Theory to Practice, RedBooks, 2015 |
| [9] Eisle, M.: Developing Reactive Microservices, Enterprise Implementation in Java, O’Reilly, 2016 |
| [10] Eisle, M.: Modern Java EE Design Patterns, Building Scalable Architecture for Sustainable Enterprise Development, O’Reilly kiadó, 2015 |
| [11] Evans, E.: Domain-Driven Design Reference, Definitions and Pattern Summaries, Domain Language Inc, 2015 |
| [12] Fowler, M., Lewis, J.: Microservices, a definition of this new architectural term, (<https://martinfowler.com/articles/microservices.html> ), utoljára megtekintve: 2017.05.06. |
| [13] Fowler, M.: MicoservicePremium, (<https://martinfowler.com/bliki/MicroservicePremium.html> ), utoljára megtekintve: 2017.05.06. |
| [14] Hoff, T.: Amazon Architecture, (<http://highscalability.com/blog/2007/9/18/amazon-architecture.html> ), utoljára megtekintve: 2017.05.06. |
| [15] <https://www.nist.gov/news-events/news/2011/10/final-version-nist-cloud-computing-definition-published> ), utoljára megtekintve 2017.05.06. |
| [16] Lewis, J: Sacrifical Architecture, (<https://martinfowler.com/bliki/SacrificialArchitecture.html> ), utoljára megtekintve: 2017.05.06. |
| [17] Miller, M.: Innovate or Die: The Rise of Microservices, (<https://blogs.wsj.com/cio/2015/10/05/innovate-or-die-the-rise-of-microservices/> ), utoljára megtekintve: 2017.05.06. |
| [18] Nadareishvili, I., Mitra, R., McLarty, M., Amudsen, M.: Microservices Architecture, Aligining Principles, Practicies, and Culture, O’Reilly, 2016 |
| [19] Ollife, G.: Microservices: Building Services with the Guts on the Outside, (<http://blogs.gartner.com/gary-olliffe/2015/01/30/microservices-guts-on-the-outside/> ), utoljára megtekintve: 2017.05.06. |
| [20] Posta, C.: Microservices for Java Developers, A Hands-On Introduction to Frameworks & Containers, O’Reilly, 2016 |
| [21] Richards, M.: Software Architecture Patterns, O’Reilly kiadó, 2015 |
| [22] Richardson, C., Smith, F.: Microservices from Design to Deployment, Nginx, 2016 |
| [23] Robinson, I.: onsumer-Driven Contracts: A Service Evolution Pattern (<https://martinfowler.com/articles/consumerDrivenContracts.html> ), utoljára megtekintve: 2017.05.06. |
| [24] Shoup, R., Pirtchett, D.: The eBay Architecture, (<http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf> ), utoljára megtekintve 2017.05.06. |

1. angol eredeti: *„Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure.”* [↑](#footnote-ref-1)
2. Az angol software DEVelopment és information technology OPerationS szavakból képzett mozaikszó. [↑](#footnote-ref-2)
3. angol eredeti: *„Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.”* [↑](#footnote-ref-3)
4. angol eredeti: *„...the microservice architectural style is an approach to developing a single application as a suite of small services...”* [↑](#footnote-ref-4)