Milyen előnyökkel és hátrányokkal jár a Mikroszolgáltatás architektúra a monolitikussal szemben? Ismertesd az alábbi Microservice tervezési minták céljait azok előnyeit, hátrányait: Shared database, Database per service, API Composition, CQRS

DIA 13

A Monolitikus alkalmazás egy olyan alkalmazás, amely telepítési szempontból egyetlen egységként jelenik meg. De ez nem jelenti azt, hogy nem bonthatjuk az alkalmazásunkat fejlesztés során modulokra bontsuk. A modulra bontás nem befolyásolja a build-et, mivel a build után ezek az alkalmazások egyetlen telepitési egység áll elő. A monolitikus kialakitást az alkalmazásban az üzleti logika telepitési egységeinek száma határozza meg.

Ezzel szemben a Mikroszolgáltatás architektúrában több üzletti logikai együtt müködése a cél, ami általában HTTP-n keresztül történik.

A Mikorszolgáltatás architectúra ebből adodó elönye, hogy több kisebb kódbázis jön létre aminek a fejlesztése és karbantartása gyorsabb. Ezzel együtt egyes frissitések emiatt a kialakitás miatt sokkal rugalmasabban telepithetőek és készithetőek el. A kialakitása miatt a mükődésben sokkal rugalmasabb skálázhatóságot érhetünk el, mivel egyes üzleti logikát képesek vagyunk skálázni.

Ezenkivül az ilyen rendszerek egyes részei nem lesznek olyan robosztusak mivel ezeket megprobáluk különálló logikákra bontani. Viszont ennek a hátránya, hogy ezek között a logikák között nehezebb a kommunikáció megoldása mint a monolitikus alkalmazásokban.

Ezenkivül az ilyen alkalmazásokban amiatt, hogy szét bontunk miatt egy nehezebben üzemeltethető alkalmazást kapunk, ahol minden modulban

Monolitikus alkalmazás

- Monolitikus (= egy tömbből álló): olyan alkalmazás, amely *telepítési* szempontból egyetlen egységként jelenik meg
  - Nem zárja ki, hogy a fejlesztés során modulokra bontsuk, amelyek
    - Külön fejleszthetők, tesztelhetők, buildelhetők
    - Több alkalmazás között újrafelhasználhatók
  - De ezekből a modulokból a teljes alkalmazás buildje során egyetl telepítési egység áll elő
  - A backend üzleti logika telepítési egységeinek száma határozza meg a monolit jelleget → hiába futnak a kliensek és a DB külön gépen, attól az még monolit marad, ha a szerver oldali üzleti logik egyetlen telepítési egység
- Telepítési egység tipikus megjelenése:
  - ASP .NET web alkalmazás: zip fájl
  - Java webalkalmazás: .war (=Web ARchive) fájl (ez is zip)
    - Spring Boot esetén akár .jar is lehet

A monolitikus alkalmazások hátrányai

- Erősen gátolja az agilis fejlesztést és üzemeltetést, mert

1. Nagy kódbázis → **lelassult fejlesztés**
   - IDE-k lassulása
   - Build + tesztfuttatás lassulása
   - Nem megfelelő tesztlefedettség esetén félelem a módosítástól
   - Új fejlesztők lassú indulása (ha van dokumentáció, az is nagy!)
   - Párhuzamos fejlesztéseknél nagyobb koordináció szükséges

A monolitikus alkalmazások hátrányai

2. Nehézkes alkalmazás frissítések
   - Minimális módosításhoz is a teljes alkalmazásból kell új verziót telepíteni → az összes komponens működését megzavarja
   - Modern rendszereken naponta több frissítési igény
3. Limitált skálázhatóság
   - Vízszintes skálázásnál csak a teljes alkalmazást tudjuk új szerverekre telepíteni, akkor is, ha pontosan tudjuk, melyik komponensnek van szüksége több erőforrásra

ismétlődik a fejlesztési költsége például a CI/CD rendszereknek. Mert azok csak egyes modulokra használhatóak.

Ezenkívül az ilyen alkalmazásokban nehezebb a tranzakció kezelés, egyes szolgáltatások verziójának számon tartása, illetve kód szinten figyelni arra, hogy a kód ne legyen redundás és egyes logikákat különböző egységekben ne valósitsunk meg külön.

A monolitikus alkalmazások hátrányai

4. Új platformok, technológiák bevezetése nehézkes
   > Hiába tudnánk bizonyos funkciókat könnyebben megvalósítani más platformon/technológiával → az egész alkalmazást át kellene migrálni arra
5. Alacsony hibatűrés
   > Bármelyik komponens végzetes hibája (pl. végtelen ciklus, memória elfogyasztása, ...) a teljes alkalmazás leállásához vezet

Microservices előnyök

- A monolitikus architektúra agilitás útjába álló problémái megoldódnak

1. Kisebb kódbázis → gyorsabb fejlesztés
   > IDE-k gyorsak
   > Build + tesztfuttatás gyors
   > Könnyebb tesztekkel lefedni → bátrabb módosítás
   > Új fejlesztők gyorsan indulhatnak
   > Kevesebb koordináció szükséges, tipikusan egy fejlesztőcsapat/szolgáltatás

Microservices előnyök

4. Platformok, technológiák átjárhatósága
   > Egy szolgáltatásnál választott platform/technológia/nyelv nem köti meg a kezünket egy másik szolgáltatásnál → elköteleződés nélkül tudunk új technológiákat kipróbálni
   > Nem tipikus, de szükség esetén (pl. elavult technológia, rosszul menedzselhető, rossz minőségű kód) egy szolgáltatás teljes újraírása is vállalható méretű feladat
   > A hálózati kommunikációban választott protokollok terén nehezebb a váltás
5. Robusztusság
   > Egy szolgáltatás kiesése nem érinti a tőle nem függő funkciókat
   > + terheléselosztó mögött akár több példányban futhat ugyanaz a szolgáltatás

Microservices előnyök

2. Alkalmazás frissítések rugalmasabbak
   > A szolgáltatások külön frissíthetők
   > Hibás release esetén egyszerűbb a rollback is
3. Rugalmas skálázhatóság
   > Elég csak a szűk keresztmetszetként azonosított szolgáltatást skálázni, vagy csökkent terhelés esetén leskálázni → költséghatékonyság
     – Ráadásul új microservice példány indítása vagy meglévő leállítása gyorsabb
   > Pontosan tudjuk, melyik funkció üzemeltetése milyen költséggel jár → üzleti döntések

Microservices hátrányok

- A legtöbb probléma Microservices esetében a monolitikushoz képest jóval nagyobb mértékű *elosztottságból* adódik:

1. Komplexebb infrastruktúra és üzemeltetés
   > Minden szolgáltatásnak külön (virtuális) szerver példány, köztük hálózati kapcsolat ...
   > A szerverek nem is feltétlen egyformák, több technológiát kell ismerni az üzemeltetőknek
   > → fejlesztői és üzemeltetői szerepek közelítése, automatizálás, lásd DevOps
2. Minden szolgáltatásnál ismétlődő költségek a fejlesztési infrastruktúrában
   > Verziókezelő, CI/CD, dev/test/prod környezetek...
   > Automatizáció segíthet

Microservices hátrányok

3. Hálózati kommunikáció okozta problémák
   - A hálózati lassulás/kiesés újabb hibaforrás → az alkalmazásokat erre felkészítve kell fejleszteni
   - Gyors lokális hálózaton is nagyobb overhead, mint egy lokális metódushívás → a funkciók szolgáltatásokra való szétbontásakor ezt figyelembe kell venni
4. Redundáns logika implementációk
   - Nagyobb a veszélye, hogy a szolgáltatásokban a független fejlesztőcsapatok ugyanazt redundánsan implementálják → fontos a csapatok közti kommunikáció, közösen használható library/szolgáltatások azonosítása

Microservices hátrányok

5. Tranzakciókezelés
   - Több szolgáltatáson átívelő atomi tranzakciók nehézsége (lásd Saga tervezési minta)
6. Szolgáltatások verziózása
   - A szolgáltatások nyújtotta API-ban lehetnek módosítási igények
     - Az API és az összes függő szolgáltatás egyidejű frissítése? →külön telepítés előnyeit elveszítjük
     - Az API módosítás legyen mindig visszafelé kompatibilis? → torzíthatja az interfészt hosszú távon
     - Legyen az API-nak több párhuzamosan élő verziója, amíg minden kliens át nem áll? → a verziók menedzselése plusz komplexitás

Microservices hátrányok

7. Sok szolgáltatást használó funkciók fejlesztése komplex
   - Sok fejlesztőcsapat közti egyeztetést és tervezést igényel
   - Kulcskérdés a szolgáltatások közti határok helyes megválasztása

Ezekre a hátrányokra ma már rengeteg dekompoziciós minta van amik azért léteznek, hogy pontositsák, hogy ezeket az alkalmazásokat hogyan bontsuk szét ahhoz, hogy ezeket a hátrányokat redukáljuk.

A Shared database -ben több szolgáltatás egy azon adatbázist használ a tranzakciók könnyitése végett. Ennek a mintának az elönye, hogy egyszerübb less az üzemeltetés és könnyebbek a lekérdezések mivel az ACID tranzakciók használhatóak a megszokott modón. Viszont ennek a mintának a hátránya, hogy függőssségeket teremtünk egyes szolgáltatások között és ennek hála fejlesztési időben állandó séma egyeztetés kell, illetve futtatási időben előfordulhat, hogy egy szolgáltatás blokkol egy másik szolgáltatást. Emiatt ezt a mintát főként anti pattern-ként szokták emlegetni.

A Databas per service mintában minden szolgáltatásnak külön adatbzisa van amiben dolgozik. Ezzel a szolgáltatások teljesen függetlenek egymástól és nem blokkolják egymás mivel mindegyik a neki megfelelő DB-t használja. Ennek viszont a hátránya, hogy több adatbázist kell egyszerre karbantartani. Illetve megvalósitási szinten nem egyértelmü a tranzakciók megvalósitása és az összetett adatok lekérése. Ez alatt lehet érteni azt ha JOIN által több tábla egyben van.

Dekompozíciós minták

- Fő kérdés: hogyan bontsuk szét szolgáltatásokra a rendszert?
  - Nem megfelelő szétbontás → a Microservices architektúra előnyök kevésbé, és/vagy a hátrányok erősebben érvényesülnek
- Célok:
  - Legyen elég kicsi a Microservice, hogy egy kisebb csapat elég legyen a fejlesztésre, tesztelésre
  - Kövessük az OO fejlesztésből már ismert Single Responsibility elvét: tartsuk egyben, ami ugyanazon okok miatt változhat, és válasszuk el, amik más okokból
    - Ha egy új vagy módosult üzleti igény több szolgáltatást érint,

Adatkezelési minták

- **Shared database**
  - Több szolgáltatás közös adatbázist használ
- Előnyök:
  - Egyszerűbb üzemeltetés
  - Join, ACID tranzakciók a fejlesztők által megszokott módon működnek
- Hátrányok:
  - Függőséget teremtünk a szolgáltatások között
    - Fejlesztési időben: egyeztetni kell a séma módosításokat az összes fejlesztőcsapat között
    - Futási időben: pl. egyik szolgáltatás által hosszan tartott zár blokkolhat egy másik szolgáltatást
    - Az egy közös DB-vel nem tudunk minden szolgáltatás igényeihez egyedileg igazodni (pl. indexek, vagy a DB típusa tekintetében)
  - A hátrányok miatt többen anti-patternnek tekintik Microservices környezetben, de bizonyos helyzetekben lehet, hogy nincs jobb megoldás
- Alternatíva: **Database per service**

Adatkezelési minták

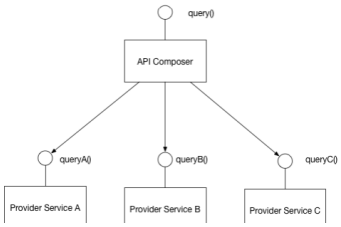
- **Database per service**
  - Minden szolgáltatás saját adatbázist használ
    - Lehet egy séma, de külön privát táblák (szolgáltatásonként más DB user)
    - Lehet egy DB, de más sémák
    - Lehet külön DB
- Előnyök:
  - Szolgáltatások függetlenek egymástól
  - Minden szolgáltatás neki megfelelő DB-t használhat
- Hátrányok:
  - Több DB üzemeltetésének overheadje
  - Nem triviális:
    - szolgáltatásokon átívelő tranzakciók megvalósítása
    - join különböző szolgáltatások adatai között
- A szolgáltatások közti joinra, de különösen a tranzakcióra mutatkozó igény mellesleg jelezheti a nem megfelelő dekompozíciót
  - De ha mégis szükségesek, és mindenképpen kerülni akarjuk a Shared database-t külön minták foglalkoznak vele

Az API Composition mintában létrehozunk egy szolgáltatást ami azért felelős, hogy több szolgáltatás által uralt adatott összegyüjtse és azt egyben oda adja. Ez megoldás arra, hogy a JOIN által egyesitett adatokat, hogyan kapjuk meg. Ennek pontosan az a célja, hogy több szolgáltatás közti join-ra megoldást adjon a Database per service mintában.

A Command Query Responsibility Segregation azaz CQRS mintának az a célja, hogy szolgáltatások adatai közti join-t megoldja. Ez szintúgy egy jó megoldás lehet a Database per service minta használata mellett. Ennek megvalósitásának céljából készitünk egy view adatbázist amely a microservice által igényelt join query-ket megvalósitja. Ezeket majd az adatokat tulajdonló szolgáltatások töltenek event alapon.

Adatkezelési minták

- **API Composition**
  - Cél: szolgáltatások adatai közti join megoldása Database per service alkalmazása esetén
- Megoldás: egy új szolgáltatás legyen felelős azért, hogy a több szolgáltatás által tulajdonolt adatot összegyűjtse



Adatkezelési minták

- **Command Query Responsibility Segregation (CQRS)**
  - Cél: szolgáltatások adatai közti join megoldása Database per service alkalmazása esetén
- Megoldás:
  - vezessünk be egy “view” adatbázist,
  - amely fölött egy microservice a join-t igénylő query-ket megvalósítja
  - és amelyet az adatokat tulajdonló szolgáltatások töltenek event alapon