**Milyen előnyökkel és hátrányokkal jár a Mikroszolgáltatás architektúra a monolitikussal szemben? Ismertesd az API Gateway tervezési mintát, és a tanult Service Discovery mintákat!**

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

**Az API Gateway minta arra ad megoldást ha szolgáltatásokat kell lekérni és ezeknek a szolgáltatásoknak a helye változik, vagy esetleg a kliensek elől szeretnénk a fukció szolgáltatások közötti szervezést elrejteni. Illetve ez megoldást adhat arra is ha több különböző tipusú klienst szeretnénk megvalósitani és ezek eltérő adatokat kérnek vagy eltérő adottságú hálozaton keresztül kérnek valamit.**

**Ebben a mintában bevezetünk egy API Gateway belépési pontot ami majd a kliensek és a szolgáltatások között kommunikál. Ezzel egyetlen belépési módot megadva a kliensek számára. Ugyanakkor ennek a variációja a Backend for frontends ahol több ilyen pontot adunk meg a kliens tipusonként.**

**Ennek a mintának az a előnye, hogy a kliensek és a mikroszolgáltatások ennek köszönhetően lazán vannak csatolva. Illetve a klienseknek nem kell keresniük a megfelelő szolgáltatást hanem az API Gateway oda irányitja őket. Ennek viszont hátránya, hogy minimum plussz egy komponenst és hálozati ugrást kapp az alkalmazásunk.**

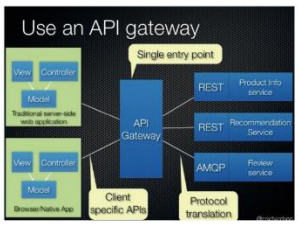
**Ebben a mintában a megoldandó feladatok amikre figyelni kell az Autentikáció, Szolgáltatások megkeresése, Kieső szolgáltatások megkeresése.**

## API Gateway

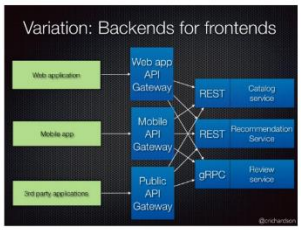
- Microservices architektúra kliensei (pl. SPA, mobil) hogyan érjék el az egyes szolgáltatásokat? Probléma:
  - > Tipikusan több szolgáltatást kell elérni
  - > Ezen szolgáltatások helye változik (példányok indulnak, leállnak, dinamikus IP-vel)
  - > A funkciók szolgáltatások közti átszervezését el akarjuk rejteni a kliensek elől
  - > Különböző kliensek
    - Eltérő adatokat kérnek
    - Eltérő adottságú hálózaton keresztül érkeznek
  - > Bizonyos szolgáltatások eléréséhez esetleg protokollt kell váltani

## API Gateway

- **API Gateway**: vezessünk be egyetlen belépési pontot a kliensek számára




- Lehetséges variáció: **Backends for frontends**
  - > Klienstípusonként külön API gateway




## API Gateway

- Előnyök
  - > Kliens és mikroszolgáltatások laza csatolása
  - > Kliensek mentesülnek az egyes szolgáltatások megtalálásának feladatától
  - > Kliensenként optimális API biztosítható
    - Pl. kevesebb kliens-szerver roundtrip
  - > Megvalósíthat protokollváltást
- Hátrányok
  - > Plusz egy fejlesztendő, üzemeltetendő komponens
  - > Plusz egy hálózati ugrás
- Az API Gateway-nél megoldandó feladatok:
  - > Autentikáció (tipikusan token alapú → **Access Token** minta)
  - > Szolgáltatások megkeresése (lásd **Service discovery** minták)
  - > Kieső szolgáltatások kezelése (lásd **Circuit breaker** minta)
  - > Nagy eséllyel alkalmazza az **API Composition** mintát

**A Service Discovery minták arra adnak megoldást, hogy a kliens hogyan és honnan éri el mégis a szolgáltásokat.**

**Erre egyik megoldás a Service Registry ami számon tartja, hogy egyes szolgáltatásokat melyik IP cimen érhetjük el. Itt viszont nem egyértelmü, hogy hogyan regisztrálódnak be ide a szolgáltatások nevei és IP cimei.**

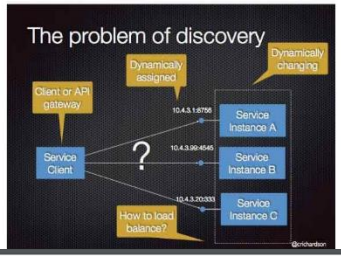
**Erre az egyik megoldást a Self Registration ahol a szolgáltatás induláskor beregisztálja magát ebbe a komponensbe. Ennek előnye, hogy a komponens teljesen tisztában van az állapotával, viszont ennek hátránya, hogy minden szolgáltatásnak ismernie kell a Service Registry komponenst. Illetve ennek hátránya, hogy minden platform-hoz és szolgáltatáshoz szükséges egy kiregisztrálást is megvalósitani. Ugyanakkor ami a legnagyobb hátrány az, hogy a komponens hibásan indul vagy futtás időben hiba lép fell akkor a kiregisztrálást képtelen less megtenni.**

**A másik megoldást a 3rd party registration ahol egy külön komponenst felel azért, hogy regisztálja ezeket a komponenseket. Ennek elönye, hogy a szolgáltatások kódja nem trehelődik ilyen logikával, viszont igy egy külön komponenset kell karban tartni amit nagy rendelkezésre állással kell szolgáltatni. Ezen kivül a komponens csak azt lesz képes megmondani, hogy futt-e a komponens vagy sem.**

**Ezeken kívül még használható a Client-side service discovery, ahol a kliens keres a service registryben és végzi a terhelés elosztást. Ebben a mintában kevesebb a hálozati ugrást, ugyanakkor a kliens függeni fogg a service registry-től. Illetve a service registry felé kommunikációt és terhelés megosztást meg kell valósitani minden platformra.**

## Service discovery minták

- Honnan tudja egy kliens, hol éri el a szolgáltatást?
  - Nem triviális, mert a szolgáltatásból több példány lehet, leállnak/újak indulnak, köztük terheléskiegyenlítésre van szükség, dinamikusan kapnak IP-t




## Service discovery minták

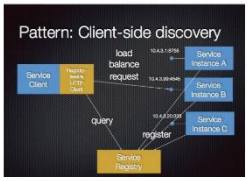
- **Service registry**
  - Legyen egy szolgáltatás, ahol nyilvántartjuk a futó szolgáltatás példányokat (név-IP páros)
  - Hogyan regisztrálódnak ide be a szolgáltatás példányok? → két alternatív minta
- **Self registration**: minden szolgáltatás maga felelős, hogy induláskor beregisztrálja, leálláskor kiregisztrálja magát a service registrynél
  - Előny: a szolgáltatás mindig pontosan ismeri a saját állapotát
  - Hátrány:
    - Minden szolgáltatásnak ismerni kell a Service registry-t
    - Minden szolgáltatás platformhoz/technológiához szükséges megvalósítani a (de)regisztrációs logikát
    - Ha a szolgáltatás abnormális módon áll le, vagy váratlaon okok miatt nem képes kérések kiszolgálására, valószínűleg deregisztrálni sem tudja magát

## Service discovery minták

- **3rd party registration**: külön komponens (registrar) felelős minden szolgáltatás (de)regisztrációjáért
  - Előnyök:
    - a szolgáltatás kódja nem terhelődik a (de)regisztrációs logikával
    - A registrar health check kérések eredményei alapján is tud (de)regisztrálni
  - Hátrány:
    - Csak running/not running állapotokat tud regisztrálni
    - Külön komponens, amit menedzselni kell, ráadásul magas rendelkezésre állással

## Service discovery minták

- Hogyan használjuk a service registryben tárolt információt? → 2 minta
- **Client-side service discovery:** a kliens keres a service registry-ben, és végzi a terheléselosztást is
  - Előny
    - Kevesebb hálózati ugrás
  - Hátrány
    - A kliens függ a service registry-től
    - Minden kliens platformra/technológiára meg kell valósitani a service registry felé a kommunikációt és a terheléselosztó logikát



**Ehhez van még egy hasonló minta és ez a Server-side service discovery ahol a szerver oldalon van a terhelés elosztó ami általában egy router. Ekkor az ő felelőssége a szolgáltatás példányok közti terhelés elosztás és a service registryben való keresés. Ennek a mintának előnye, hogy a kliens oldal megmarad egyszerűbbnek, illetve erre már sok felhő platform már kész megoldást nyújt. Ugyanakkor erre nincs beépitett megolást vagy plussz egy komponens vagy magas rendelkezésre állás. Illetve itt több hálozati ugrás is lehet és ha nem TCP szinü a router akkor lehet több protokollt is támogatni kell, mint például a HTTP.**