Microservices

Imre Gábor

Q.B224

gabor@aut.bme.hu

Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

- Monolitikus alkalmazások
- SOA
- Microservices
- Elterjedt tervezési minták Microservice architektúrákban
- Konténerizáció

Monolitikus alkalmazások

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 egyetlen 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 logika 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

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

Szolgáltatásorientált architektúra

Szolgáltatásorientált architektúra

- SOA (=Service Oriented Architecture)
- A 2000-es években terjedt el, egyik fő célja a monolitikus alkalmazások hátrányainak kiküszöbölése
- Alapötlet: az alkalmazások olyan lazán csatolt *szolgáltatásokból* épüljenek fel, amelyek
  - Hálózaton elérhetők, nyílt protokollokon keresztül
  - Jól definiált interfészekkel rendelkeznek
  - Kliensei számára “fekete dobozok”
  - Önállóan telepíthetők
  - Elemi, vagy más szolgáltatásokat komponáló összetett szolgáltatások lehetnek

Szolgáltatásorientált architektúra

- A SOA alapelvek nem kötik meg a konkrét hálózati protokollokat, de a megjelenése idején a legtöbben SOAP + WSDL alapú XML webszolgáltatásokkal implementálták
- A kezdeti lelkesedést követő sikertelen projektek + a 2000-es évek végi gazdasági válság miatt nem terjedt el a várt mértékben
- A SOA tapasztalatokra is építve a 2010-es években jelent meg a Microservices architektúra (MSA)

Microservices

Microservices alapelvek

- Alapötlet: a független (autonóm) szolgáltatásokból való építkezés jó irány, de a SOA-ban a szolgáltatások mérete tipikusan túl nagy volt, ez vezetett problémákhoz → építkezzünk kisebb szolgáltatásokból, amelyek
  - Jól körülhatárolt üzleti logikát valósítanak meg
  - Saját processzben futnak
  - Külön telepíthetők
  - Pehelysúlyú protokollokon hívhatók (tipikusan REST API)

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

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 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 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

Microservices tervezési minták

Forrás: https://microservices.io/patterns

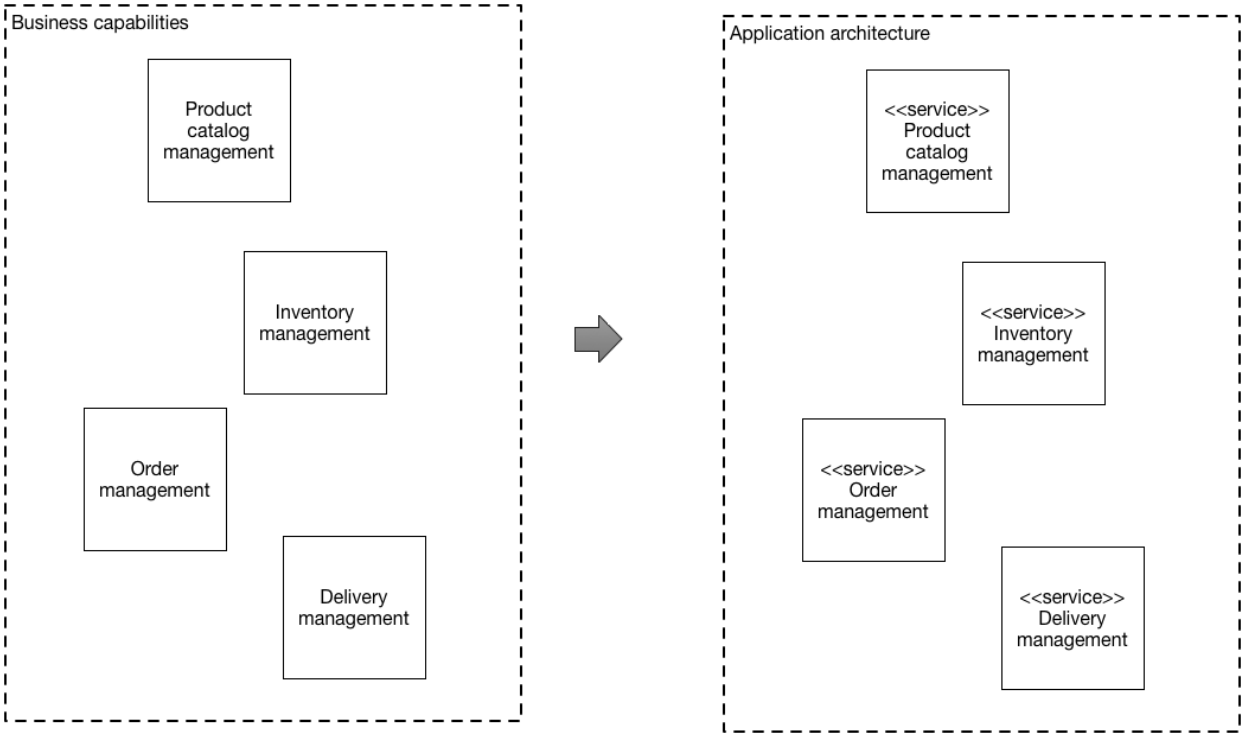
Előzményei: SOA tervezési minták, EAI tervezési minták, Elosztott rendszerek tervezési mintái

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, az mindig komplexebb fejlesztéssel + telepítéssel jár

Dekompozíciós minták

- **Decompose by business capability**
  - Üzleti tevékenységek/képességek mentén azonosítsuk a szolgáltatásokat, pl.



Dekompozíciós minták

- Az üzleti tevékenység felmérése szükséges
- Lehetséges kiindulási pontok (de nem mindig esik egybe egy jó felbontással):
  - Szervezeti struktúra
  - Domain model

Adatkezelési minták

- Kérdés: milyen DB architektúrát alkalmazzunk Microservices környezetben?
- Szempontok:
  - A szolgáltatások maradjanak lazán csatoltak
  - Felmerülhet az igény, hogy egy üzleti funkció több, különböző szolgáltatások által kezelt adatot módosítson vagy kérdezzen le
    - Akár join-ra is szükség lehet
  - Különböző szolgáltatásokhoz különböző típusú DB passzolhat jobban (relációs vagy NoSQL: dokumentum/kulcs-érték/gráf)

Adatkezelési minták

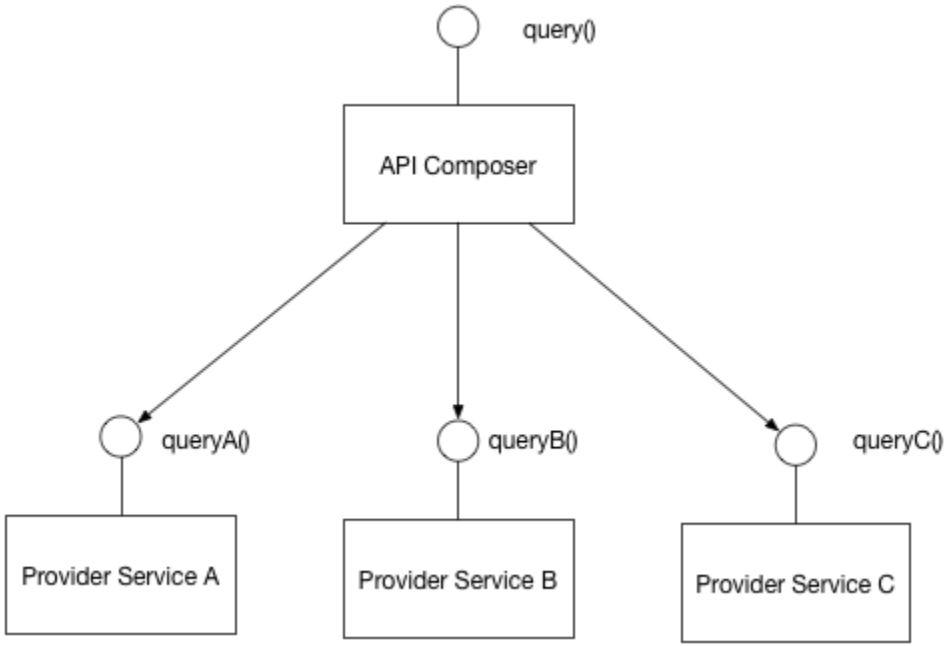
- **Shared database**
  - Több szolgáltatás közös adatbázist használ
- Előnyök:
  - Egyszerűbb üzemeltetés
  - Join, ACID tranzakciók a fejlesztők által megszokott módon működnek
- Hátrányok:
  - Függőséget teremtünk a szolgáltatások között
    - Fejlesztési időben: egyeztetni kell a séma módosításokat 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

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

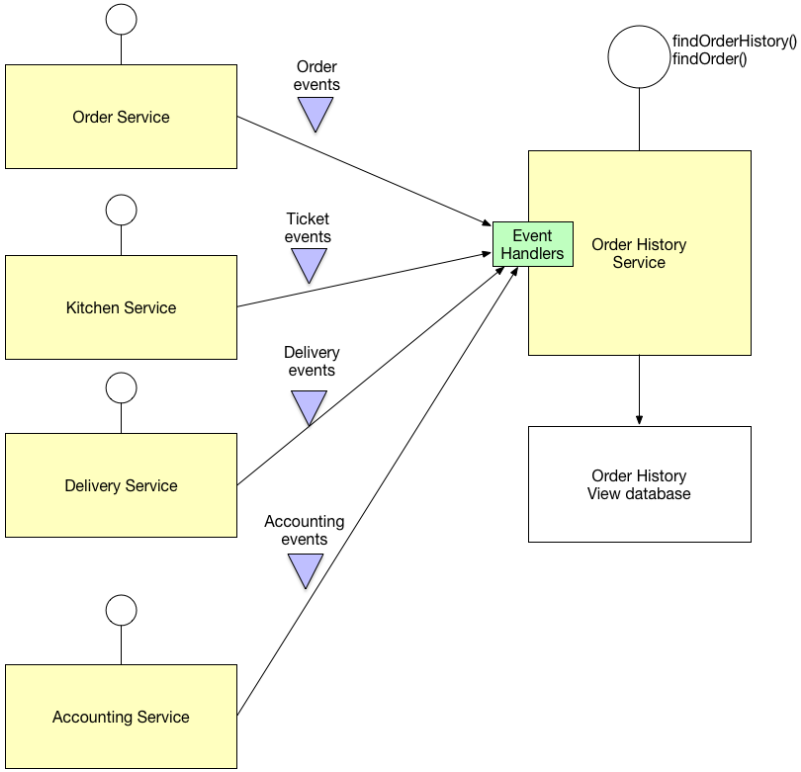


Adatkezelési minták

- Előnyök: Egyszerű
- Hátrány:
  - Nagy mennyiségű adat in-memory joinja nem hatékony
- Alternatíva: **CQRS**

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



Adatkezelési minták

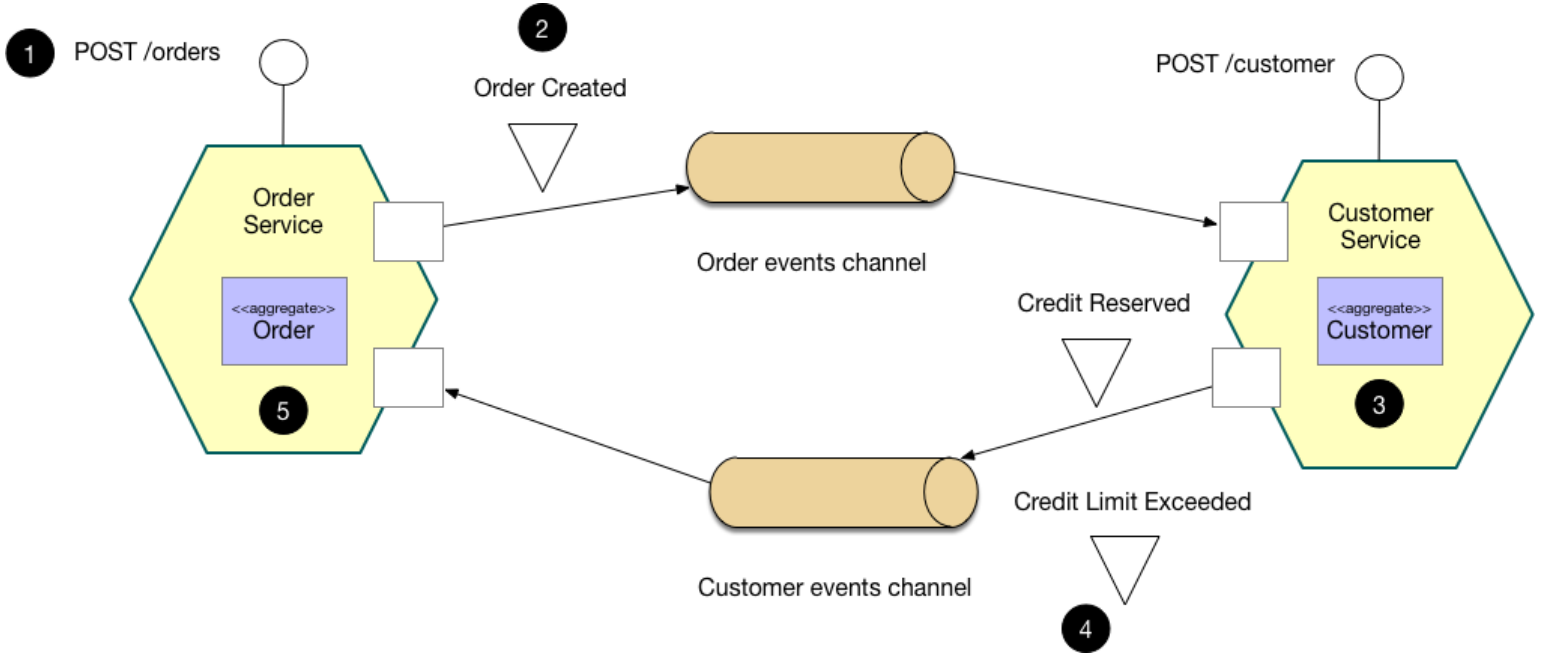
- Előnyök:
  - A view DB-ben triviális a join,
  - Sőt, ha a szükséges lekérdezésekre optimalizált, denormalizált sémát vezetünk be, lehet, hogy nem is lesz szükség a joinra
- Hátrány:
  - Plusz komplexitás
  - Esetleges kód duplikáció
  - Replikációs késleltetés/esetleges konzisztencia

Adatkezelési minták

- **Saga**
  - Cél: szolgáltatásokon átívelő adatmódosításokat akarunk úgy egybefogni, hogy ha valamelyik módosítás sikertelen, a már elvégzett módosítások visszagördüljenek
  - Fontos szempont: a hagyományos ACID tranzakciót megvalósító 2 fázisú kommitot REST környzetben tipikusan kerülik az állapottárolás szükségessége, komplexitása és a hosszú futás miatt

Adatkezelési minták

- Megoldás
  - A saga lokális tranzakciók egy sorozata
  - Minden tranzakcióhoz definiált a művelet inverzét elvégző kompenzáló tranzakció is
  - A lokális tranzakciókat végrehajtó szolgáltatások eventeken keresztül értesítik egymást
    - Siker esetén indulhat a következő lokális tranzakció
    - Hiba esetén kompenzálni kell az összes korábbit



Adatkezelési minták

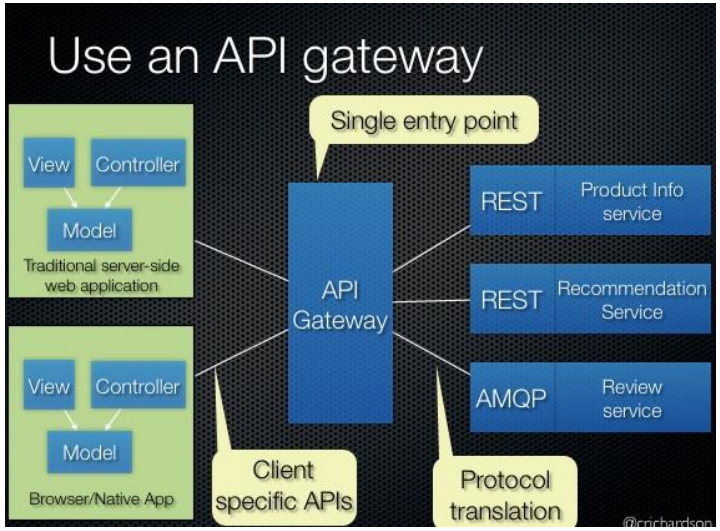
- Megjegyzések:
  - A fenti példa egy ún. Koreografált Saga: minden lokális tranzakció eventet dob, ami más lokális tranzakciókat triggerel
  - Egy másik opció az Orkesztrált Saga: külön orkesztrátor küldi a tranzakciókat indító eventeket, minden tranzakció a saga-nak küld válaszüzenetet
  - A robusztus megoldás érdekében a lokális tranzakció az üzenetküldéssel atomi kell legyen, de a DB-t és a message brokert sem akarjuk 2-fázisú committal összefogni → erre is külön minták
    - **Event sourcing**
    - **Transactional Outbox**
- Előnyök:
  - Elosztott tranzakciók nélkül megoldja a szolgáltatások közti adatkonzisztenciát
- Hátrány:
  - Plusz komplexitás (pl. minden tranzakcióhoz kompenzáló tranzakció)
  - Nem szigorú konzisztencia: egy folyamatban lévő saga minden már sikeresen lefutott lokális tranzakciója látható a userek számára (illetve ezek a módosítások eltűnnek, ha egy későbbi lépés kompenzációt triggerel)

API Gateway

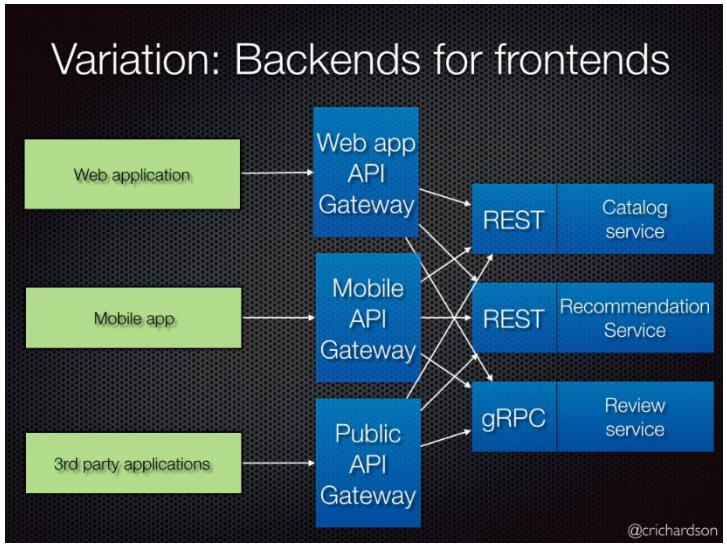
- Microservices architektúra kliensei (pl. SPA, mobil) hogyan érjék el az egyes szolgáltatásokat?
  Probléma:
  - Tipikusan több szolgáltatást kell elérni
  - Ezen szolgáltatások helye változik (példányok indulnak, leállnak, dinamikus IP-vel)
  - A funkciók szolgáltatások közti átszervezését el akarjuk 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

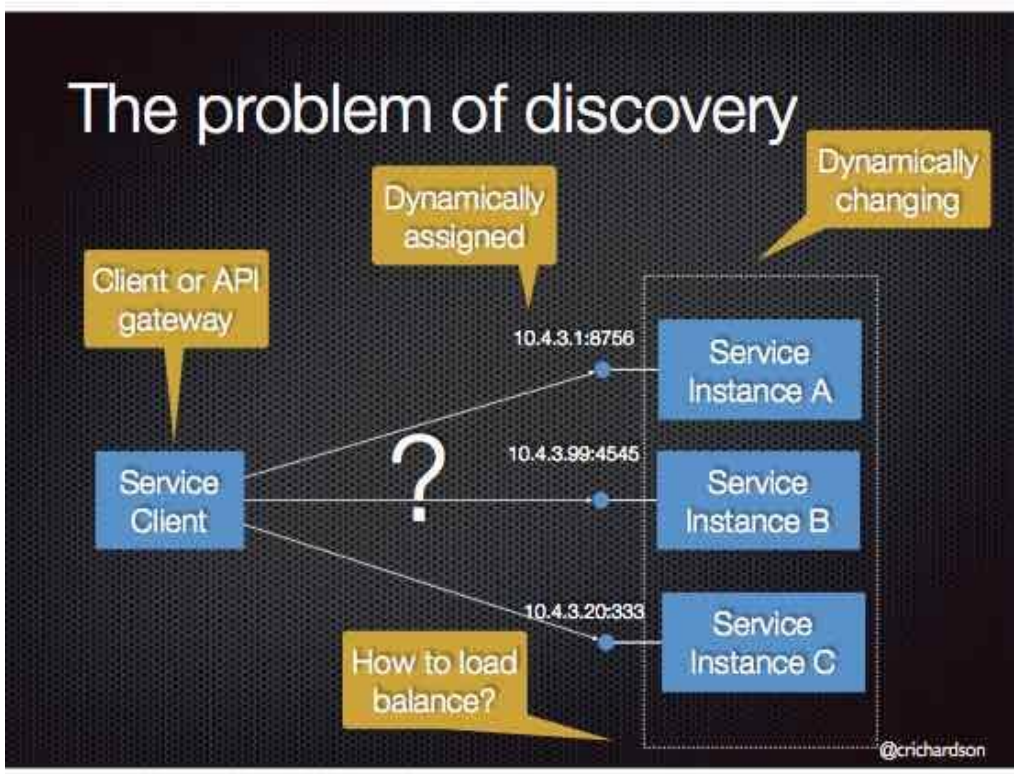


API Gateway

- 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

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

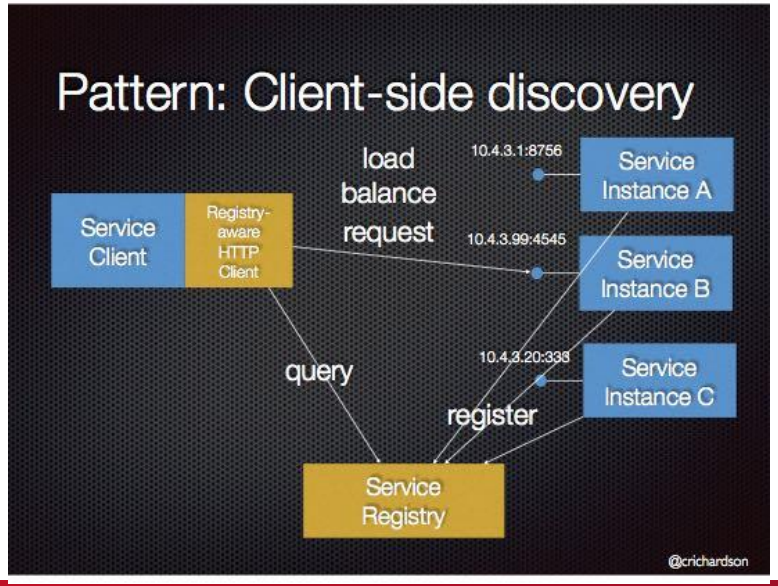
- **Service registry**
  - Legyen egy szolgáltatás, ahol nyilvántartjuk a futó szolgáltatás példányokat (név-IP páros)
  - Hogyan regisztrálódnak ide be a szolgáltatás példányok? → két alternatív minta
- **Self registration**: minden szolgáltatás maga felelős, hogy induláskor beregisztrálja, leálláskor 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

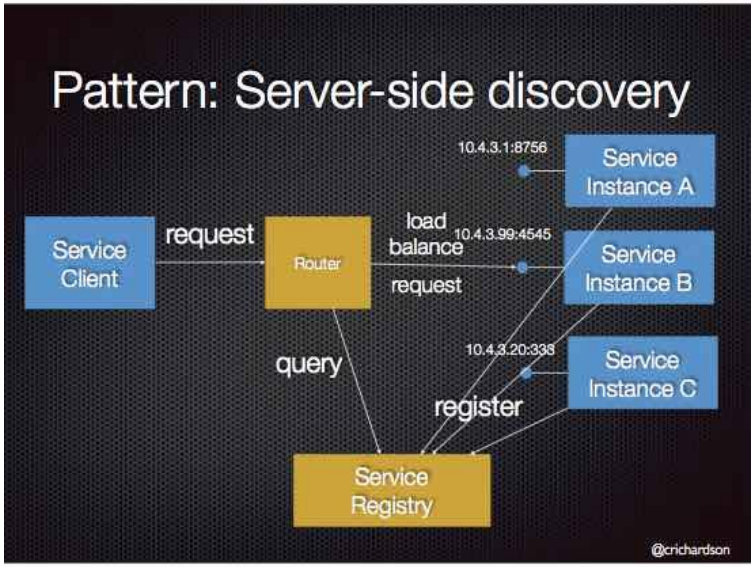
Service discovery minták

- 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 valósítani a service registry felé a kommunikációt és a terheléselosztó logikát



Service discovery minták

- **Server-side service discovery:** szerver oldalon van a terheléselosztó (router), az ő felelőssége a szolgáltatás példányok közti terheléselosztás és a service registrybeli keresés
  - Előny
    - Kliens oldal egyszerűbb
    - Sok felhő platformon kész megoldás
  - Hátrány
    - Ha nincs beépített megoldás, plusz egy komponens, magas rendelkezésre állással
    - Több hálózati ugrás
    - Ha nem TCP szintű a router, lehet, hogy több protokollt kell támogatnia (HTTP, Thrift, gRPC…)



Circuit Breaker

- Probléma: ha egy szinkron módon hívott szolgáltatás nem elérhető, az a hívó számára csak egy timeout lejárta után derül ki. Addig viszont a hívó oldali szál is blokkolódik. Ha sok kérés fut bele ilyen timeoutba, a hívó oldali szálak elfogyhatnak, és a hívó a kieső szolgáltatástól nem függő funkcióit sem tudja ellátni → egy szolgáltatás kiesése végiggyűrűzhet akár az összes szolgáltatáson
- Megoldás:
  - Egy olyan proxyn keresztül hívjuk a szolgáltatásokat, amely szükség esetén "megszakítja az áramkört", vagyis adott limit feletti sikertelen kérés után meg sem próbálja hívni a kiesett szolgáltatást, hanem azonnal hibával tér vissza
  - + időnként egy-egy kérést mégis átenged, hogy kiderüljön, ha közben feléledt a kieső szolgáltatás → ha ezt észleli, onnantól visszaáll a normál üzem

Microservice chassis

- Látható, hogy az eddig minták jó része gyakorlatilag elengedhetetlen egy működőképes Microservices architektúrához
  - + még nem említett minták: konfiguráció kezelése, naplózási kérdések, health check, metrikák
- Probléma: ha ezt mind magunk valósítjuk meg, az rengeteg erőforrást von el a konkrét alkalmazás fejlesztésétől
- Megoldás: alkalmazzunk valamilyen Microservices keretrendszert (chassis = alváz), amely számos mintát megvalósít

Egy microservice keretrendszer: Spring Cloud

- A Spring Boot-ban megismert fejlesztői modelt követi: annotációk + properties/yaml alapú konfig
- Sok Netflix által fejlesztett open source komponenst integrál, majd 2018 végétől ezek helyett más megoldásokat támogat
- Néhány annotáció + konfig révén készen kapunk
  - Az összes szolgáltatás konfigurációját központilag kezelő config server
  - Service registry (Eureka, majd Consul)
  - API gateway (Zuul, majd Spring Cloud Gateway)
  - Circuit breaker (Hystrix, majd ResilienceJ)
  - Client-side service discovery (Ribbon, majd Spring Cloud Loadbalancer)

Konténerizáció

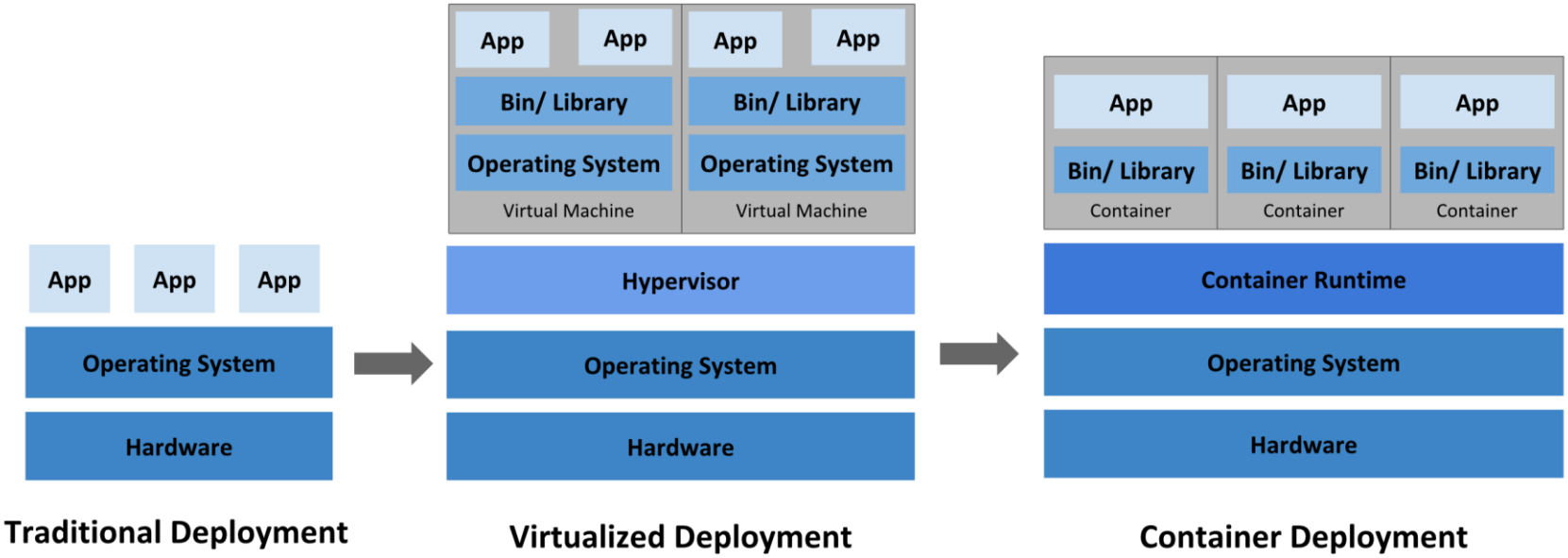
Microservices architektúra üzemeltetése

- Hova telepítsük a microservice-eket?
  - Mindegyiket külön fizikai szerverre? → pazarlás
  - Több microservice-t egy fizikai gépre? → nem megfelelő izoláció (pl. library verzió ütközések, erőforrások dedikálása, végzetes hibák)
  - Külön virtuális gépekre? → izoláció OK, de a VM futtatáskor nagyobb overheaddel jár, létrehozása is több idő, a példányok ki-/bekapcsolása nem elég gyors
  - Külön konténerekbe → izoláció is megvan + kb. 100x gyorsabb az alkalmazás becsomagolása, mint egy virtuális gépnél
- → A microservice-eket leggyakrabban konténerekbe telepítik, ezek közül a legelterjedtebb a Docker

Konténerek jellemzői

- Rugalmas: Bármilyen alkalmazást belerakhatunk, mindegy milyen bonyolult
- Gyors: A gazda gép kernelén osztozik
- Hordozható: Ha van egy docker image-ünk, akkor azt használhatjuk lokálisan, a felhőben vagy bárhol máshol is
- Skálázható: Használható automatikus skálázásra is
- Legózható: Szolgáltatások egymásra pakolhatóak könnyen

Telepítés konténerekbe



Konténerek előnyei

- Közvetlenül az operációs rendszeren fut, megosztva használja a gazda gép kerneljét
- Gyorsan elsajátítható: lényegében a VM-hez hasonló nagyon
  - Saját fájlrendszer, környezeti változók, CPU, memória, folyamat menedzsment, stb …
  - Saját hálózat!
- Sokkal könnyebb kezelni mint VM-eket
- Ugyanaz a környezet előállítható könnyedén egy laptopon, mint az éles környezet
- Ha kell egy fejlesztő eszköz, nem kell a gazda gépre telepítenünk
  - Nincs környezeti változó állítgatás
  - Nem marad szemét utána

Docker fogalmak

- DockerFile: leíró fájl, meghatározza, hogy miből álljon az Image
- Image: Végrehajtható csomag, ami tartalmaz mindent az alkalmazás futtatásához
  - Kód
  - Könyvtárak
  - Runtime
  - Konfigurációs és környezeti változók
- Container: Egy image futó példánya

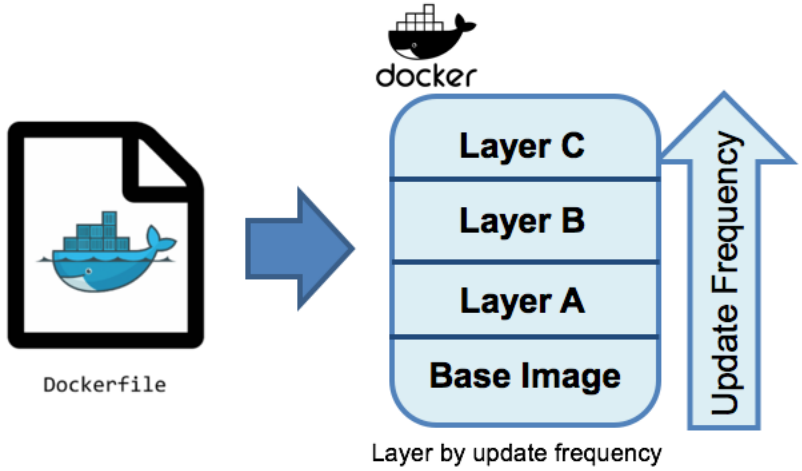
Docker Hello World

- Docker file

```
FROM openjdk:11
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac Main.java
CMD ["java", "Main"]
```

Docker image

- Hasonlóak a VM imagekhez
- Minden Image rétegekből áll
- A DockerFile tartalmazza a Base Image-et
- Minden új tartalom új réteget generál
- Minden réteg cache-elődik
- Csak a változásokat követi a build
  - Csökkenti a build időt
  - Kevesebb tárhelyet igényel
- A rétegek sorrendje számít



Docker image megosztása

- Docker Repository – online repository, ahonnan az image-ek letölthetők
- DockerHub – hivatalos repository (http://hub.docker.com):
  - MySQL, Redis, Node.js, Tomcat, Java, Ubuntu, Wordpress, Elasticsearch …
  - És rengeteg nem hivatalos
- Egyszerűen kezelhető parancssorból
  - docker tag
  - docker push
  - docker run
  - docker pull

Kubernetes

- Microservice architektúrában számos egymástól függő konténert kell felügyelni, terhelésnek megfelelően újakat indítani/leállítani → ennek megoldására a legelterjdtebb eszköz a Kubernetes
  - Számos Microservices tervezési mintát is megvalósít
- A Google megoldása

Miben segít a Kubernetes?

- Service discovery és load balancing
- Storage orchestration
  - Automatikus tárolórendszer felcsatolás (lokális, publikus cloud, stb.)
- Automatizált rollout és rollback
- Vízszintes skálázás, akár automatikusan
- Erőforrások (CPU, RAM) automatikus elosztása
- Self-healing
  - Újraindítja a hibás konténereket
  - Kicseréli, leállítja azokat a konténereket, amik nem válaszolnak
  - Csak a futásra kész konténereket ajánlja ki a klienseknek
- Secret és konfiguráció menedzsment
  - Érzékeny adatok tárolása (jelszavak, Oauth tokenek, ssh kulcsok)
  - Secretek és konfigurációk lecserélése anélkül, hogy a konténer image-eket módosítani kellene
- Mindent a konténerek szintjén kezel → Mindegy, milyen platformon/technológiával fejlesztettük a microservice-eket