Szoftvertechnológia

ÓBUDAI EGYETEM
NEUMANN JÁNOS INFORMATIKAI KAR

MODUL 10

Domain-driven design

CQRS

Mikroservices

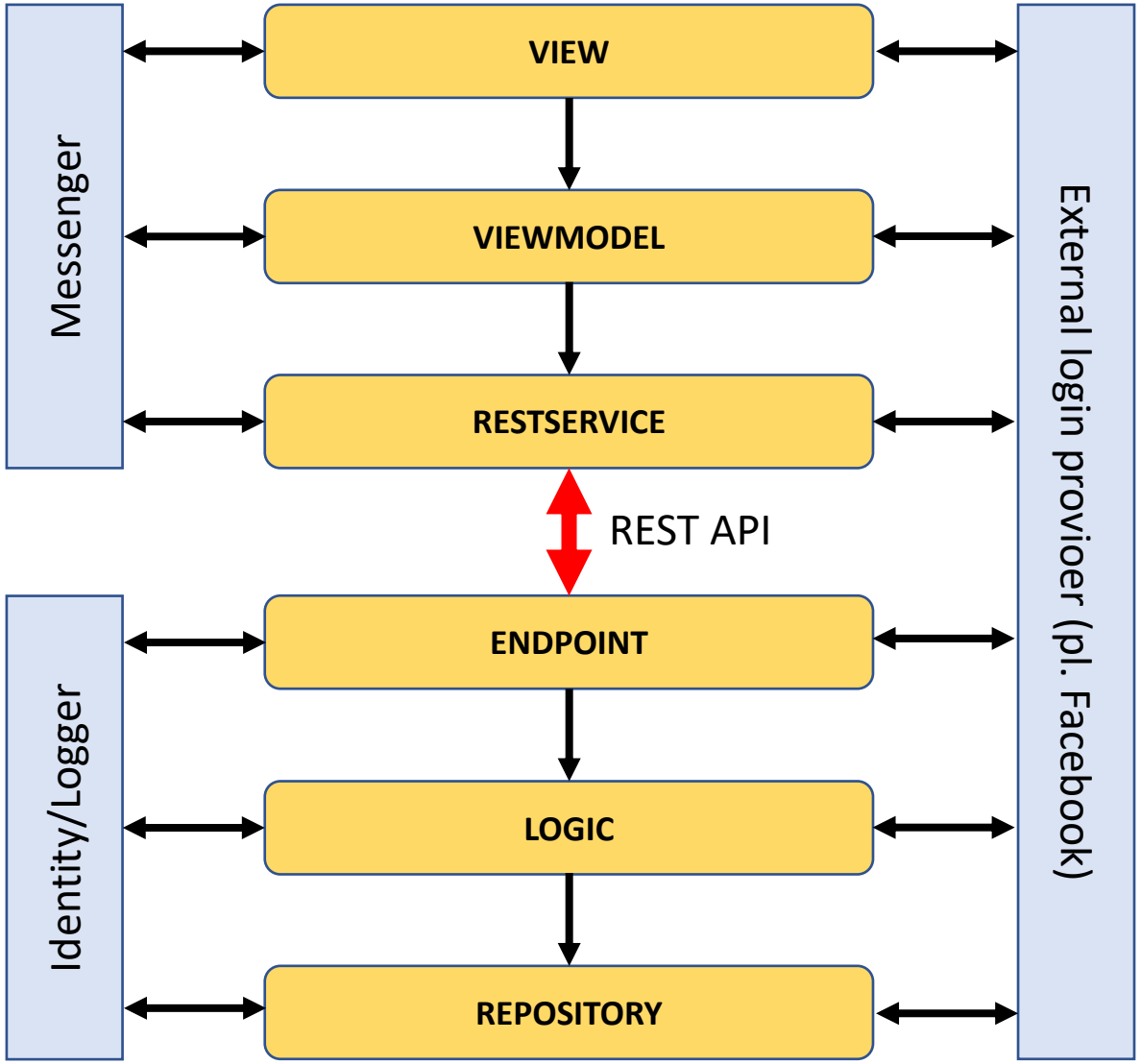
Event brokers

Eddigi rétegzési elvek

- Adatelérés → Üzleti logika → Megjelenítés
- Ezek akkor igazán hatékonyak, hogyha
  - Felhasználó használja a rendszert
  - Alapvetően CRUD funkcionalitásra van az egész kihegyezve
- Mi használhatja még a rendszert?
  - Automata tesztek
  - API végpontok
  - Scriptek
  - Belső és külső automatizmusok
- Vannak olyan komponensek, amelyekre minden rétegben szükség van
  - Naplózás
  - Security
  - Messenger

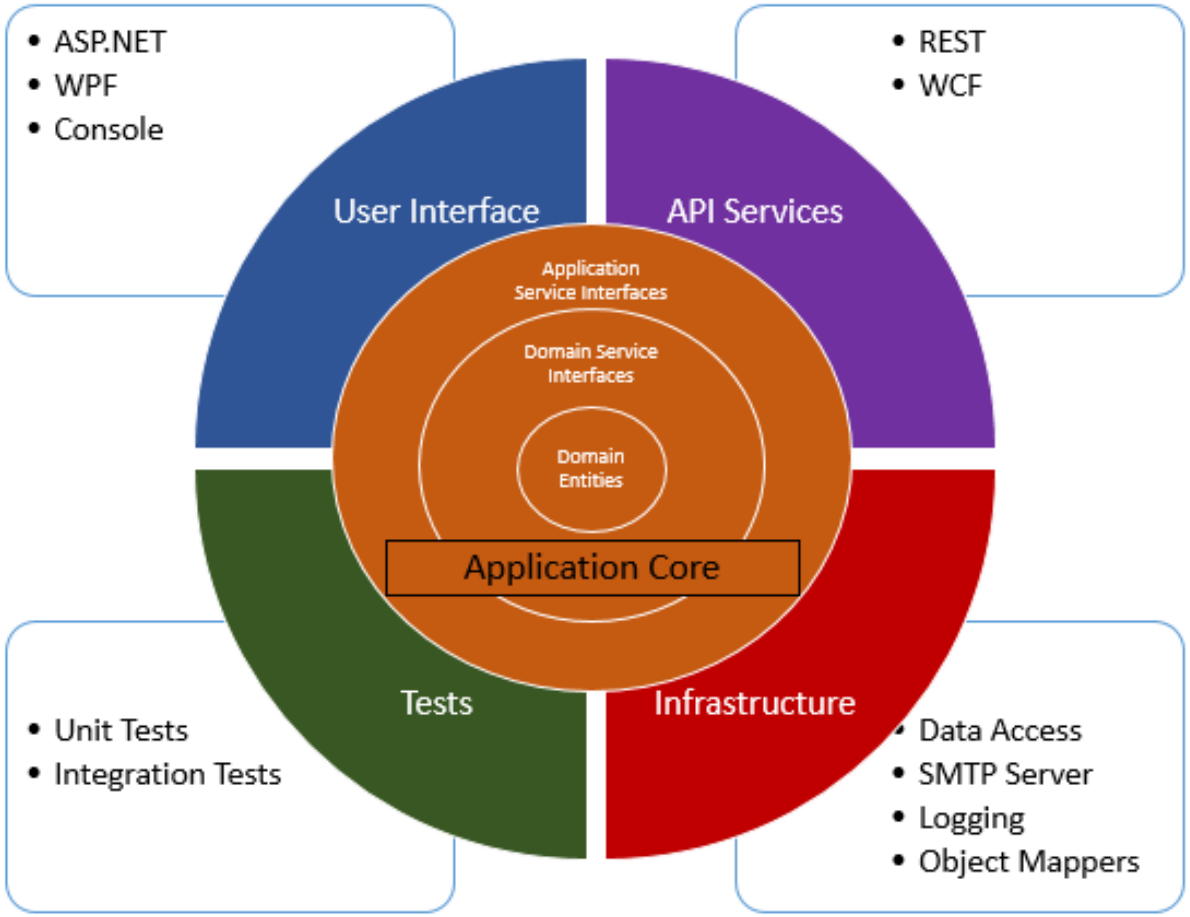
Aspect-ek

- Olyan komponensek, amelyeket minden réteg használ
- Tipikus elvárások az aspect-ekkel szemben
  - Ne kövessenek el rétegsértést
  - Legyenek szűk funkcionalitásúak



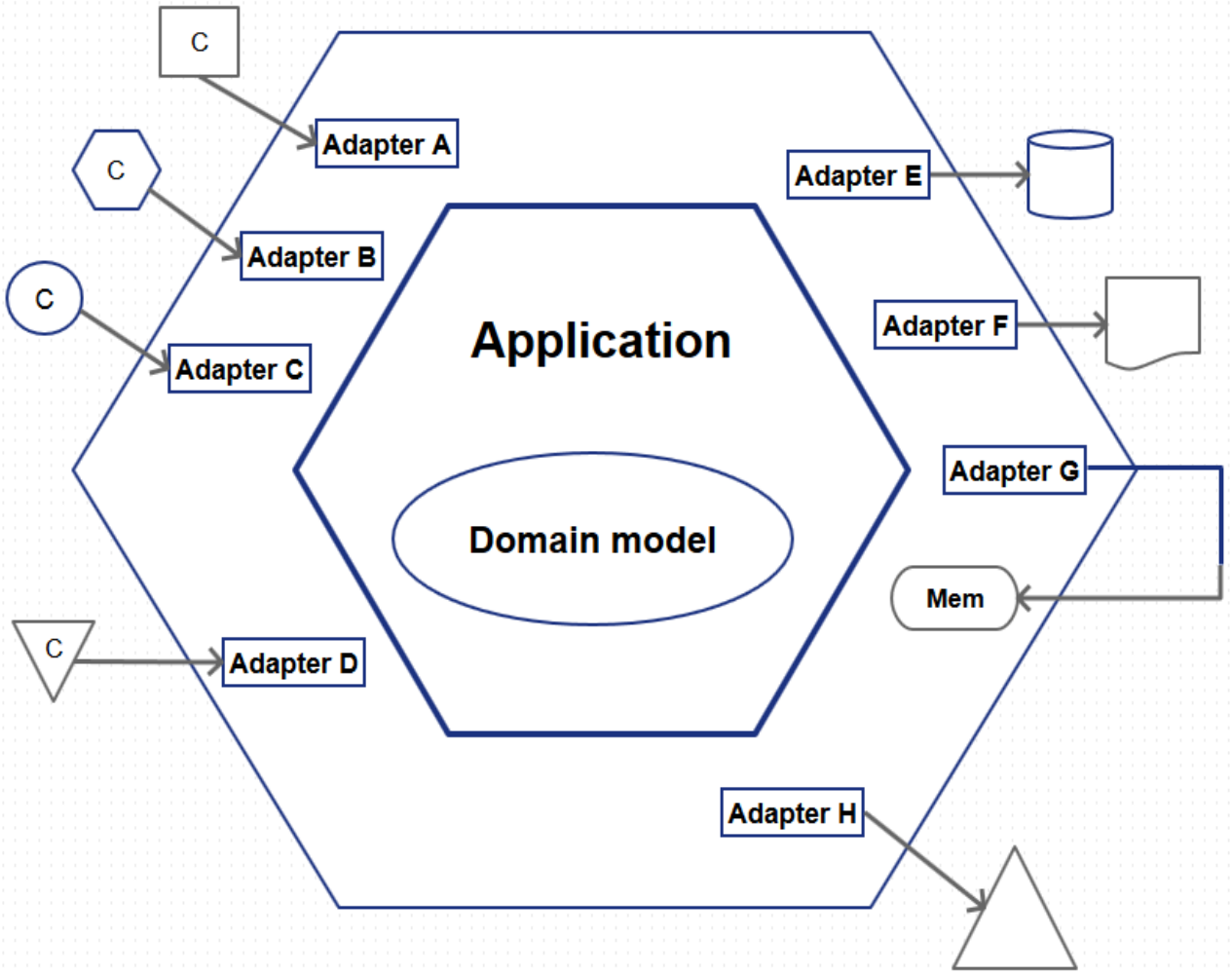
Onion architecture

- Application core = Logic
- Logic felett több réteg is elhelyezkedik
- Mindegyik más-más célt szolgál ki
- Pl: webalkalmazás
  - Logic adott
  - Logic felett
    - MVC-vel webes UI
    - API végpontok mobilapphoz



Hexagon architecture

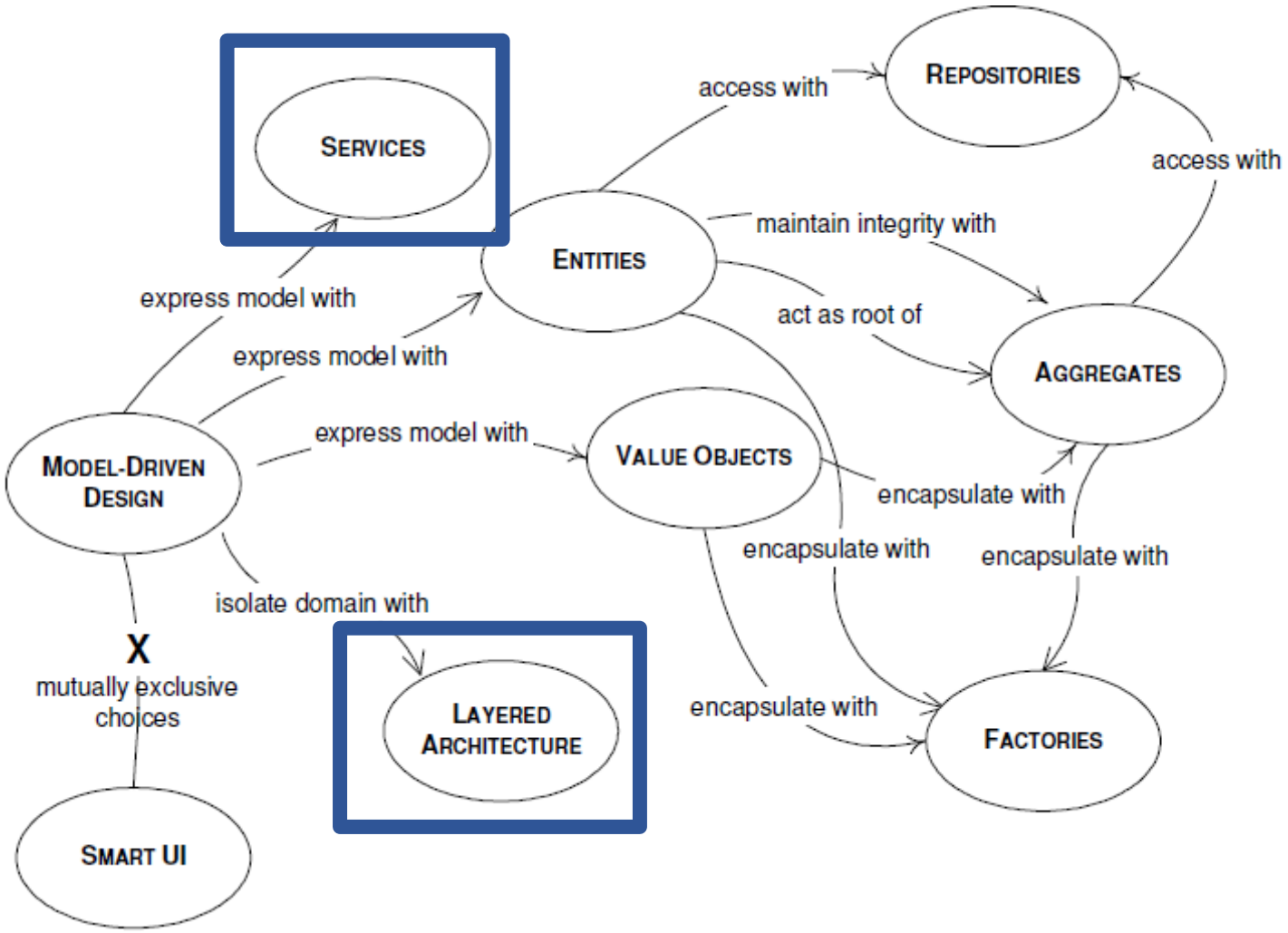
- Application core = Logic
- Logichoz külső rendszereket csatolunk adaptereken keresztül
- Külső rendszerek
  - Adattárolás
  - API végpontok
  - Webes UI
  - Email küldés
  - Logolás
  - Felhasználókezelés
  - Stb.



Domain-driven design

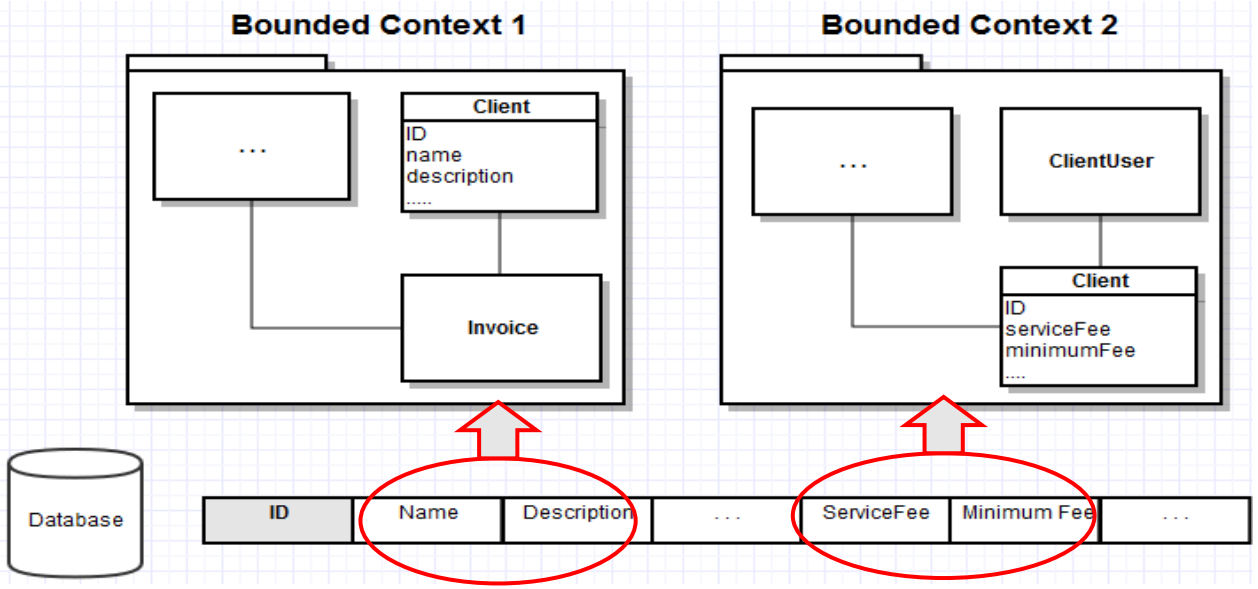
- Martin Fowler, Vaughn Vernon, Eric Evans, Udi Dahan → DDD
- Lényege: a rétegzés ne attól függjön, hogy MVC vagy API vagy bármi a UI elérési technika
  - **Attól függjön a rétegzés, hogy mit akarok csinálni az adattal**
- Irodalom a témában
  - Refactoring (Improving the Design of Existing Code, 2018 2nd ed)
  - https://martinfowler.com/books/eaa.html
  - Patterns of Enterprise Application Architecture (2003/2004) → újabb 51 db pattern!
    - Domain Logic Patterns
    - Data Source & Object-Relational Behavioral & Structural Patterns → ORM
    - Object-Relational Metadata Mapping Patterns
    - Web Presentation Patterns
    - Distribution Patterns
    - Offline Concurrency Patterns
    - Session State Patterns
    - Base Patterns

Domain-driven design



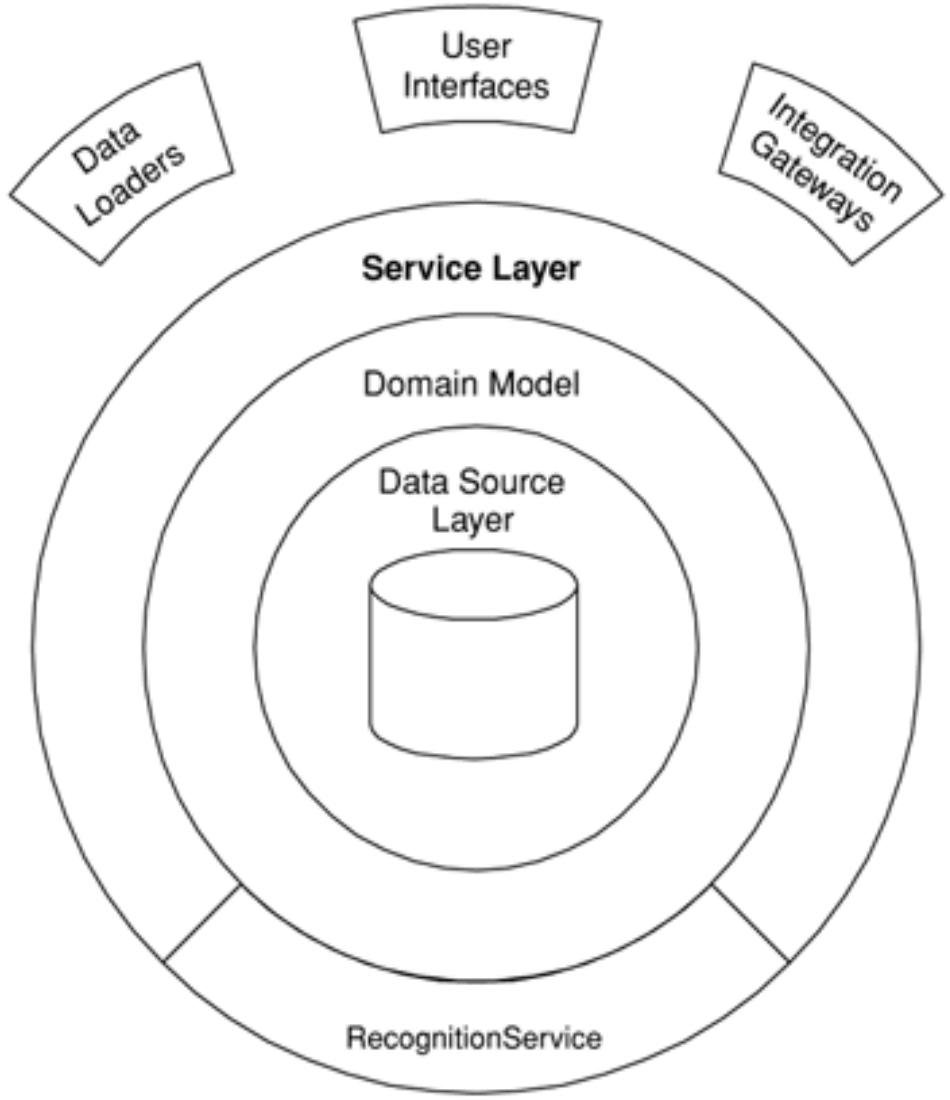
Domain-driven design

- Nem az adatbázissal kezdjük a modellezést, hanem a funkciókkal
- **Bounded Context**-eket hozunk létre → Domain model lesz belőlük
  - Jelentése: Egy User tábla szerepelhet a Szállítás domain modelben és a Számlázás domain modelben is
  - Ez így DRY elveknek ellentmond…
  - De csak látszólag, mert a Data Mapper / ORM majd valójában ugyanarra az 1 db táblára mappeli le
  - Hibalehetőségek: Bloated domain objects (túl sok felelősség), Anemic domain objects (túl kevés felelősség)



Service Layer

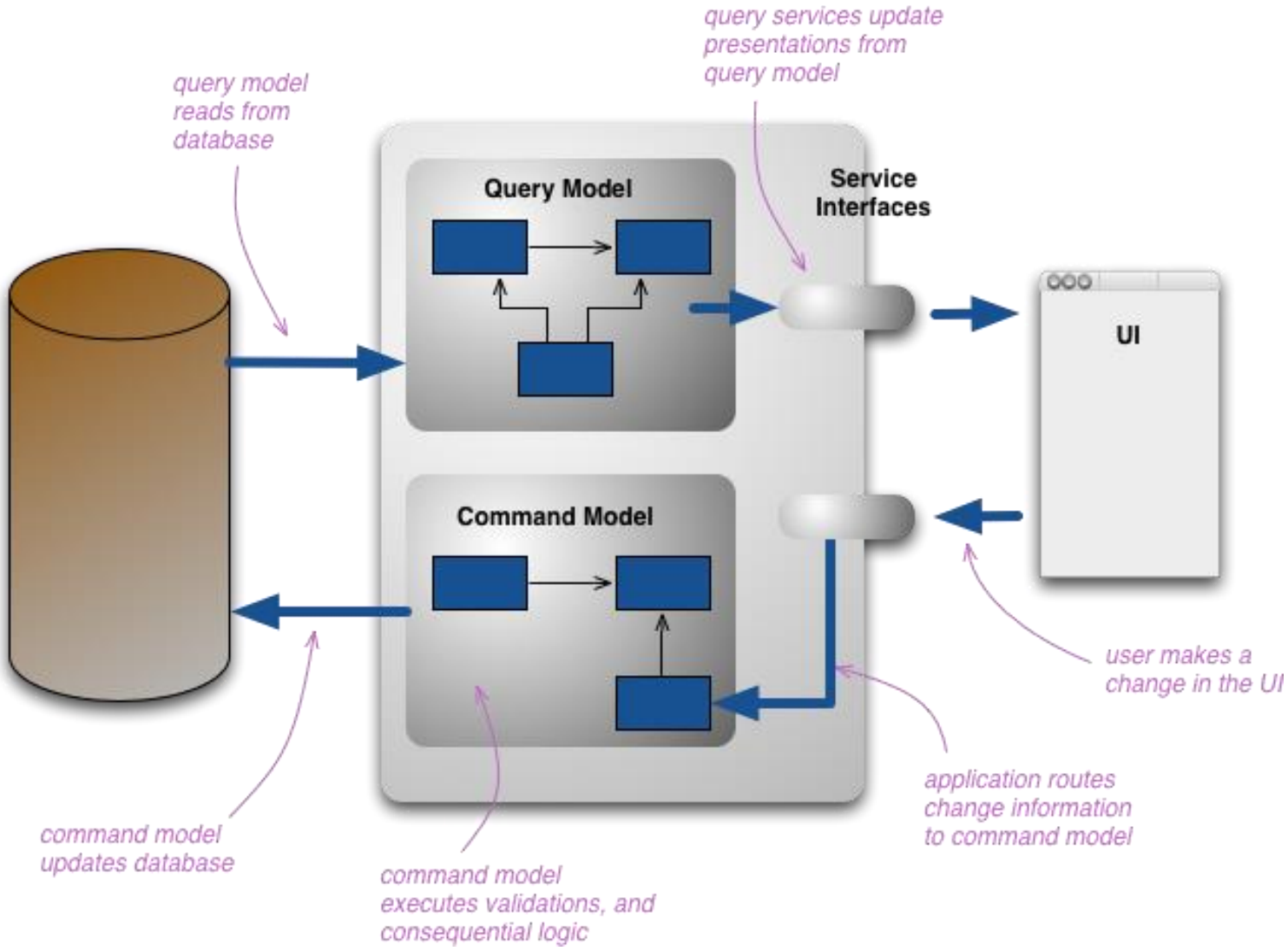
- Domain model egységbezárja a Domain Entityket és azok üzleti műveleteit
- SOA filozófiát valósít meg (Service-Oriented Architecture) → Microservices
- Feladata: hívások fogadása, továbbítása a Domain Logic felé
- Feladata lehet még a tranzakciókezelés és a lock is
- Alsóbb rétegekben megjelenik ettől függetlenül az adatbázis szintű tranzakciókezelés is
- UI csak egy service a sokból

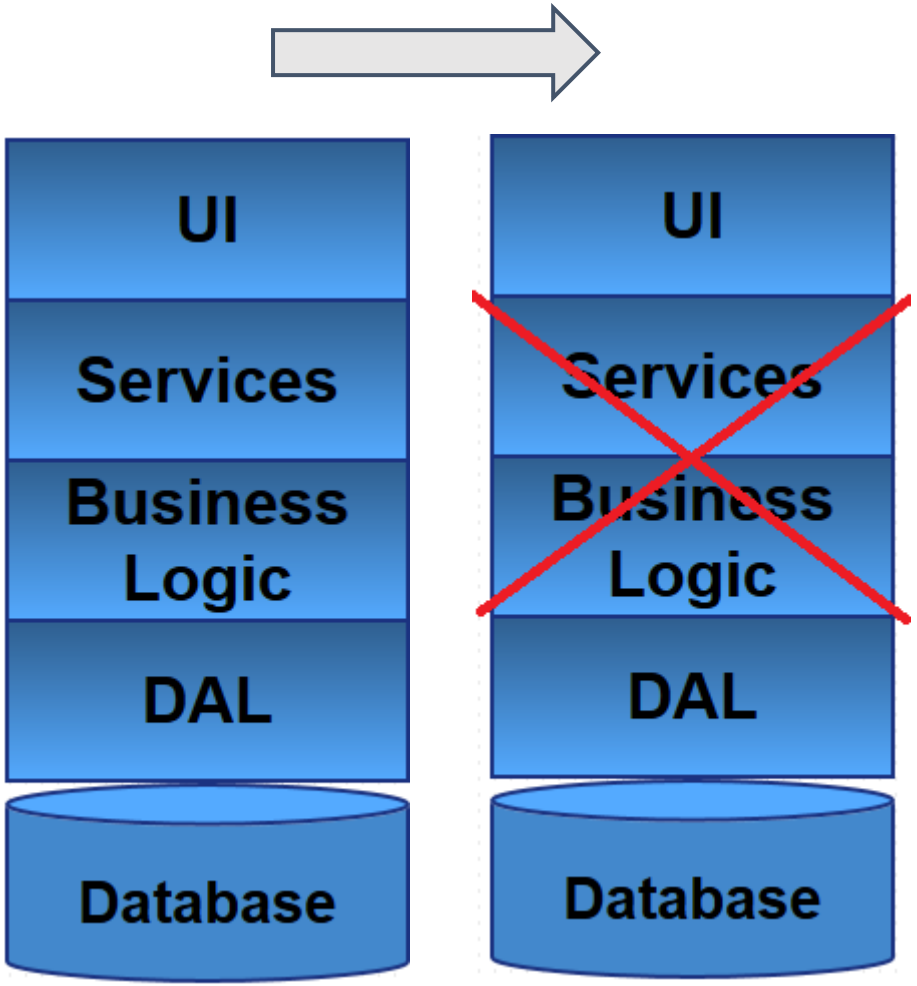


Írás vs. olvasás

- Nagy rendszerek esetén általában SOK olvasási művelet és KEVÉS írási művelet történik
- Írási műveletek
  - Tipikusan egy bounded context-be akarunk írni (pl. egy számla létrehozása)
  - Kell minden alrendszer hozzá (jogosultság kezelés, validáció, tranzakciókezelés, stb.)
- Olvasási műveletek
  - „Dashboard” és „Reports” funkcionalitás nagyon gyakori
  - Általában több bounded context-ből kell összeszedni az adatokat (group by/join lassú…)
  - Egy csomó alrendszer kikerülhető akár (nem kell validáció, naplózás, tranzakciókezelés)
- Ez szétválasztható lenne két nagy alrendszerre
  - CQRS (Command-Query Responsibility Segregation)

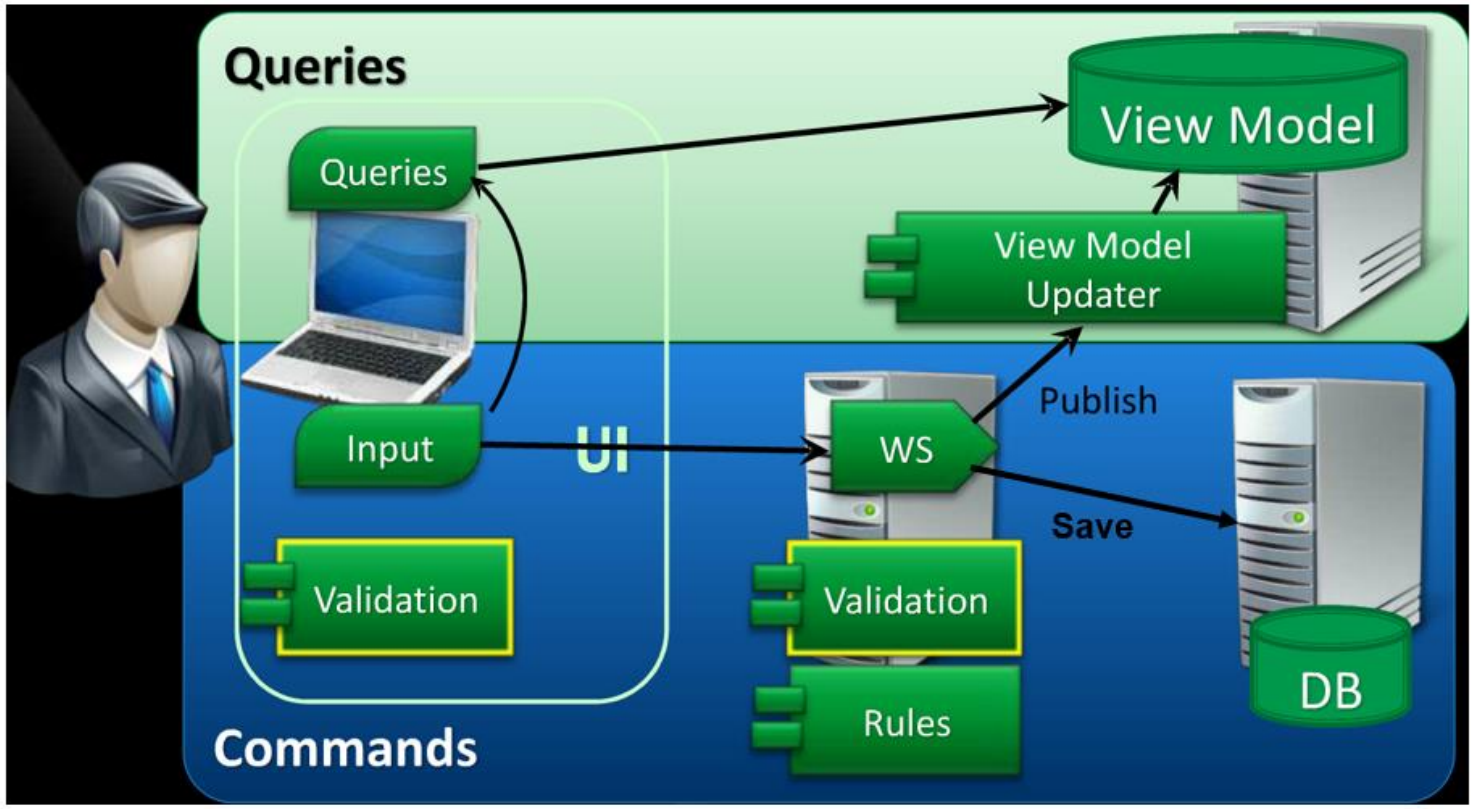
CQRS


query model reads from database
query services update presentations from query model
Query Model
Service Interfaces
Command Model
UI
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic


UI
Services
Business Logic
DAL
Database
UI
Services
Business Logic
DAL
Database

Olvasás gyorsítása

- A User lásson mindig régi adatot
- Pl. Neptun → Képzettség adatok (dashboard) → sok bounded contextből áll össze
- Megjelenítése lehet, hogy 3-4 mp is lenne
- **Megoldás**
  - Persistent View Model → minden éjfélkor pl. egy adatbázisba mentsük ki az összes user dashboardját
  - Innen a betöltést 1 db select 1 db where záradékkal



CQRS hibalehetőségek

- **Query oldal** – Olvasási műveletek
  - Egyedi keresések problémája
    - Generálható minden éjjel a top 100 népszerű keresés találati oldala (pl: budai penthouse lakások)
    - A ritka keresések majd tovább tartanak (gond?)
    - Adatbázisok optimalizálhatóak keresésekre
    - Gyors keresésre optimalizált DB: ElasticSearch
- **Command oldal** – Írási műveletek
  - Hibára futás ritka
  - Szinkron hibajelzés feleslegesen lassít
  - Aszinkron hibajelzés
    - Azt mondjuk, hogy sikeres a foglalás ☺
    - Ha netán valami baj van, akkor 5-10 perc múlva küldünk egy mailt, hogy mégis hiba történt
    - Aszinkron reagáló mechanizmusok → akár kézi megoldások
  - Azért ez nem mindenhol használható!!! → jegyfoglalás, tárgyfelvételnél tipikusan nem
  - Ahol szóba jöhet: hozzászólás, üzenetküldés, értékelés, megrendelés, jelentkezés, regisztráció, stb.

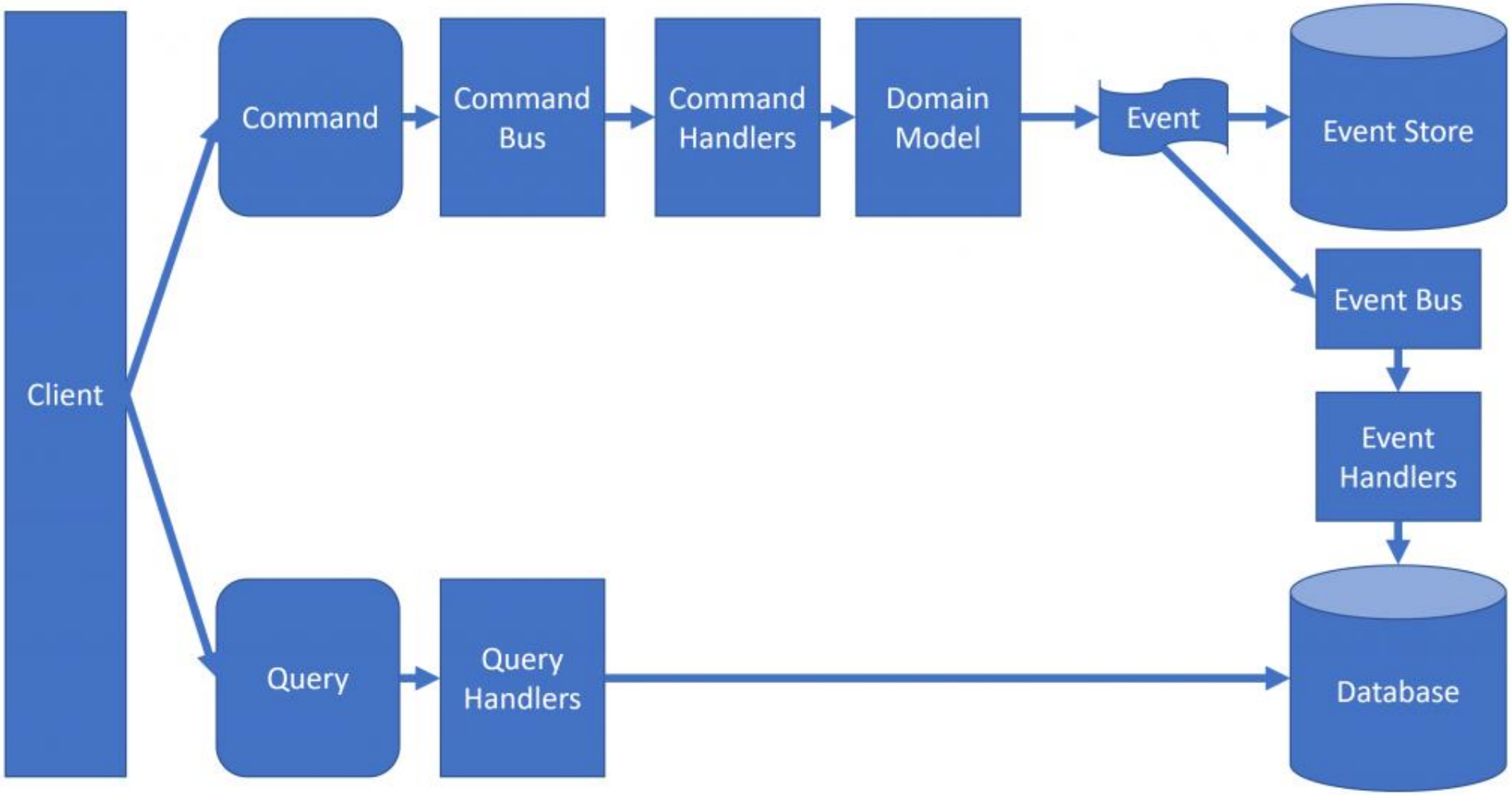
Event Sourcing

- Tipikus probléma, hogy gyorsabban jön az input, minthogy fel tudnánk dolgozni
- Pl: Egy szenzor akarna 1mp-enként adatot küldeni, de a szerver annyira túlterhelt, hogy 3 mp múlva jön meg a HTTP response
  - Gyakorlatilag feltorlódnak a kérések és használhatatlan lesz a rendszer
- Pl: Kijön az új Iphone, van belőle 100.000 db készleten
  - Éjféltől lehet előrendelni
  - Korábbi évek statisztikái alapján 90.000 – 100.000 rendelés fog jönni
  - Nem ellenőrzünk minden rendelés előtt, hogy van-e biztosan még
  - Mindenkinek visszaigazoljuk azonnal, hogy megkaptuk a rendelést
  - Elmentjük a rendeléseket egy várósorba
  - Később elkezdjük ténylegesen feldolgozni a rendeléseket

Event Sourcing

- Megoldás: várósorba mentés → Event Store
- Technika
  - Redis
  - RabbitMQ
  - MQTT
  - HiveMQ
- Ezek az adatbázisok arra vannak optimalizálva, hogy villámgyorsan képesek legyenek elmenti kéréseket, nagyságrendekkel gyorsabban, mint egy relációs adatbázis
  - Előző szenzoros problémafelvetést is megoldja
  - Majd a valós szerverek aszinkron módon kiszedegetik innen a beérkezett kéréseket

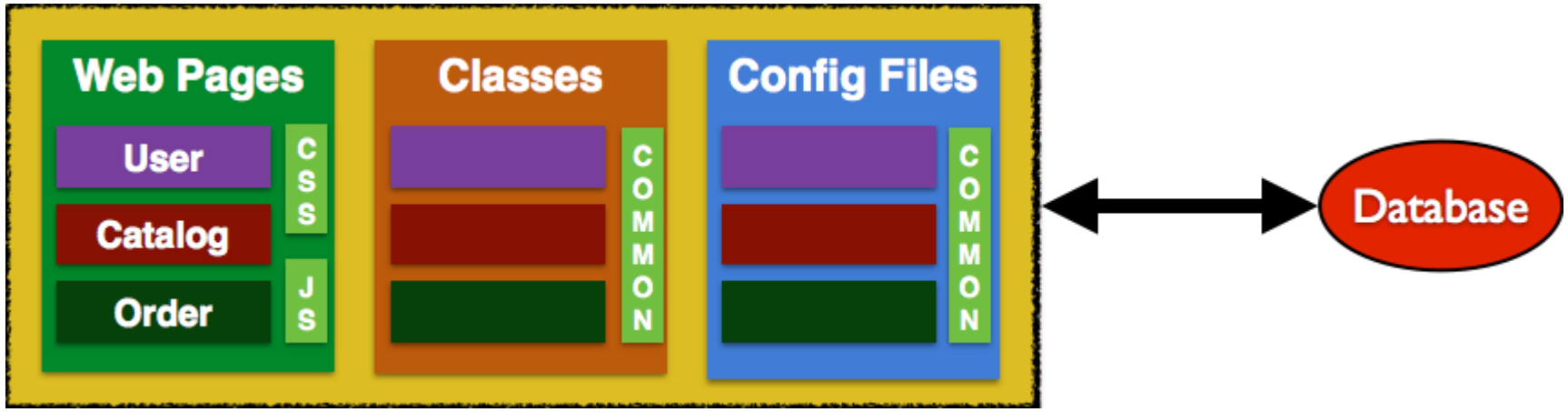
CQRS (+ Event sourcing, DB szétvágás nélkül)



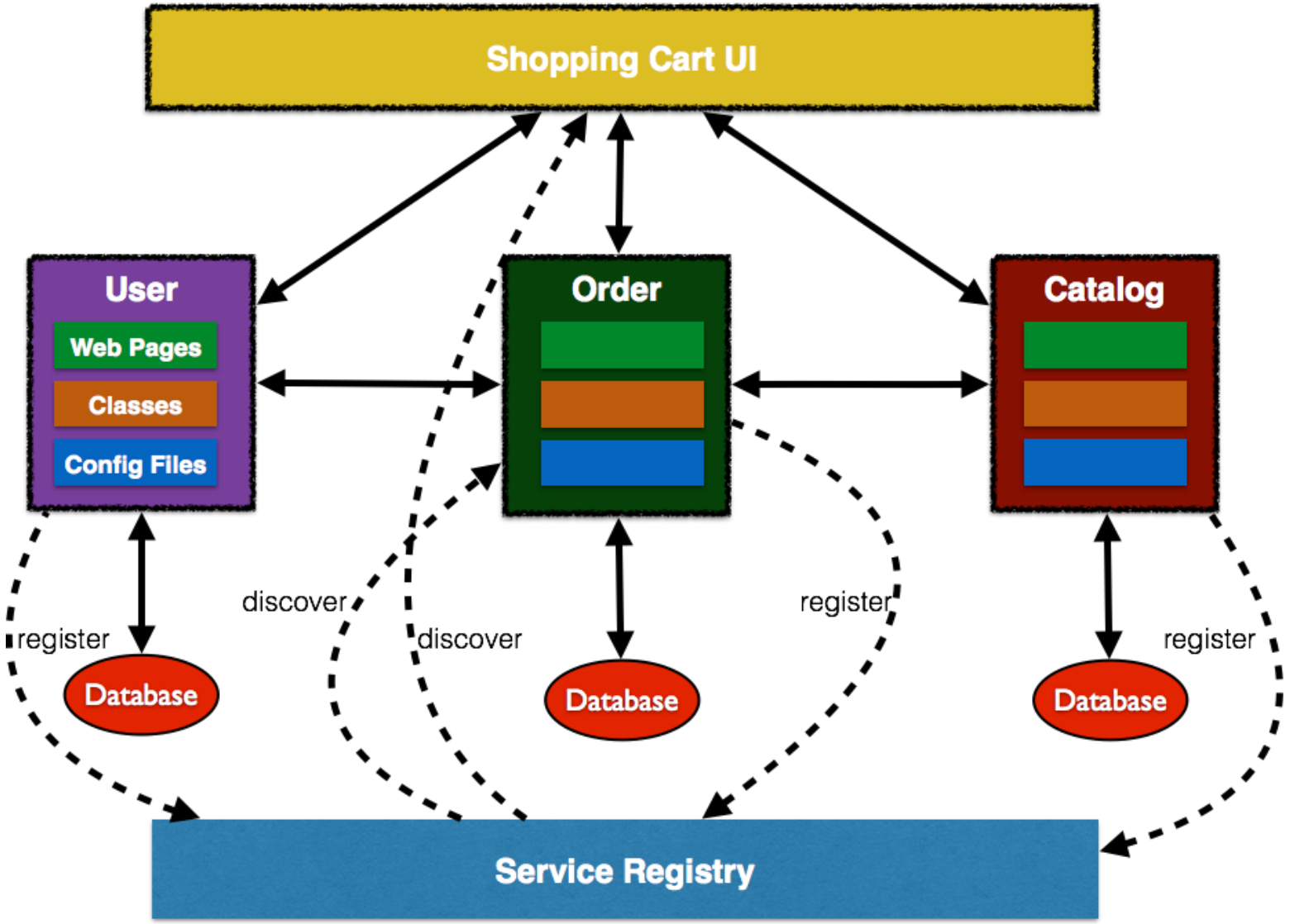
Event sourcing + CQRS + DDD

- **Előnyei**
  - Nagy teljesítmény
  - Egyszerűbb a rendszerek összeépítése
  - Könnyű hibakeresés, tesztelés
  - Event Store-ból extra üzleti adat is kinyerhető
- **Hátrányai**
  - Reporting bonyolult
  - Nagyobb tárigény → Persisent View Model eltárolása
  - Hibás kérések visszajelzése nem azonnali

Monolitikus alkalmazás



Mikroszolgáltatások

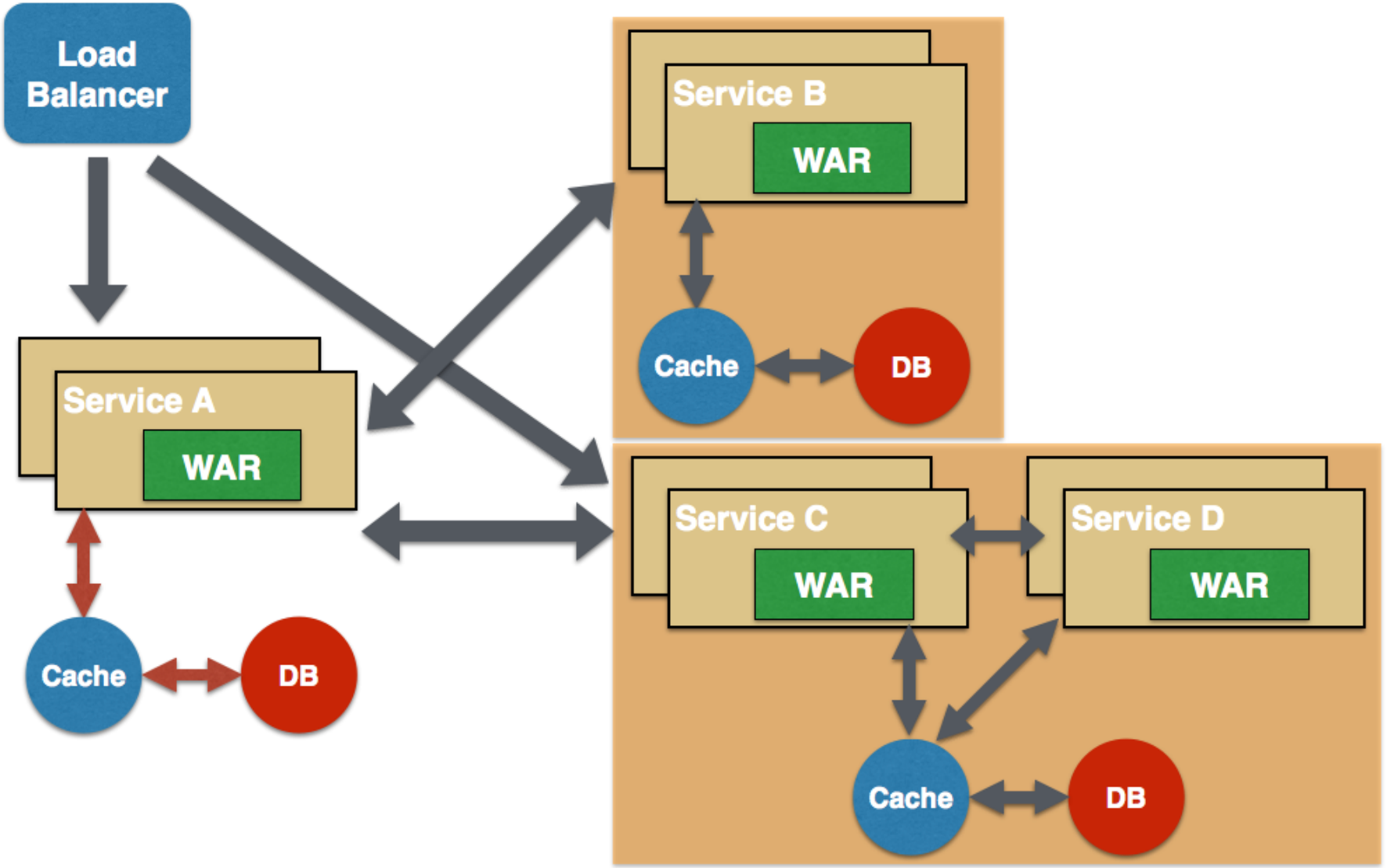


- **Jellemzői**
  - Önálló alkalmazások
  - Felbontás alapja: domain model → bounded context
  - Önálló fejlesztési ciklus
  - Önállóan tesztelhetőek (service stubokkal)
  - Önállóan skálázhatóak
  - Akár más-más nyelven írhatóak
  - Egymással valamilyen közösen ismert protokollon keresztül beszélgetnek
  - Tipikusan konténerbe zárunk egy-egy mikroszolgáltatást

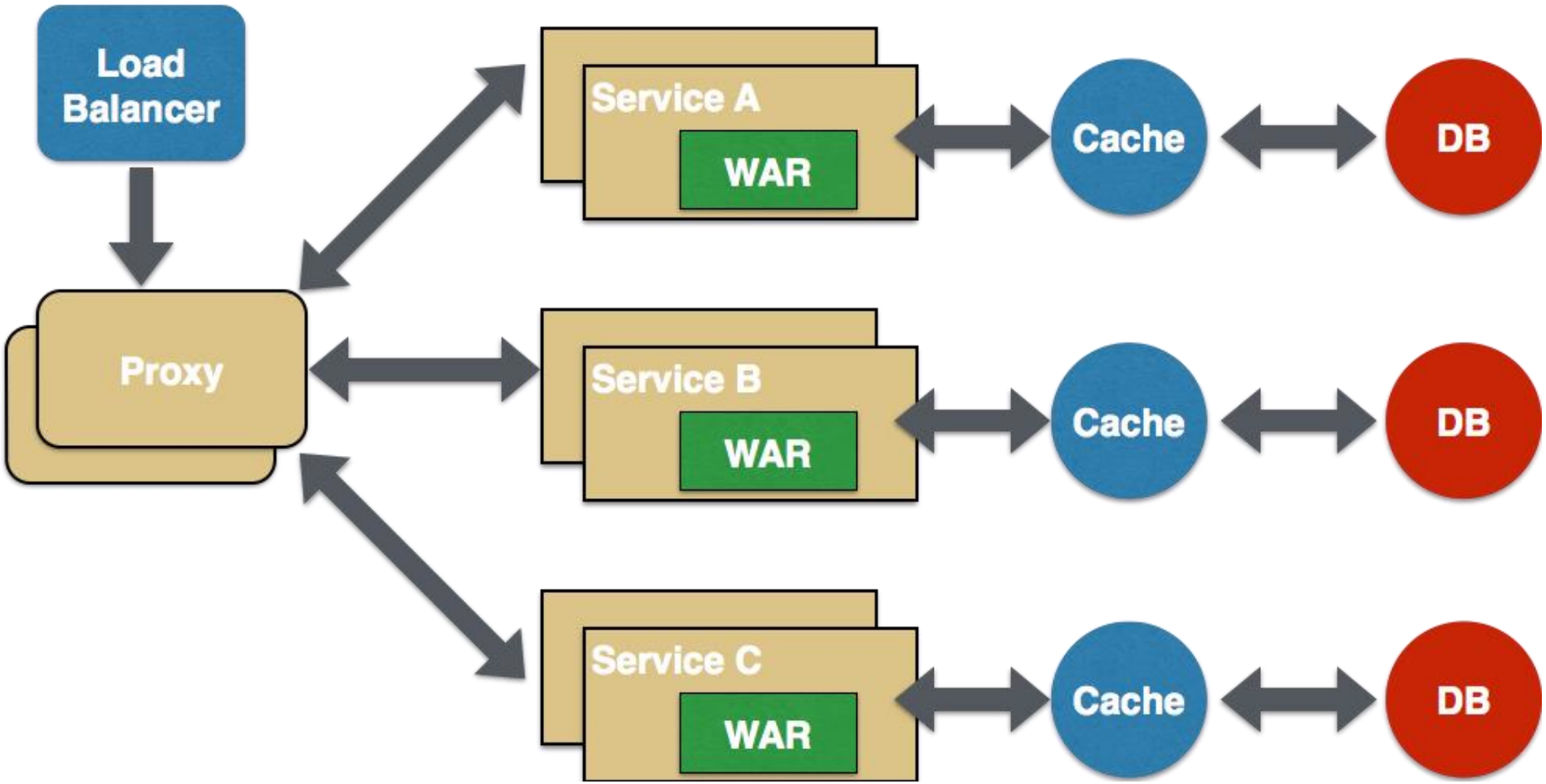
Mikroszolgáltatás felépítése

- **Belső működés**
  - Hagyományos rétegezéssel épül fel
  - Saját adatbázissal rendelkezik/rendelkezhet
  - Kommunikáció pl. REST API vagy Message bus protokollok
- **Külső elérés**
  - Tipikusan REST API-n keresztül
  - UI is egy mikroszolgáltatás, ami megjelenítésért felelős

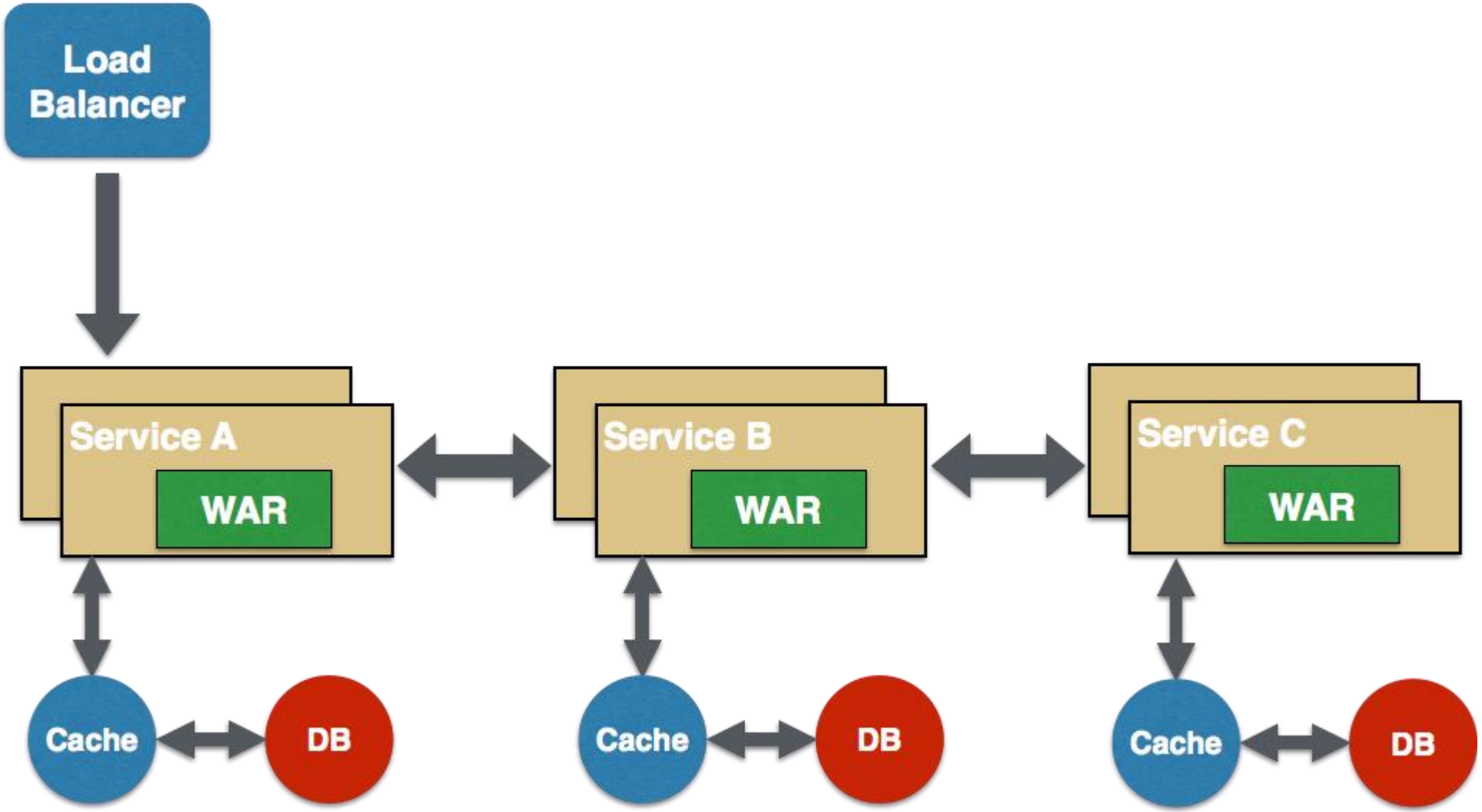
Tervezési minták – nem strukturált



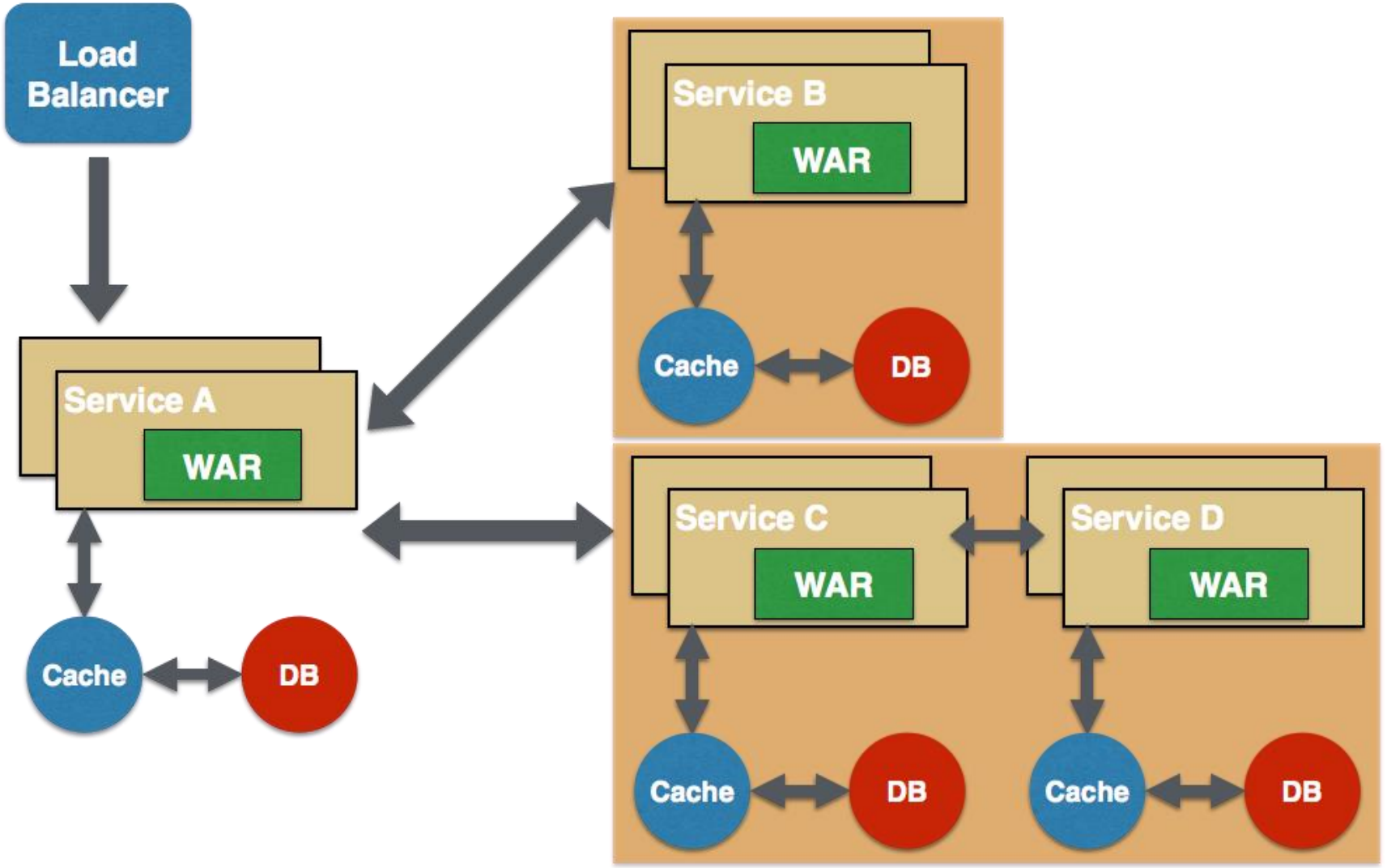
Proxy/Facade



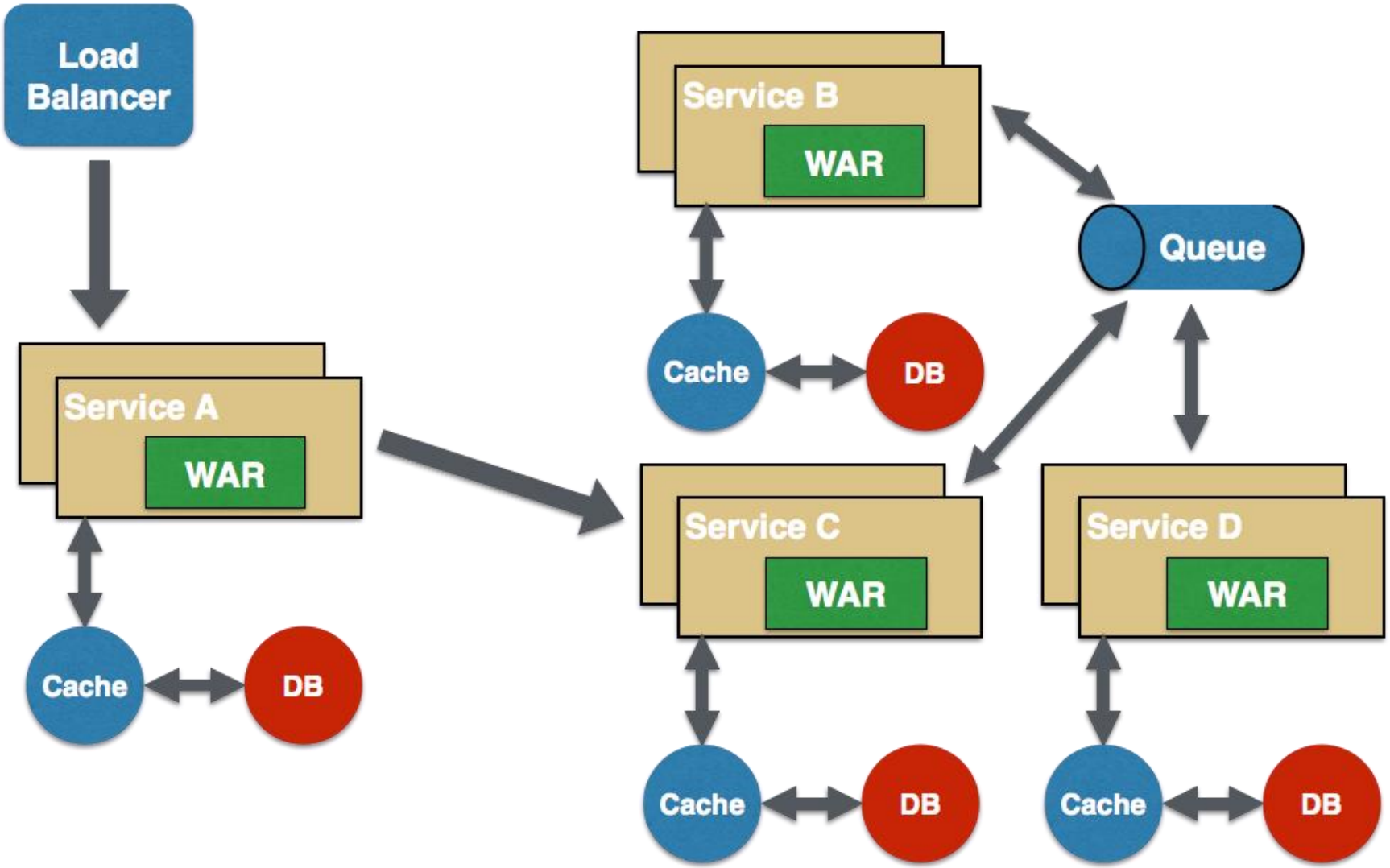
Chain or Responsibility



Composite


Load Balancer
Service A
WAR
Cache
DB
Service B
WAR
Cache
DB
Service C
WAR
Cache
DB
Service D
WAR
Cache
DB

Mediator


Load Balancer
Service A
WAR
Cache
DB
Service B
WAR
Cache
DB
Queue
Service C
WAR
Cache
DB
Service D
WAR
Cache
DB

Mikroszolgáltatás kommunikáció

- **Mediator technológia**
  - **AMQP**: Advenced Message Queueing Protocol
    - Általános, nyílt protokoll
    - Tipikusan PC/WEB
  - **MQTT**: Message Queue Telemetry Transport
    - ISO szabvány
    - Publish – Subscribe üzenetküldésre tökéletes
    - Kis overhead
    - Tipikusan mobil/IOT
    - Brokers
      - Mosquitto: 30k msg / sec
      - Moquette: 30-100k msg / sec
      - HiveMQ: 800k msg / sec
      - Redis: 1M msg / sec

Köszönöm a figyelmet!

Kérdés esetén e-mailben szívesen állok rendelkezésre.