Domain-Driven-Design filozófia

# Domain-Driven-Design

## Lényege

* A rétegzés ne attól függjön, hogy MVC vagy API vagy bármilyen a UI elérési technika.
  + Attól függjön, hogy mit akarok csinálni az adattal.
* Nem az adatbázissal kezdjük a modellezést, hanem a funkciókkal.

## Bounded Context

* **Bounded context-eket hozunk létre:**
  + Domain model lesz belőlük
  + Jelentése, hogy egy user tábla szerepelhet a szállítás domain model-ben és a számlázás model-ben is.
  + DRY-elveknek ellentmond, de csak látszólag mert a Data Mapper / ORM majd valójában ugyanarra az 1db táblára mappeli le.
* **Hibalehetőségek:**
  + Bloated domain objects (túl sok felelőség)
  + Anemic domain objects (túl kevés felelőség)

# Hexagon/Onion architektúrától a flexibilis rétegzésig

### Onion architektúra

### Hexagon architektúra

# DDD megközelítés az írás-olvasás szétválasztásához

* Nagy rendszereknél általában SOK olvasási művelet és KEVÉS írási művelet történik.
* Írási műveletek
  + Tipikusan egy bounded context-be akarunk írni.
  + Kell minden alrendszer hozzá
* Olvasási műveletek
  + Dashboard és Reports funkcionalitás nagyon gyakori.
  + Általában több bounded context-ből kell összeszedni az adatokat.
  + Egy csomó alrendszer kikerülhető akár.

# CQRS optimalizáció a lekérdezés (query) és az utasítás (command) oldalon

### Query oldal – Olvasási műveletek

* Egyedi kérések problémája
  + Adatbázisok optimalizálhatóak kérésekre.
  + Gyors keresésre optimalizált DB: ElasticSearch

### Command oldal – Írási műveletek

* Hibára futás ritka
* Szinkron hibajelzés feleslegesen lassít
* Aszinkron hibajelzés
  + Sikeres foglalás
  + Ha baj van, akkor email küldése, hogy hiba történt.
  + Aszinkron reagáló mechanizmusok

# Event sourcing

* Probléma, hogy gyorsabban jön az input, minthogy fel tudnánk dolgozni.
* Például egy szenzor akarna 1mp-enként adatot küldeni, de a szerver annyira túlterhelt, hogy 3 mp múlva jön meg a HTTP response.
  + **Feltorlódnak a kérések és használhatatlan lesz a rendszer.**
* **Megoldás:**
  + Várósorba mentés, vagyis Event Store
  + Technika
    - Redis
    - RabbitMQ
    - MQTT
    - HiveMQ
  + Ezek az adatbázisok arra vannak optimalizálva, hogy villámgyorsan képesek legyenek elmenteni kéréseket, nagyságrendekkel gyorsabban, mint egy relációs adatbázis.

### 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
  + Hibás kérések visszajelzése nem azonnali