12. Domain-Driven-Design filozófia

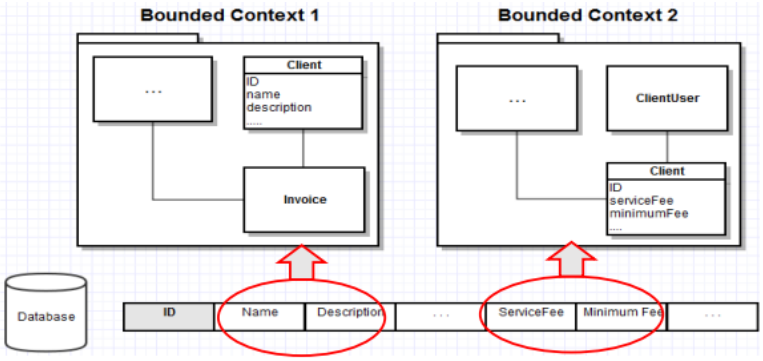
Domain-Driven-Design - DDD

Lényege

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

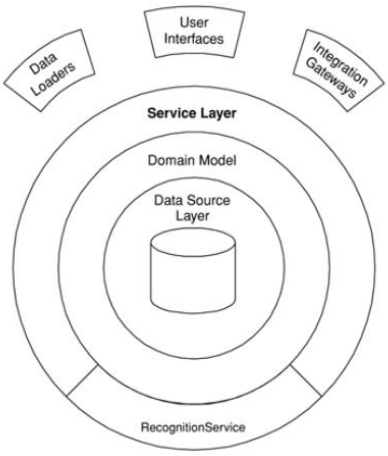
Bounded Context

- **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)



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.
    - Tranzakciókezelés, lock
- Alsóbb rétegekben megjelenik ettől függetlenül az adatbázis szintű tranzakciókezelés is.
- UI csak egy service a sokból.



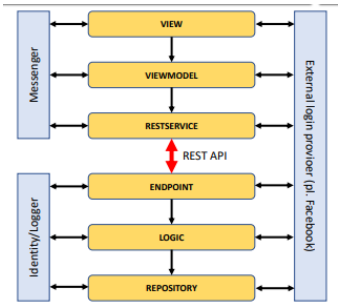
Eltérés a rétegzéstől

Eddigi rétegzési elvek

- Adatelérés → Üzleti logika → Megjelenítés
- Ezek akkor hatékonyak, ha
    - Felhasználó használja a rendszert
    - CRUD funkcionalitásra van kialakítva
- Mi használhatja még a rendszert?
    - Automata tesztek
    - API végpontok
    - Scriptek
    - Belső és külső automatizmusok

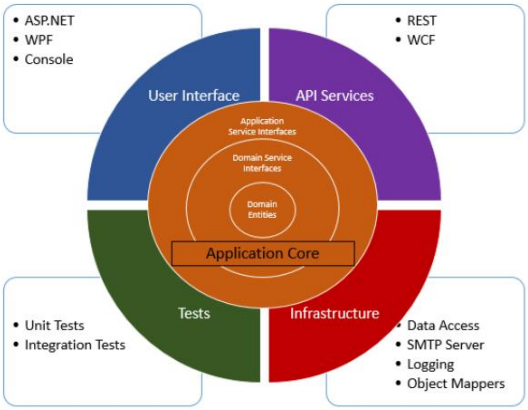
Aspect-ek

- Olyan komponensek, amiket minden réteg használ.
- Elvárások az aspect-ekkel szemben
    - Ne kövessenek el rétegsértést
    - Legyenek szűk funkcionalitásúak



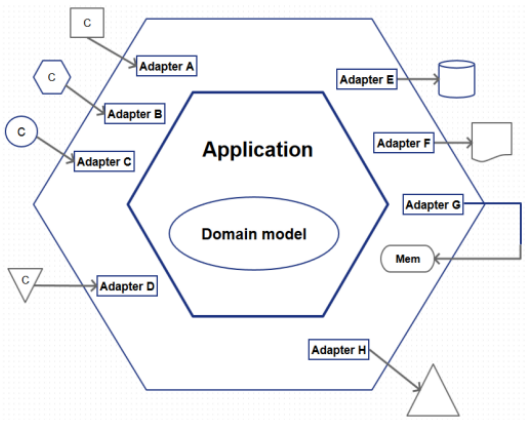
Onion architecture

- Application core = Logic
- Logic felett több réteg is elhelyezkedik
- Mindegyik más-más célt szolgál ki
- Például webalkalmazásnál
    - Logic adott
    - Logic felett



Hexagon architecture

- Application core = Logic
- Logichoz külső rendszereket csatolunk adaptereken keresztül.
- Külső rendszerek. (**Adapter tervezési minta**)
    - Adattárolás
    - API végpontok
    - Webes UI
    - Email küldés
    - Logolás
    - Felhasználókezelés



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. (pl.: számla létrehozása)
    - Kell minden alrendszer hozzá. (pl.: jogosultság kezelés, validáció)
- 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)
- **Szétválasztható két nagy alrendszerre**
    - **CRRS (Command-Query Responsibility Segregation)**

Olvasás gyorsítása

- Felhasználó lásson régebbi adatokat.
- Megjelenítés x mp lehet.
- **Megoldás:** Persistent View Model, ahol különböző szabályokat állítunk fel, hogy mikor legyen frissítve.

CQRS hibalehetőségek

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 (akár kézi megoldások)

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 3mp múlva jön meg a HTTP response.
    - **Feltorlódnak a kérések és használhatatlan lesz a rendszer.**

Event sourcing példa

- Kijön egy új termék, amiből van x db a készleten, ahol rendeléseket fogadunk.
- **Megoldás**
    - Nem ellenőrzünk minden rendelés előtt, hogy van-e biztosan még.
    - Mindenkinek visszaigazoljuk azonnal, hogy megkaptuk a rendelést.
    - Elmentjük a rendeléseket egy várósorba.
    - Később kezdjük el ténylegesen feldolgozni a rendeléseket.
        - **Később hozzuk meg a tényleges döntést!**

Event sourcing technikák

- Várósorba mentés, vagyis Event Store
- Technika
    - Redis, RabbitMQ, MQTT, HiveMQ
- Ezek az adatbázisok arra vannak optimalizálva, hogy gyorsan 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
        - Persistent View Model tárolása miatt (adat duplikáció)
    - Hibás kérések visszajelzése nem azonnali