Domain driven design – Entity

Entitatea (Entity) este un obiect fundamental definit nu prin atributele sale, ci printr-un thread de continuitate și identitate.

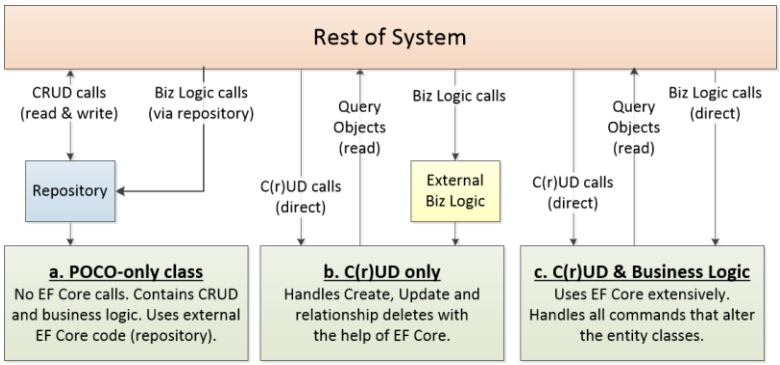
Abordarea DDD pentru scrierea clasei de entități în EF Core face ca fiecare proprietate să fie doar citită. Singura modalitate de a crea sau actualiza date despre entitate sunt constructorii, fabricile sau metodele din clasa entității.

Argumente pro si contra pentru o clasa de entitati standard si o clasa cu abordare DDD:

- Avantaje pentru o clasa de entitati standard: Simplu. Cod minim
- Avantaje pentru o clasa cu abordare DDD : Foarte observabil, numele variabilelor cu sens. O bun control a accesa datele.
- Dezavantaje pentru o clasa de entitati standard: Un sistem mare devine foarte greu de inteles. Posibilitatea de cod dublicat.
- Dezavantaje pentru o clasa cu abordare DDD: Cod putin de scris.

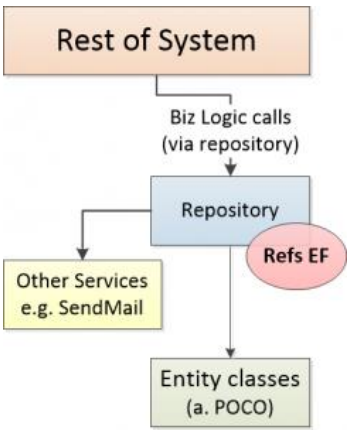
Pe langa Aborarea DDD exista inca 3 abordari DDD care sunt prezentate in diagrama de mai jos indicand ce cod contine fiecare clasa entitate.

- **Repository**: un model de depozit oferă un set de metode pentru a accesa baza de date. Aceste metode "ascund" codul necesar implementării diferitelor caracteristici de bază de date.
- **Query Objects**: Acesta este un model de proiectare pentru construirea de interogări de bază de date eficiente pentru EF.
- **C(r)UD**: Aceasta este Creare, (Citire), Actualizare și Ștergere - Folosind termenul C (r) UD pentru a separa citirea de funcțiile care schimbă baza de date (Creare, actualizare sunt șterse).
- **Business logic**: procese mai complexe care depășesc validarea simplă. Acestea pot avea calcule complexe (de exemplu, un motor de tarifare) sau pot solicita acces la sisteme externe (de ex. Trimiterea unui e-mail unui client).



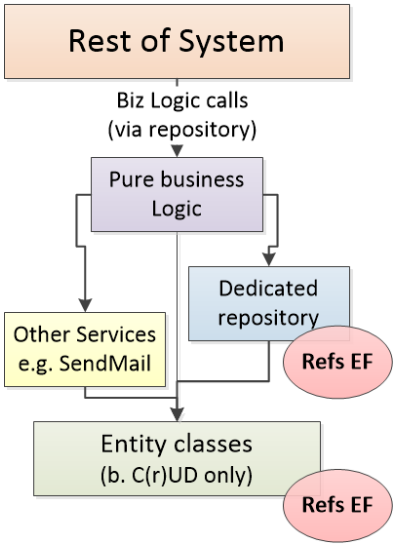
Cele 3 abordari: POCO-only class, C(r)UD only si C(r)UD & Business Logic

Abordarea POCO-only



Abordarea exclusivă a clasei POCO utilizează un depozit pentru accesarea bazelor de date, astfel încât atunci când logica de afaceri are nevoie de date pe care le bazează pe depozit. Deoarece nu puteți avea referințe circulare la ansambluri, aceasta înseamnă că o parte din logica de afaceri se mută în depozit. În mod obișnuit, rezultă că depozitul ia o coordonare a construcției ordinului (model de comandă).

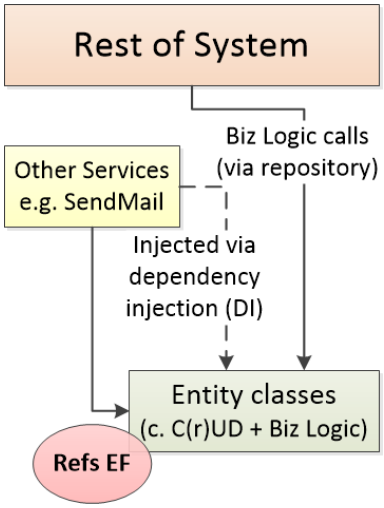
Abordarea C(r)UD



De-a lungul anilor s-a dezvoltat o serie de reguli care să puna în aplicare logica afacerii. Și unul dintre ele este "Logica de afaceri ar trebui să creadă că lucrează la date în memorie". După ce a decis că modelul generic de depozit nu funcționează bine cu EF Core.

1. BizLayer: acesta conține clase care sunt logică de afaceri pură, adică nu accesează direct EF Core, ci se bazează pe propriul mini-depozit pentru toate acțiunile bazei de date
2. BizDBAccess: Aceasta se ocupă de toate accesările bazelor de date pentru o singură clasă de logică de afaceri, adică este un depozit dedicat logicii de afaceri.

Abordarea C(r)UD si business logic



Versiunea finală are metode / fabrici pentru TOATE manipularea datelor din clasa entității. Aceasta înseamnă că nu există un depozit și nici un nivel de afacere, deoarece totul este tratat de clasa entității în sine.

În acest caz, codul logicii de afaceri care a fost în depozit (a se vedea 2.a) este mutat complet în clasa entității. Problema tehnică cu aceasta este metoda SendMail se află într-un ansamblu care este conectat la ansamblul care conține clasele de entități, ceea ce vă oprește trimiterea directă la SendMail. Este destul de simplu să rezolvați acest lucru definind o interfață (de exemplu ISendMail) în clasa entității și utilizând injecția de dependență pentru a furniza instanța SendMail la momentul executării.